Master Thesis

Achieving Unified Data Quality Representation by Constraints Transformation

Author(s):
Szymanek, Jakub

Publication Date:
2015

Permanent Link:
https://doi.org/10.3929/ethz-a-010510131

Rights / License:
In Copyright - Non-Commercial Use Permitted
Achieving Unified Data Quality Representation by Constraints Transformation

*Master Thesis*

**Jakub Szymanek**
<szjakub@student.ethz.ch>

Prof. Dr. Moira C. Norrie
David Weber

Global Information Systems Group
Institute of Information Systems
Department of Computer Science
ETH Zurich

23rd August 2015
Abstract

The reliability and usefulness of Information Systems, that due to their omnipresence are of immense importance, is largely dependent on the results they are producing. But their output, in turn, is largely determined by the quality of data that serves as input and is processed by the system. In order to ensure the high level of data quality, one needs to precisely define data quality requirements – constraints. However, this task is currently still very troublesome due to lack of widely accepted standards, approaches, technologies and tools.

In this thesis, we present a way how to facilitate integration of data quality requirements with information systems. In order to do that, we created a system that follows the Model Driven Architecture (MDA) approach of the utilization of a technology independent model that serves as a base for technology specific code generation. In order to provide a user with means of constraint specification we propose a very powerful Domain Specific Language (DSL), called UnifiedOCL, that is capable of denoting data structure together with constraints of a great variety of types. Furthermore, we studied various approaches and technologies of bidirectional transformations between various models (representations). This resulted in a extendable toolkit capable of transforming constraints to- and from- UnifiedOCL.

As a consequence, the system we present allows for transformation from any source representation into any target representation. Our proof-of-concept implementation supports three representations: object-oriented language (Java), relational database (SQL) and business rules (Drools), which let us explore and cover a broad range of diverse constraints. Moreover, it is easily extendable with any other representations.

Finally, we proposed a way how constraints can be mapped into various data quality dimensions, which together with our data quality calculation model lead to a complete framework for providing the user with feedback about data quality for given data. The Data Quality Visualizer is an application designed to visualize values of data quality dimensions computed for the user’s data, which enables the data quality analysis for any Information System.
# Contents

1 Introduction ................................................................. 1
   1.1 Motivation ............................................................ 1
   1.2 Thesis structure ..................................................... 2
   1.3 Demonstration scenario .............................................. 3

2 Background and Related Work ........................................... 5
   2.1 Constraints & data quality .......................................... 5
   2.2 Constraints ............................................................ 5
      2.2.1 Definition and types ........................................... 5
      2.2.2 Classification .................................................. 7
      2.2.3 Representation .................................................. 9
      2.2.4 Design phase ................................................... 9
      2.2.5 Construction phase ............................................ 9
      2.2.6 Problems with constraints .................................... 14
      2.2.7 Constraints related software engineering challenges ...... 17
      2.2.8 Platform independence ........................................ 17
   2.3 Models ................................................................. 17
   2.4 Transformations ....................................................... 18
   2.5 OCL constraint transformations .................................... 19
   2.6 Abstract vs concrete syntax ....................................... 21
   2.7 Domain Specific Language (DSL) .................................. 21
   2.8 Data quality .......................................................... 22
      2.8.1 Overview ........................................................ 22
      2.8.2 Data quality management frameworks ....................... 24

3 Unified Constraint Representations ................................... 27
   3.1 The need for a unified representation ............................ 27
### 5.9 Formulae
- **5.9.1 Single class level**
- **5.9.2 Multiple class level**
- **5.9.3 Overall data quality**

### 5.10 Feedback types

### 5.11 System overview

### 6 Implementation

- **6.1 Overview**
- **6.2 Domain Specific Language frameworks**
- **6.3 Eclipse platform**
- **6.4 Eclipse plugin**
- **6.5 EMF Ecore**
- **6.6 Technology overview**
  - **6.6.1 Template languages**
  - **6.6.2 Model to model transformation**
  - **6.6.3 Other used frameworks**
  - **6.6.4 Other considered frameworks**
- **6.7 UnifedOCL**
- **6.8 UDQAM dictionaries**
- **6.9 Supported constraints**
- **6.10 Data quality mapping**
- **6.11 UML to UnifiedOCL transformation**
- **6.12 Technology specific to- and from- UnifiedOCL transformations**
- **6.13 UnifiedOCL to pure OCL transformaton**
- **6.14 UnifiedOCL (de)serialization to XMI**
- **6.15 Data Quality Visualizer**
- **6.16 Used technology overview**
- **6.17 Transformation architecture**
  - **6.17.1 Source files**
  - **6.17.2 Traversing & analyzing – input & intermediary models**
  - **6.17.3 Matching constraints**
  - **6.17.4 Regular expressions**
  - **6.17.5 Building constraints**
6.17.6 Constraint dictionaries ............................................ 100
6.17.7 Producing the target representation ............................. 100
6.18 DSL project structure .................................................. 101
6.19 All bundles – details .................................................... 104
6.20 All bundles – overview ................................................ 115
6.21 Extensibility .............................................................. 115
  6.21.1 A new representation .............................................. 115
  6.21.2 Eclipse independence .......................................... 118

7 User Guide ................................................................. 119
  7.1 Eclipse ....................................................................... 119
  7.2 Installation guide ....................................................... 120
  7.3 Eclipse configuration GUI ............................................. 121
  7.4 Supported file types ................................................... 121
  7.5 Transformation step-by-step ....................................... 122
  7.6 Working with the UnifiedOCL editor .............................. 124
  7.7 Data Quality Visualizer .............................................. 126
    7.7.1 General information ........................................... 126
    7.7.2 Visualizing data ............................................... 127

8 User Study ................................................................. 131
  8.1 Tasks ....................................................................... 131
  8.2 Participants ................................................................ 133
  8.3 Goals ....................................................................... 133
  8.4 Procedure .................................................................. 135
  8.5 Results ..................................................................... 136
    8.5.1 Task I .................................................................. 136
    8.5.2 Task II ................................................................ 139
    8.5.3 Task III .............................................................. 139
    8.5.4 Task IV .............................................................. 143
  8.6 User feedback .......................................................... 145
  8.7 Summary .................................................................. 146

9 Future Work ............................................................... 147
  9.1 More representations, more constructs, more constraints ...... 147
1 Introduction

1.1 Motivation

In the modern world we can find Information Systems (ISs) in almost every area of our daily life. They have an impact on countless many aspects of our existence – they are responsible for managing our finances, organizing our travels or compiling our menus, etc. However, to be able to rely on their support and have confidence in them, we need to ensure that the data they are operating on is of good quality. In that way, we make sure that the obtained output results are meaningful to us. Otherwise, this process would adhere to the rule garbage in, garbage out and the consequences of that fact might be disastrous. Just imagine what would happen if an aircraft, that is already in the air, at some point runs out of fuel. This could happen due to an incomplete data in the take-off procedure check-list or due to an incorrect value of the tanked petrol (an unintentional mistake) or an inappropriate representation in the system (an insufficient accuracy when rounding values) or just due to the lack of regulations of the minimum amount of fuel for a given flight length. All these reasons can be identified as data quality problems. As a remedy to this problems one can specify some data quality requirements, which are more technically called constraints, and take care of their enforcement in terms of data, that is being used by a given information system. Any infringement of defined constraints leads to decreased quality of data. Additionally, not only does the input data have to be validated against constraints, but also the data that is being processed, modified or updated by the system (the amount of fuel is decreasing automatically during the flight). As a consequence, maintaining the desired level of data quality is a continuous task that should be handled with an appropriate care. This leads to the emergence of a need not only to specify data quality requirements and validate real data against them but also to monitor the current state of data quality. Nowadays, the issues related to data quality, though their importance is widely accepted, are still put on a side. Their crucial role is belittled. Although this research topic has been
active for the last couple of decades, everlastingly there does not exist universally recognized approach how to integrate data quality aspects into information systems during their whole life-cycle and how to support this process across software development methodologies.

Currently, trends in the field of software development are changing rapidly. As a consequence we can observe a massive growth of new technologies, frameworks and programming languages that are available at the market. At the same time many other existing tools very quickly become obsolete, leading to increasing the number of legacy systems. Moreover, individual information systems are getting more and more complex, they span over multiple layers (from the persistence layer through the business logic layer till the presentation layer) and are composed of various parts, that are built using considerable variety of interweaving technologies. As a result, currently we have to be concerned with the overwhelming plethora of data quality requirements (constraints), notations and frameworks. All of these make the management of constraints and the maintenance of the system in a coherent state an extremely difficult but at the same time absolutely important task. Not only should the pilot know the fuel level in his machine along with other parameters of the flight but also he is supposed to be aware of the ground handling, air traffic control, aviation safety agencies, etc. All of these information might operate on completely different systems, procedures, policies, etc., but in order to make our flight safe, quick, enjoyable and comfortable (good quality of service), they all need to cooperate and they all need to rely on the same (or perfectly corresponding/matching) data, the good quality data.

The goal of this thesis is to propose a solution to the aforementioned problem by introducing unified constraint representation along with transformations to- and from- specific technologies. Through adhering to the model driven development principles, it will separate the conceptual data quality requirements from their actual representations. As a consequence, the number of supported environments (platforms, programming languages, frameworks, libraries) will be virtually infinite. The implementation of bidirectional transformation between platform independent and platform specific models will not only contribute to coherent data quality between systems or their parts but also simplify conversion between various representations. Additionally, it will offer a support for reasoning about data quality of manifold notations by allowing for operation of comparison. Last but not least, we will study the way how to provide feedback to users about achieved data quality by analyzing and visualizing its various aspects (dimensions).

We believe that this thesis will not only contribute to unified data quality representation, achieved by means of constraint transformations, but will be also a step forward towards good practices in software engineering by enhancing model driven development with the way to handle one aspect of an immense importance - the concept of data quality.

1.2 Thesis structure

In the next chapter we provide the theoretical background of various aspects of data quality, constraint representation, domain specific languages and model driven development. While
discussing various concepts we present also existing related work. Chapters 3-5 contain a detailed description of the contribution of this thesis (our approach):

- unified data quality representation (Chapter 3) – we present our Domain Specific Language (DSL) – UnifiedOCL and discuss our compact way of specifying constraints – Unified Data Quality Assurance Markups (UDQAM)

- constraint transformations (Chapter 4) – we introduce a way how in our system, using transformations, a user can obtain a technology specific representation from a technology independent model and how one constraint representation can be converted in the other

- data quality feedback (Chapter 5) – we show how, in our approach, constraints are related to data quality dimensions and a way how we can compute values of particular data quality dimensions for given data. Also here we present our idea how to provide a user with feedback about data quality – we are presenting our Data Quality Visualizer

Chapter 6 encloses the description of implementation details of our system, which consists of all three parts discussed in the previous three chapters. Chapter 7 aims at users of our system and explains how to use it (user guide). In Chapter 8, we describe the user study we conducted to evaluate our system, and we are discussing obtained results. Based on the user study and our observations and ideas we present in Chapter 9 ways of further work in the topics that are covered in this thesis. Moreover, we discuss potential possible extensions of our system. Finally, in Chapter 10 we conclude this thesis and we summarize what we managed to achieve.

1.3 Demonstration scenario

Throughout this thesis, in order to illustrate various concepts related to constraints or data quality, we will use an airline domain example. This domain should be familiar to all readers, therefore, grasping even complex concepts should not cause any problems. Even though, a careful reader may find that the airline examples we are using are not the most realistic ones, we have to bear in mind that their major goal is to serve as real life related background and not to provide the best model of an existing airline.

The overview of the airline model can be found in Appendix E.
2.1 Constraints & data quality

Simply saying, data is of a good quality when it fulfills all the constraints that are defined specifically for a given domain. Therefore, in order to achieve a desired level of data quality one needs to define constraints thoroughly. By constraints we mean every limitation that prevents a particular piece of data from being out of any value (narrowing the set of all possible values that a certain variable, entity or object may take).

2.2 Constraints

2.2.1 Definition and types

In general constraint is a very broad term that is not defined precisely in the literature. It is mentioned in the literature almost always in the context of databases. In such cases it is usually understood as an integrity constraint. Of course this is not the only type of constraints. Because of its imprecise definition, the group of artifacts, that we can call constraints, is quite vast. From the point of view of this thesis it is important to state clearly that constraint types may vary due to following reasons:

Context

Context independent Such constraints are always well defined, regardless the external conditions, i.e. their semantics is not changing according to the location where they are defined. They are always validated in the same way, independently of the context, e.g. temperature has to be always greater than -273.15 Celsius degrees.
2.2. CONSTRAINTS

Context dependent Those constraints are usually related to more than just one object, e.g. a car plate must be unique within a particular country, but it does not have to be true in a global scope. In general, they may be satisfied or not depending on the conditions of the environment they are evaluated in. Another example might be any time-dependent constraint, e.g. the pilot license validity expressed as a date is questionable depending on the real time when such a constraint is being validated (before or after the expiration period).

Meta-model hierarchy

Such a division is suggested in [70] in the context of data quality problems – according to the definition of data quality, mentioned in Section 2.1, data quality problems may be considered as constraint violations. A schema is considered as the model whilst an instance as the model instance – correspondingly these are M1 and M0 levels according to Object Management Group (OMG) Meta-Object Facility (MOF) specification [31] (generic reflective model - chapter 7.3)

Schema related Such constraints are related to the way how the data is organized. The most obvious representatives of that group are referential integrity constraints in relational databases. E.g. they are responsible for maintaining relations between entities by means of primary/foreign keys (which according to our understanding are the data organization constraints).

Instance related Those constraints take into account the content (value) of each instance of the model entity (it doesn’t matter whether they are considered in isolation or aggregated all together). For instance, the condition denoted as pilot.age>18 states that every pilot should be older than 18 years – the value of the attribute age of the object pilot (object model) or respectively column age of the table pilot (relational model) should have the value greater than 18.

Technology specificity

Technology independent This type of restrictions could be imposed by using any available technology. Representatives of that group are conceptually very simple, therefore the great majority of technologies (if not all) are capable of expressing and enforcing such constraints. A good example is the requirement that the name of the pilot cannot be empty or null.

Technology specific Those constraints are highly coupled with a particular constraint definition framework. Due to their specificity, it might not be possible to express such a constraint in any other technology specific representation. For instance, using Java Bean Validation [6] one can define the precision of the numeric representation (maximal number of fraction digits) using the @Digits annotation. Another example of a technology specific constraint from the relational domain is the auto increment constraint, which states that a table column with this constraint defined, should contain
numbers in an increasing order. Such a constraint in the Java language would be less reasonable or would make no sense at all. Constraints like relational database keys have completely no sense in object-oriented languages.

**Data vs process orientation**

**Static – data oriented** Such constraints may be related to the static features of the model objects and therefore they should be valid anytime during the program execution. *Object Constraint Language (OCL) invariants* are perfect examples of such a group. The object/attribute that is restricted using this type of constraint, always has to satisfy the imposed limitation. E.g. a plane’s maximal weight cannot exceed 125 tonnes. This constraint could be expressed in OCL as follows

```oclnonumber
context Plane inv maximalWeight:
    self.weight < 125000;
```

**Dynamic – process related** Those constraints refer to the inputs and outputs of a particular operation (process, method). They correspond to the concept of pre- and postconditions of a method, which is discussed by [60]. These concepts are also the core elements of OCL. As an example one can imagine the method `takeOff` of the class `Plane` which would be responsible for the process of lift-off of the aircraft. A precondition may be the appropriate speed of the airplane that would be enough for becoming airborne.

**Data origin**

Some constraints may be also related to the default (initial) value of a piece of data. E.g. the initial speed of the plane should be equal 0.0 km/h. By default, the value of data may also be computed using sophisticated algorithms that process many data sources. A representative example of this group of constraints is a specification that a particular number should represent the *Social Security Number (SSN)*. SSNs are usually computed in a way that they contain a checksum that could be used to verify whether such a number is not forged. Both concepts (initial and derived data) are also supported by OCL.

### 2.2.2 Classification

Kim et al. [45] introduce a constraint classification according to their complexity. It is mainly focused on the constraints defined in relational databases, but it could be easily generalized into the object-oriented world, which is also mentioned by the authors. For the purposes of this thesis it is important to emphasize that constraints may be very complex and their validation may require traversing a huge network of interrelated objects. The following OCL code snippet shows one of such constraints (it is taken from the well-known example of *Royal and Loyal* as defined by Warmer and Kleppe in [83])

```oclnonumber
context LoyaltyAccount inv invariant_oneOwner :
    (self.transactions
```
2.2. CONSTRAINTS

```ocl
collect( i_Transaction : Transaction | i_Transaction.card )
collect( i_CustomerCard : CustomerCard | i_CustomerCard.owner )
asSet()->size() = 1
```

The meaning of this constraint could be explained as follows: any customer that is a member of a loyalty program (in a retail network) must be the owner of all customer cards that are used in all transactions (payments) that belong to the loyalty program account whose owner is the given user.

This classification is relevant from the point of view of this thesis as one of the challenges we have to face is the impedance mismatch between relational and object models and, as the consequence, some constraints expressed using OCL are not possible to be expressed in Structured Query Language (SQL). It is due to the object nature of OCL – objects that are attributes of other objects (and navigational links between them) are impossible to be modeled in a relational domain, where the attributes might be of only simple type (of course bearing in mind only solutions that are clean and conforming to the standards). In other words, object models allow for expressing more complex constraints comparing to relational models and as the consequence a relational system supports narrower classes of constraints classified according to [45]. Our system is also affected by the fact that some constraints expressed in OCL cannot be expressed in SQL, but it is important to notice that it is not a specific problem of our system but a more general limitation of relational databases. Moreover, in [45] Kim et al. list all the features of the object-oriented world that are making constraint handling more difficult comparing to the relational model:

- objects may contain methods – e.g. constraints for pre- and post- conditions
- objects may contain complex nested objects – constraints may navigate over a hierarchy of objects, e.g. OCL expression `dog.owner.car.engine.piston.capacity>250`, which is not possible in standard relational databases as the relational model is rather value oriented. However, in relational databases there is a similar concept – joins but according to Kim they require querying much larger sets of entities
- objects may contain collections – it allows for constructing more complex type of constraints as a single field of a class represents collections of objects. Therefore, evaluating such an attribute may require aggregation functions.
- objects have a hierarchy of types – as a consequence some constraints should be evaluated not only for a given type but also for all of its subtypes. Additionally, constraints from subtypes may specialize constraints from supertypes, attributes of types that are queried by constraints may change their visibility, etc. It is important to notice that some Relational Database Management Systems (RDBMSs) allow for table inheritance, e.g. PostgreSQL, but this inheritance is understood just as the inclusion of all columns and constraints from supertables into subtables (concepts like overriding or encapsulation levels are not available).
2.2.3 Representation

In information systems constraints are usually defined aside from the main implementation or specification of the system. It is equally true for the phase of requirements gathering/system design and the phase of the system construction, regardless used software development methodologies like the waterfall model [76] or any of the agile models (e.g. Scrum [72]). The reason is just not to clutter the existing code with too many assertions/validations, which would make the code more difficult to maintain and may lead to an incoherent constraint specifications.

The following sections will introduce various constraint representations with regards to the software development life-cycle phases:

- **Design phase** – platform independent modeling
- **Construction phase** – platform specific modeling

2.2.4 Design phase

Information systems are very often designed using Unified Modeling Language (UML) diagrams proposed by the OMG [29]. The most basic one is the class diagram. It reflects the structure of the system by means of classes, attributes, relations, etc. It is also capable of expressing a simple behavior by means of operations (members of the classes). Even though UML was initially considered by the OMG as the standardized way of designing object-oriented systems, there exist systematic approaches that allow to derive a relational model from UML class diagrams ([25] and [1]).

The Object Constraint Language (OCL), also standardized by OMG [30], is a language that should complement UML in the field of the constraint specifications. It is not a programming language but just a convenient way of defining constraints in a formal way. A set of constructs available in OCL is powerful enough to define complex constraints over hierarchies of interrelated objects. The OCL specification includes the OCL standard library, which defines types and available operations for them. However, as it will be stated in Section 3.6 OCL is not powerful enough to allow for the definition of all the necessary constraints in modern technologies.

Some attempts to overcome problems of dynamic aspects of OCL language are presented in [20].

2.2.5 Construction phase

For the purposes of this thesis, we decided to focus on three different programming languages in order to cover as many types of constraints as possible and to investigate a broad spectrum of potential problems. Moreover, we wanted to investigate representations used by object-oriented languages and relational databases. We also considered two popular programming paradigms: the imperative and the declarative. We considered following representatives of the programming languages:
2.2. CONSTRAINTS

- object-oriented (the imperative paradigm) – Java\(^1\)
- relational (the declarative paradigm) – SQL\(^2\)
- business rules – Drools\(^3\)

At this phase of the software development process a developer has to deal with technology specific constraint (data quality requirement) specification.

Java

The decision of selecting Java as the main imperative, object-oriented language is justified by the fact that at the time of writing this thesis it is the most popular programming language in the world according to TIOBE\(^4\).

A common way of defining constraints in Java is the **Bean Validation** specification. It is a standardized framework [5], [6] which is part of the Java Enterprise Edition (Java EE). Java EE is an extension of the standard Java platform for enterprise application, which includes plenty of Application Programming Interfaces (APIs) and components such as messaging system, object-relational mappings, web services etc.

Bean Validation contains the set of predefined Java annotations used for expressing constraints. It is important to notice that this is just a specification with minimal requirements regarding constraints, that is implemented by lots of vendors, e.g. Hibernate Validator\(^5\) or OVal\(^6\). For the purpose of this thesis we’ve chosen Hibernate Validator which is a reference implementation of the standard [6], which contains also an extended version of the minimal set of constraints defined in [5]. A full list of standard constraints is provided in [2] and the Hibernate extension [19].

Bean Validation constraints, which are represented as annotations may be declared at the following levels

- class – constraints of that type may refer to multiple fields of the class
- field/property – refers to a single field of the class (property is understood as the possibility to specify a constraint annotation for the getter method of the field)
- method parameter – constraints over inputs of a certain methods - could be considered as method preconditions
- method return value – a constraint over the output of the method - could be considered as method postconditions

---

\(^3\)http://www.drools.org/ [Online, accessed 19-August-2015]
public abstract class Person {
    @NotNull
    public String name;
    @NotNull
    public String surname;
    @Past
    public Date birthDate;
    public String title = "Sir/Madam";
    @Email
    public String email;
    public Integer id;
    @NotBlank
    public String address;
}

Figure 2.1: Class Person with defined constraints

Figure 2.1 presents a very simple class Person with defined constraints – annotations for various fields. The first three ones (@NotNull and @Past) are parts of the Bean Validation standard while the later two (@Email and @NotBlank) are parts of the Hibernate Validator extension. The default value of the attribute title is also considered as a constraint defined in Section 2.2.1 as data origin.

In addition to constraints defined for attributes, Figure 2.2 depicts how constraints are also defined for the method takeOff – imposing restrictions over the return value (@Null) and over the parameters (@Min and @AssertTrue). Moreover, there is a constraint defined for the whole class – @EnoughSafetyEquipment. This constraint is expressed by a user-defined annotation (definition shown in Figure 2.3), which is validated by a custom validator (Figure 2.4). The isValid method is a function of the values of various Plane fields.

SQL

The Structured Query Language (SQL) is not only an example of a declarative language but also a standard way of manipulating and defining relational databases. Relational databases are still the most popular ones according to the DB-Engines ranking\(^7\), therefore addressing the problems with constraints in this systems are of unquestionably high importance.

Constraints used in RDBMS are called integrity constraints. An overview of the available constraints in PostgreSQL can be found in [33]. A theoretical study of existing integrity constraints could be found also in [45].

Figures 2.5 and 2.6 show constraints for the same entities like in the Java section but this time defined in SQL. Constraints may be expressed as:

- special keywords – e.g. the NOT NULL keyword for the column name in the table Person or the PRIMARY KEY keyword for the column id also in the table Person
- default values – e.g. the title column in the table Person

```java
@EnoughSafetyEquipment
public class Plane implements Machine {

    public Integer lifeJacketsNumber;
    @Max(value = 25)
    public Integer age;
    @DecimalMax(value = "125.75")
    @Digits(integer = 4, fraction = 2)
    public Double weight;
    public Integer seatsNumber;
    public Integer cabinCrewNumber;

    @Null
    public String takeOff(@Min(value = 350) Integer speed,
                           @AssertTrue @AssertFalse Boolean landingGear) {
        return null;
    }
}
```

Figure 2.2: Class Plane with defined constraints

```java
@Constraint(validatedBy = EnoughSafetyEquipmentValidator.class)
@Target({ TYPE, METHOD, FIELD, ANNOTATION_TYPE, CONSTRUCTOR, PARAMETER })
@Retention(RUNTIME)
public @interface EnoughSafetyEquipment {
    String message() default "EnoughSafetyEquipment constraint violated!";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Figure 2.3: Annotation EnoughSafetyEquipment definition
Drools

Business rules are a convenient way of describing complex aspects of business. They may be used for defining and constraining processes depending on the current condition, e.g. actions to be taken when some events occur. In other words, they can be used to express either integrity constraints (data validation) or conditions of operations. The latter ones are also types of constraints – we can think of them as preconditions. In this thesis we decided to consider Java based rule engine – Drools.

Any business rule in Drools consists of two parts: a condition and a consequence. The
consequence is the action that is taken when the condition is evaluated to be true. Such an
action may be, e.g. appending a log entry when an object, for which the rule is called, is not
valid. The condition is used for matching any object reachable by the rule engine with the
corresponding action. The body of this condition is therefore our constraint – a limitation
that restricts the whole domain of objects into just a subset (in case of object validations – to
those objects that do not fulfill constraints).

Figures 2.7 and 2.8 show business rules defined in Drools. The condition part follows the
keyword when while the consequence follows the keyword then. The consequence part is
denoted as ellipses, as this is not relevant for us what action should be taken when objects are
positively matched with the condition of particular rules. In Drools there is no compact way
to express constraints. A user has to define explicitly the body of the condition. They can use
either Drools or Java languages expressions syntax.

2.2.6 Problems with constraints

The concept of constraint may refer to a very broad spectrum of features/aspects of data
as it was stated in Section 2.2 and Subsection 2.2.2. As a natural consequence of this fact,
constraints are not homogeneous and they cannot be expressed in a unified way. There is
no language that would cover all the cases, similarly, as not all constraints make sense in
all programming paradigms. To sum up, constraints are difficult to be handled due to the
following facts:

• they describe a very broad spectrum of data aspects

• there is no unified way of their representation and no widely accepted versatile frame-
work for constraint specification/management

• they are managed in a decentralized way due to the fact that they may be defined at
various application layers (persistence layer, business logic layer, presentation layer,
etc), using various technologies

• data quality requirements may come from various actors existing in the software de-
velopment process lifecycle that present interests of different stakeholders. Constraints
may come from software architects, business analysts, clients, external systems etc.
Figure 2.7: Rules for class Person

```plaintext
rule "Person_address_notblank"
when
  $a : Person (! (address!=null && !address.trim().isEmpty()) )
then
  ...
end;

rule "Person_birthDate_past"
when
  $a : Person (birthdate!=null, ! (birthdate.before(new Date())) )
then
  ...
end;

rule "Person_email_email"
when
  $a : Person (! (validateEmail(email)==true) )
then
  ...
end;

rule "Person_id_primarykey"
when
  (($a : Person(id==null)) and ($b : Person(this==$a))) or
  (($a : Person()) and ($b : Person(id!=null, this==$a, id==$a.id)))
then
  ...
end;

rule "Person_name_notnull"
when
  $a : Person (! (name!=null) )
then
  ...
end;

rule "Person_surname_notnull"
when
  $a : Person (! (surname!=null) )
then
  ...
end;
```
rule "Plane_age_max"
when
  $a : Plane (! (age<=25) )
then
  ...
end;

rule "Plane_plane_enoughsafetyequipment"
when
  $a : Plane ( !( lifeJacketsNumber!.equals((((seatsNumber + (cabinCrewNumber * new Integer(2))) + new Integer(5))) )) )
then
  ...
end;

rule "Plane_weight_decimalmax"
when
  $a : Plane (! (weight<125.75) )
then
  ...
end;

rule "Plane_weight_digits"
when
  $a : Plane (! (validateDigits(weight, 4, 2)==true) )
then
  ...
end;

Figure 2.8: Rules for class Plane
CHAPTER 2. BACKGROUND AND RELATED WORK

2.2.7 Constraints related software engineering challenges

The real life consequences of the problems with constraints specified in the Subsection 2.2.6 are the software engineering challenges that any software developer may encounter and have to deal with quite often. Those challenges are:

1. How to model constraints in a technology/representation independent way?
2. How to generate the technology specific code from the technology independent constraint representations?
3. How to check whether given constraint representations are equivalent?
4. How to maintain coherent data quality requirements across various parts of a complex information system?
5. How to convert one technology specific representation into another, which is especially important when working with legacy systems or trying to migrate data between systems?
6. How to manage and monitor data quality requirements in a central place?

2.2.8 Platform independence

Investigating and solving the problems mentioned in the previous section 2.2.7 are the ultimate goals of this thesis. Precisely, the aim of this project is to allow a user to specify, manage, process and validate constraints independently of:

- application layers
- programming languages
- programming paradigms
- technologies
- frameworks
- programming environments

in order to achieve a coherent and unified data quality representation.

2.3 Models

Model Driven Architecture (MDA) [75], [58] is an approach proposed by OMG\(^8\) to deal with the problem of constantly changing technologies and business requirements during the software development lifecycle. The main idea is to, at first, model a system using technology

2.4 Transformations

In the MDA approach, except models, the key elements are transformations. The technical report [28] introduces a standardized version of model transformations language proposed by OMG. Even though it is considered as a standard, there is currently no widely accepted tool at the market that fully implements that standard. One of the most common implementation is the Eclipse Query/View/Transformation Operational (QVTo)\[^9\].

[11], [59] and [73] give an overview of various approaches of model transformations and propose classifications/taxonomies based on various metrics. Moreover, Sendall and Kozaczynski [73] try to define desirable characteristics of model transformations with regards to the usage in real industrial processes.

Mens and van Gorp [59] define a concept of the program transformation as the special case of a model transformation where the artifact to be transformed is a source code or a byte-code or a machine code. With this in mind they are concluding that the source and target models (source codes) can be understood in terms of their structure. In this master thesis we deal mostly with the source code transformation expressed by syntax trees (reflecting structures of the source codes).

Czarnecki and Helsen [11] discuss major categories of model transformations with distinction to model to code (model to text) transformations and model to model transformations. The former one could be considered as the special (but very common) case of the latter one, where the output format is a source code of the target programming language (text). In the same paper they also describe various frameworks allowing for the model transformation that belong to both categories.

Further studies of the model-to-model (M2M) and model-to-text (M2T) are presented

in [87]. This classification is particularly important in our project as it is also used by the Eclipse Modeling Project\textsuperscript{10}11. The book by Richard C. Gronback [26] provides an excellent overview over Eclipse platform related M2T and M2M frameworks such as Query/View/Transformation (QVT)\textsuperscript{12} or Xpand\textsuperscript{13}.

Taking the Eclipse platform frameworks for M2M and M2T transformation, as the most popular and probably the most advanced and mature ones, we have to be aware of the following issue. From the technical (implementation) point of view, the model to model transformation seems to be much more difficult (illustrated in Figure 2.9) than the model to text transformation. First of all, the transformation between two models is implemented as two unidirectional transformations. Even though it seems to be more reasonable (conceptually simpler) it is not a single bidirectional transformation. In other words, such a M2M transformation is usually implemented in an imperative way (describes how to obtain the second model given the first one) and not in a declarative way (finding corresponding concepts – what are the identities in the models). Second, for the purposes of this thesis inputs and outputs are source code files. Models are the abstract in-memory structures. To obtain instances of such models we need to parse source code files. The reverse operation is required in order to produce an output source code file from the model instance. One has to traverse to a model instance (usually the abstract syntax tree) and ‘print it out’ or, more technically, serialize it. To sum up, for just one representation one needs to handle:

1. model loading from file
2. model storing/saving (serialization)
3. to- and from- model transformations

With this in mind the model to text transformation concept depicted in Figure 2.10 seems to be conceptually much easier. A model serialization is a form of a model to text conversion. Continuing this reasoning, we may consider the model serialization and the model to text transformation for a particular representation as an identity. As a consequence, having the model to text transformation, one does not need to implement the model serialization. Of course it is noticeable that this approach requires in worst case enormously many different model serialization-transformations – one for each target representation. For the reasons that will be explained in the next chapters, this particular drawback will turn out to be one of the greatest advantages.

### 2.5 OCL constraint transformations

Current efforts in the OCL constraint transformations studying are mostly focused on constraint simplifications or mapping to relational database queries/integrity constraints.

---

\textsuperscript{10}https://eclipse.org/modeling/ [Online, accessed 19-August-2015]
\textsuperscript{13}https://eclipse.org/modeling/m2t/?project=xpand [Online, accessed 19-August-2015]
Cosentino and Martinez discuss in [10] a reverse engineering approach focused on extracting integrity constraints from relational databases and mapping them into OCL expressions. They are also providing a working tool capable of inferring those OCL constraints from SQL schema elements (column constraints, triggers). Furthermore, they study also more complex constructs such as PL/SQL procedures. Because OCL constraints cannot exist in a detached state from the UML class diagram (they are not context-free), the authors introduce also a concept about the SQL schema mapping into the UML class diagram. Moreover, they provide a study of a corresponding concept in SQL and OCL (e.g. a foreign key representation, unique constraints etc.). In this thesis we highly rely on their findings as this is probably the only one paper targeting SQL to OCL transformation problems.

Much more has been done with regards to the opposite transformation. Demuth and Hussmann [13] give a very comprehensive analysis of how to create relational databases from UML and OCL constraints using a systematic approach. They discuss in details various OCL mapping patterns and mention also natural limitations due to the model impedance mismatch. Moreover, DresdenOCL[^14], described in [12] and [14], is a proof of concept implementation of their approach.

A more theoretical study of the problem of transforming class diagrams into object-relational schemas is presented in Golobisky and Vecchietti [25]. They provide an analysis of the UML class diagram transformation to a relational database schema conforming to the SQL:2003 standard [36], [18]. However, they put little or no effort regarding constraints. The UML to object-relational schemas transformations are also discussed in [62] by Mok and Paper.

OCL expressions may be the output of model transformations. Therefore, in some situations they are too complex to be human readable. Giese and Larsson [24] provide a solution to this problem by repeated applications of some simple rules.

class diagram is a rather straightforward task. The transformation in the opposite way also
shouldn’t be troublesome. Generation of Java is simple due to the intrinsic object nature
of UML. Moreover, there are many tools capable of transforming UML to Java – e.g. Ma-
gicDraw\textsuperscript{15}, Visual Paradigm\textsuperscript{16, 17} and IBM Rational Software Architect\textsuperscript{18, 19}. Many tools
provide also the reverse transformation, e.g. MagicDraw or IBM Rational Software Archi-
tect\textsuperscript{20}. However, they all provides either completely no or very limited support of constraints.
Tools capable of converting OCL into Java code are DresdenOCL or the OCL implementation
for Eclipse Integrated Development Environments (IDE)\textsuperscript{21, 22}. Briand et. al in [8] presents
another systematic approach of transforming OCL into Java. To our best knowledge there
does not exist any literature about Java to OCL transformations.
Transformation of OCL to- or from- specific programming platforms is a crucial step in the
matter of this thesis as it is used extensively. However, delivering a tool for OCL transform-
tation to specific technologies is beyond the scope of this thesis due to the complexity of that
task (syntax transformation is not that difficult as a semantic code validation, type checking,
standard library compliance, etc.). Without loss of generality we can assume that implement-
ing such transformations is possible but very time-consuming.

2.6 Abstract vs concrete syntax

In the context of model transformations a very important term is the \textit{syntax}. When we are
referring to the model, usually we bear in mind the \textit{abstract syntax}. Gronback [26] iden-
tifies the abstract syntax directly as the metamodel of the data. The abstract syntax is the
representation of a model (data) that is completely independent of any notation. The notation
dependent syntax is the \textit{concrete syntax}. It may be either textual or graphical ([26]), for in-
stance the class diagram or the Backus-Naur Form (BNF) [56] of the language grammar. A
very convenient way of traversing the model is the abstract syntax tree. It is a data structure
that reflects the abstract syntax which is usually the product of parsing a concrete represent-
ation. It is an important issue for the purposes of this thesis that an abstract syntax tree is
always considered to be a starting point in any transformation.

2.7 Domain Specific Language (DSL)

A Domain Specific Language is the programming language that is designed to be very con-
venient to describe very specific, narrow areas. It allows to simplify the software develop-
ment process by providing a more suitable representation for concrete domains by letting a

\textsuperscript{16}http://www.visual-paradigm.com/ [Online, accessed 19-August-2015]
\textsuperscript{17}https://www.visual-paradigm.com/tutorials/sdeeccodegen.jsp [Online, accessed 19-August-2015]
\textsuperscript{20}http://www.ibm.com/developerworks/rational/library/09/reverseengineerumlplacediagramsrationalsoftwarearchitect/
[Online, accessed 19-August-2015]
[Online, accessed 19-August-2015]
\textsuperscript{22}http://www.eclipse.org/modeling/mdt/?project=ocl [Online, accessed 19-August-2015]
2.8. DATA QUALITY

user operate on entities that are more natural in the given discipline. Domain Specific Languages (DSLs) are in opposition to General Purpose Language (GPL) which are designed to be versatile, but may be awkward to use in some particular application areas.

DSLs may be represented by both concrete syntaxes: graphical and textual ones [26]. The complete overview of various classes of DSLs are described in [50] together with frameworks for developing DSLs. DSLs are one of the examples of models that could be transformed. In this thesis the proposed unified constraint representation is expressed by using such domain specific language.

Kurtev et al. [48] go one step further stating that DSLs are a generalization of MDE which is a generalization of MDA. Without any doubts, we can say that the DSL is a form of representation of a model, that describes it using a textual (or graphical) representation. With this in mind, in this thesis, when we are discussing models (or model transformations) we also refer to DSLs as instances or examples of models.

2.8 Data quality

2.8.1 Overview

According to the ISO 9000:2005, quality refers to the ‘degree to which a set of inherent characteristics fulfills a set of requirements ’[37]. For us the data quality is an important concept, because it can help us to assess how reliable a particular information system is. If it is good, it increases our belief that system works as intended and the results are meaningful. If it is bad, it not only causes suspicious and erroneous results but also increases the operational cost of the systems – the time and money needed to fix/improve/correct the data.

As it was stated in the introduction data quality is not a precisely defined concept. Presser et al. [67] point out that this concept might be understood differently by various groups of information system users or stakeholders. In [21], Fox et al. discuss diverse interpretations of what data is and tries to provide the best definition. They conclude that the most exhaustive definition of data is a triple \((e,a,v)\), where \(v\) is the value of the attribute \(a\) of the entity \(e\) together with the definitions of data representation and data recording. Although, they do not define explicitly the term of data quality, they are implicitly assuming that quality could be expressed using the concept of dimensions. The practical explanation of what data quality dimensions are, is to identify them as some metrics to express certain aspects of data. They could be also studied as classifications of some attributes of data according to their relevance for particular application fields.

Fox et al. [21] define a set of data quality dimensions, however probably the most famous work is done by Wang and Strong [82]. They came up with not only data quality dimensions (4 categories, 15 dimensions) but also with a framework how to find out which data quality dimensions are relevant to data users. Batini and Scannapieco [4] provides a very detailed study and a literature overview of the data quality and its dimensions. In this thesis for the purpose of real life examples, we adopted data quality dimensions proposed by them. These data quality dimensions are depicted in Figure 2.11:
accuracy – how close is the given value from to real world value

completeness – is the given data breadth and depth enough to describe a given domain sufficiently (definition taken from [82])

currency – how up-to-date is the data

volatility – how often does the data change

timeliness – is the data available at the time it is needed

consistency – is a single piece of data coherent with the whole data set

There are lots of slightly different data quality dimensions definitions in the literature besides the ones just presented, e.g. [57], [42]. Moreover the international Data Management Association (DAMA United Kingdom) [3] proposed six data quality dimensions as their official recommendation to be used when handling data quality. They present a more industrial categorization of data quality dimensions:

- accuracy
- consistency
- completeness
- uniqueness
2.8. DATA QUALITY

- timeliness
- validity

In [71] Scannapieco et al. provide a table with a summary of what dimensions are defined by individual authors in the literature. For the purposes of this thesis it does not matter which proposal of the set of data quality dimensions are the best. It is only important that the data quality dimensions are identified by a name, and various constraints are mapped to one or more data quality dimensions. In other words in this thesis we consider data quality dimensions as labels for the bags including various constraints from various locations for particular sets of data.

Weber et al. state in [85] that the data quality can be analyzed at two levels of the data life-cycle. Because of this they introduced two types of data quality feedback:

- **Data Quality Input Feedback (DQIF)** – deals with individual data entities during their input into the system
- **Data Quality Analysis Feedback (DQAF)** – deals with data already stored in the system

In this thesis we consider the data quality feedback only of the second type. However, it is important to notice under what circumstances the constraints are held (during entering data into the system or anytime when it is stored) is just the matter of a particular technology, which is not affected in any way by our system.

### 2.8.2 Data quality management frameworks

Currently there is no available and widely accepted, versatile tool for constraints (data quality requirements) specification. As discussed earlier in Section 2.2.6 it is very difficult to manage constraints in a central repository, at best in a technology independent way. As it has been pointed out before, such a tool would significantly contribute to simplifying the process of software development during its whole lifecycle. Such a central constraint repository would not only raise the abstraction level of the software but also simplify the maintenance of the data quality coherence.

Most of the work that has been done in that field applies just to a very narrow domain or is merely technology/platform specific. An ideal tool should first of all consists of a powerful constraint definition language (for the constraint specifications/definitions). Second, it should allow for usage of the previously defined constraints in real projects, either by transforming constraint rules into technology specific form (model driven architecture approach) or allowing for external constraint verification, that could be called from the real applications. As the latter approach requires cooperating parts from both the constraint modeling framework and the applications, in this thesis we are focused on a more realistic approach - the MDA case. OCL, an example of such a language that is capable of expressing constraints was mentioned before. In [88] one can find overview of the other languages that can be used to specify constraints for object languages, e.g. ALICE by Urban and Wang [79].

Another approach of technology independent constraint definition is proposed by Weber et al. in [84]. Their approach supports both simple and complex (involving more than one object) constraints. Moreover, they introduced a distinction into soft- and hardconstraints. The
difference between those two constraint types is the fact that the latter ones can be either satisfied or violated while the former ones, when violated, add just some penalty to the overall data quality (the lower the better). The constraint management framework for object databases proposed by them uses an external (from the database point of view) component for a centralized management (maintenance) of integrity constraints ([88], [17], [78], [15]).

Liu and Kochut [53] proposed a process constraint modeling framework for web services and workflow applications. It allows to define constraints using Process Constraint Language (PCL) and Process Constraint Ontology (ProContO), which could then be validated during the design phase and during runtime as well.

In [74] Sneed and Demuth discuss the need for a tool support for assessing the data quality in relational databases (for both: schema and instance level integrity constraints). They also provide a case study how such a tool should work.

To sum up, most of the existing work that has been done is focused on either just theoretical studies or more practical ones, but covering only one requirement (constraint representation or constraint applications/usages). Finally, the frameworks satisfying both requirements cover just a very narrow field of application.
3 Unified Constraint Representations

The goal of this chapter is to present our unified data quality representation which is implemented as a new DSL. In this chapter, we are discussing and elaborating its syntax and constraint representation capabilities. Moreover, we are presenting a newly introduced compact way of specifying constraints – the UDQM labels – which was created to overcome limitations of constraint representation imposed by the OCL syntax.

3.1 The need for a unified representation

A remedy to all the problems described in Subsection 2.2.6 and the challenges mentioned in Subsection 2.2.7 could be a black box, that would take any source constraint representation as an input and produce an output – a desired target representation. This solution is conceptually truly simple, but the evident challenge is how to construct such a black box.

First of all, we need to have in mind that there may exist considerably many input representations and substantially many output representations as well. We may assume that the aggregated input and output sets of various representations would have the same elements. This simplifies our consideration – now every transformation, independent of its concrete source and target representations, does not go beyond this set. Therefore, our goal is to provide transformations between any two representations chosen from the set of all given representations. Having \( n \) various representations, we would need \( n(n - 1) \) transformations to be implemented in order to satisfy the requirement that conversions between all representations are possible. Moreover, adding the \( (n + 1) \)th representation to that set would require additional \( 2n \) implementations of transformations \( (n \) in each direction). This approach seems to be extremely inefficient.
Thus, our approach is to introduce one additional constraint representation that would serve as the core one – in this thesis we called it a unified constraint representation or a unified data quality representation. Now, we can simplify the approach of the transformations construction. We would introduce the middle step in all transformations from any specific representation into any other. As a consequence, the transformation from the representation $A$ to the representation $B$ would firstly require the transformation from $A$ to the unified representation and then from the unified representation to $B$. The benefit of this setting is pretty straightforward – having $n$ various representations we need just $2n$ transformations (for each specific representation one needs to provide transformations to- and from- the unified representation). Moreover, adding the $(n + 1)$th representation would require implementation of just 2 transformations.

For the purposes of this thesis, we decided to pick the name $UnifiedOCL$ for the unified constraint representation. This name consists of two parts – the first one indicates that this representation is a unification of all the other representations, and the second part brings to our mind the OCL, which is unofficially considered as a synonym of constraints. Moreover, the $UnifiedOCL$ internally uses OCL to represent some constraints.

### 3.2 Core part - a new DSL

The source and target representations that the system should process are various source code files. Because of this, we decided that the unified representation should also have a textual representation. Therefore, we decided that the core part of our system, the $UnifiedOCL$, would be the DSL whose concrete syntax would be textual.

Other benefits of the textual representation are as follows:

- **human readability** – DSLs are simple languages with a limited number of constructs. Therefore, they are understandable also for non-programmers

- **easy processing** – textual documents could be easily parsed and, in that way, the traversable structure representing our notation could be easily obtained for further processing

- **efficiency** – instances of a textual representation could be created much faster than corresponding graphical ones as they do not require any interactions with the complicated Graphical User Interfaces (GUIs) (as in most of the Computer-aided software engineering (CASE) tools)

- **portability** – textual files can be easily exchanged among various platforms as they do not require any external libraries.

Moreover, choosing the DSL based solution perfectly fits into the Model Driven Engineering (MDE) paradigm. If we look from the further perspective on our system, we will see that this is a perfect example of a typical MDE approach. $UnifiedOCL$ could be considered as the Platform Independent Model (PIM) while all the other representations are Platform Specific Models (PSMs) indeed. The last component binding all the representations together, in a way that they constitute the single system, are transformations. Transformations are also the crucial parts of the MDE doctrine.
3.3 Overview of UnifiedOCL

In Figure 3.1 we are presenting the overview of UnifiedOCL. This figure would be explained in details in the next sections of this chapter. The gray rectangles reflects the implementation aspects of UnifiedOCL. Thus, they would be explained in details in Chapter 6.

UnifiedOCL should capture two concepts (the green rectangles presented in Figure 3.1):

- **data structure** – the context in which constraints exist. E.g. the SQL table schema or a Java class.
- **constraints**
The OMG standard for MDA assumes that the data structure could be denoted using UML while the constraints using OCL. Therefore, we decided not to completely drop this legacy, but to adjust what is possible and implement/create what is missing.

The UML is a graphical representation of the model, which has no widely accepted standardized textual representation that is human readable. The existing one – UML Human-Usable Textual Notation (HUTN)[27] – is almost never used, while the standardized XML Metadata Interchange (XMI) is not human readable [32].

The most common diagram that is capable of expressing data structures is the UML class diagram. Therefore, the first significant contribution of this thesis is to come up with a textual representation of data structures that would be based on the UML class diagram. UML is a very complex language, which could be also extended in numerous way. As a result, in our specification, we are focused on the subset of the most commonly used components, but powerful enough to express most of the modeling concepts (in order to satisfy most of the design requirements). Our goal was to create a representation which would be able to express modern object-oriented languages, e.g. Java.

The OCL language already uses textual representations but the set of constraints that could be expressed are mostly limited to the constraints defined over interrelated context objects (specified using UML). However, as OCL is a standardized way of a constraint specification we decided not to drop it but also to include it into our unified representation.

UnifiedOCL should not be just a theoretical study existing only in the academic world. For that reason, our goal was to ensure that UnifiedOCL would closely cooperate with currently used industrial standards. Nowadays, the most common and standardized way to model real systems uses the OMG framework – UML and OCL. Therefore, UnifiedOCL should also work well with already existing standards. This lead us to the assumption that UML+OCL could be also considered as one of the representations that should be handled by our system. In other words it should be also possible to convert a system denoted using UML and OCL into UnifiedOCL. With this in mind, basing our syntax on the UML concepts and using OCL internally appears to be the right solution.

### 3.4 Data structure in UnifiedOCL

The syntax of UnifiedOCL is based on the OCLinEcore\(^1\). We decided to reuse most of its components as they fitted very well our needs. We adjusted concepts for expressing data structure to be more suitable for UML concepts rather than Ecore ones. The details are explained in Subsection 6.7. The data structure syntax is designed to be compatible with the object-oriented paradigm, as most of the real systems are built using this paradigm (UML is also based on this paradigm). Moreover, it is also based on the Ecore metamodel, which is the core element of the Eclipse Modeling Framework (EMF). It is a similar concept as the OMG MOF explained in Section 2.2.1. Following subsection presents various elements

---

of UnifiedOCL syntax. Mostly this are concepts taken over from OCLinEcore, but slightly adjusted to our needs. Here, we are presenting the final form of all the concepts used by the UnifiedOCL syntax (OCLinEcore concepts that are adjusted by us). Figure 3.2 provides a general overview of the main constructs to denote data structures in UnifiedOCL. Most of the examples we are using in the subsections below are taken from the UnifiedOCL representation of the airline demonstration scenario that is available in Appendix E.2.

Package

The top level element of UnifiedOCL is a package. It is determined by its name. It serves as a top level container for all other components.

Example:

```java
package model {

...}
```

Global concepts

Certain aspects of our unified representation are common for its various elements. These are supplementary features that complement the object-oriented nature of the data structure.

Visibility

The concept of an encapsulation from the object-oriented model is captured by the concept of the visibility. It tries to specify who (classes, interfaces, methods) may access certain components (taking into account inheritance). Inspired by Java, we defined four levels of encapsulation:

1. public
2. protected
3. package
4. private

The interpretation which level allows for what kind of access is dependent of the architect of the transformation/technology specific representation. In the unified representation we assumed that this concept is optional, so the unified representation does not require a specification of any visibility.

Example:

```java
public attribute lifeJacketsNumber : Integer ;
...
private reference aircraft : Plane ;
```
Figure 3.2: Syntax of UnifiedOCL
CHAPTER 3. UNIFIED CONSTRAINT REPRESENTATIONS

Multiplicity

*Names* are a common way to refer to underlying objects/entities/data in any programming paradigm. However, the interpretation of what is behind such a *name* might be different depending on the representation. It might be a single piece of data or it might be a collection of objects or entities. To simplify the understanding how the data structure is organized, in our unified data structure representation, a user may explicitly define the number of objects (more precisely a *range*) that are referenced by a given *name*. It is a similar concept as the *cardinality* from the relational database model or UML class diagrams. We decided to use the name *multiplicity*, because the meta-model that underlies our representation (the Ecore) and *OCL in Ecore*, which we also took as our basis, use such a naming convention. If nothing is specified, we assume a default multiplicity – just a single object. If one wants to define another *multiplicity*, they have to use square brackets `[]` and two numbers separated by a double dot `. .`. Instead of the second number a wildcard `*` can be used which would denote that the upper bound is not defined. From the syntactical point of view, the specification of the multiplicity directly follows the type specification of one of the supported objects:

- **structural feature**
- **method parameters**
- **method return types**

Example:

```java
public reference pilots : Pilot[2..2];
...
public reference departures#sourceAirport : Flight[0..*];
```

Qualifiers

Sometimes it is useful to mark some components of a data structure representation with additional information. It justifies the introduction of *qualifiers*. Those are just textual labels attached to the components that convey some well-known object-oriented or relational programming aspects. The name *qualifier* is inspired by the similar Ecore concept, while the set of values for particular supported components are derived from the UML class diagram. *Qualifiers* are supported by the following data structure definition components:

- **classes**
- **interfaces**
- **structural features**
- **methods**
Classifier

The classifier is a common name for all the components that are used to group entities/objects depending on their similarities and taking into account either their structure or their behavior. Classifiers could be seen as types from object-oriented languages, though in general, it is a broader concept. Classifier is the common name for five types of elements, which have their own sort of classification:

1. class
2. interface
3. enumeration
4. exception
5. primitive type

Moreover, some types of classifiers may have members – components that are contained by them.

Class

This component, identified by the class keyword followed by a name, has the same meaning as the class from the UML class diagram. It is the basic concept from object-oriented paradigm that could be identified with the type. Such a type may reveal its own behavior which is expressed by methods. Moreover, it has an internal structure that is reflected by some structural features. The overview of this component is summarized in Table 3.1.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support (+/−) or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>class</td>
</tr>
<tr>
<td>Visibility</td>
<td>+</td>
</tr>
<tr>
<td>Qualifiers</td>
<td>‘abstract’, ‘final’, ‘static’</td>
</tr>
<tr>
<td>Specialization</td>
<td>+</td>
</tr>
<tr>
<td>Realization</td>
<td>+</td>
</tr>
<tr>
<td>Members</td>
<td>methods, structural features, invariants</td>
</tr>
</tbody>
</table>

Example:

```java
public abstract class Person
{
    ...
}
```
CHAPTER 3. UNIFIED CONSTRAINT REPRESENTATIONS

Interface

This element is very similar as the class, but it does not exhibit any internal structure – structural features are not allowed for a classifier of this type. The interface keyword followed by a name, is used to denote for a classifier of this type. The corresponding concept from the UML class diagram is the class marked with the «interface» stereotype. The overview of this component is summarized in Table 3.2.

Table 3.2: UnifiedOCL – Interface

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support (+/−) or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>interface</td>
</tr>
<tr>
<td>Visibility</td>
<td>+</td>
</tr>
<tr>
<td>Qualifiers</td>
<td>'abstract', 'final', 'static'</td>
</tr>
<tr>
<td>Specialization</td>
<td>+</td>
</tr>
<tr>
<td>Realization</td>
<td>+</td>
</tr>
<tr>
<td>Members</td>
<td>methods, invariants</td>
</tr>
</tbody>
</table>

Example:

```java
public interface Machine
{
    ...
}
```

Enumeration

The goal of this simple concept is to define a type that can take just a single value from the predefined set of values. It is denoted by the keyword enum followed by a name. The overview of this component is summarized in Table 3.3.

Table 3.3: UnifiedOCL – Enumeration

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support (+/−) or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>enum</td>
</tr>
<tr>
<td>Visibility</td>
<td>+</td>
</tr>
<tr>
<td>Qualifiers</td>
<td></td>
</tr>
<tr>
<td>Specialization</td>
<td>−</td>
</tr>
<tr>
<td>Realization</td>
<td>−</td>
</tr>
<tr>
<td>Members</td>
<td>literals</td>
</tr>
</tbody>
</table>
Example:

```java
enum TravelClass {
    literal FIRST;
    literal BUSINESS;
    literal ECONOMIC;
}
```

**Enum literals**  *Enum literals* represent the values that a certain *enum* can take. These are just names of certain values – unquoted strings. They are denoted using the keyword `literal`.

**Exception**

A creation of this component was justified by the need to handle errors. Therefore, using the keyword `exception`, we can create a very simple type whose only feature is its name. It is used by *methods* to mark that they may throw an error, which happens when the *method* execution cannot be performed successfully. *Exceptions* are specified in the *method* declaration using the keyword `throws` followed by a name. The overview of this component is summarized in Table 3.4.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support (+/−) or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>exception</td>
</tr>
<tr>
<td>Visibility</td>
<td>−</td>
</tr>
<tr>
<td>Qualifiers</td>
<td>−</td>
</tr>
<tr>
<td>Specialization</td>
<td>−</td>
</tr>
<tr>
<td>Realization</td>
<td>−</td>
</tr>
<tr>
<td>Members</td>
<td></td>
</tr>
</tbody>
</table>

Example:

```java
exception systemError
```

**Primitive type**

This is a very simple classifier that cannot exhibit neither any behavior nor any internal structure. The only feature of a *primitive type* is its name. It can be considered as a type representing a single, inseparable value. It is defined using the keyword `type` followed by a name. The overview of this component is summarized in Table 3.5.

Example:

```java
type Char
```
Methods

Classes and interfaces are the elements that can exhibit some behavior. The declarations of such behaviors are expressed using a well known object-oriented concept – methods. Methods are operations that may take some parameters as the input and may also return some value as a result of their execution. Moreover, they may throw an exception, if their executions fails. Both labels and concurrency markers are carrying additional information about the method, they are optional, so this extra feature may be helpful when dealing with various technology specific representations (they may store extra information which otherwise wouldn’t be possible). Methods are defined using the keyword operation followed by a name. The overview of this component is summarized in Table 3.6.

Example:

```java
public operation takeOff (speed : in Integer,
landingGear : inout Boolean) : String {
  ... 
}
```

Parameters

Parameters are primarily used to pass data into methods that is required for the execution. Usually, they just bring some values into methods. However, they may also be used for the results – in such a case they are passing to a method a reference pointing to the location where
the results should be stored. Finally, they may also serve for both purposes. The way they are used is called a \textit{directionality}. The \textit{parameter} definition consists of two parts - its name and its type. The type is used to determine what kind of data may be referenced or stored by a given \textit{parameter} (\textit{primitive type} vs \textit{classifier}). Moreover, similarly as methods they may also be labeled with additional information, desired for particular representations. The overview of this component is summarized in Table 3.7.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support (+/− ) or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td></td>
</tr>
<tr>
<td>Visibility</td>
<td>−</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>+</td>
</tr>
<tr>
<td>Directionality</td>
<td>‘in’, ‘out’, ‘inout’</td>
</tr>
<tr>
<td>Qualifiers</td>
<td></td>
</tr>
<tr>
<td>Labels</td>
<td>‘ordered’, ‘unique’</td>
</tr>
<tr>
<td>Members</td>
<td></td>
</tr>
</tbody>
</table>

\textbf{Return types}

A \textit{method} may return a value. This particular type of value that is the result of the method execution is named the \textit{return type}. This value may be of a \textit{primitive type} or a reference to a \textit{classifier}.

\textbf{Structural features}

This type of unified data structure representation elements are parts of the \textit{classifiers} and reflects their internal structure. Structural features may store values of the built-in primitive types or custom primitive types. Finally, they may point also to instances of classifiers. There are two types of \textit{structural features}:

- \textit{attribute}

- \textit{reference}

They are essentially similar, but they differ according to the \textit{containment} relation. This difference is described below. Both subtypes of \textit{structural feature} make use of the concept of \textit{labels} to store some additional information required by some concrete representations.

\textbf{Attribute}

An \textit{attribute} is a \textit{structural feature} that is exclusively contained within the \textit{classifier}. It exists only in the scope in which it is defined. In other words, such a \textit{structural feature} cannot exist independently. It is considered to be solely owned by the \textit{classifier} – it cannot be shared among various objects. When it is not used with primitive types, it reflects the concept of
**CHAPTER 3. UNIFIED CONSTRAINT REPRESENTATIONS**

**composition** from the UML class diagram. An attribute is defined using the attribute keyword followed by a name. The type of the attribute denotes either the type of value that is stored or the type of a referenced classifier. This type of structural feature contains also an additional feature aggregation that may be also used by a certain representation to convey some valuable extra meta information (a concept derived from the UML class diagram). The overview of this component is summarized in Table 3.8.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support (+/−) or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>attribute</td>
</tr>
<tr>
<td>Visibility</td>
<td>+</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>+</td>
</tr>
<tr>
<td>Qualifiers</td>
<td>'const', 'static'</td>
</tr>
<tr>
<td>Labels</td>
<td>'derived', 'ordered', 'readonly', 'unique'</td>
</tr>
<tr>
<td>Aggregation</td>
<td>'none', 'composite', 'shared'</td>
</tr>
<tr>
<td>Members</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8: UnifiedOCL – Attribute

Example:

public attribute age : Integer ;

**Reference**

A reference is a very similar concept as the attribute. The difference is that the reference always refers to an external instance of a classifier. It is recommended not to use it for primitive types. However, when used with primitive types, then it has exactly the same meaning as the attribute. It is due to the fact that primitive values are always stored in the location where they are defined. The classifier, within which the reference is defined, is never the owner of the referenced classifier. The referenced object can exist without a surrounding container (classifier). The possession relation does not take place between the referenced object and the containing classifier. Referenced objects may be shared by many other objects. The second difference to the attribute is also a consequence of the fact mentioned above – the concept of aggregation (as described for the attribute) cannot be used for references as it makes completely no sense. Additionally, references makes use of the concept of a directionality. To put it simply, it means that two classifiers, used as the type of references, can mutually reference each other. Therefore, using the hash character (number sign) # just after the reference name, one may point the name of the reference on the other side of the relation i.e. the name of the reference member of the classifier specified as the type of the reference. This concept is entirely taken from the OCLinEcore representation². The overview of this component is summarized in Table 3.9.

Example:

public reference plane : Plane;
public reference departures#sourceAirport : Flight[0..*];

Table 3.9: UnifiedOCL – Reference

<table>
<thead>
<tr>
<th>Feature</th>
<th>Support (+/−) or Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyword</td>
<td>reference</td>
</tr>
<tr>
<td>Visibility</td>
<td>+</td>
</tr>
<tr>
<td>Multiplicity</td>
<td>+</td>
</tr>
<tr>
<td>Qualifiers</td>
<td>'const', 'static'</td>
</tr>
<tr>
<td>Labels</td>
<td>'derived', 'ordered', 'readonly', 'unique'</td>
</tr>
<tr>
<td>Members</td>
<td></td>
</tr>
</tbody>
</table>

**Built-in primitive types**

*Built-in primitive types* are used to store basic values. Those simple kinds of data are either adapted from OCL or created specifically for the purposes of the common data structure representation. Types of data adapted from OCL are listed below:

- **Boolean**
- **Integer**
- **Real**
- **UnlimitedNatural** – it is just a * used by the concept of *multiplicity*
- **String**

Moreover, we introduced the type **Date** which is a subtype of **Integer** and is aimed to store time and date related values. In the current implementation, there is no support for **literals** for that type.
The second specifically introduced type is **void**. The motivation for this type was to allow for a definition of methods returning nothing.

**Literals**

**Literals** represent the values of *built-in primitive types*. In other words these are the pieces of data of a given kind that could be assigned to *structural features* of the same given kind. They may be also passed as **parameters to methods** and returned from **methods** as the results of their execution.

**Inheritance**

The relationships between classes are based on the similar concepts that is used in the UML class diagrams. The two types of inheritance are:

- **specialization**
- **realization**
The difference between them is conceptual: the first one is focused on a code reuse while the second one over a type hierarchy. Such a motivation is justified by the fact that we are assuming that the type hierarchy is behavior based, like in Python (Duck test\(^3\)) and not structure based. In the structure based approaches, the multiple inheritance causes lots of problems. The goal was to obtain the unified representation compatible with modern languages, so we decided to allow for the multiple inheritance. This is why the behavior based inheritance seems to be more suitable.

In UML one will also find two types of relationships among classes: Generalization and Realization. The justification for their separation is a bit different – the first one is more structure oriented while the second one is more behavior oriented. But overall, the UML class relationship distinction is a bit fuzzy. If we consider translating them into Java concepts, the generalization corresponds to the extends keyword, while the realization can be expressed by the keyword implements.

**Specialization**

This type of inheritance is conceived to reflect the concept of code reusability. It can also be understood in terms of the structural inheritance and it is similar to the UML concept of generalization. To denote specialization, the specializes keyword is used.

Example:

```java
public class Passenger specializes Person {
    ...
}
```

**Realization**

The inheritance of this type denotes behavioral aspects of relations. In other words the subclass (subtype) implements and realizes behaviors (methods) of the superclass/supertype. This relationship is denoted using the keyword realizes. It is very similar to the concept of realization from UML.

Example:

```java
public class Plane realizes Machine {
    ...
}
```

### 3.5 Constraints in UnifiedOCL

OCL proved to be a very convenient way to define constraints for single objects and relations. It is also a very powerful language for a constraint specification in complex networks of

---

\(3\)https://docs.python.org/2/glossary.html\#term-duck-typing [Online, accessed 19-August-2015]
interrelated objects. Moreover, it is designed to harmonize well with UML. With this in mind, we decided not to drop the OCL legacy, but try to benefit from it using it internally in UnifiedOCL.

Similarly as discussed in Section 3.4 the syntax for OCL has been taken over from OCLinEcore and adjusted to our needs.

The OCL expressions are embedded into the data structure representation of UnifiedOCL in the same way as in OCLinEcore. They are specified within the scope of classifiers or methods or structural features. Therefore, the original OCL scope specification – the context clause – is not needed any more. As a result, the usage of OCL is significantly simplified.

In order to specify OCL constraints within the body of one of the allowed data structure representation components, one just needs to use the keyword (type of the OCL statement), optionally followed by its name, and the body of the OCL expression.

Another change comparing to the original OCL language is the unification of the init and derived types of constraints (see data origin in Section 2.2). Those two types of constraints can be used only with structural features and they both have impact on the initial (default) value of the structural feature. Taking this into consideration we merged them under the common constraint type default.

Finally, we introduced a new primitive type of data that is particularly suitable for holding time and date related values. It is denoted by the keyword date (subject to regulations from Section 3.4).

Below we present the list of supported types of constraints, grouped by the scopes in which they are allowed to occur:

- **Classifier** – within this scope we may define the OCL invariants denoted by the keyword invariant. They represent constraints involving one or more structural features. They should be valid all the time (during the whole execution of the program written in a specific technology).

- **Structural feature** – within this scope one can specify the OCL init and derive constraints using the common keyword default. They represent the value with which the certain structural feature should be initialized.

- **Method** – within this scope we can define three types of OCL constraints:
  - **preconditions** – denoted by the precondition keyword. They represent conditions that should be satisfied before the method execution.
  - **postconditions** – denoted by the postcondition keyword. They represent conditions that should be satisfied after the method execution.
  - **body specifications** – denoted by the body keyword. It is not really a type of a constraint but the way how the actual operation performed by a method is specified. The OCL expression is used to specify how the input is transformed into the output (result).

Examples of how those constraints can be used could be found in Appendix E.2.
3.6 OCL limitations

The OCL manual [30] states that OCL is ‘a formal language used to describe expressions on UML models’. This definition, however, says nothing about constraints. And if we take a closer look on the OCL usage, we will see that OCL is a very powerful language when it comes to defining constraints over relations/associations of objects but it lacks of certain features for defining many other types of constraints. Among others, it does not provide tools to validate certain values of attributes or determine the accuracy of values. Defining schema constraints is also a very intricate task. In order to expose some of the imperfections of OCL we will show a couple of examples.

Primary keys

In OCL there is no dedicated keyword to express the primary key constraint, one of the most popular ones in the relational domain. Using the theoretical definition of this constraint we may use a less unambiguous representation of this constraint – the combination of the not null and the unique constraints.

primarykey = notnull + unique

However, then the OCL expression for such a determined constraint will look as follows (example for a class FooClass and the barAttribute attribute):

context FooClass
invariant PrimaryKey:
not self.barAttribute.oclIsUndefined()
and self.allInstances() ->forall(
tmp : FooClass | tmp<>self implies
tmp.barAttribute<>self.barAttribute);

This expression is very complex and would be used quite frequently in modeling of relational databases. As a result, specifying constraints in that way is extremely error-prone and requires a lot of patience. Both factors may lead to an aversion of programmers to use any unified constraint representation where constraints could be expressed using only OCL.

Email

The email address has a specific form of two parts separated with the @ character. Moreover, the second part is a domain address, so most probably it should contain at least one dot and a suffix matching one of the top level domains⁴. At the same time, the email address is stored just as a string. Therefore, to check whether the email address is correct, one needs to validate the email string against a specified regular expression or using any other sophisticated algorithm. Unfortunately, the latest specification of the OCL standard library⁵ does not specify regular expressions or a pattern matching for the type string. In other words, it is not possible to check the correctness of the email address using OCL. But even if we assume that the OCL standard library supports pattern matching, there is a high probability that each software developer would specify the email validation regular expression differently. It is getting even worse if there are

---


many developers that would like to work on the same source/project or when email addresses occur many times in the modeled domain.

**Precision of numbers** There is also no way to enforce a particular precision of numbers (type `real`) in OCL – e.g. by defining the number of decimal places. For instance, in Java we can enforce the precision of numbers using the `@Digits` annotation. The desired precision of numbers is extremely important in any system processing floating point numbers, therefore, the lack of support for a number precision definition questions the OCL usability as a language for constraint specifications.

**Time** The OCL standard does not have a dedicated type to store time or date related values. This was the reason why we introduced the `date` type. However, the `date` literals are not supported due to the fact that it would require to redesign the large part of the OCL standard library (operators overloading, syntax checking, etc.). Unless OMG extends OCL with datetime types, it won’t be possible to fully support any constraints that refer to time. E.g. a date of birth should be a past date with regards to the date when the constraint is being validated – for instance in Java there exists a special annotation for this type of constraint – `@Past`.

**Credit card number** In order to validate the credit card number (compute the checksum) one needs to use the *Luhn algorithm*\[54\]. This is not a trivial algorithm and due to the lack of typical programming constructs in OCL, it would be extremely difficult or impossible to implement it in OCL. In other words, the OCL specification is not powerful enough to allow for the validation of numbers that require complex algorithms computing checksums.

### 3.7 Unified Data Quality Assurance Markups

#### 3.7.1 Motivation

The common part of all the OCL problems stated in Section 3.6 is the fact that they are either technology specific or require demanding computational engines to be validated (see Subsection 2.2.2). The technology specificity in those examples involves a constraint types specificity, e.g. `primary key` as well as a data type specificity, e.g. `date`. Even though, these constraints are technology specific, they might be used by more than just one representation (platform), so it should be possible to denote them anyway in *UnifiedOCL*. On the other hand, the unified representation does not necessary have to be aware of how such constraints should be implemented or validated. The only thing the unified representation should be aware of is the fact that such constraints exist. The same is true with complex computational engines for validating constraints. The unified representation just needs to record the existence of a constraint of that type, but does not need to know how to exactly validate it (does not need to understand the semantic of that constraint and the algorithm). Of course, the full algorithm of validation is one of the forms of the representation of the constraint. But in the technology independent constraint representation it is not the most desired one (the easier and more compact one, the better).
All these considerations lead to the necessity of a separation of a constraint definition from a constraint meaning. We can think of it as a pair constraint name and constraint body. The constraint name is always related to a particular component of the data structure. E.g. constraint name is the CreditCard whilst the constraint body is the implementation of the Luhn algorithm. The unified representation should be aware that some data fields are marked as credit card numbers. This information is taken into account when generating the technology specific constraint representation, but the validation (performing Luhn algorithm) is the concrete platform implementation task.

As a result, for the purposes of the unified constraint representation, the only information required is the constraint name. The constraint body is the part that is required by a specific technology representation. The goal of the constraint transformation is to ensure that during a conversion of a technology independent to a technology dependent form of a constraint, it is possible to match a constraint name with a constraint body, and use the latter one as the validation code.

3.7.2 Approach

Syntax

Taking into account our considerations from Subsection 3.7.1, we decided to introduce another way of a constraint specification – Unified Data Quality Assurance Markups (UDQAM). This language consists of a very simple parametrized labels. In this thesis, we are considering the application of the UDQAM labels just for structural features, but this concept may be used also for other components of the unified data structure representation – e.g. method parameters or whole classifiers.

The syntax of the UDQAM language is very simple (expressed using EBNF notation[38]):

```
UDQAM-EXPRESSION := '{', { label-name,
                           [ '(' '(', PARAMETER-EXPRESSION}, ')') ]
                    }, '}'
PARAMETER-EXPRESSION := parameter-name, '=',
                        parameter-value, ['','']
```

where

- label-name and parameter-name are unquoted strings i.e. just a name or an identifier
- parameter-value is one of the following literals: an integer number, a real number, a boolean value, or a string (unquoted or single/double quoted)

In order to express a constraint using the UDQAM labels, one just needs to put the label names in curly brackets (optionally separated with commas). If the label is parametrized then
 parameters should be specified as pairs – a name and a value separated by the equality sign. Optionally pairs can be separated by commas defined in brackets following the label names. E.g. definition of labels telling that the size of the underlying structural feature should be between 5 and 10 and this structural feature cannot be null:

\{ notnull size(min=5 max=10) \}

The UDQAM labels are designed to be used within the UnifiedOCL code. The places where labels can occur within structural features are defined by UnifiedOCL syntax. UDQAM labels are depicted as black ovals at Figure 3.2.

Parametrization

Parameters of labels are used to convey some additional information about constraints. Such information is used by technology specific constraint representations. In the example from the Subsection 3.7.2 the size constraint would make no sense without at least one of the parameters (min and max). Parameters may also be used to pass arguments to the constraint validator – e.g. the length of the hashing function to compute the checksum. Parameters may pass the following types of values (some of them are consistent with the OCL types [30]):

- integer numbers – consistent with the OCL type Integer
- real numbers – consistent with the OCL type Real
- boolean values – consistent with the OCL type Boolean
- strings – consistent with the OCL type String
- identifiers – unquoted sequence of characters (numbers, letters and the underscore sign), without whitespaces, cannot start with a number

Integration with UML

Constraints in UML are expressed in the standardized way using OCL. The designers of UML wanted to make it extensible. Therefore, they didn’t limit the constraint specification languages just to OCL. They introduced the OpaqueExpression – the component that should contain constraints\(^6\). The specification says that an OpaqueExpression has two attributes:

- language
- body

As a consequence we can use any representation we want to express constraints in UML. Particularly, we can use directly UDQAM by setting the language attribute to the value

"UDQAM". Depending on which components of the UML class diagram are affected by the constraint it may be required to extend the UDQAM syntax with a pointing field (used by the body attribute of an OpaqueExpression). In our case, the structural feature from UnifiedOCL corresponds to Property of the Class from the UML class diagram. Constraints in UML (OpaqueExpression) can be only defined within a context. Property cannot serve as a context. Therefore, constraints for just a single property have to be defined within the class context. As a consequence, we need to extend the UDQAM syntax for the UML class diagram with the pointing element. Constraints defined within the context of a Class have to point to the affected property. The way how such an indicator is implemented, is dependent on the transformation that would link the UML diagram and UnifiedOCL. Additionally, it should be human-readable and clear for the requirement engineers that would use UML diagrams. In the transformation from UML to UnifiedOCL we decided to represent the syntax extended with pointers as follows:

```
property_name : UDQAM-EXPRESSION
```

This construct is sufficient as it unambiguously states which Property is affected as it is used within the scope of just a single Class (context).

### Integration with UnifiedOCL

UnifiedOCL allows for seamless integration with UDQAM. The syntax supports UDQAM labels within the scope of attributes and references. It allows a user to specify constraints in UnifiedOCL not only using OCL but also using UDQAM. All the problems mentioned in Section 3.6 can now be expressed in a compact and human-readable way. The following table shows the solutions to all the problems:

<table>
<thead>
<tr>
<th>Constraint</th>
<th>OCL</th>
<th>UDQAM</th>
</tr>
</thead>
</table>
| Primary key                    | context FooClass
invariant PrimaryKey:
not self.barAttribute
.oclIsUndefined()
and self.allInstances()
->forall(
tmp : FooClass |
tmp<>self implies
tmp.barAttribute<>
self.barAttribute); | { primarykey } |
| Email                          | --                                                                  | { email }                   |
| Precision of numbers           | possible parsing the string representation of a number              | { digits(integer=2, fraction=4) } |
| Time                           | --                                                                  | { past }                    |
| Credit card number             | --                                                                  | { creditcard }              |
The last column shows the UDQAM representation of constraints. Please notice that the UDQAM constraints have to be put in the correct place within the unified data structure representation i.e. within the scope of the particular structural feature.

Example:

public attribute salary : Real { range(min=2500, max=5300)} ;
...
public reference plane : Plane{ unique};

UDQAM labels can be grouped into sets called dictionaries. This division is particularly reasonable when one thinks of it as sets of labels, where each member of the same set describes the same area, is related to the same domain, etc. E.g. a set named SQLLabels may contain typical SQL constraint types: primarykey, foreignkey, unique, autoincrement.

Those sets are identified as plugins, which are plugged-in the UnifiedOCL syntax definition. It is motivated by the fact that UnifiedOCL has to be aware of all allowed constraints in order to ensure the correctness of the unified constraint representation. It means that all the possible UDQAM labels from all plugins are known in advance to enable the syntax check. Moreover, the label definitions contains information about their parameter names and types, which also allows for semantic checking for UDQAM\(^7\).

3.8 Summary

To conclude, UnifiedOCL – our unified constraint representation – enables to specify in a textual form both the data structure and constraints. Constraints can be specified in two ways:

- using OCL – especially useful where constraints involve graphs of interrelated objects or are easy to express using imperative statements
- using UDQAM – where the usage of OCL is not possible due to the lack of constructs or inappropriateness of a representation

This approach provides, on the one hand, flexibility and, on the other hand, fault-tolerance. First of all, it is because we rely on the standards (OCL), secondly because we provide validation rules for the UDQAM constraint dictionaries. Moreover, this solution is compatible with widely used standards (UML), benefits from reusing existing products and delivers the basis for the robust extensible frameworks. UDQAM dictionaries can be provided by third parties which does not close the doors against the constantly emerging technology specific representations and new kinds of constraints. The UDQAM language might be extended to cooperate with various kinds of data structure definition elements, enabling almost unlimited capabilities of constraint specifications.

\(^7\)in case of UDQAM: name and type checks
In Chapter 3 we explained our unified data quality representation (UnifiedOCL). We stated that it consists of the data structure definition and the constraint definition which combined together constitute a coherent model. We also said that constraints can be expressed using either OCL or UDQAM, and therefore, the designed framework is very flexible and extensible. What we intentionally skipped in Chapter 3, was the detailed explanation of transformations, which binds UnifiedOCL with technology specific representations. We also mentioned that the pure UML model enhanced with OCL constraints can also be considered as a form of an input – a technology specific representation, where the technology is understood as the OMG standard. The goal of this chapter is to elaborate the concept of transformations, which together with UnifiedOCL embodies the heart of our system. We will describe the transformations by confronting them with potential user requirements/use cases.

### 4.1 Overview

Figure 4.1 shows the overview of the system which is our approach to solve the problem of constraint transformations to achieve a unified data quality representation. The architecture of our system will be elaborated in the subsequent sections. Briefly, our system is structured as follows:

- The central point of the system is the unified data quality representation/uniﬁed constraint representation. It is a DSL called UnifiedOCL that is capable of reﬂecting both: the data structure and constraints.

- The standard way of expressing constraints is OCL. However, it is not powerful enough to specify all types of constraints. This is why the UDQAM constraint representation was introduced. It is natively supported by UnifiedOCL, but can be also used in the
UML class diagrams. Dictionaries of *UDQAM* labels representing constraints for different domains may be dynamically loaded into the *UnifiedOCL* module.

- A user can model a system, in a technology independent way, using either UML class diagrams enriched with constraints and then transform them in *UnifiedOCL* or use directly the textual representation of *UnifiedOCL*. The second step would be always to convert the *UnifiedOCL* representation to generate a technology specific output representation.

- The conversion between all concrete (technology specific) representations is possible via a two step transformation. The first step is a conversion form the source representation to *UnifiedOCL*, the second step is a transformation from *UnifiedOCL* to the target representation.

- There is a possibility to retrieve just the OCL representation from the *UnifiedOCL* source

- The *UnifiedOCL* representations can be serialized as the Extensible Markup Language (XML) documents.

- All the transformations (except the pure OCL extraction) use the M2T approach to simplify the architecture.

We will use Figure 4.1 in the following sections in order to explain the architecture of our system in details.

### 4.2 From UML to UnifiedOCL

According to the MDE doctrine, one of the first actions the information system designer should do after gathering requirements is to analyze and formalize them and then covert them into the model of the system. The model of the system is denoted using UML, most probably using the class diagram. Next, the model is enhanced with constraints using either the standard OCL notation or the *UDQAM* notation which we discussed in Chapter 3 emphasizing that it offers a seamless integration with UML. The final UML model enriched with constraints is depicted in Figure 4.1, in the left upper corner, as a rounded green rectangle. According to MDE, this model constitutes the platform independent model (PIM), which in the next step should be turned into the platform specific model (PSM). However, in our approach users can obtain any technology specific constraint and data structure representation from *UnifiedOCL* (the UML diagram is also considered as technology specific). Taking this into consideration they have two options (depicted as orange arrows in Figure 4.1):

- stick to the UML graphical representation enhanced with constraints and convert it first to *UnifiedOCL* and then to the desired representation

- choose the textual representation of *UnifiedOCL* and convert it in just one step to the desired representation
Figure 4.1: System overview
Being aware that some users prefer a graphical representation over the textual one, the emerging requirement to our system is to provide a transformation from the graphical UML diagram into the unified data quality representation (*UnifiedOCL*).

The difficulty in transforming the graphical UML class diagram with constraints into *UnifiedOCL* concerns the processing of the serialized form (possibly the binary one) of this diagram. As we mentioned in Section 3.3 the standardized textual UML Human-Usable Textual Notation (HUTN) notation is not popular. Moreover, each of the big market players offering CASE tools for UML use different formats for storing UML diagrams. The comparison of various CASE tools is available in [86]. It is also mentioned at the OMG website\(^1\). E.g. StarUML2 uses its own *MDJ format* (JavaScript Object Notation (JSON) based)\(^2\), MagicDraw supports XMI 2.1 and Eclipse implementation of UML (UML2)\(^3\), Visual Paradigm uses its own binary *VPP format*\(^4\). As there are so many serialization formats, there is no silver bullet – if one wants to provide a support for the transformation from graphical UML to *UnifiedOCL*, they need to implement the transformation for each tool separately. This is not a specific drawback of our system, but one of the most significant drawbacks of UML – the lack of portability due to the CASE tool dependence. Luckily, there is one format that seems to be supported by most of the big players (if not directly then at least via export libraries, plugins or tools).

The UML2 format\(^5\) is the Ecore based implementation of the OMG UML standard. It is stored in the XMI format, that uses the UML2 schema definition. Natively this format is supported by the Papyrus UML editor\(^6\), which is a powerful tool, available as an Eclipse plugin. Papyrus advertises itself as aiming to cover 100% of the OMG UML standard. The great advantage of this UML2 serialization format is its popularity, which results in a quite broad support by most of the big players (it is presented in the column *Can be integrated with* in the table with features of UML tools available in [86]).

To summarize this subsection, one of the transformation that our system provides is the transformation from the graphical UML representation (including constraints) denoted with the UML2 format to *UnifiedOCL*. Due to the lack of portability between UML CASE tools, supporting other formats is not a great conceptual challenge but it requires a lot of work and this is why our system supports by default only the Ecore based UML2 format.

The reverse transformation (*UnifiedOCL* to the graphical UML) is also not supported as it requires anticipation about the graphical aspects of the model (colors, positioning, layers, etc.). Such a transformation is possible to implement but it is completely dependent on the UML editor, and investigation of this topic is not relevant for achieving the goal of this thesis.

---


4.3 From and to UnifiedOCL

One goal of this thesis is to provide a tool that will be able to transform constraints between various technology specific representations. To investigate this problem thoroughly we decided to choose representatives of three different programming paradigms which were described in the Section 2.2.5. To fulfill the architectural requirements of our approach we need to implement just six transformations:

1. from Java to UnifiedOCL
2. from UnifiedOCL to Java
3. from SQL to UnifiedOCL
4. from UnifiedOCL to SQL
5. from Drools to UnifiedOCL
6. from UnifiedOCL to Drools

These transformations are depicted in Figure 4.1, in the bottom part, as green arrows. The supported technologies are depicted as rounded green rectangles at the very bottom of this figure. As it was discussed in Section 3.1, a transformation between randomly chosen two representations requires always two steps of conversion (from a source representation to UnifiedOCL and from UnifiedOCL to a target representation). It is justified by the fixed number of transformations to be implemented when adding a new representation (assuming that a transformation between any two representations should be possible).

4.4 M2M vs MT2

In Section 2.4 we described the differences between the M2T and M2M transformation. Here, we are describing the applicability of both approaches for the seven required transformations described in the previous two subsections (1 for UML to UnifiedOCL and 3 * 2 for other technologies to and from UnifiedOCL).

In order to minimize the number of transformations that should be implemented, we decided to introduce a central element – UnifiedOCL – our unified data quality requirements representation.

From UnifiedOCL

Let’s first consider the transformation from UnifiedOCL to a concrete technology, which is the second step of two-step transformations (depicted in Figure 4.1 as green arrows going out of the UnifiedOCL blue rectangle). The input is always a text file containing our unified
data quality representation. The outputs are various text files depending on the selected target representation. In other words: one input form, many output formats. Implementing this transformation as M2M requires parsing the *UnifiedOCL* input to build a model (in-memory tree-like traversable representation), then transforming this model into a technology specific model and finally serializing (printing into a file) this technology specific model to obtain technology specific sources. Using the M2T approach sounds much easier: one needs to parse the input text file to build a model and then just to serialize this model. Therefore, the step with model to model transformation is not necessary any more. Of course this approach (as mentioned in Section 2.4) requires as many serializations as specific representations. But still this decreases the amount of operations to be implemented twice ($n$ – number of technology specific representations):

- the *UnifiedOCL* source parsing – this step is present in both approaches
- serialization into a target platform – requires $n$ versions (does not matter whether the serialized model is for a unified M2T or a concrete M2M representation)
- a unified model to a concrete model transformation – requires $n$ versions for M2M, and 0 for M2T

**To UnifiedOCL**

The first step of an any-to-any representation transformation is the transformation from a technology specific representation to *UnifiedOCL* (depicted in Figure 4.1 as green arrows going in the *UnifiedOCL* blue rectangle) The inputs are various types of sources depending on the technology. The definition of the output is more complicated. Of course it is some form of *UnifiedOCL* – either the textual file or the in-memory traversable model. Later in this section we will explain why we decided on the textual one. At this point, we will show that it doesn’t matter what we’ve chosen, the number of operations (components of the whole transformation) is always the same for both M2M and M2T ($n$ – number of technology specific representations):

- the concrete representation source parsing – this step is always present whether we choose M2M or M2T
- transformation into *UnifiedOCL* – requires $n$ operations (one per each technology specific representation)

So the remaining decision to be taken was whether the output format should be a text file or an in-memory representation. Choosing the in-memory representation (M2M) would seem more reasonable as then the second step does not require the *UnifiedOCL* source parsing. On the other hand the textual representation would allow a user for modifying the *UnifiedOCL* source, before further processing. Therefore, we decided to choose the M2T approach also for the first step of the transformation, and as a consequence the text file as the representation. Currently there are no other benefits of using the M2M over M2T transformation, especially that the operation of parsing the *UnifiedOCL* source is not one of
the computationally expensive (see Section 6.17) in the whole transformation process.

Also from the pragmatic point of view, the M2T approach appears to be more convincing. There are plenty of tools supporting M2T at the market, while at the same time there are almost no mature frameworks supporting M2M. The existing ones are either poor quality or awkward to use, but the most essential drawback is the fact that they are based on the imperative paradigm (Query/View/Transformation Declarative (QVTd) is not yet finished). As long as the model to model transformation is not a declarative one (see Section 2.4), which would simplify the process of the transformation implementation, there is no justified reason to use this approach. The elaborated technological aspects of those transformations are described in Chapter 6.

4.5 Additional transformations

Through analyzing potential use cases and requirements for our system, we decided to add two additional transformations:

1. a pure OCL extraction (depicted in gray color, in the right part of Figure 4.1) – OCL is a typical textual representation of constraints that coexists well with UML. There exist tools, e.g. DresdenOCL, that allow for a validation of objects (e.g. Java Beans described in Section 2.2.5), which take as an input the pure OCL specification. Therefore, we decided that the extraction of OCL statements from UnifiedOCL would increase the usefulness of our system. The extracted OCL statements have to be wrapped within the context information and the correct type of constraint (e.g. an invariant). In case of the data origin type of a constraint, it is not possible to distinguish between the derive and the init OCL constraint type (we merged them). Therefore, this is the only part of a transformation which is not reversible.

2. XML serialization (depicted in gray color, in the left part of Figure 4.1) – in order to ensure the portability of UnifiedOCL we decided to enhance our system with the capabilities of a bidirectional conversion between the XML format, that could be easily parsed by external tools that are not aware of the UnifiedOCL DSL syntax.

4.6 Architecture of transformations

As we discussed in Section 4.4, we decided to use only M2T transformations. In our system each of such a transformation consists of three main steps indirectly mentioned in the previous sections:

1. obtaining a traversable in-memory model instance from the source file(s)

2. model discovery and analysis

3. performing a model to text serialization

The goal of this section is to elaborate our approach how the transformation should be implemented. As there are enormously many technology specific representations, it may happen that any of them would require an individual approach. Here we are just trying to point out the common parts of all transformations from the conceptual point of view. In our work, a transformation is understood as just one step of a two-step conversion between various technologies (does not matter whether the first or the second one). I.e. in the following descriptions either the source or the target representation is UnifiedOCL.

### 4.6.1 Traversable model instance

To be able to process a textual file we need to represent it as an in-memory object that is traversable and one can easily navigate between its various components. For most of the specific technologies the abstract syntax tree would fulfill those requirements. Sometimes it could be just a navigable graph of objects reflecting a particular source code file. In our system we can distinguish three ways of obtaining a traversable model instance.

1. Use the existing parser. Some technologies are accompanied or supplied with parsers, e.g. Drools, that we can reuse to obtain an abstract syntax tree that corresponds to the input source file.

2. Use the language grammar. If a language is expressed by a known grammar, but no parser is available, we can adapt such a grammar and generate a parser directly from the grammar. We can use for this, e.g. the ANTLR\(^8\) parser generator.

3. Use a model discovery tool. Many reverse engineering tools or libraries provide a way to discover an underlying model and represent it in the form of a navigable tree or graph that we can reuse for the purposes of the transformation implementation. E.g. MoDisco\(^9\) is a tool capable of revealing and exploring various input formats including Java packages.

### 4.6.2 Model discovery and analysis

Usually, a single pass over the traversable model is not enough to produce the output – the target representation. Very often language concepts are divided among various constructs or scopes, they may be nested or embedded in some external scopes, they may include circular dependencies etc. In such cases, we need to analyze the whole model instance, gather the relevant information, process them and then use them to produce the output. E.g. constraints in SQL can be defined:

- as a part of a column definition
- at the table level – interchangeable with columns
- as an `ALTER TABLE` statement defined completely outside of the table, possible in a different SQL script

\(^8\)http://www.antlr.org/ [Online, accessed 19-August-2015]

CHAPTER 4. CONSTRAINT TRANSFORMATIONS

But they all may mean the same, e.g. a single column should contain just positive numbers. Therefore, we cannot produce the output at the same time when we are traversing because we may not be aware of the existence of a constraint when it is defined in a different posterior location. Let’s consider the SQL to Java conversion. In Java Bean Validation such a constraint is defined as an annotation (e.g. `@Min(value=0)`) in a location directly preceding the affected class attribute. Therefore, it should be serialized (printed) before serializing the class field. Due to an impedance mismatch between Java and SQL it won’t be possible. This is why the additional step of analysis of the model is required. Moreover, the intermediate results have to be buffered.

At the same stage discovered constraints should be transformed into the valid intermediary or directly to the target representation. Due to the fact that in many technologies constraints are stored a bit aside from the data structure definition it is important to know the correct output form of the constraints before the model serialization as within this phase they should be smoothly merged with the target model constructs. Part of the constraint discovery process is matching the constraint with the affected element, as this information would be required to correctly merging the constraint model with the data structure model. The used naming convention may be specific for a particular transformation depending on whether the source and the target representation naming convention are similar.

4.6.3 Model serialization

The last step assumes either traversing the intermediary model from the previous step or directly the source model (depending on the particular representation) or both of them and printing the target textual representation. Printing is a complicated operation as it has to be aware how to represent the intermediary or the source model in the available terms of the target model. Moreover, it has to be aware also where (a location in the model) the constraints could be used. If during the process of serialization a model into the target representation such a constraint possible location in encountered, the printer should check what element of the data structure definition could be affected by the constraint occurring in the given place and check whether the intermediary constraint representation created in the previous step contains a constraint for this targeted element. If the answer is positive then the new constraint representation is embedded into the target representation.

4.7 Summary

The benefits of our system are as follows:

- support for system design and modeling
- unified constraints (data quality requirements) representation
- transformation from any technology specific representation to any other
- cooperation with currently used industrial standards (UML)
• ability to compare constraints defined using different notations (by bringing them into a common representation)

Moreover, our system is built as a set of plugins and it is designed to minimize the number of transformations required to convert one representation to any other. As a consequence, our system could be easily extended with new constraint representations, e.g. JavaScript (Form Validation\textsuperscript{10}) or PHP with Symfony\textsuperscript{11}.

The less obvious advantages of our approach that significantly augments the automation of laborious software engineering tasks are as follows:

**Coherent system** A single system may be composed of many parts. Those parts could be related to different layers of the real application (e.g. the persistence layer – SQL or the business logic layer – Java) or can just be the units of business logic responsible for some subsets of the domain problems. All the parts could be written in different technologies. Therefore, the usage of our tool increases the consistency of constraint definitions in the whole information system by allowing for transformations from one representation to the other. E.g. a software developer creates a constraint just once – at the database level, and later during developing the business logic or the presentation layer, they just generate corresponding constraints based on the job that they thoroughly did at the beginning for the persistence layer.

**Constraint central repository** In case of complex information systems the \textit{UnifiedOCL} can serve also as a central repository of data quality requirements. Constraints defined once, in the central place could be used during further phases of the software development process to generate constraint representations for various components of the system. In that sense we can consider our system as a data quality management framework. Even though the current implementation is not capable of a partial generation of the specific representation (e.g. a regeneration of the whole data structure but preserving the method bodies), it can be extended in such a way (a modification of transformation implementation) and as the result it could evolve to the fully functioning MDE generation framework that supports constraints.

\textsuperscript{10}http://www.w3schools.com/js/js_validation.asp [Online, accessed 21-August-2015]
5

Data Quality Feedback

5.1 From requirements to the approach

In the previous chapter we have discussed how the unified data quality could be achieved by constraint transformations. However, we did not present so far how the data quality could be assessed. In this chapter, we are describing our approach to provide feedback about the achieved data quality.

The concept of data quality in our system could refer to:

- instance data quality – the data quality of real instances of various classifiers of the concrete technology data model. In other words, this is the overall measure of all objects whether they are satisfying the constraints imposed on them.

- schema data quality – the quality of the target technology specific representation. We are interested whether it fulfills the meta-model constraints. This concept could also be extended with the definition of unified meta constraints that would serve as the meta model for UnifiedOCL.

The second approach is very complex and goes beyond the scope of this thesis, therefore, we will address only the first sense of the concept. Our goal is to provide a tool that would inform a user about the achieved quality of the given set of data entities. The data quality is computed using violations of constraints that are defined for that data.
5.2 Data quality dimensions

In Section 2.8 we discussed various definitions of data quality dimensions. For us, it is only important that there are some named categories of data quality aspects. Our goal is to express those aspects in a numerical way in order to equip a user with a tangible feedback regarding data quality.

Our idea is to, first of all, group constraints defined for some data structure into sets, each of which could be named with different data quality dimensions. In other words, data quality dimensions serve as labels for various categories of constraints. How a user would categorize the constraints, and what labels would be used, are not the subjects of our considerations. We only recommend to use data quality dimensions for the names of the labels, e.g. depicted in Figure 2.11 and we will provide a sample categorization of constraints for the airline domain presented in Appendix E.3.

5.3 Identification of constraints

In order to classify constraints into various data quality dimensions we need to use a constraint name and a constraint location. Using just a name would lead to ambiguous situations. E.g. two fields of the class Flight have a constraint size, but one is of the type string (description) while the other one is the collection passengers. Therefore, violations of the string length (size) would be considered as an accuracy problem while if the collection has the wrong size it could be considered as the consistency problem. The constraint location within a data schema solves this problem by pointing to which object is affected (in which scope the constraint is used). And this information, together with the constraint name, suffices to identify a constraint uniquely. E.g. a constraint identified by the pair (Flight.description, size) is different from the constraint identified by (Flight.passengers, size).

5.4 Data Quality Indicators – mapping penalty

In our approach we define the mapping of constraints into data quality dimensions, but it has nothing to do with the numerical value informing about the overall data quality which could be used for a user feedback. We may assume that a particular constraint is satisfied or not, but this information cannot be aggregated, so we can only present one number for a particular data quality dimension as a single numerical value. We may just say how many objects of the total number of all considered objects violates the constraint.

We decided to enhance the solution presented in Section 19.3 of the master thesis of Olivier Probst [69] and adapt it for the purposes of data quality feedback. Moreover, we are using similar approach as presented by Konstantina Gemenetzi [23] with regards to the usage
of the concept of data quality indicators to calculate the value of data quality. The term data quality indicator is defined by Karl Presser as follows: ‘A data quality indicator is a concrete intrinsic or contextual constraint to data that can help to satisfy a data quality dimension’ [68].

In essence every constraint is accompanied with the percentage value (0-100%) which corresponds to the numerical scale of \([0.0, 1.0]\). This number is considered as a penalty value (the higher the more severe is the violation). A data quality indicator with the penalty value of 1.0 is identical to the concept of a constraint where the validation evaluates to only true or false. All violations are accumulated into one formulae soft constraint fulfillment (introduced by Probst [69]) that constitutes to a single value informing about the overall data quality.

In this thesis, we are using the same approach, but we are computing a separate value for each data quality dimension i.e. we are computing the value of a soft constraint fulfillment taking into account only constraints classified as belonging to the given data quality dimension. Moreover, we are enhancing this formula with a way to capture types inheritance.

### 5.5 Mapping representation

In order to represent a mapping we use, a so called, data quality mapping file. In such a file every line stores a single mapping of a uniquely identified constraint (Subsection 5.3). Therefore, each line consists of three components:

1. constraint location
2. constraint name
3. mapping to data quality dimension

A constraint location is an unquoted string without whitespaces that contains a path to the place where the constraint is located. The interpretation of such a path depends on the specific technology, therefore the only general requirement is to separate path segments with an underscore sign (_). E.g. for a method parameter name in Java this could be a package name + class name + method name + parameter name, but the underscores are used instead of dots. A constraint name is an identifier (a unquoted string without whitespaces).

The syntax for expressing the third component, the mapping to a particular data quality dimension, is very simple. It is specified with pairs of data quality dimension names and penalty values (from the range of 0.0 to 1.0) separated by commas. The whole component should be placed inside square brackets. The following listing shows a sample entry from the data quality mapping file:

```
model_person_address NotBlank [Completeness 0.2, Consistency 0.1]
```

It assigns the NotBlank constraint in the location model_person_address (for instance in Java it could be interpreted as the model package, the Person class and the address field) to two data quality dimensions: Completeness with the penalty value of 0.2 and Consistency with the penalty value of 0.1.
5.6 Data Quality Mapping in UnifiedOCL

Data quality mappings can be directly represented in UnifiedOCL. In order to do that, we extended the original UnifiedOCL syntax with the following grammar concept in order to be able to represent any mapping of an UDQAM label or OCL invariant.

\[
\text{DQ-MAPPINGS_DEFINITION := ['\', DQ-MAPPING,}
\{['\',] DQ-MAPPING}, ',']
\]

\[
\text{DQ-MAPPING := dimension-name, penalty-value}
\]

This grammar element can be used directly after the definition of a single UDQAM label or directly after the invariant specification.

Example:

```
public attribute salary : Real { range(min=2500, max=5300) [Correctness 0.2] } ;
```

5.7 Computing data quality feedback

5.7.1 Feedback levels

We assume that the quality of a flawless data (that does not violate any constraint) equals 100% (or 1.0). In order to compute the value of data quality for a particular dimension, we are subtracting a normalized sum of all penalties (violated constraints that are mapped into the given data quality dimensions) from this maximum value. Which constraints are taken into account and how the normalization is applied depends on the desired abstraction level.

For the purposes of this thesis we decided to present the data quality feedback at two different levels of granularity:

1. Single class
   (a) Single object
   (b) Multiple objects separately
   (c) Multiple objects aggregated
2. Multiple classes (class = all instances combined)
   (a) Single class
   (b) Multiple classes separately
   (c) Multiple classes aggregated

In the following subsections we present formulae to compute the values of data quality dimensions – v\text{dqd} at particular levels.

It is important to notice that if a single instance of an object (a class of objects) has no constraints belonging to a particular data quality dimensions, but we still want to compute the value of data quality, it does not require any modification in the following formulae – the penalty would be 0, which would not distort the final value.
5.7.2 Notation

In this subsection we present the entire notation that is used in the following subsections to explain the formulae which are used to compute values of data quality dimensions.

Table 5.1: Symbol legend of equations for values of data quality dimensions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>a constraint defined by a pair of a constraint location and a constraint name defined in Subsection 5.3</td>
</tr>
<tr>
<td>d</td>
<td>a data quality dimension</td>
</tr>
<tr>
<td>p</td>
<td>the penalty value of a constraint $c$ mapped to a data quality dimension $d$</td>
</tr>
<tr>
<td>t</td>
<td>a single type of data (e.g. a class in Java or a table in SQL)</td>
</tr>
<tr>
<td>$T$</td>
<td>a set of types of data</td>
</tr>
<tr>
<td>$T^+$</td>
<td>a closure of the set of types of data i.e. a set of types consisting of all types existing in the set $T$ and all supertypes of those types (recursively)</td>
</tr>
<tr>
<td>$T^*$</td>
<td>a set of all types of data</td>
</tr>
<tr>
<td>i</td>
<td>a single instance of a data object</td>
</tr>
<tr>
<td>$i_t$</td>
<td>a single instance of a data object pointed by a reference of the type $t$ (either the type $t$ or any of its subtypes)</td>
</tr>
<tr>
<td>$I_t$</td>
<td>a set of instances of data objects pointed by a reference of the type $t$ (either the type $t$ or any of its subtypes)</td>
</tr>
<tr>
<td>$I_t^*$</td>
<td>a set of all instances of data objects pointed by a reference of the type $t$ (either the type $t$ or any of its subtypes)</td>
</tr>
<tr>
<td>$C_t$</td>
<td>set of all constraints defined for the type $t$</td>
</tr>
<tr>
<td>$C_t^+$</td>
<td>set of all constraints defined for the type $t$ including constraints defined also for supertypes of $t$ (transitively)</td>
</tr>
</tbody>
</table>

Figure 5.1: Definition of the penalty function for a specified dimension

$$penalty(c, d) = \begin{cases} 
0.0 & \text{if constraint } c \text{ is not mapped to the data quality dimension } d \\
p \in (0,1] & \text{if constraint } c \text{ is mapped to the data quality dimension } d \text{ with the penalty value } p
\end{cases}$$

Figure 5.2: Definition of the penalty function for all dimensions

$$penalty_{ALL}(c) = \begin{cases} 
0.0 & \text{if constraint } c \text{ is not mapped to any data quality dimension} \\
p \in (0,1] & \text{if constraint } c \text{ is mapped to any data quality dimension with the penalty value } p
\end{cases}$$

Figure 5.3: Definition of the violation function

$$violation(i, c) = \begin{cases} 
1 & \text{if instance } i \text{ violates constraint } c \\
0 & \text{otherwise}
\end{cases}$$
5.8 Constraint inheritance

One of the tricky issues that we have to address is the constraint inheritance. Some classifiers may form relationships of the type subclass-superclass constituting complex hierarchies. Constraint inheritance affects the way how constraints are evaluated and which are taken into account when computing a value for the selected data quality dimension at a particular level.

These issues are addressed also in the subsequent subsection, when discussing particular granularity levels of presentation.

5.8.1 Single class – hierarchy of constraints

We may refer to a single object using its type, or any of its supertypes (Liskov Substitution Principle\cite{52}\cite{55}). Similarly, constraints may be defined directly in the given type or in any of its supertypes. As a consequence, during validation of constraints for a single object, not only constraints defined in its type are taken into account, but also all the constraints defined in all its supertypes (recursively). For the data quality visualization, we assumed that the type of an object is determined by the type of the reference that is used to point to that object. E.g. if we use the reference type Person for an object of type Pilot, which is a direct subclass of the Person, the constraint defined in the type Pilot will not be taken into account during calculation, but only the constraints from the type Person and all its supertypes. However, if we use the reference of the type Pilot for an object of the type Pilot, constraints included into calculation would come from Pilot, Person and other supertypes.

5.8.2 Multiple classes – only current type specific constraints

The approach from the previous subsection is safe when we are working with the first level of granularity defined in Subsection 5.7 (a single class). Unfortunately, it fails if we would like to obtain a combined feedback for more than just one class. E.g. showing the data quality feedback for two classes: Pilot and Person from our airplane scenario should result in showing a feedback for all instances of the classes Pilot and Passenger. However, a Pilot is also a Person, therefore, having two targeted types, constraints for the type Pilot would be taken twice – more precisely: once when the computation runs for the class Pilot (constraints defined in classes Pilot and Person) and the second time, when the Person class is considered (instances of the Pilot are casted to their supertype Person – only constraints defined in the type Person are taken into account). To solve this problem we came up with a solution that consists of three steps:

1. First we are collecting all the objects that should be taken into this combined data quality feedback. In our case, these would be all the instances of classes Pilot and Passenger. In general it would be a set of all instances of all selected classes (including recursively all subclasses of those classes).
2. As a second step we are computing a closure of the selected types. This is a set of all selected classes plus all superclasses of those classes (recursively). If the superclass of the class Person would be the class Vertebrate, then our set would contain both classes Pilot and Person (taken not only because this class was specifically selected but also as a super type of Pilot) as well as the class Vertebrate.

3. Finally we are iterating over all classes (types) found in the previous step (Pilot, Person and Vertebrate). For all instances obtained in the first step, we are checking whether they belong to the current class. If this condition is evaluated to true, then we are taking into consideration only constraints (more precisely existing constraint violations) that are defined in the current class – without constraints from sub- or superclasses.

One has to be aware that in this approach constraints defined specifically in the type Passenger class would not be taken into consideration. This is a correct situation, as instances of the type Passenger are pointed using references of the type Person, for which those constraints are not accessible.

5.9 Formulae

5.9.1 Single class level

For a single class level, regardless the number of selected instances, we are taking into account violations of constraints defined in that class and in all of its superclasses (recursively) – described in Subsection 5.8.1. Of course, computations are run for each dimension separately and with the respect to constraints that are mapped to the given data quality dimension. We are summing up all the violations for this single/multiple instance(s) and normalizing this sum using the sum of all possible violations for the class of which representatives are taken into account. In case of multiple instances, the number of instances is also taken into consideration during the normalization process. To obtain the overall value of data quality for each dimension we are subtracting the obtained ratio (actual violations divided by the sum of all possible violations) from the value representing the flawless data (1.0). The achieved result is the value of the selected data quality dimension for selected object instances. Depending on the level of granularity we are using one of the following formulae to compute the result:

Single instance

\[
vdqd(d, i_t) = 1 - \frac{\sum_{c \in C_t^+} violation(i_t, c) \cdot penalty(c, d)}{\sum_{c \in C_t^+} penalty(c, d)}
\]

Figure 5.4: Value of data quality dimension for a single instance
Multiple instances separately

\[
for \ i \in I_t : \\
vdqd(d, i) = 1 - \frac{\sum_{c \in C_t^+} violation(i, c) \cdot penalty(c, d)}{\sum_{c \in C_t^+} penalty(c, d)}
\]

Figure 5.5: Value of data quality dimension for multiple instances separately

Multiple instances combined

\[
vqd(d, I_t) = 1 - \frac{\sum_{i \in I_t} \sum_{c \in C_t^+} violation(i, c) \cdot penalty(c, d)}{|I_t| \cdot \sum_{c \in C_t^+} penalty(c, d)}
\]

Figure 5.6: Value of data quality dimension for multiple instances combined

Single class (all instances combined)

\[
vqd(d, t) = 1 - \frac{\sum_{i \in I_t^*} \sum_{c \in C_t^+} violation(i, c) \cdot penalty(c, d)}{|I_t^*| \cdot \sum_{c \in C_t^+} penalty(c, d)}
\]

Figure 5.7: Value of data quality dimension for a single class (all instances combined)

Example

Our scenario system consists of three classes: Person, Pilot and Airport (we assume that the Person class is not abstract). The class Pilot is a subclass of the class Person. In this example we want to compute the value for Timeliness for the class Pilot. This implies that constraints defined in the classes Pilot and Person will be included. Instances of all the classes and violations of the constraints are given in tables 5.3 and 5.2. The constraints to data quality dimensions mapping is presented in Figure 5.8.

<table>
<thead>
<tr>
<th>Constraint (location, name)</th>
<th>Pilot 1</th>
<th>Pilot 2</th>
<th>Pilot 3</th>
<th>Person 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>model_pilot_licenceexpired, Future</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>model_pilot_salary, Range</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>model_person_birthdate, Past</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>model_person_email, Email</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Table 5.3: Constraint violations - class Airport

<table>
<thead>
<tr>
<th>Constraint (location, name)</th>
<th>Airport 1</th>
<th>Airport 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>model_airport, AvoidCongestion</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>model_airport_arrivals, Size</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

model_airport AvoidCongestion [Correctness 0.5]
model_airport_arrivals Size [Correctness 0.2, Consistency 0.2]
model_person_birthdate Past [Timeliness 0.2]
model_person_email Email [Correctness 0.2]
model_pilot_licenceexpired Future [Timeliness 0.5]
model_pilot_salary Range [Correctness 0.2]

Figure 5.8: Constraints to data quality mapping for the classes Pilot, Person and Airport

In order to compute the value for the Timeliness for the class Pilot we have to use the formula presented in Figure 5.7. Let’s first compute the value for the denominator:

\[
|I_1^*| \cdot \sum_{c \in C_1^+} \text{penalty}(c, d) = \\
\#pilots \cdot \text{penalty}((\text{model_pilot_salary, Range}), \text{Timeliness}) \\
+ \text{penalty}((\text{model_pilot_licenceexpired, Future}), \text{Timeliness}) \\
+ \text{penalty}((\text{model_person_birthdate, Past}), \text{Timeliness}) \\
+ \text{penalty}((\text{model_person_email, Email}), \text{Timeliness}) \\
= 3 \cdot [0.0 + 0.5 + 0.2 + 0.0] = 2.1
\]
and for the nominator.

\[
\sum_{i \in I^*_t} \sum_{c \in C^+_i} \text{violation}(i, c) \cdot \text{penalty}(c, d) = \\
\left[\text{violation}(\text{Pilot1}, (\text{model\_pilot\_salary, Range})) \right. \\
\left. \times \text{penalty}(\text{model\_pilot\_salary, Range}, \text{Timeliness}) + \text{violation}(\text{Pilot1}, (\text{model\_pilot\_licence\_expired, Future})) \right. \\
\left. \times \text{penalty}(\text{model\_pilot\_licence\_expired, Future}, \text{Timeliness}) + \text{violation}(\text{Pilot1}, (\text{model\_person\_birthdate, Past})) \right. \\
\left. \times \text{penalty}(\text{model\_person\_birthdate, Past}, \text{Timeliness}) + \text{violation}(\text{Pilot1}, (\text{model\_person\_email, Email})) \right. \\
\left. \times \text{penalty}(\text{model\_person\_email, Email}, \text{Timeliness}) + \text{violation}(\text{Pilot2}, (\text{model\_pilot\_salary, Range})) \right. \\
\left. \times \text{penalty}(\text{model\_pilot\_salary, Range}, \text{Timeliness}) + \text{violation}(\text{Pilot2}, (\text{model\_pilot\_licence\_expired, Future})) \right. \\
\left. \times \text{penalty}(\text{model\_pilot\_licence\_expired, Future}, \text{Timeliness}) + \text{violation}(\text{Pilot2}, (\text{model\_person\_birthdate, Past})) \right. \\
\left. \times \text{penalty}(\text{model\_person\_birthdate, Past}, \text{Timeliness}) + \text{violation}(\text{Pilot2}, (\text{model\_person\_email, Email})) \right. \\
\left. \times \text{penalty}(\text{model\_person\_email, Email}, \text{Timeliness}) + \text{violation}(\text{Pilot3}, (\text{model\_pilot\_salary, Range})) \right. \\
\left. \times \text{penalty}(\text{model\_pilot\_salary, Range}, \text{Timeliness}) + \text{violation}(\text{Pilot3}, (\text{model\_pilot\_licence\_expired, Future})) \right. \\
\left. \times \text{penalty}(\text{model\_pilot\_licence\_expired, Future}, \text{Timeliness}) + \text{violation}(\text{Pilot3}, (\text{model\_person\_birthdate, Past})) \right. \\
\left. \times \text{penalty}(\text{model\_person\_birthdate, Past}, \text{Timeliness}) + \text{violation}(\text{Pilot3}, (\text{model\_person\_email, Email})) \right. \\
\left. \times \text{penalty}(\text{model\_person\_email, Email}, \text{Timeliness}) \right] = [1 \times 0.0 + 0 \times 0.5 + 1 \times 0.2 + 0 \times 0.0] \\
+ [0 \times 0.0 + 1 \times 0.5 + 0 \times 0.2 + 0 \times 0.0] \\
+ [0 \times 0.0 + 0 \times 0.5 + 0 \times 0.2 + 1 \times 0.0] \\
= 0.2 + 0.5 + 0.0 = 0.7
\]

Taking into account the values for the nominator and denominator we can compute the final result.

\[
vqd(Timeliness, Pilot) = 1 - \frac{\sum_{i \in I^*_t} \sum_{c \in C^+_i} \text{violation}(i, c) \cdot \text{penalty}(c, d)}{|I^*_t| \cdot \sum_{c \in C^+_i} \text{penalty}(c, d)} \\
= 1 - \frac{0.7}{2.1} = 0.667 \approx 67\%
\]

The Timeliness of the class Pilot results in about 67%.
5.9.2 Multiple class level

When computing a data quality feedback for this level of abstraction, we must adhere to the rule described in Subsection 5.8.2 (the only exception is computing a feedback for multiple classes by treating each of them separately – this adheres to the same rule as for the single class level). Therefore, we consider in the isolation every level of type hierarchy (up to the root) for each instance of each of the selected classes. This means that for each level of type hierarchy we are considering only constraints defined for this type and not for the supertypes. As described before, it prevents from including some violations more than once in the value of data quality dimensions. Similarly as in the case with just a single class, we are normalizing the actual sum of penalties by the number of instances of the given class multiplied by the sum of all penalties. The obtained value is then subtracted from the maximum value of 1.0. Depending on the level of granularity we are using one of the following formulae to compute the result:

**Multiple classes separately**

\[
vdqd(d, t) = 1 - \frac{\sum_{i \in I_t^*} \sum_{c \in C_t^+} violation(i, c) \cdot penalty(c, d)}{|I_t^*| \cdot \sum_{c \in C_t^+} penalty(c, d)}
\]

Figure 5.9: Value of data quality dimension for multiple classes separately

**Multiple classes combined**

\[
vqd(d, T) = 1 - \frac{\sum_{t \in T^+} \sum_{i \in I_t^*} \sum_{c \in C_t} violation(i, c) \cdot penalty(c, d)}{\sum_{t \in T^+} (|I_t^*| \cdot \sum_{c \in C_t} penalty(c, d))}
\]

Figure 5.10: Value of data quality dimension for multiple classes combined

**The whole system**

\[
vqd(d) = 1 - \frac{\sum_{t \in T^+} \sum_{i \in I_t^*} \sum_{c \in C_t} violation(i, c) \cdot penalty(c, d)}{\sum_{t \in T^+} (|I_t^*| \cdot \sum_{c \in C_t} penalty(c, d))}
\]

Figure 5.11: Value of data quality dimension for the whole system (all instances of all classes)

**Example**

Our scenario system consists again of three classes: Person, Pilot and Airport (we assume that the Person class is not abstract). The class Pilot is a subclass of the class Person. In this example we want to compute the value for Correctness for the whole system. This
imply that all the classes would be included. Instances of all the classes and violations of constraints are given in tables 5.3 and 5.2. The constraints to data quality dimensions mapping is presented in Figure 5.8.

In order to compute the value for the Correctness for the whole system we have to use the formula presented in Figure 5.11. Let’s first compute the value for the denominator:

\[
\sum_{t \in T^*} \left( |I_t^*| \cdot \sum_{c \in C_t} \text{penalty}(c, d) \right) =
\]

\[
#\text{pilots} \cdot \text{penalty}((\text{model\_pilot\_salary}, \text{Range}), \text{Correctness})
+ \text{penalty}((\text{model\_pilot\_licence\_expired}, \text{Future}), \text{Correctness})] +
\]

\[
#\text{persons} \cdot \text{penalty}((\text{model\_person\_birthdate}, \text{Past}), \text{Correctness})
+ \text{penalty}((\text{model\_person\_email}, \text{Email}), \text{Correctness})] +
\]

\[
#\text{airports} \cdot \text{penalty}((\text{model\_airport\_arrivals}, \text{Size}), \text{Correctness})
+ \text{penalty}((\text{model\_airport}, \text{Avoid\_Congestion}), \text{Correctness})]
\]

\[
= 3 \cdot [0.2 + 0.0] + 4 \cdot [0.0 + 0.2] + 2 \cdot [0.2 + 0.5] = 2.8
\]
and for the nominator.

\[
\sum_{t \in T^*} \sum_{i \in I_t^*} \sum_{c \in C_t} \text{violation}(i, c) \cdot \text{penalty}(c, d) = \\
\text{violation}(\text{Pilot1}, (\text{model}_\text{pilot}_\text{salary}, \text{Range})) \\
\ast \text{penalty}(\text{model}_\text{pilot}_\text{salary}, \text{Range}, \text{Correctness}) \\
\text{violation}(\text{Pilot1}, (\text{model}_\text{pilot}_\text{licensureexpired}, \text{Future})) \\
\ast \text{penalty}(\text{model}_\text{pilot}_\text{licensureexpired}, \text{Future}, \text{Correctness})] \\
\text{violation}(\text{Pilot2}, (\text{model}_\text{pilot}_\text{salary}, \text{Range})) \\
\ast \text{penalty}(\text{model}_\text{pilot}_\text{salary}, \text{Range}, \text{Correctness}) \\
\text{violation}(\text{Pilot2}, (\text{model}_\text{pilot}_\text{licensureexpired}, \text{Future})) \\
\ast \text{penalty}(\text{model}_\text{pilot}_\text{licensureexpired}, \text{Future}, \text{Correctness})] \\
\text{violation}(\text{Pilot3}, (\text{model}_\text{pilot}_\text{salary}, \text{Range})) \\
\ast \text{penalty}(\text{model}_\text{pilot}_\text{salary}, \text{Range}, \text{Correctness}) \\
\text{violation}(\text{Pilot3}, (\text{model}_\text{pilot}_\text{licensureexpired}, \text{Future})) \\
\ast \text{penalty}(\text{model}_\text{pilot}_\text{licensureexpired}, \text{Future}, \text{Correctness})] \\
\text{violation}(\text{Pilot1}(\text{Person}), (\text{model}_\text{person}_\text{birthdate}, \text{Past})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{birthdate}, \text{Past}, \text{Correctness}) \\
\text{violation}(\text{Pilot1}(\text{Person}), (\text{model}_\text{person}_\text{email}, \text{Email})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{email}, \text{Email}, \text{Correctness})] \\
\text{violation}(\text{Pilot2}(\text{Person}), (\text{model}_\text{person}_\text{birthdate}, \text{Past})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{birthdate}, \text{Past}, \text{Correctness}) \\
\text{violation}(\text{Pilot2}(\text{Person}), (\text{model}_\text{person}_\text{email}, \text{Email})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{email}, \text{Email}, \text{Correctness})] \\
\text{violation}(\text{Pilot3}(\text{Person}), (\text{model}_\text{person}_\text{birthdate}, \text{Past})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{birthdate}, \text{Past}, \text{Correctness}) \\
\text{violation}(\text{Pilot3}(\text{Person}), (\text{model}_\text{person}_\text{email}, \text{Email})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{email}, \text{Email}, \text{Correctness})] \\
\text{violation}(\text{Person1}, (\text{model}_\text{person}_\text{birthdate}, \text{Past})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{birthdate}, \text{Past}, \text{Correctness}) \\
\text{violation}(\text{Person1}, (\text{model}_\text{person}_\text{email}, \text{Email})) \\
\ast \text{penalty}(\text{model}_\text{person}_\text{email}, \text{Email}, \text{Correctness})] \\
\text{violation}(\text{Airport1}, (\text{model}_\text{airport}, \text{AvoidCongestion})) \\
\ast \text{penalty}(\text{model}_\text{airport}, \text{AvoidCongestion}, \text{Correctness}) \\
\text{violation}(\text{Airport1}, (\text{model}_\text{airport}_\text{arrivals}, \text{Size})) \\
\ast \text{penalty}(\text{model}_\text{airport}_\text{arrivals}, \text{Size}, \text{Correctness})] \\
\text{violation}(\text{Airport2}, (\text{model}_\text{airport}, \text{AvoidCongestion})) \\
\ast \text{penalty}(\text{model}_\text{airport}, \text{AvoidCongestion}, \text{Correctness}) \\
\text{violation}(\text{Airport2}, (\text{model}_\text{airport}_\text{arrivals}, \text{Size})) \\
\ast \text{penalty}(\text{model}_\text{airport}_\text{arrivals}, \text{Size}, \text{Correctness})] \\
= [1 \ast 0.2 + 0 \ast 0.0] + [0 \ast 0.2 + 1 \ast 0.0] + [0 \ast 0.2 + 0 \ast 0.0] \\
+[1 \ast 0.0 + 0 \ast 0.2] + [0 \ast 0.0 + 0 \ast 0.2] + [0 \ast 0.0 + 1 \ast 0.2] + [0 \ast 0.0 + 0 \ast 0.2] \\
+[1 \ast 0.5 + 0 \ast 0.2] + [0 \ast 0.5 + 1 \ast 0.2] \\
= 0.2 + 0.0 + 0.0 + 0.0 + 0.2 + 0.0 + 0.5 + 0.2 = 1.1
Taking into account the values for the nominator and denominator we can compute the final result.

\[
vdqd(Correctness) = 1 - \frac{\sum_{t \in T^*} \sum_{i \in I_t^*} \sum_{c \in C_t \text{violation}(i,c) \cdot \text{penalty}(c, Correctness)}}{\sum_{t \in T^*} (|I_t^*| \cdot \sum_{c \in C_t \text{penalty}(c, Correctness))}}
\]

\[
= 1 - \frac{1.1}{2.8} = 0.607 \approx 61\%
\]

The Correctness for the whole system results in about 61%.

5.9.3 Overall data quality

In subsections 5.9.1 and 5.9.2 we discussed the data quality feedback for a particular dimension. As we know all dimensions represent some aspects of the overall data quality. For the purposes of this master thesis we would like to provide also a single value that is representing the overall quality of the data, called also the average data quality. Therefore, we derived the concept of all dimensions data quality which takes into account violations of all constraints that are mapped to any of the data quality dimensions. Constraints that are not mapped to any of the data quality dimensions are not included in the output data quality feedback value.

In other words we eliminate the \(d\) parameter (a given data quality dimension) from all variations of \(vdqd(d, \ldots)\) function. The concept of violating a constraint that is mapped into any data quality dimension is expressed by the function \(\text{penalty}_{\text{ALL}}(c)\). This is shown in the following formula:

\[
vdqd(d, \ldots) \Rightarrow vdqd(\ldots)
\]

\[
\text{penalty}(c, d) \Rightarrow \text{penalty}_{\text{ALL}}(c)
\]

As a consequence we can use all the formulae from sections 5.9.1 and 5.9.2 for computing the data quality feedback through the application of the substitution from the formula presented above.

We cannot further generalize this concept by including also the constraints that are not mapped into any data quality dimensions, because for such constraints we have no penalty value. However, if we assume some default penalty then this concept leads to the soft constraint fulfillment proposed by Oliver Probst [69].

5.10 Feedback types

So far we have discussed how to map constraints to particular data quality dimensions and how to compute the numerical value of the achieved data quality. In this subsection we will cover the last part regarding the data quality feedback – the form in which the feedback is presented.

We decided to base our visualization tools on findings of Reto Mock [61] and Sebastian Hafen [35]. Finally, we have chosen a textual report (description) representation and two graphical ones: a bar plot and a spider web plot.
Report

Figure 5.12 presents a sample textual report. This form contains detailed numerical values about data quality dimensions with regards to the selected granularity level and result aggregation options. This form of feedback is suitable for particular data quality dimensions and for the overall data quality feedback.

```
--- DATA QUALITY FEEDBACK ---

----- MULTIPLE CLASSES
----- model.Airport
Consistency : 0.8333
Correctness : 0.6667
AVERAGE DQ : 0.7778

----- model.Flight
Appropriateness : 0.9777
Consistency : 0.6626
Correctness : 0.5802
Timeliness : 1.0000
AVERAGE DQ : 0.8074

----- model.Machine

----- model.Passenger

----- model.Person
Completeness : 1.0000
Correctness : 0.6111
Timeliness : 1.0000
AVERAGE DQ : 0.7941

----- model.Pilot
Completeness : 0.7333
Correctness : 0.8000
Timeliness : 0.8000
AVERAGE DQ : 0.7600

----- model.Plane
Completeness : 0.5556
Correctness : 0.6667
Timeliness : 0.6667
AVERAGE DQ : 0.6275

----- model.Plane
Accuracy : 0.5000
Consistency : 1.0000
Correctness : 0.5000
AVERAGE DQ : 0.5786
```

---

Figure 5.12: Textual report – data quality feedback

Spider web plot

The spider web plot, presented in Figure 5.13, illustrates differences in the data quality either for different classes/instances or different data quality dimensions (depending on the aggregation mode). Every axis is scaled from 0.0 to 1.0. The further the point lies from the origin the better the data quality is. This plot allows also for the comparison of various data quality dimensions for many classes/instances by drawing them all in a single plot. It is not suitable
for the *overall data quality* as it should reflect similar features, while the overall data quality is an aggregated feature of all data quality dimensions. Presenting just one value in the spider web plot does not bring any cognitive values as for the numerical presentation every other kind of the representation seems to be more reasonable.

Figure 5.13: Spider web plot – data quality feedback

**Bar plot**

Figure 5.14 illustrates a sample bar plot. The function of the bar plot is very similar to the textual report – they show detailed numerical values, but the graphical way makes the perception of differences between values much easier. It is suitable for both overall data quality feedback and values of particular data quality dimensions.

5.11 System overview

Figure 5.15 presents the overview of the architecture and components of the data quality feedback visualizer.

*Data Quality Visualizer* is the central component of the data quality feedback system which computes the values of data quality dimensions for the given, uniquely identified, instances of the real data. It matches constraint violations with constraints to data quality dimensions mappings (taking into account constraints and types hierarchy).

The input of the whole system are a *data definition* and *data instances (real data)*. The data definition is used to generate a stub data quality dimensions mapping file described
in the Subsection 5.5. Such a stub contains only two of three components: a *constraint location* and a *constraint name* collected from the input data definition. The data definition could be represented as particular technology source files, e.g. Java classes. The *UnifiedOCL* specification may also serve as the data definition. Moreover, the mapping file stub could be also retrieved directly from the running technology specific system, e.g. RDBMS.

Having the stub, a user should fill in each line with the third component – the data quality dimension mapping (a dimension name and the penalty value) mentioned in subsections 5.4 and 5.5.

The real data (objects, entities, etc.) is the second input argument to our system. It is used for three purposes:

1. It indicates all the objects that the visualization should present. Therefore, every instance of the real data should be uniquely *identified*. Such an identification is needed to represent all objects in *(Data Quality Visualizer)*. The benefit of using identifiers instead of objects is straightforward – a memory efficiency (an instance identifier could be a `toString` method or hash function instead of passing the whole complex object).

2. In order to compute the values of particular data quality dimensions the system should know the hierarchy of constraints, which is determined by the hierarchy of types (discussed in Subsection 5.8). This hierarchy can be obtained either directly from the data, e.g. Java objects or, similarly as in the case with the data definition, from the running technology specific system, e.g. an RDBMS.

3. Finally, the real data is validated by some external frameworks, e.g. Java Bean Validation or an RDBMS. The output of the validation – constraint violations – is the last input parameter for our data quality feedback visualizer. Therefore, also in this step the real data is an indirect input to our system.
Figure 5.15: Data Quality Feedback Visualizer – system overview
All three steps of processing real data described above are external from the point of view of the system central element (Data Quality Visualizer). It is due to the fact that we wanted to keep our system technology independent. All the steps cannot be implemented in a technology independent way as they need to process technology specific instances. Therefore, to separate a technology independent tool from a technology specific input we decided not to include the real data processing into our visualizer. We assumed that the output of the real data processing steps are well structured using JSON files (the schema of those files is imposed by the visualizer – standardized interfaces). It ensures a technology and validation framework independence, enforcing at the same time a well defined interface usage.
6

Implementation

In this chapter we are presenting the technical aspects of UnifiedOCL, our unified data quality requirements representation along with the transformations between various constraint representations. Moreover, this thesis provides also an overview of widely used technologies available at the market and frameworks for developing DSLs and inter model transformations.

6.1 Overview

Our system supports three technology specific representations:

- **Java**

- **SQL** – we decided to use PostgreSQL 9.4\(^1\). This is one of the most powerful open-source relational database management systems (RDBMS).

- **Business rules** – after comparison of Drools\(^2\) with JRules\(^3\)\(^4\) we’ve finally chosen the former one (version 6.2) as the open-source product with a large community of users.

The main parts of our system are implemented using the Java programming language. We used the latest available version of Java – i.e. Java SE 1.8 Update 31 (at the time of beginning of this thesis).

---

\(^1\)http://www.postgresql.org/ [Online, accessed 19-August-2015]
6.2 Domain Specific Language frameworks

The central element of our system is *UnifiedOCL* – our domain specific language for denoting unified data quality requirements. As a consequence, the decision which DSL modeling framework should be chosen would be the most influential one. The aftermaths of this decision would have an impact on almost all aspects of our system, therefore this decision should be made with prudence. Kurtev et al. [49] compares various DSL modeling frameworks. Unfortunately, at the time of writing this thesis some technologies discussed there are out of date. Another comparison can be found in [65], but technologies described there are more academic projects rather than real industrial standards. Regarding the implementation of our DSL we decided to rely on industrial standards rather than purely academic studies, to ensure the highest possible quality of the product. One of the main goals of developing DSLs is to deliver also high quality editors supporting their concrete syntaxes in order to make their usage as convenient and efficient as possible. We took into account the following frameworks:

**Textual Concrete Syntax (TCS)** ⁵, described in [40], is an Eclipse project closely related to ATL Transformation Language (ATL)⁶, the model to model transformation framework (they both use the Kernel Meta Meta Model (KM3)⁷ – an abstract syntax representation). TCS attaches syntactic information to a metamodel, which can then be used to generate an annotated grammar [26]. It also provides an Eclipse based editor for DSLs that is capable of syntax coloring, providing an outline view, etc. However, after some online investigation, this project turned out to be a bit out-of-date (in comparison with Xtext) and not widely used.

**EMFText** ⁸ is an Eclipse framework for developing textual syntax for Ecore based metamodels. It also supplies an Eclipse based editor for a DSL, equipped with features like auto-completion or quick fixes. It separates the creation of an abstract syntax from the concrete syntax. It is capable of generating default syntaxes (HUTN or Java-like). Moreover, there exist a JaMoPP⁹ project (Java ModelParser and Printer) that is based on EMFText through which the Java code can be also processed in a similar way as any DSL.

**Xtext** ¹⁰ seems to be the most mature of all the Eclipse based DSL frameworks. It can work either with existing Ecore metamodels (abstract syntaxes) or can generate Ecore metamodels from the concrete syntaxes. It is based on the ANTLR parser and designed to work with LL(∗) class grammars. It also generates a powerful Eclipse editor. Moreover, it provides a seamless integration with model to text transformation tools like Xpand and Xtend. Except this, it supports a Java-like expressions, which can be embedded into DSLs (XBase¹¹). Finally, it seems to be the most widely used tool of its kind and has a large community of users.

---

Modeling SDK for Visual Studio (MSDK) is a Microsoft framework supporting the DSL creation. It is mostly focused on graphical DSL representation, that could be integrated into Microsoft (MS) Visual Studio. It provides a rich editor, supporting features like tree explorer, serialization into XML, etc. It can be used with the Text Template Transformation Toolkit (T4) Text Templates for the code generation.

Due to the fact that MSDK is graphical based and our UnifiedOCL has a textual representation, we decided to go for one of the Eclipse based frameworks. Taking into account the comparison done by Christopher Guntli [34] and bearing in mind the fact that Xtext is a massively growing in popularity framework we decided to finally choose the Xtext. Xtext seems appealing also due the fact that there are many project using it and many plugins extending its core functionalities. The growing popularity of Xtext is also driven by the annual conference on it – Xtextcon.

6.3 Eclipse platform

The DSL language implemented using Xtext is delivered as a set of Eclipse plugins, which can be easily installed in the Eclipse platform, extending its core features with new capabilities. As a result the UnifiedOCL is delivered as a set of such plugins. Therefore, we decided to make an additional step forward and implement also all the transformations as Eclipse plugins. It is justified by the fact that as we stated earlier all the inputs and outputs in our system are text (code) files. Usually, source code files are developed using IDEs. Eclipse is an example of such IDE, so there is a high probability that the users of our system would also use Eclipse to work with source code files (e.g. Java or SQL). Thus, the integration of all types of sources would be appreciated as a great simplification in the development process. Of course, the usage of Eclipse for the technology specific sources is not a requirement, and Eclipse plugins can be used just to generate/convert technology specific files, which are then further processed in other IDEs. Another benefit of choosing the Eclipse platform is the vastness of frameworks and toolkits available as plugins or libraries for the Eclipse platform.

Used Eclipse version: Luna Service Release 1a (4.4.1) with preinstalled Eclipse Modeling Tools

6.4 Eclipse plugin

The Eclipse platform is designed to be extendable by means of plugins. An Eclipse plugin is a bundle (component) running on the Open Service Gateway Initiative (OSGi) component system. A bundle is a Java Archive (JAR) archive that is equipped with a special MANIFEST file, containing among others, the plugin version and dependencies. This file is interpreted by the OSGi system. The Eclipse platform is based on one of the implementations of the

---

OSGi standard – OSGi Equinox\(^\text{16}\).

OSGi reduces the complexity of the software by encapsulating internal implementation details by enclosing them within components – bundles that communicate using clearly specified services. The whole life cycle of the component is managed dynamically, so e.g. the bundles can be installed and started/stopped at runtime. This increases also the performance of the application – bundles are loaded only when needed (lazy initialization).

Our system consists of over a dozen of plugins described in Section 6.19. Benefits of such an architecture are as follows:

• self-containment – every component encapsulates a part of the logic that represents a coherent subsystem. For instance the responsibility of the component could be a support for a single technology specific representation or a particular transformation

• extensibility – the system can be easily extended without a necessity of changing the other plugins

• customization – by selecting plugins it can be chosen what to install

• easy deployment – plugins can be grouped together and installed using the Eclipse Update Sites\(^\text{17}\)

6.5 EMF Ecore

The Eclipse Modeling Project (EMP)\(^\text{18}\) is a common name for a group of projects supporting model driven software development for the Eclipse platform. These include: model to model and model to text transformations and modeling of abstract and concrete syntaxes (including both textual and graphical representations) for DSLs\(^\text{[26]}\). Eclipse Modeling Framework (EMF)\(^\text{19}\) is the central component providing the basis for the actual building of models and code generation. One of the goals of this framework was to unify Java, XML and UML as stated in \([77]\). This project is extensively used by many other model driven development projects for the Eclipse platform. It comes with the Ecore metamodel, which is used to describe a DSL abstract syntax (models in EMF), and could be serialized using the XMI format. To avoid confusion we consider Ecore as model or metamodel depending on the context – whether we are bearing in mind a particular concrete model denoted using Ecore, or whether we mean Ecore concepts (Ecore is an EMF model, therefore it is a metamodel for itself). The Ecore metamodel was designed to describe object-oriented domains. Moreover, the EMF framework can generate Java code from the Ecore models. For the purposes of this thesis we are highly relying on the Ecore metamodel concepts, therefore to get acquainted with the Ecore details we recommend to refer to \([77]\).


CHAPTER 6. IMPLEMENTATION

6.6 Technology overview

In this section we are providing an overview of the available model driven development technologies compatible with the Eclipse platform, that we previously selected.

6.6.1 Template languages

**Acceleo**[^1][63] is a template language for code generation. It is an implementation of the OMG MOF Model To Text Transformation Language (MOFM2T)[^2] standard. As a part of EMP it is capable of processing Ecore based models like UML2. It is supplied with a dedicated Eclipse based editor that provides a very convenient code auto-completion which makes it very easy to navigate over very complex models. Through reflection it can call methods from the external Java libraries.

**Xtend**[^22] is a very powerful general purpose language advertised as Java 10 by its creators. Among others it supports lambda expressions, multiple dispatch, extension methods, active annotations and map/reduce constructs. Moreover, it comes with a very expressive *switch expression* that takes into account polymorphic types of objects. Xtend source code is compiled into Java 5 source code, therefore it allows for a seamless integration with existing Java libraries. One of the greatest advantages of Xtend is its code generation capabilities. It not only allows for mixing template expressions with static text but also allows for *pretty-printing* of the output generated code. It is seamlessly integrated with Xtext and they together provide a complete suit for creating DSLs and processing them in order to generate a target platform code [7].

In our system we are using both above mentioned template languages depending on from which we can benefit more in a particular transformation.

6.6.2 Model to model transformation

**ATL Transformation Language (ATL)**[^23][39] is a model transformation toolkit. It is capable of generating various target models from a set of source models. It uses internally EMF and it is integrated with the Eclipse platform. ATL consists of a textual editor (Eclipse plugin) and a language to define model to model transformations. This language matches concepts from one model to the corresponding ones in the other model. It can work with metamodels conforming to Ecore and KM3[^24]. Unfortunately, the ATL language cannot seamlessly support external Java (or any other common general purpose language) code/libraries, what significantly limits it adequacy in many applications. ATL is one of the attempts of the implementation of the OMG QVT standard[^25].

QVT Operational \textsuperscript{26}, described in [16], is a EMP component responsible for model to model transformations. It partially implements one of the family of QVT languages defined in the MOF 2.0 QVT standard – the imperative mapping language (Operational Mapping Language). This is a language that provides a way how to map (step by step) concepts from the source model into the target model. It supports models represented in Ecore. A detailed description of that language can be found in [26]. At the time of writing this thesis, this language seemed to be the most mature from the whole Eclipse QVT family.

QVT Declarative \textsuperscript{27} is a partial implementation of the declarative approach of model to model transformations defined in the OMG standard – the languages QVT Core and QVT Relational. At the time of writing this thesis this project is still under development.

The Eclipse QVT languages family is currently a standard of QVT model transformation within the MDA paradigm. Kurtev [47] discusses in details the QVT in the MDA context providing details about various types of QVT languages, their requirements and supporting tools. It consists of many languages corresponding to various approaches of model to model transformations. However, the Eclipse implementation of the QVT family is still very limited and this is also why we decided to go for the model to text transformation instead of model to model ones.

However, before the decision about selecting the model to text type of transformation had been taken, we had implemented one transformation using the model to model approach to investigate its usability in practice. We have chosen the QVT Operational framework due to the fact that it is compatible with the OMG standard, it is well documented ([26]) and it is imperative. We found that the ATL declarative approach, without the seamless possibility to refer to the code written in general purposes languages, complicates implementation of transformation between significantly differing models. A theoretical comparison between ATL and QVT is presented by Jouault and Kurtev in [41].

6.6.3 Other used frameworks

This section gives an overview of the other Eclipse platform based technologies/frameworks that are used by our system.

MoDisco \textsuperscript{28}, described in [9], as its name suggests, is a reverse engineering tool that is capable of model discovery. It analyzes a given system (one of the goal of the architects of this framework was to discover legacy systems) and builds a model exposing its internal architecture, components and structures of elements. Not only does it help to understand the existing system, but also helps to transform it into different representations, facilitating many MDA software development processes. The default installation

\begin{itemize}
\item \textsuperscript{26}https://projects.eclipse.org/projects/modeling.mmt.qvt-oml [Online, accessed 19-August-2015]
\item \textsuperscript{27}http://projects.eclipse.org/projects/modeling.mmt.qvtd [Online, accessed 19-August-2015]
\item \textsuperscript{28}https://eclipse.org/MoDisco/ [Online, accessed 19-August-2015]
\end{itemize}
of MoDisco contains the Java project definition, allowing for discovery and analysis of entire Java packages.

**Eclipse implementation of UML (UML2)**[^29] is an implementation of UML 2.x for the Eclipse platform. It is based on EMF, therefore, it can be seamlessly integrated with other model driven software development tools for the Eclipse platform. UML2 files are stored in the XMI format, which simplifies interchanging of semantic models[^30] between various CASE tools. As mentioned in Section 4.2 it is supported either natively or by a set of plugins by many UML tools existing on the market.

**Papyrus**[^31], described in [51], is a graphical editor for the EMF Ecore based models, delivered as an Eclipse plugin. Therefore, it is especially suitable for creating UML models. It natively supports the Eclipse implementation of UML – UML2. Besides this it can support other modeling languages like **MARTE** or **SysML**. As mentioned in Section 4.2 it tends to fully support the OMG specification of UML.

**Eclipse OCL**[^32] is an Eclipse implementation of OCL, designed to cooperate with EMF based models. It allows for both specification of OCL constraints and validating existing models (Ecore or UML) against them. This Eclipse plugin is accompanied with an interactive console for evaluating OCL expressions. The Eclipse implementation of OCL consists of four languages[^33]:

1. **EssentialOCL** – an implementation of the core OCL expressions, but without any reference to the model (data structure)
2. **CompleteOCL** – a language that binds together OCL statements with the context where they are used (a model). The syntax conforms to the OMG OCL specification.
3. **OCLinEcore** – a language that allows to embed the OCL statements within the definition of a data structure. It mixes the Ecore textual syntax with OCL constraints.
4. **OCLstdlib** – the OCL standard library. It contains data types and operations defined by the OMG standard together with Eclipse extensions, e.g. string pattern matching.

**DresdenOCL**[^34], described in [12] and [14], is a very comprehensive framework supporting OCL constraints. It provides an editor to specify constraints denoted using OCL for various types of models, e.g. EMF, Java and UML. Instances of those models can then be evaluated against constraints. One of the greatest strengths of DresdenOCL is its ability to generate Java/AspectJ[^35] and the SQL code from the OCL expressions. DresdenOCL is provided as an Eclipse plugin or can be used as a standalone library.

6.6.4 Other considered frameworks

This section presents other technologies that we considered but finally decided not to use.

**Epsilon** [36], described in [46], is a framework supporting various aspects of model driven software development. It consists of several languages used for code generation, model-to-model transformations, model validation, model migration and refactoring. It can cooperate with EMF based models and also with other types of models via the *Epsilon Model Connectivity*. We decided not to use Epsilon due to the fact that all the components of that framework work nicely together, constituting a powerful framework, though when we just want to pick one part of the Epsilon environment, we believe we can always find a better alternative.

**The Spoofax Language Workbench** [37], described in [43], is a framework for developing DSLs. It is provided together with *Stratego* [80][81] – a language for a program transformation. Even though Spoofax seems to be a mature and complete product it is not as widely used as its competitors, e.g. Xtext. Thus, we decided not to use it.

**OCL Compiler for EMF** proposed by Garcia and Shidqie in [22] is a tool for OCL transformation into Java code. Regrettably, it is a research project that is not well known, therefore, we decided to choose DresdenOCL as the more mature product.

6.7 UnfiedOCL

As we mentioned earlier our *UnifiedOCL* is implemented as a DSL language using Xtext. It is based on the *OCLinEcore* language, which is a part of the *Eclipse OCL* project. *OCLinEcore* allows to combine an Ecore model with OCL statements. The *OCLinEcore* grammar extends the *EssentialOCL* grammar, which allows for specifying OCL expressions, but has no relation to the Ecore (or any data) model. The *OCLstdlib* is used to provide standard OCL types and operations, but also some exclusive Eclipse extensions (e.g. string pattern matching).

In order to implement our *UnifiedOCL* we decided to use the same approach – to extend the *EssentialOCL* language. Moreover, we derived *UnifiedOCL* by reusing many of the components of the *OCLinEcore* language and the Ecore metamodel, adjusting them and introducing new ones. We modified the *OCLinEcore* [38] in three ways:

1. We changed the metamodel by extending Ecore concepts with attributes or relationships required to reflect the structure of a UML class diagram, e.g.
   - we introduced the grammar element (UniOCLIVisibilitable) to reflect the concept of encapsulation levels – modifiers public, private, protected, package

---

• we introduced simple exceptions (changed existing way of handling exceptions)
• we introduced a direction of the parameter
• we introduced user defined primitive types and the Date built-in primitive type
• we introduced the grammar element (UniOCLConcurrency) to allow for specifying whether the operation could be multithreaded

2. Further changes are related to the constraint representation. The UnifiedOCL metamodel, besides OCL, supports also our UDQAM constraint representation. Therefore, we created the super class UniOCLLabelable which is a base class for all concepts that can define UDQAM labels (currently only references and attributes).

3. Finally, we extended the metamodel with a way to express constraints to data quality dimensions mappings as described in Section 5.6

An example of the UnifiedOCL syntax can be found in Appendix E.2.
We chose .uniocl as the extension for the files containing UnifiedOCL specifications.

6.8 UDQAM dictionaries

As mentioned in Section 3.7 UDQAM labels are grouped into dictionaries and plugged into the UnifiedOCL grammar. Such a dictionary is a JSON file with the extension .udqam. The structure of such a JSON file consists of a map, where the key is the constraint (label) name and the value is another map containing constraint parameters (or null if there are no parameters). The map of parameters consists of keys which are the parameter names and parameter value types which are represented by one of the strings from the following set: STRING, INTEGER, REAL, BOOLEAN, IDENTIFIER, ARRAY, COLLECTION, MAP, ANY. Below we are presenting a sample UDQAM dictionary (JSON file content) containing constraints for SQL:

```
{
  "primarykey": null,
  "foreignkey": {
    "column": "IDENTIFIER",
    "table": "IDENTIFIER"
  },
  "autoincrement": null,
  "unique": null,
  "default": {
    "value": "IDENTIFIER"
  }
}
```

The location of UDQAM dictionaries can be configured as other UnifiedOCL or transformation parameters using the Eclipse preference configuration (discussed in Section 7.3).
6.9 Supported constraints

We decided to provide a support for the following constraint types:

- Java Bean Validation standard – the list of supported constraints is presented in Table B.1 in Appendix B
- Hibernate Validator – the list of supported constraints is presented in Table B.2 in Appendix B
- PostgreSQL – the list of supported constraints is presented in Table B.3 in Appendix B

Drools constraints (rule conditions) are defined in an imperative way, there is no compact representation, and a user has to define explicitly constraint expressions. As a result we did not extended our set of supported constraints for specific Drools constraints. However, due to the fact that in Drools conditions are very complex, we are supporting some expressions, that may appear in Drools conditions and that may constitute constraints, e.g. the collection `containsAll` method or the object `equals` method.

For constraints defined in the technologies mentioned above we created corresponding UDQAM labels or constructs in UnifiedOCL as presented in the second column of all three tables from the appendix that are mentioned above. Some UDQAM labels are used for several constraints allowed in various technologies. Moreover, in some cases we defined UDQAM labels to be more powerful than corresponding concepts for instance in the Java Bean Validation standard, e.g. the `min` UDQAM label, which allows for specification whether the extreme values from the ranges should be included, is more powerful than the @Min annotation.

For most of the constraints from these three technologies, we defined also corresponding invariants in OCL where it was possible. As a result, a user may choose whether they prefer OCL over UDQAM or vice versa during code generation.

Moreover, we provided support for default values of fields defined in Java and columns defined in SQL.

Finally, we are also supporting constraints that involve more than just one structural feature from UnifiedOCL. These constraints are defined in OCL – complex constraints – and are transformed into a Java Bean custom (user defined) annotation, whose validator takes more than just one field of the Java class for evaluation, or into the SQL table level CHECK constraints. For this purpose we integrated DresdenOCL in our system, that provides required features. These types of constraints are only supported in transformations from UnifiedOCL to Java, SQL or Drools.
6.10 Data quality mapping

The constraints to data quality mapping notation as discussed in Section 5.5 is also implemented as Xtext DSL. The grammar is available in Appendix A.2. Moreover, our system provides a plugin to generate mapping file stubs for a given specific technology (e.g. in our system we implemented it for Java) or for UnifiedOCL sources. The implementations of those transformations are parts of the plugins responsible for a concrete representations. Moreover, we are providing a library methods that can connect to the PostgreSQL RDBMS and retrieve defined constraints for a given set of table names (for SQL). It is due to the fact that PostgreSQL uses it’s own constraint names, if they are not explicitly defined by a user. The file extension for data quality mappings is .dqm.

6.11 UML to UnifiedOCL transformation

In our system we assume that the UML2 Eclipse format is the standard way how UML diagrams are serialized. We also decided to use Papyrus as the default UML editor because it is also available as an Eclipse plugin. The transformation from UML2 into UnifiedOCL is implemented using Acceleo as this template language particularly facilitates the navigation over the XMI files, which is the format of serialized UML2 files (.uml).

6.12 Technology specific to- and from- UnifiedOCL transformations

All those transformations are much more complex than the transformation from UML into UnifiedOCL. Therefore, the key factor in the choice of the appropriate technology was the expressiveness of the language rather than the easy navigation through the model. As a consequence, all those transformations are implemented using Xtend. Details are explained in sections 4.6 and 6.17.

6.13 UnifiedOCL to pure OCL transformation

This transformation was implemented using QVTo. As mentioned in Subsection 6.6.2 we wanted to check the usability of a model to model approach in our system and this is why we decided to implement this transformation using the QVT language. This transformation is fairly simple (just an extraction of OCL constraints and wrapping them into contexts, which can be easily obtained from the UnifiedOCL abstract syntax tree), but the amount of code needed to implement this transformation confirmed that the model to text paradigm is better suited for our system.
6.14 UnifiedOCL (de)serialization to XMI

*UnifiedOCL* files (.uniocl) are serialized into XMI in two steps:

1. the desired source file is loaded using Xtext (parser for *UnifiedOCL* files)
2. the constructed model is an EMF model. Therefore, it can be straightforward serialized into XMI using the API provided by EMF.

The deserialization is the reversed process of the serialization.

It is important to notice that every *UnifiedOCL* file is serialized into two XMI files. It is justified by the fact that the internal implementation of *Eclipse OCL* uses a so called *pivot model* to bridge the gap between UML and Ecore. Therefore, we decided to use two extensions for the generated files: .xmi and .oclxmi. The former one is *UnifiedOCL* aligned, while the latter one is pivot model aligned.

6.15 Data Quality Visualizer

The *Data Quality Visualizer* is built as a Java application with the Swing\(^{39}\) toolkit used for the Graphical User Interface (GUI). We decided to use the Model View Presenter (MVP) [66] approach. In that way we can have a model and a view that are completely independent. Also the code processing the data is not mixed with the code responsible for the presentation/visualization. As a result we can easily replace the existing visualizer with another one without any alteration of the model. In the same way we can expand the model without the necessity to change the presentation layer. The common part that binds the model and the view is a presenter. As presented in Figure 6.1, the presenter is the element that mediates the communication between those two components. The classes represented with green rectangles are the main classes of the *Data Quality Visualizer* that implement the MVP design pattern. Moreover, the DQFeedbackWindow, the main class responsible for the visualization, implements the DQFeedbackView which further increases the modularity of our application by stating clearly which components of the presentation layer could be accessed by the presenter. The classes denoted using pink rectangles are the main components of the model. Those are: constraints to data quality dimension mappings (DQMapping), as defined in the .dqm mapping file and constraint violations (ConstraintViolationDescriptions). The latter ones are used by the DQComputer to obtain the values of various data quality dimensions for the given set of entities. The detailed explanation of the responsibilities of particular classes is provided in Section 6.19.

Additionally, our *Data Quality Visualizer* provides the graph showing the hierarchies between classifiers: classes or tables is implemented using jgraph v5.13\(^{40}\) (the legacy version) and jgraphx\(^{41}\).

---

\(^{39}\)http://docs.oracle.com/javase/7/docs/technotes/guides/swing/ [Online, accessed 19-August-2015]
Figure 6.1: Data Quality Visualizer – MVP architecture – UML class diagram
The Spider Web plots and Bar plots are created using the JFreeChart\textsuperscript{42} library. Moreover, we used the Jaspersoft\textsuperscript{43} implementation of the check box lists\textsuperscript{44}.

The Data Quality Visualizer can be used in three different ways:

1. as a standalone application – delivered as a JAR archive – dqvisualizer.jar
2. can be run from the Eclipse platform as a part of the plugin
3. can be called programatically using the provided API, contained in the same JAR archive as in the first point. Classes responsible for input data preparation for the Data Quality Visualizer are DQUtil and PostgresUtil. The sample project delivered with our system (Demo_DQVisualizer) shows how to run the Data Quality Visualizer. The detailed API documentation (javadoc) is also delivered together with this thesis.

In the first two cases the Data Quality Visualizer inputs (described in Section 5.11) are provided as JSON files (described in the appendix) whilst in the third case as Java objects (maps).

6.16 Used technology overview

In Figure 6.2 we present the architecture of our system with particular emphasis on the used technologies. This is Figure 4.1 enriched with used technologies and libraries. Technologies presented on the left side of the thick black stroke are used in various parts of the system by many plugins. The sources of images and trademark policies of the used technologies are mentioned in the footnote \textsuperscript{45}.

\textsuperscript{42}http://www.jfree.org/jfreechart/ [Online, accessed 19-August-2015]
\textsuperscript{43}https://www.jaspersoft.com/ [Online, accessed 19-August-2015]
\textsuperscript{44}http://opensourcejavaphp.net/java/report/com/jaspersoft/ireport/designer/fonts/CheckBoxList.java.html [Online, accessed 19-August-2015]

\textsuperscript{45}Sources of graphics (all below – Online; accessed 20-August-2015):
EMF - https://eclipse.org/modeling/emf/images/emf_logo.png
Papyrus - https://projects.eclipse.org/sites/default/files/Papyrus.gif
Eclipse OCL - https://projects.eclipse.org/sites/default/files/ocl.png
QVTo - https://projects.eclipse.org/sites/default/files/QVTo.PNG
Xtext - https://wiki.eclipse.org/images/thumb/d/db/Xtext_logo.png/450px-Xtext_logo.png
Xtend - http://www.eclipse.org/xtend/images/favicon.png
Postgresql - http://www.postgresql.org/; https://wiki.postgresql.org/images/3/30/PostgreSQL_logo.3colors.120x120.png
Java - http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html (provided in Java SE Development Kit 8u60 Documentation)
Figure 6.2: Used technologies
6.17 Transformation architecture

In this section we are presenting the architecture of transformations in both directions between UnifiedOCL and concrete technologies. The flow of operations is similar for all the plugins implemented by us (Java, SQL, Drools). Here, we are presenting the general overview, which should be adjusted for particular plugins, and simplified when possible. In this section we are elaborating concepts described in Section 4.6 taking into account implementation factors.

Figure 6.3 presents the overview of the whole transformation architecture. On the left we can see the green arrow callouts presenting the flow of the subsequent operations in the process of a source to the target representation transformation. The generator is the core element of the transformation implementation. The red rectangles on the right denote some mechanisms used by the generator which are not internal parts of it. The orange rectangles on the right are parts of the generator that are not delegated into special, external classes/components – they are just internal parts of the generator. The gray rectangle denotes the third party tools used to transform some types of constraints, e.g. DresdenOCL or the tool for SQL to OCL constraint transformation proposed by [10]. This tool is available online and is not integrated with our system.

As we already mentioned the core element of the transformation implementation is a special class – the generator. This class manages all the other components of the transformation implementation and is responsible for the general way how the transformation is executed. Moreover, it contains the code for the data structure transformation.

The generator is responsible for all the steps described below except the building of the in-memory model of the source file that should be transformed. This task is executed before the actual generation process. E.g. for Java to UnifiedOCL transformation the...
Figure 6.3: Transformation architecture – overview
Java2UnifiedOCLGenerator is the generator class.

### 6.17.1 Source files

The first step to transform a source representation into a target representation is to build an in-memory model of the given source code file. In other words, we need to obtain an abstract syntax tree or any other traversable model corresponding to the given sources. In our system we have three options to do this:

1. use the existing parser – we used this approach for the Drools plugin. Drools is supplied together with a parser (org.drools.compiler.compiler.DrlParser) that builds an abstract syntax tree from a given input .drl file.

2. take the existing grammar, transform it to the Xtext format and generate the parser – we used this approach in the SQL plugin. We adjusted the existing SQL grammar to meet the PostgreSQL syntax and denoted it using the Xtext grammar language. We had to change also the SQL expressions to remove grammar disambiguities. Details would be discussed in Section 6.19 under the ch.ethz.unifiedocl.sql plugin. The benefits of using Xtext is its ability to generate parsers automatically. In that way, we can obtain an access to the abstract syntax tree for the SQL language.

3. use reverse engineering tools – in the Java plugin we used MoDisco to discover the structure of a Java project. The benefit of using this tools is its ability to automatically merge all the Java sources from the given package and build a single traversable model. The discovered model contains also information about the used annotations therefore we can easily extract information about Bean Validation constraints.

### 6.17.2 Traversing & analyzing – input & intermediary models

The second step of the transformation is to process the obtained technology specific model instance. We are traversing this model either to build an intermediary model used to generate the target representation or directly converting it into the desired output representation. An example of the intermediary model is given in the SQL to UnifiedOCL transformation. Constraints may be defined at the column level, along with the column specification (at the table level) or as external statements outside of the table definition. In order to get the information about existing constraints we need to first search through the whole SQL script given as an input, collect all the constraints and then produce the output. This order of operations is very important because when we are generating the target representation we are basically creating a string with a textual representation. Rewinding back such a string is very unnatural and leads to various implementation problems. Another example of intermediary models are discovered linking tables (many to many relations). For such tables there shouldn’t be generated any individual class but just references in other existing classes instead. To discover linking tables the whole SQL database script has to be parsed first. Then we are constructing the intermediary model using various heuristics, e.g. if all column in the table are foreign keys then it indicates that the given table is a linking table. Such an intermediary model is

used later when we are generating the entities corresponding to non linking tables. However, if there is no impedance mismatch between the source and target representations then the target model can be produced at the same time as traversing the source model (such an approach is called a single pass).

To store the parts of the intermediary model we are very often using various maps or lists, e.g. `TABLE_2_PRIMARY_KEY` map from the SQL2UnifiedOCLGenerator. Intermediary models may be also required to manage the dependencies between components of the input model.

### 6.17.3 Matching constraints

A developer implementing the concrete technology specific plugin knows where the constraint in the target representation may occur. Therefore, while traversing the source model we are paying particular attention to all known constraint locations. If a constraint exists in such a location then it has to be further analyzed. For this task we are using the concept of matchers. The goal of a matcher is to create an instance of the `ConstraintTransformationValue` class for a given constraint occurrence. The `ConstraintTransformationValue` class contains all the information about the constraint, which are stored in the technology independent way. This class is afterwards used to produce the target textual representation of the constraint. Information stored by `ConstraintTransformationValue` include:

- a constraint location – expressed by the name of the constrained entity/method and the name of the container containing this entity
- a name of the constraint that may be used by a constraint dictionary, e.g. an *UDQAM* label
- constraint parameter names and values
- a type of the entity (a data type of the constraint) – used by dictionaries when the constraint representation is different depending on the constrained data type. The type is defined using the `UnifiedOCLKeywordType` enumeration.

We can also consider instances of the `ConstraintTransformationValue` class as an intermediary model – abstract representations of various constraints. There are two ways of building `ConstraintTransformationValue` data structures:

1. they can be obtained pretty straightforward when constraint occurrences (or more general some types of constraints in a particular technology) are denoted in a compact way. In such a case we can build the object `ConstraintTransformationValue` directly using its constructor. For instance this approach is used when we are transforming Java into *UnifiedOCL*. We assumed that in most cases Bean Validation annotation names correspond to our abstract constraint names. Exceptions, like `@CreditCardNumber` – which is mapped into `creditcard`, have to be handled manually.
2. they can be also obtained using matchers. One of the overloaded versions or modification of the method `getConstraintsAndValues` is called for a given object wrapping the occurrence of the constraint in a source model and returns a set of `ConstraintTransformationValues`. A single constraint in one technology specific representation may be transformed into many constraints in the other technology specific representation. The method `getConstraintsAndValues` uses the concept of the constraint `dictionary`. E.g. for the matcher `SQL2UnifiedOCLMatcher` the `SQL2UnifiedOCLDictionary` is used. The method `getConstraintsAndValues` tries to match the textual representation of the constraint existing in the source model against all the entries in the dictionary. If a match is positive, then using the information from the `ConstraintDefinition`, that accompanies a regular expression in the dictionary, the source textual representation of a constraint is parsed, processed and the `ConstraintTransformationValue` is produced.

### 6.17.4 Regular expressions

The best way to match a constraint from the source model with the recognized set of constraints, for which we know how to generate the target representation, would be to compare their semantic meanings. This of course is not a trivial task, especially taking into account the extensibility of the system, it would probably require the usage of sophisticated machine learning/artificial intelligence methods.

On the other hand we can easily analyze the meaning of the constraint when it has a simple structure. This is given for example for Java Bean Validation. In such a case the only required action is to match the annotation name with the list of recognized constraints (dictionary) and, knowing the structure of this annotation, process its parameters. E.g. `@Length(min=2, max=4)` can be matched with an abstract constraint from the dictionary – the `size` constraint. It would also have well defined parameters, e.g. `lowerBound` and `upperBound`. Such a translation is not a difficult task. Another real life example are SQL `UNIQUE` constraints which can also be easily translated into technology independent (abstract) representations.

However, if we consider the Drools representation or OCL, where there are no compact ways to represent constraints, the process is getting more convoluted. We need to analyze the textual content of a Drools `conditions` or composite OCL invariant expressions. As we discussed earlier it is not possible to find the semantic meaning of such a constraint, so we decided to use regular expressions to match the known constraints. For this reason we prepared a set of dictionaries containing such regular expressions. Every constraint is matched against all the regular expressions from the dictionary in order to recognize its type. The presented approach is of course not an ideal solution. One may write constraints in various ways, e.g. a constraint limiting the number of elements of the collection of `students` may be specified as `students.size() > 5` or `5 < students.size()`. To overcome this difficulty we made our regular expressions very flexible and we defined several regular expressions matching a single abstract constraint.
6.17.5 Building constraints

The structure `ConstraintTransformationValue` containing the abstract representation of a constraint is not sufficient to build the target constraint representation. We may consider this structure as a *parametrization* that needs to be matched with a corresponding constraint definition for the particular target representation. In other words an instance of the `ConstraintTransformationValue` class has to be matched with the instance of a `ConstraintTransformationDescription` class. The latter one contains the information how to produce a target constraint representation i.e.:

- a constraint name that should correspond with the constraint name from the `ConstraintTransformationValue` data structure
- a constraint type that is used by dictionaries when the constraint representation is different depending on the entity type it constraints. The type is defined using the `UnifiedOCLKeywordType` enumeration.
- a map matching a parameter name with the textual output of the constraint in the target representation. The values coming from the parametrization would be placed in the locations pointed by place holders (numbers in curly brackets), e.g. the `UnifiedOCL` to Java transformation considers a size constraint with parameters `min` and `max` (if there would be just the `min` parameter or just the `max` parameter then a different `ConstraintTransformationValue` would be matched) would be translated as follows:
  
  \[
  \begin{align*}
    min & \quad \Rightarrow \quad @Size(min={2}, \max={2}) \quad \text{(if there would be just the `min` parameter or just the `max` parameter then a different `ConstraintTransformationValue` would be matched)} \\
  \end{align*}
  \]

- another map matching parameter name but this time with some value. This parameter is not printed (is not a part of the target constraint representation), but changes the way how the output is produced, e.g. the parameter `inclusive` with the value `true` instead of `false` changes the sign from `<` to `≤`.
- a processor handling some custom user defined operations required to produce the target representation. It is an optional element (could be `null`). A processor very often uses values from the map mentioned in the previous point (parameters changing the way of how a constraint is generated).

`ConstraintTransformationDescriptions` for a particular plugin are also stored in *dictionaries*. For instance `UnifiedOCL2JavaDictionary` is the dictionary containing those structures for `UnifiedOCL` to Java transformation. All constraint dictionaries are extending (directly or indirectly) the `ConstraintsBuilder` class. It (and possibly also the classes lying in the hierarchy between the `ConstraintsBuilder` class and the dictionary class) contains various versions of the `buildConstraint` method. This method for a given constraint parametrization (an instance of the `ConstraintTransformationValue`) finds the best matching `ConstraintTransformationDescription` taking into account constraint parameters, type and name. Subsequently, it uses either the standard procedure to built the target
representation of constraints by merging information from the constraint transformation description and parametrization (instance of ConstraintTransformationValue and ConstraintTransformationDescription) or uses a special processor that could be provided as a part of the ConstraintTransformationDescription. Regardless the way how the constraint representation in the target model is produced, it has always a textual form (string) obtained by pretty printing. This step we call also a constraint serialization. The constraint builder is also a constraint serializer in that sense. Sometimes we need to use external libraries to produce the output format, e.g. for UnifiedOCL to Java/SQL transformation we are using the DresdenOCL framework to transform OCL expressions into either Java or SQL expressions. Such a tool takes as an input the textual representation of the constraint (possibly also with the data structure definition) and produces also the output in textual form.

6.17.6 Constraint dictionaries

There are three types of dictionaries used in our system.

• The first type of a dictionary stores a map that binds together a regular expression matching a constraint in the source model with an instance of the class ConstraintDefinition. A ConstraintDefinition contains information about the abstract constraint name and the way how to parse the regular expression i.e. which positions (brackets) of a regular expression string corresponds to which parameters of the constraint. For instance, Drools2UnifiedOCLDictionary is a dictionary of that type.

• The second type of dictionaries are the ones that map an abstract constraint name with ConstraintTransformationDescription as discussed in Subsection 6.17.5. The great simplification in our architecture is the fact that all transformations from a technology specific representation to UnifiedOCL may use the same dictionary of that type. In our implementation it is the GeneralToUnifiedOCLDictionary.

• The last type of dictionaries are some auxiliary dictionaries. They contain additional or supplementary information required by transformations. E.g. how to translate particular primitive types (SQL2UnifiedOCLPrimitiveTypeDictionary) or which methods should be included/excluded from the generation (the field EXCLUDED_METHODS in AbstractFromJavaGenerator, which is the super class of Java to UnifiedOCL generator). They may be stored in separate classes or they may be represented just as a map or list within the generator code.

6.17.7 Producing the target representation

The last step in the process of transformation between various constraint representations is to print out the output. A generator goes through the input model and produces the output taking into account the input model data structure, all the intermediary models and built
The output is a textual file containing data structures and constraints in the target representation. We may also call this process a *model serialization*.

### 6.18 DSL project structure

In our system we have designed two new DSLs: *UnifiedOCL* and *Data Quality Mapping (DQM)*, both are implemented using Xtext. Moreover we modeled one technology specific representation – SQL – also as the Xtext DSL. Every Xtext DSL project consists of four plugins. We will explain them using our DQM as an example:

**ch.ethz.unifiedocl.dqm** contains the language grammar and all other classes needed for parsing of DSL sources, code generation, code validation, code formatting (pretty printing) etc. Figure 6.4 shows the standard packages of the Xtext project. The packages denoted using green color represent the top level packages; the *pink* packages are generated automatically by Xtext (*src-gen* directory); the *blue* packages are the Xtext stubs (Xtend classes contained in the *src* directory), that could be modified by the DSL developer. All the stubs are automatically translated from Xtend to Java (Java classes are stored in the *xtend-gen* directory). In our system we are extensively making use of the package generator which contains the main class implementing the transformation between two given representations. Any *generator* is a very complex class, e.g. Figure 6.8 shows a sample *generator* class for the SQL bundle. The attributes on the light pink background denote parts of the *intermediary model* (Subsection 6.17.2); the attribute on the green background is a dependency to the *matcher* (Subsection 6.17.3); the attributes on the blue background are the steering flags/parameters; while the operations on the yellow background implement various actions of the transformation process.

For the *UnifiedOCL* bundle we also implemented the *UnifiedOCL* code validation (in the editor when the validation engine return an error, the corresponding code is underlined with a red stroke and the popup error message window is provided).

**ch.ethz.unifiedocl.dqm.ui** contains packages with classes contributing to the user interface of the Eclipse platform. Those are presented in Figure 6.5. Same as in the previous point, the green color means the top level packages, the pink – Xtext generated packages while the blue – Xtext stubs for content assist, code quick fixes, outline view, etc.

**ch.ethz.unifiedocl.dqm.tests** contains tests for both **ch.ethz.unifiedocl.dqm** and **ch.ethz.unifiedocl.dqm.ui** bundles.

**ch.ethz.unifiedocl.dqm.sdk** allows to prepare an installation package of the DSL that can be used by an Eclipse Update Site.
Figure 6.4: Structure of an Xtext project – DQM language
Figure 6.5: Structure of an Xtext user interface project – DQM language
6.19 All bundles – details

In this section we are briefly discussing all the bundles (plugins) in our system, presenting their contents and responsibilities. We are presenting only the code written by us (excluding the automatically generated). In other words all the packages and classes mentioned below originate from the directory src of particular bundles. Other directories like src-gen, xtend-gen or emf-gen are not discussed as they are generated automatically by various tools, e.g. Xtext, Xtend.

ch.ethz.unifiedocl.tests, ch.ethz.unifiedocl.sql.tests, ch.ethz.unifiedocl.dqm.tests are designated to store tests for the the appropriate DSLs: UnifiedOCL, Constraint to Data Quality Mapping representation or SQL (we are considering it also as DSL).

ch.ethz.unifiedocl.sdk, ch.ethz.unifiedocl.sql.sdk, ch.ethz.unifiedocl.dqm.sdk contain scripts for preparing installation archives for the appropriate DSLs: UnifiedOCL, Constraint to Data Quality Mapping representation or SQL. Such an archive can be used by an Eclipse Update Site to install the plugin.

ch.ethz.unifiedocl.ui contains code for customizing the visual editor of the UnifiedOCL, e.g. content assist, highlighting, outline view, quick fixes etc. Moreover, this plugin contains code contributing to the context menu and the menu bar of Eclipse. The package ch.ethz.unifiedocl.ui.handler contains following classes:

- RunDGVisualizer – provides a menu bar item for running the Data Quality Visualizer from the Eclipse platform
- ShowPreferences – provides a menu bar item allowing for the configuration of various options (flags) of transformations in our system
- UnifiedOCL2XMIHandler, UnifiedOCL2XMIHandlerEditor, XMI2UnifiedOCLHandler, XMI2UnifiedOCLHandlerEditor – provide context menu items responsible for the serialization of UnifiedOCL file into the XMI format and the deserialization from XMI into UnifiedOCL

Package ch.ethz.unifiedocl.ui.util contains the utility class for the XMI serialization and is responsible for handling some aspects of the user interface related to this bundle.

ch.ethz.unifiedocl is the main bundle of the whole system. It contains the grammar of UnifiedOCL - UnifiedOCL.xtext along with all the supplementary classes generated by Xtext supporting various basic aspects of the DSL processing, e.g. building the abstract syntax tree.

- ch.ethz.unifiedocl.builder – contains a class responsible for preparing the environment for external tools used by some transformations – currently only used by DresdenOCL
- ch.ethz.unifiedocl.formatter – a stub for UnifiedOCL source formatters (pretty printing)
- ch.ethz.unifiedocl.fragments – contains a class EssentialOCLFragment that is required for the generation of DSL artifacts (e.g. parsers) that are uses internally by the Eclipse implementation of OCL
• ch.ethz.unifiedocl.generator – contains a stub for the code generation from *UnifiedOCL* sources and a generator for a constraint to data quality mapping file

• ch.ethz.unifiedocl.matcher – contains *UnifiedOCL* related classes and interfaces matching constraint occurrences found in the source model with the appropriate constraint definitions for both directions of the transformations. These are various versions of the `getConstraintsAndValues` method that return an object of the class `ConstraintTransformationValue`, which stores, in the abstract form, information about a constraint.

• ch.ethz.unifiedocl.scoping – a stub generated by Xtext

• ch.ethz.unifiedocl.services – contains a value converter service returning value converters for various types of data that are used when the *UnifiedOCL* model is processed

• ch.ethz.unifiedocl.validation – contains the *UnifiedOCL* code validator used by the *UnifiedOCL* code editor (if the code contains semantic errors, the corresponding statement is underlined with red color and a diagnostic message is available)

**ch.ethz.unifiedocl.core** is the bundle that contains common parts of all the technology specific bundles i.e. all the data structures and operations used by many concrete technology transformations

• ch.ethz.unifiedocl.core.dictionary
  - `ConstraintDefinition` – a structure storing the definition of a single constraint supported by the system, e.g. its name and its parameter names and values
  - `ConstraintsBuilder` – a class with methods for building constraints. It matches the abstract representation of the constraint that occurred in the source model with the corresponding representation of this constraint in the target model. Definitions of constraints in the target model are passed as a dictionary that contains `ConstraintTransformationDescription`. When there exists a satisfactory mapping, then using the `ConstraintTransformationDescription`, the output string containing the target constraint representation is produced.
  - `GeneralConstraintsDictionary` – contains a map that binds regular expressions matching of constraint occurrences with the abstract representation of those occurrences (class `ConstraintsDefinition`)
  - `GeneralToUnifiedOCLDictionary` – contains a map matching a constraint name with `ConstraintTransformationDescription` that contains the definitions of the output representation of constraints (identified by their names). It is the common dictionary for all the transformations where the target representation is the *UnifiedOCL*. It contains definitions allowing for producing constraints either in *UDQAM* or in OCL form (the list of all supported constraints (names) can be found in Appendix B and in Section 6.9)
- **IFromUnifiedOCLDictionary** and **IToUnifiedOCLDictionary** – interfaces that should be implemented by all classes containing dictionaries for to- and from- specific technology transformations
- **UnifiedOCLKeywordsDictionary** – loads the **UDQAM** dictionaries and returns them as a map: constraint names to map of parameters

- **ch.ethz.unifiedocl.core.generator**
  - **ConstraintTypes** – an enumeration type that contains various constraint types: various OCL types and **UDQAM**

- **ch.ethz.unifiedocl.core.matcher**
  - **CoreConstraintsMatcher** – contains top level methods matching a constraint occurrence found in the source model with the appropriate constraint definition for both directions of the transformation. These are various versions of the `getConstraintsAndValues` method that returns an object of the class `ConstraintTransformationValue` which stores, in an abstract form, information about the matched constraint.

- **ch.ethz.unifiedocl.core.processor** – contains an interface `Constraint2OCLProcessor` along with implementations of that interface for some constraints. This interface allows to specify custom actions when merging a `ConstraintTransformationValue` (actual constraint parameters) with a constraint definition (`ConstraintTransformationDescription`) to produce a textual representation of the constraint.

- **ch.ethz.unifiedocl.core.util**
  - **ConstraintTransformationDescription** – a data structure to store all information required to produce the target textual representation of the constraint, which includes the constraint name, type, parameters (names and types) and **processors** (if a customized action is required to produce the textual representation of the constraint). This structure can be considered as the final definition of the target constraint representation.
  - **ConstraintTransformationValue** – contains a runtime information about a constraint occurrence that is required to produce the target textual representation of that constraint. These include: constraint location, name and values of the constraint parameters. It can be considered as a **parametrization** of the corresponding constraint definition (`ConstraintTransformationDescription`).
  - **StringConversionUtils** – an utility class containing methods to convert various string representations required by source or target constraint representations, e.g. single or double quoted strings. Moreover, this class contains methods for extracting values of various data types from text.
  - **UnifiedOCLKeywordType** – an enumeration type containing names of types of constraint parameters (in **UDQAM**)

Moreover, this plugin contains the directory `dictionaries` containing **UDQAM** dictionaries with basic constraint definitions. Additionally, it contains dictionaries for the extensions of **UnifiedOCL** supported by our system (Java and SQL).
ch.ethz.unifiedocl.core.ui contains common parts of the user interface of all plugins that contains various technology specific transformations. It contains enhanced dialog windows informing about the results of the executed transformation or details of errors in case when the transformation was not successful.

ch.ethz.unifiedocl.dependencies is designed to store all the libraries required by all other plugins of our system. We decided to designate one plugin to store all the JARs used by other bundles to simplify the dependency management in our system (stored in one place). For instance DresdenOCL is provided with our tool as a part of this bundle.

ch.ethz.unifiedocl.ocl2java, ch.ethz.unifiedocl.ocl2sql contain builders (subclasses of the ConstraintBuilder) capable of transforming OCL expressions into correspondingly either Java expressions or a SQL code using the DresdenOCL framework. They are used to process complex OCL constraints for which there are no compact representations (UDQAM labels, Java annotations, etc.) and therefore there are no regular expressions matching those complex constraints.

ch.ethz.unifiedocl.uml contains the implementation of the transformation from the UML class diagram (UML2 format, .uml file) into UnifiedOCL. The generate.mtl file contains the exact code of the transformation which is written in Accceleo. Moreover, it contributes to the context menu in a similar way as ch.ethz.unifiedocl.ui, providing menu items to trigger the action of the transformation (package ch.ethz.unifiedocl.uml.handler).

ch.ethz.unifiedocl.ocl contains the implementation of the extraction of pure OCL from the UnifiedOCL representation. The transforms directory contains a .qvto file with the body of the transformation written in Query/View/Transformation Operational (QVTo). Moreover, it contributes to the context menu in a similar way as ch.ethz.unifiedocl.ui, providing menu items to trigger the action of the transformation (package ch.ethz.unifiedocl.ocl.handler).

ch.ethz.unifiedocl.java is responsible for transformations between UnifiedOCL and Java in both directions. It contains the actual code executing the transformation and context menu item contributions to trigger those transformations. Figure 6.6 presents the class diagram of this plugin. Classes drawn outside of the ch.ethz.unifiedocl.java bundle come from the same plugin but are considered as the user interface (not the core part).

- ch.ethz.unifiedocl.java – stores classes required by OSGi and the Eclipse framework (JavaExecutableExtensionFactory and Activator). Moreover, it contains classes required by the configuration GUI to set desired flags/parameters of the to- and from- Java transformation (classes JavaPreferenceInitializer and JavaPreferencePage).
- ch.ethz.unifiedocl.java.builder – contains data structure storing information required to generate Java Bean Validation annotations. StandardAnnotationBuildData is responsible for holding data about standardized constraints (including Hibernate Validator extensions), while both CustomAnnotationBuildData and
Figure 6.6: ch.ethz.unifiedocl.java – UML class diagram
DresdenOCLConstraintBuildData keep data required by DresdenOCL to produce customized constraint annotations.

- `ch.ethz.unifiedocl.java.dictionary` — contains a class `UnifiedOCL2JavaDictionary` with a map containing `ConstraintTransformationDescriptions` that specifies how constraint occurrences in `UnifiedOCL` are transformed into Java code. The corresponding class for the opposite direction of a transformation (`Java2UnifiedOCLDictionary`) is just a stub. It is due to simplicity reasons: in Java we assume that `UDQAM` label names correspond to the names of the standard Bean Validation annotation names (same is true for parameters). However, if more Java Bean Validation annotations would be introduced, which do not have corresponding `UDQAM` labels then this dictionary should be filled with the missing entries.

- `ch.ethz.unifiedocl.java.generator` — this package contains Xtend classes that define the body of transformations in both directions
  - `IFromJavaGenerator` and `IToJavaGenerator` — these are interfaces for classes implementing transformations. They are required by Google Guice\(^{48}\) to correctly inject dependencies/classes of any plugin.
  - `AbstractFromJavaGenerator` — contains the common code of transformations originating either from Java or from Drools into `UnifiedOCL` (Drools uses a Java definition of the data structure).
  - `Java2UnifiedOCLGenerator` — an Xtend class containing a Java Bean Validation specific code for the Java to `UnifiedOCL` transformation
  - `UnifiedOCL2JavaGenerator` — an Xtend class that contains the code for `UnifiedOCL` to Java transformation
  - `Java2DQMGenerator` — an Xtend class generating data quality mapping DQM out of the Java code (collects constraint locations and names)

- `ch.ethz.unifiedocl.java.handler` — contains context menu contributions for executing to- and from- Java transformations and generation of data quality mappings as well

- `ch.ethz.unifiedocl.java.matcher` — contains a class `UnifiedOCL2JavaMatcher` implementing the `FromUnifiedOCLConstraintsMatcher` interface and containing methods for processing constraint occurrences in the `UnifiedOCL` code. It returns a set of recognized constraints that are stored together with their locations and their parameter values required for further processing in order to produce the target representation. A matcher for the opposite direction of the transformation does not exist due to the simplification exclusively for Java code (`UDQAM` labels matching Java annotation names and parameters). In general, this package should contain also the corresponding matcher (constructed in the same way) for the reversed transformation.

- `ch.ethz.unifiedocl.java.test` — test classes

• **ch.ethz.unifiedocl.java.ui** – contains the **JavaModule** class required by the OSGi platform and the dependency injection engine (injection configuration entries)

**ch.ethz.unifiedocl.drools** is implemented in a very similar way as the plugin for Java (**ch.ethz.unifiedocl.java**). For this reason here we are presenting just the differences, all the other classes and interfaces act in the same way – they differ only in names (Java vs Drools). The overview of this plugin is presented in Figure 6.7. It shows the complex structure of that bundle (including **builders** and **processors**) and the classes inherited from the Java bundle (**AbstractFromJavaGenerator** and **CustomAnnotationBuildData**). The other classes drawn outside of this bundle belong to the user interface part.

• **ch.ethz.unifiedocl.drools** – contains a **.drl** file with the bodies of validating methods for complex built-in constraints (e.g. email or creditcard) denoted in Drools. Those methods are taken directly from the Hibernate Validator implementation and are slightly adapted to be used with Drools. The file **HibernateValidatorConstraintMethodsForDrools** serves just as a reference file – the same code is embedded in the generator classes performing the transformations.

• **ch.ethz.unifiedocl.drools.builder** – the Drools representation of constraints is not a compact one. One simple constraint may consist of a sequence of conditions. Therefore, one needs to discover all those conditions and check whether they all together constitute a single, recognized by our system constraint. In this package the class **DroolsConstraintsPartsMerger** is responsible for merging those conditions and building **ConstraintTransformationValues**, which represents the occurrences of known constraints. The class **ConstraintParametersNamesMapping** stores the information about constraint parameters that are obtained from conditions constituting a given constraint. If merging conditions requires a custom action this class stores also a **processor** – implementation of the interface **ConstraintsPartsParametersMergingProcessor** that allows a user to define and execute custom merging operations.

• **ch.ethz.unifiedocl.drools.dictionary** – for Drools we cannot use the simplification that was possible for Java Bean Validation, thus both dictionaries contain definitions of constraints for target platforms (Drools or **UnifiedOCL** depending on the direction)

• **ch.ethz.unifiedocl.drools.generator** – the generation from Drools to **UnifiedOCL** requires two steps:
  
  1. parsing data structures expressed as Java classes
  2. parsing constraints expressed using Drools

As a consequence there exist two generators: first – **Drools2UnifiedOCLGenerator** – builds all the constraints and then the second – **Java2UnifiedOCL4DroolsGenerator** – generates the final output (target representation) by merging constraints with the data structure
obtained from Java classes. The transformation in the opposite direction also requires two steps. We decided to create a generator capable of producing Drools constraints (.drl file) out of the UnifiedOCL file and for the Java classes (used as a model for constraints defined in Drools) we entirely reused the UnifiedOCL to Java generator (used with different parameters as for Bean Validations – should be called directly by user). To sum up: a UnifiedOCL file taken as input contains all required information to build Drools constraints, so we can generate them directly. However, as Drools does not contain any information about the data structure (similarly as OCL they are context dependent), it is required to provide the data structure (Java Beans) for the opposite direction of transformation.

• ch.ethz.unifiedocl.drools.matcher – similarly as in the case of the ch.ethz.unifiedocl.drools.dictionary package we cannot use the simplification that was possible for Java Bean Validation. Thus, both matchers producing the parametrization for constraint occurrences in the source representation are required (Drools and UnifiedOCL).

• ch.ethz.unifiedocl.drools.processor – this package contains the interface ConstraintsPartsParametersMergingProcessor that is supposed to be implemented by processors responsible for custom (user defined) actions for merging Drools conditions constituting a single constraint. The OrEqualMergingProcessor processor is an example of such a processor. It is responsible for the parameter inclusive which indicates if the inequality was strict or not.

ch.ethz.unifiedocl.sql plugin is implemented in a very similar way as the plugins for Java (ch.ethz.unifiedocl.java) and for Drools (ch.ethz.unifiedocl.drools). For this reason, we are also presenting here just the differences in comparison with those plugins. All the other classes and interfaces play the same roles – they differ only in names (Java vs Drools vs SQL). The only significant difference is the fact that in our system SQL is modeled as a DSL. This is due to the fact that we were not able to find a suitable parser for SQL scripts. Therefore, we created our own SQL grammar based on the SQLite grammar available at the GitHub implemented in ANTLR v449, adjusting it to the PostgreSQL syntax50 and using the approach for expressions discussed by Lorenzo Batini [4]. As a consequence of the fact that the SQL plugin is also an Xtext bundle, some packages are stubs generated by Xtext (the same as discussed in the bundle ch.ethz.unifiedocl), the only difference is again only the name. Moreover this plugin does not contain user interface contributions, e.g. context menu items which are stored in the bundle ch.ethz.unifiedocl.sql.ui

- ch.ethz.unifiedocl.sql – contains the definition of our SQL grammar – SQL.xtext
- ch.ethz.unifiedocl.sql.dictionary – contains two additional dictionaries
  - SQL2UnifiedOCLPrimitiveTypeDictionary – stores how SQL (PostgreSQL) primitive types are mapped into UnifiedOCL primitive types (the latter ones conform to OCL primitive types)

Figure 6.7: ch.ethz.unifiedocl.drools – UML class diagram
Figure 6.8: Sample generator class: ch.ethz.unifiedocl.sql.generator.SQL2UnifiedOCLGenerator – UML class diagram
• UnifiedOCL2SQLPrimitiveTypeDictionary – defines how the UnifiedOCL primitive types are mapped into SQL (PostgreSQL) primitive types

• ch.ethz.unifiedocl.sql.builder – contains two auxiliary data structures: TableDescriptor and ColumnDescriptor. They hold information about the discovered in the SQL script tables and columns (SQL to UnifiedOCL transformation). Those two descriptors could be considered as an intermediary model produced to simplify the transformation (avoid multiple source file passing and backtracking).

The model directory contains the Ecore model of our SQL language.

ch.ethz.unifiedocl.sql.ui contains the user interface contributions for SQL language transformations. Similarly as bundles ch.ethz.unifiedocl.drools and ch.ethz.unifiedocl.java it contains the context menu items to trigger the to- and from- SQL transformations for a selected source.

ch.ethz.unifiedocl.dqm is a bundle storing the definition of the data quality mappings (DQM language). We modeled constraints to data quality mappings as an Xtext DSL, whose syntax is described in Section 5.5. This bundle does not contain anything more than the code generated directly by the Xtext. The file DQM.xtext contains the grammar of the DQM language that describes the content of the data quality mapping (.dqm) files.

ch.ethz.unifiedocl.dqm.ui is a stub bundle for user interface contributions for the Eclipse platform with regards to the constraints to data quality mappings (DQM language)

ch.ethz.unifiedocl.dqm.gui contains the code of the Data Quality Visualizer

• ch.ethz.unifiedocl.dqm.gui – stores the GUI components required by Data Quality Visualizer. Those include list or check box selectors, bar renderers (nested classes), etc. As mentioned in Section 6.15 the architecture of our Data Quality Visualizer conforms to the MVP design pattern. The DQFeedbackView is an interface defining all the ways how a presenter may interact with the view. The class DQFeedbackWindow is an actual implementation of the graphical user interface. Through implementing the view interface it can be easily replaced with other implementations without disturbing any other parts of the applications.

• ch.ethz.unifiedocl.dqm.model – contains the model of the data that is visualized. The class DQFeedbackModel stores all the data needed to visualize the values of the particular data quality dimensions i.e.:
  – constraint to data quality dimension mappings (DQMapping)
  – list of constraint violations – collection of ConstraintViolationDescription
  – all entities for which the data quality feedback should be presented
  – a graph of types hierarchy required for correctly computing values for particular data quality dimensions as discussed in Section 5.8
  – the name of the top level package (scope) that should be visualized
• ch.ethz.unifiedocl.dqm.mapping – contains classes that all together constitute a traversable and searchable representation of constraint to data quality mappings.
  - DQMapping – stores the list of single definitions of constraint to data quality dimension mappings (collection of ConstraintMappingDefinitions)
  - ConstraintMappingDefinition – contains information about constraint mappings to data quality dimensions for a particular location (e.g. in Java package name + class name + attribute name). For every location a map is stored that contains entries binding constraints defined for that location (key of the map) with a list of mappings for a given constraint (value of the map – list of DQDimensionRelevance).
  - DQDimensionRelevance – this is a small structure containing the dimension name and the penalty value (dimension relevance/weight) which indicates how important a constraint owning this structure, for a particular data quality dimension, is
• ch.ethz.unifiedocl.dqm.computer – this package is responsible for correctly computing the value of a particular data quality dimension for a given set of data. It takes into account constraint violations and the given mapping file. It contains the implementation of all formulae discussed in Section 5.9 (class DQComputer). The class ConstraintViolationDescription is a structure to store which entity (identified uniquely by a string) violates which constraint defined in a concrete location. It is mostly used by the DQComputer.
• ch.ethz.unifiedocl.util – contains two utility classes (DQUtil and PostgresUtil) that help to preprocess the raw data obtained from the PostgreSQL server or Hibernate Validator in order to use them in Data Quality Visualizer (constraint violations). Constraint violations are converted into a format discussed in Section 6.15.

UniOCLAllTogether – this is a special project that is used to build an installation archive containing all the bundles. Such an archive can then be used by an Eclipse Update Site to facilitate the process of installing our system (all plugins) on the Eclipse platform.

6.20 All bundles – overview

Figure 6.9 presents the dependencies between all the bundles discussed in the previous section. Each bundle is considered as a separate component within the OSGi framework.

6.21 Extensibility

6.21.1 A new representation

As we discussed in Section 4.7 our system could be easily extended with new constraint representations by means of plugins. In this section we are describing the steps required to
Figure 6.9: All bundles/plugins – component diagram
implement such a plugin. We recommend for a new plugin developer to take one of the three representations implemented by us as a reference. The decision which one would be the most suitable could be justified as follows:

- **ch.ethz.unifiedocl.sql** – should be chosen when the desired representation is also declarative or is kind of a relational database (SQL dialect). This would be also the right choice when there exists no parser for the representation to be implemented, and it has to be generated from a language grammar.

- **ch.ethz.unifiedocl.java** – would be the right choice when the new representation is also object-oriented

- **ch.ethz.unifiedocl.drools** – this would be the best choice when the new representation is also kind of a business rules or it requires to merge various types of sources

Each of the representations is different, as a result, there is no silver bullet how it should be implemented. We assume that a developer of a new representation is familiar with the Eclipse plugin development. We also recommend to consult sections 6.17 and 6.19. Below we are providing just an overview of the steps that should be implemented, therefore, our three supported representation implementations should serve as examples:

1. Create a new Eclipse project:
   - if there exists no parser, the Xtext Project should be chosen. The description how to create a grammar and generate all the required artifacts can be found in the Xtext Documentation
   - if there exists an available parser or reverse engineering tools (as described in Subsection 6.17.1), then the Plug-in Project should be chosen

2. Specify the plugin dependencies in the file MANIFEST.MF. Depending on the concrete representations various plugins may be required by your representations – consult Figure 6.9 for dependencies to the common parts of our system.

3. Adjust the bundle activator (specified in the file MANIFEST.MF) to allow for dependencies injection. E.g. class ch.ethz.unifiedocl.drools.Activator

4. Provide all dictionaries required by your representation as described in Subsection 6.17.6. This step involves also the creation of regular expressions to parse various types of constraints in the source representation (described in Subsection 6.17.4). E.g. package ch.ethz.unifiedocl.drools.dictionary.

5. Implement constraint matchers as described in Subsection 6.17.3. Check if you can extend any existing class from the bundles required as dependencies e.g. class ch.ethz.unifiedocl.matcher.FromUnifiedOCLConstraintsMatcher.

6. Implement constraint builders as described in Subsection 6.17.5. Notice that in case when the class

---

ch.ethz.unifiedocl.core.dictionary.ConstraintsBuilder is used, the builder can be implemented by the same class containing a dictionary of the first type (see Subsection 6.17.6).

7. Implement generators as described at the beginning of Section 6.17, which is the main class responsible for the transformation. We recommend to use the Xtend template language as it simplifies the implementation of a model serialization (model to text transformation). E.g. package ch.ethz.unifiedocl.java.generator.

8. Create the GUI handlers, i.e. context menu items, that would allow for the transformation execution. Therefore, adjust also the plugin.xml. E.g. package ch.ethz.unifiedocl.java.handler.

9. Install the plugin. One may also want to modify the project UniOCLAlltogether to integrate the new plugin with the existing ones and deliver all of them as a single archive.

6.21.2 Eclipse independence

Currently our system heavily relies on EMF, therefore, it is not possible to easily migrate it to another IDE, e.g. IntelliJ IDEA. However, as it is implemented in Java, which is portable, and mostly uses Xtext and Xtend (for the unified data quality requirements representation and constraint transformations) in the future it should be possible to at least migrate to IntelliJ IDEA since according to the Sven Efftinge, the project lead of Xtext, there are plans to develop the Xtext and Xtend plugins also for IntelliJ IDEA. Therefore, one would need to reimplement just the parts that are based on other Eclipse plugins, which includes Acceleo, QVTo, etc.

Moreover, if one would like to use our system outside of the Eclipse GUI, it should be possible to create a Headless Build, which is a light Eclipse instance running without the GUI, but which includes all the required OSGi bundles. An introduction to this topic is described by Lars Vogel and Simon Scholz.

---

In this chapter we are presenting how to install and use our system. We are discussing first the Eclipse plugins responsible for the UnifiedOCL representation and implementation of all transformations and then our Data Quality Visualizer. This chapter is dedicated to the prospective users but we are also providing some information for future developers.

7.1 Eclipse

Our system is compatible with Eclipse Luna (version 4.4), available online\(^1\), running on Windows 7/8/8.1. The sufficient Eclipse package is the standard one – Eclipse IDE for Java Developers. However, for the future developers and advanced users we recommend Eclipse Modeling Tools. This version provides a better integration with the EMP projects, e.g. Xtext or Papyrus via the Install Modeling Components menu entry (under the Help menu). In that way, a user would be able to easily install Papyrus and create UML class diagrams. For the developers it also facilitates the process of installation of all additional EMP tools required by our system.

Moreover, we recommend to install the Eclipse JBoss plugin\(^2\) which includes the Drools editor (syntax coloring, source code validation, etc.). This would make the Drools development more user friendly.


7.2 Installation guide

Our system is packaged as a *Deployable feature* – this is .zip archive that is compatible with the *Eclipse Update Site*. We delivered our system as the `alluniocl.zip` file. In order to install our system (set of bundles) one needs to follow the steps described below:

1. From the Eclipse Help menu select the item *Install New Software*...
2. Click the button *Add*...
3. Using the button *Archive*... choose the archive `alluniocl.zip`
4. Fill the form providing the name of the feature repository as illustrated in Figure 7.1

![Figure 7.1: The UnifiedOCL installation – Add Eclipse Update Site](image)

5. Click the *OK* button
6. Choose all the plugins from the category *UnifiedOCL All-in-one* as illustrated in Figure 7.2
7. Click the *Next* button and then the *Finish* button in the subsequent window
8. Confirm the dialog window *Security Warning* informing about the unsigned content by clicking the *OK* button
9. The progress bar is informing about the software installation progress
10. Restart Eclipse – click the Yes button on the dialog prompting for Eclipse restart

7.3 Eclipse configuration GUI

The second step one need to do to use our system is to configure various flags and parameters of our system using the Eclipse Preferences. In order to configure transformation and UnifiedOCL parameters, one need to follow the steps described below:

1. From the Eclipse UnifiedOCL menu select the item Configure...
2. From the tree on the left choose one of the preferences pages, e.g., concerning only the Java plugin presented in Figure 7.3
3. In the window on the right configure the parameters
4. Click the OK button.

7.4 Supported file types

The input and output artifacts of all the transformations are the source code files. In Table 7.1 we are presenting all the input file types (identified by their extensions) supported by our
system. The table shows also all available transformations for a given input representation. In Table 7.2 we are presenting other file types that are processed by our system or that are the output artifacts of some transformations.

### 7.5 Transformation step-by-step

In order to execute one of the transformations, a user needs to follow the steps described below:

1. In the Eclipse Package Explorer (Figure 7.4 – red number 1) right-click on the source that should be taken as the transformation input according to Table 7.1. The column Input scope indicates what other sources would be taken as the input of the transformation as the consequence of selecting the given input source file.

2. From the context menu choose the desired transformation as presented in Figure 7.4 (red number 2 – e.g. a transformation taking a UnifiedOCL file as source)

3. Wait for the confirmation window informing about the success or displaying errors if the generation failed. A sample confirmation window is presented in Figure 7.5.

Alternatively to the first step a user may right-click on the opened editor and the same context menu items would appear. Supported editors are:

- *UnifiedOCL* editor
- default Eclipse Java editor
Figure 7.4: UnifiedOCL transformations in Eclipse

Figure 7.5: A transformation confirmation window
Table 7.1: Supported input file types

<table>
<thead>
<tr>
<th>Input file extension</th>
<th>Input file type</th>
<th>Input scope</th>
<th>Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>.uml</td>
<td>file containing a UML class diagram denoted with the UML2 specification</td>
<td>all the classes, interfaces and relationships from the diagram</td>
<td>UML → UnifiedOCL</td>
</tr>
<tr>
<td>.uniocl</td>
<td>UnifiedOCL specification</td>
<td>entire file</td>
<td>UnifiedOCL → {Java, SQL, Drools, OCL, XMI, DQM}</td>
</tr>
<tr>
<td>.java</td>
<td>Java classes</td>
<td>every class contained in the same package as the selected class</td>
<td>Java → {UnifiedOCL, DQM}</td>
</tr>
<tr>
<td>.sql</td>
<td>SQL script with a table schema definition and ALTER TABLE ADD CONSTRAINT statements</td>
<td>entire file</td>
<td>SQL → UnifiedOCL</td>
</tr>
<tr>
<td>.drl</td>
<td>Drools file – constraints expressed as business rules</td>
<td>entire Drools file and all the Java classes from the same package</td>
<td>Drools → UnifiedOCL</td>
</tr>
<tr>
<td>.xmi or .oclxml</td>
<td>serialized UnifiedOCL specification using XMI</td>
<td>always both files with the same name but different extensions (.xmi and .oclxml)</td>
<td>XMI → UnifiedOCL</td>
</tr>
</tbody>
</table>

- default JBoss Eclipse Drools editor
- SQL editor provided as part of our system (corresponds to our SQL Xtext project)

7.6 Working with the UnifiedOCL editor

One of the benefits of the usage of Xtext for the implementation of UnifiedOCL (our DSL language) is the editor it provides. The editor that is generated by Xtext can be adjusted in various ways by filling the generated stubs in the ch.ethz.unifiedocl.ui bundle. In that way we achieved a very powerful editor for UnifiedOCL that supports features described below. Figure 7.6 presents various features that are available when working with the UnifiedOCL sources. We can see that the syntax is colored to increase human readability of the UnifiedOCL sources – e.g. a gold color for the UDQAM constraints. Moreover, our editor provides the outline view (red number 1) displaying a tree-like structure of the source. The UnifiedOCL syntax is being validated when the source is modified. Errors are marked in the code (a red underline) and a popup window with error messages is available (a red number 2). Moreover, the summary of all errors is visible in the Eclipse Problems view (a red number 3). Additionally, a limited support is provided for auto-completion of the UnifiedOCL code.
Figure 7.6: The Eclipse editor for UnifiedOCL
7.7 Data Quality Visualizer

7.7.1 General information

As discussed in Section 6.15 the Data Quality Visualizer may be used in three different ways. In order to run it from the Eclipse platform one should choose the option Data Quality Visualizer from the UnifiedOCL menu.

In case of running Data Quality Visualizer not programatically the first thing one needs to do is to load the required data for which the data quality feedback should be presented. The input files are mapping files (\dqm) and JSON files described in the Section 5.11 and in the Appendix D. In order to load those files one needs to follow the steps described below:

1. From the menu DQFeedback select the item Load model

2. Provide the paths to all the required input files:
   
   - Data quality mapping – a \dqm file containing constraints to data quality dimensions mappings
   - Entities – a \json file containing all entities (objects, table rows) for which the data quality feedback should be presented
   - Types hierarchy – a \json file containing the hierarchy of entity types that should be visualized
   - Violations – a \json file containing data about all the constraint violations by some entities

   A sample input configuration is presented in Figure 7.7.

3. Choose the Default package from the combo box (appears only when the Types hierarchy file is specified) that corresponds to the package from which the entities should be visualized

4. Click the button Load model

### Table 7.2: Supported other file types

<table>
<thead>
<tr>
<th>File extension</th>
<th>File type</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.dqm</td>
<td>constraint to data quality dimensions mapping</td>
<td>output of the transformations: {UnifiedOCL, Java} (\rightarrow) DQM</td>
</tr>
<tr>
<td>.udqam</td>
<td>UDQAM dictionaries (JSON files)</td>
<td>definitions of UDQAM labels</td>
</tr>
<tr>
<td>.ocl</td>
<td>Eclipse OCL file</td>
<td>output of the transformation UnifiedOCL (\rightarrow) OCL</td>
</tr>
</tbody>
</table>
7.7.2 Visualizing data

Below we are describing all the components of the graphical user interface of *Data Quality Visualizer*. All the numbers of components corresponds to the numbers presented in Figure 7.8. Table 7.3 provides the overview how components 2, 3 and 4 are linked to the formulae discussed in Section 5.9.

<table>
<thead>
<tr>
<th>Scope</th>
<th>Grouping mode</th>
<th>Display mode</th>
<th>Formulae as discussed in Section 5.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple classes</td>
<td>Combined</td>
<td>By Class</td>
<td>Multiple classes combined</td>
</tr>
<tr>
<td>Multiple classes</td>
<td>Separately</td>
<td>By Class</td>
<td>Multiple classes separately</td>
</tr>
<tr>
<td>Multiple classes</td>
<td>–</td>
<td>By Dimension</td>
<td>Multiple classes separately</td>
</tr>
<tr>
<td>Single class</td>
<td>Combined</td>
<td>By Class</td>
<td>Multiple combined instances</td>
</tr>
<tr>
<td>Single class</td>
<td>Separately</td>
<td>By Class</td>
<td>Multiple separately instances</td>
</tr>
<tr>
<td>Single class</td>
<td>–</td>
<td>By Dimension</td>
<td>Multiple separately instances</td>
</tr>
</tbody>
</table>

1. switches between various feedback types as discussed in Section 5.10. The last tab – *Class dependency graph* illustrates the hierarchy of types (classes) that are visualized. E.g. a sample graph is presented in Figure 7.9. The red node represents the class that is currently being visualized.

2. allows to select the *scope* i.e. either the concrete class (type) should be visualized or
Figure 7.8: Data Quality Visualizer – a GUI overview
all types. In the former case by *entity* we mean a class/table, while in the latter one we mean an object (class instance)/a table row.

3. allows to decide whether the data quality dimension values should be presented for each entity separately or for all together

4. allows to define whether the data quality is presented with regards to the classes/types (values of data quality dimensions grouped by entities) or data quality dimensions (values of data quality dimensions grouped by dimensions)

5. this option allows to compare differences between two various selections in components 6 and 7. When the button *Make snapshot* is clicked then the current selection of entities and data quality dimensions is recorded. When the check box *Display differences* is selected, then not only the current selection of components 6 and 7 is visualized but also the one stored as a snapshot. The snapshot selection is labeled as *SNAPSHOT*. Changing the value in components 2, 3 or 4 clears the recorded snapshot (the comparison for different scopes makes no sense).

6. allows to choose multiple entities: classes (types) or instances

7. allows to choose data quality dimensions that should be visualized. Only the dimensions specified for the particular scope (concrete class) are displayed, if multiple classes are selected in component 2 then all dimension are listed.

8. allows to choose either all entries of the check box list or no entries. This is possible for both: the entity selector and the data quality dimensions selector.
8

User Study

In order to evaluate our approach and the usefulness of our system we conducted a user study. We wanted to obtain not only a general feedback about the constructed system but also some insight about the further direction of evolution of our system.

8.1 Tasks

Users had to solve four tasks. First three of them were related to the unified constraint representation and constraint transformations, while the last was related to our Data Quality Visualizer. Before each task we had given users a detailed explanation of our system that were related to the task. In case users had no previous experience we had been also giving some general information about used technologies, representations or programming languages. After each task they were asked to fill out a corresponding questionnaire (provided in Appendix C.4 of this thesis). The questionnaires were created using Google Forms.\footnote{https://www.google.com/forms/about/ [Online, accessed 19-August-2015]}

Task I In this task participants had to choose one of the following three technologies: Java, SQL or Drools. It had to be the one they feel the most comfortable with. Next they were given a UML class diagram consisting of two classes: a Student taking part in a Study. This diagram contained just the model related information – no constraints (presented in Appendix C.1). Participants were asked to create the code corresponding to the given diagram. The syntax did not have to be absolutely correct – our goal was just to stimulate users to think about the problems they are facing when converting UML class diagrams into code (this was required to gather information about the problems that users are encountering and which could be solved by our system). The second part of that task was the same as the first one, but this time the participants
were given the same UML class diagram which included also constraint specifications (presented also in Appendix C.1). Constraints were expressed in UDQAM, but we did not provide any introduction to UDQAM – we wanted to observe whether the UDQAM syntax is intuitive and is understood by the users. Participants had to enrich their model with the constraints defined at in UML class diagram. Moreover, they were instructed to include only those constraints that make sense in the technology of their choice. For both parts we were taking time the users needed to solve those parts of the task.

The third part of the first task was to generate the same technology specific representation using our tool, given the same UML diagram (with constraints) in a digital form. We were also measuring the time required by participants to obtain the target representations of the model with constraints denoted using the UML class diagram. Next the users were ask to analyze the generated code and adjust it if it didn’t meet their expectations. First, we wanted to check whether the users understood the concept of two steps of transformations (from the source representation to UnifiedOCL, and from UnifiedOCL to the target representation). Second, we wanted to know if our tool generates the correct code, that fulfills users’ requirements. Finally, we wanted to investigate if potential users consider our tool as useful and as facilitating software engineering tasks (especially taking into account the MDA approach).

**Task II** The goal of that task was to investigate to what extent the UnifiedOCL syntax is user friendly. Participants were given a short introduction (based on an example) about the constructs and ways of expressing constraints available in UnifiedOCL. Then they were asked to solve a task that required to add one field of the class and specify a constraint for that field. There were more than just one correct solution (e.g. additionally involving creation of a new class). We were not imposing the way how this task should be solved – every solution that made sense was accepted. Users were also not expected to know the correct name of the constraint as defined in the UDQAM dictionary – we asked them to use the name that would be intuitive according to them. The task description is given in Appendix C.2.

**Task III** In this task we wanted to explore the possible use cases for our system. Participants were given two representations of the entities Flight and Pilot – a code in Java and in SQL. We explained that UnifiedOCL may serve as the constraint repository which could be used to generate the code for various layers of the same system (e.g. persistence layer – SQL and presentation layer – Java). The goal is to achieve consistency of constraint definitions. Participants were asked to manually compare two representations and think of whether the constraints are equivalent. Then, the participants were requested to use our tool to obtain the same format of representation of the two given sources. In that way, we wanted to investigate which format would be the preferred one by them. They could solve this task by converting Java to SQL or SQL to Java or both files to UnifiedOCL. We wanted the users to discover the potential of our system and explore some possible usecases. Furthermore, we had a chance to evaluate the memorability aspect of our system – whether users still remember how to use our system (this was explained in Task I). The source codes for the files used in this task are available in Appendix C.3.
Task IV  This task involved the usage of our Data Quality Visualizer. Participants were given an introduction to our visualizer along with explanations of different modes and forms of feedback that are available in this application. Next they were given five questions related to the data quality of the loaded set of data (airline model available in Appendix C.4). We wanted to observe which form of the feedback is the most appreciated one by the users (spider web plot, bar plot or raw data) and whether they find it useful providing a feedback about data quality using our approach. Questions are listed in the Google Forms provided in Appendix C.4.

8.2 Participants

The prerequisite for participants to take part in our study was to have a basic knowledge about Java and SQL. Participants were supposed to have also some basic knowledge about UML class diagrams and software engineering in general.

Our goal was to attract potential users of the system that have either an academic background or are working for industry. In that way, we would be able to indicate possible directions of further research in the field of constraint transformations and data quality and at the same time to point out extensions/improvements required to turn our system into a widely used tool.

Our participants were master and doctoral students, postdoctoral researcher and people working for several companies in industry (including people with diversified experience: from junior software engineers up to senior software engineers). Some participants had both of the backgrounds. One participant was neither working nor studying. That’s why the percentage for the type of background in Table 8.1 does not add up to 100%. In total we had 21 participants. However, one of the participants was not focusing on the tasks and showed reluctance to complete several tasks which lead to incomplete results in his case. Moreover, he did not provide constructive feedback. Based on that we excluded the incomplete results from this participant. Therefore, the summary of results in the following section includes answers from 20 participants. Table 8.1 summarizes the information about the participants.

8.3 Goals

Through conducting the user study we wanted to achieve the following goals:

- We wanted to evaluate how important the problems related to constraints and data quality for software engineers are
- How likely is our system to become a widely used CASE tool (as discussed in Section 2.5)
- Is our tool providing what users expect regarding constraint transformations
- Which aspects of constraint transformations are the most important for users
Figure 8.1: Knowledge and experience of participants
Table 8.1: Participants - summary

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Male</td>
<td>75%</td>
</tr>
<tr>
<td>Female</td>
<td>25%</td>
</tr>
<tr>
<td>20-25 years old</td>
<td>10%</td>
</tr>
<tr>
<td>26-30 years old</td>
<td>35%</td>
</tr>
<tr>
<td>31-35 years old</td>
<td>25%</td>
</tr>
<tr>
<td>36+ years old</td>
<td>30%</td>
</tr>
<tr>
<td>Academic background (including MSc students)</td>
<td>50%</td>
</tr>
<tr>
<td>Industrial background</td>
<td>75%</td>
</tr>
<tr>
<td>Achieved highest level of education:</td>
<td></td>
</tr>
<tr>
<td>Bachelor in Computer Science</td>
<td>35%</td>
</tr>
<tr>
<td>Master in Computer Science</td>
<td>50%</td>
</tr>
<tr>
<td>Doctorate in Computer Science</td>
<td>15%</td>
</tr>
</tbody>
</table>

- Which aspects of the development environment are important for users when working with constraints
- What is the usability of our system
- What are the possible ways of future development of our system
- What are the possible ways of further research in the field of constraints and data quality

8.4 Procedure

In order to evaluate the usefulness of our system we decide to follow guidelines provided by Jakob Nielsen [64]. Therefore, we decided to adopt the definition of usability provided by Nielsen as the composition of:

- learnability
- efficiency
- memorability
- errors
- satisfaction

Moreover, we decided to adopt usefulness as defined by Nielsen as the sum of usability (described above) and utility (‘whether it provides the features you need’[64]). The questionnaires were designed to allow for estimate usefulness defined in that way.

Additionally, we were observing participants solving their tasks to gather as much data as possible and we asked them about their general comments regarding our system.
In many questions asking whether a participant agrees with some statements we used the following five-level Likert type scale:

1. totally disagree  
2. disagree  
3. neutral  
4. agree  
5. totally agree

In the next section, where we are presenting results, we consider the first two values as disagree and last two values as agree.

For assessing the users’ knowledge we used a numerical sevenfold scale where 1 corresponds to the complete lack of knowledge and 7 means that the participant is an expert in the given discipline. By good knowledge we understand the score of 4 or more.

The duration of the user study (1 participant) was about 1 hour. 17 participants did the study separately being supervised by one interviewee, while 3 participants did the study simultaneously, being supervised by 2 interviewees. In our opinion all participants had the same condition during the study, therefore, we are presenting all the results together.

### 8.5 Results

#### 8.5.1 Task I

Figure 8.2 presents the overview of the results for this task, which are discussed in details below.

As depicted in Figure 8.3, most of the participants (85%) have chosen Java. Nobody took Drools. It indicates that in the future work on that project we have to be focused mostly on various object-oriented languages as these are most commonly used. Only 30% of the participants were able to solve the task manually (to create the code that corresponds to the UML class diagram with constraints). 70% solved the task partially. Moreover, almost half of those who completed the task partially (43%) declared their knowledge in the selected technology as very good (5+) – for those who selected Java we took into account the columns Java and Java - Bean Validation, for SQL we took the columns SQL - Queries and SQL - DDL. The understandablity of UML is declared as good (4+) the great majority of users (80%), therefore, we can eliminate the impact of misunderstanding the diagram as the main reason of not solving the task completely, especially taking into account that the diagram was very simple. As the main difficulties they mentioned:

- evaluating which constraints are available in the selected technology (e.g. the uniqueness in Java)
Figure 8.2: Task I – results

- The obtained result is what I expected
- The tool was easy to use
- The tool was easy to learn
- The tool simplifies the process of technology specific code generation
- The tool makes the process of technology specific code generation much faster
- I am satisfied with the results
- Using the tool was efficient

Legend:
- 1 - totally disagree
- 2 - disagree
- 3 - neutral
- 4 - agree
- 5 - totally agree
138 8.5. RESULTS

Figure 8.3: Task III – chosen technology

- forgetting the syntax
- forgetting details about constraints (mostly expert users)
- how to map primitive types from UML into the specific technology (e.g. `Real`)
- modeling relationships (aggregation)

What is striking is the fact that participants declaring their knowledge of the given technology at a very high level (6 or 7) were also not sure about the existence of some constraints in the given technology.

The average time users spent on writing code corresponding to the UML class diagram in the selected technology is 185 sec (with a standard deviation of 70 sec). Enriching the representation with constraints took users on average 224 sec (with a standard deviation of 73 sec). As the most time consuming aspects:

- the large number of attributes/constraints to be defined – six in the class `Student` and two (including the relationship) in the class `Study`
- typing
- creating the skeleton of the class/table
- thinking about how to model relationships

The average time people needed to solve the task automatically (using our tool) was 46 sec (with a standard deviation of 21 sec). Taking into account that they didn’t know the system before, and the operations required to transform representations take a constant amount time in relation to the size of the problem, we may definitely say that our system is efficient and saves a lot of users’ time. Moreover, all participants agreed that the obtained code meets
their expectations and they were satisfied with the results. Only 5 of them needed to slightly adjust the code by applying formatting, organizing imports, or adjusting names (for SQL linking tables). The great majority of the participants agreed that our tool solved all the difficulties that they encountered during the manual process of code creation. The only difficulty that according to the users was not solved by our tool was the generation of *unique* and *primary key* constraints for Java representation, but those constraints are not supported by the Java Bean Validation standard, therefore, we do not consider those as flaws (they would not make sense in Java).

Moreover, all participants agreed that our tool was easy to use and easy to learn (learnability). Also all the participants admitted that the tool is making the process of code generation faster and nobody disagrees that the tool simplifies the process of the technology specific code generation.

8.5.2 Task II

Figure 8.4 presents the overview of the results for this task, which are discussed in details below.

This task was solved successfully by almost all participants (90% of them knew how to define the field and 85% how to define the constraint). The reason for failing this task was misunderstanding what to do rather than not knowing how to do it. The great majority of users think that the *UnifiedOCL* syntax is intuitive (90%). Moreover, all the participants believe that they will not encounter any troubles in learning the syntax. Similarly, almost all of them (75%) think that they will remember the *UnifiedOCL* syntax required to solve this task in the future – good memorability.

8.5.3 Task III

Figure 8.5 presents the overview of the results for this task, which are discussed in details below.

All the participants knew how to transform files from the source representation into the target representation (80% totally agree, 20% agree). Moreover, we observed that almost all the participants have no doubts about the action required to execute the transformation. Those who asked for additional explanation where not sure what to do rather than how to do it. This leads to the conclusion that they remember how to use the system, which was explained in Task I. To sum up, we achieved a very good level of memorability of our system. Furthermore, participants were not making mistakes using our tool, therefore we may conclude that regarding the users’ errors, the usability of our system is very good. None of the participants said they don’t understand the concepts of constraint transformations and unified data quality representation used by our system.

The vast majority of users consider the manual comparison of two representations as time-consuming (85%). 55% consider this as not an easy task. Moreover, they see the transform-
Figure 8.4: Task II – results

Task II

- I knew how to define the new field
- I knew how to define the constraint
- The UnifiedOCL syntax is intuitive
- I believe I can learn it quickly
- I believe in the future I would not have trouble recalling how to do it again

Legend:
- 1 - totally disagree
- 2 - disagree
- 3 - neutral
- 4 - agree
- 5 - totally agree
Figure 8.5: Task III – results
ation between those two representations also as a difficult (75%) and time-consuming task (95%).

As depicted in Figure 8.6, most of the users (60%) decide to choose UnifiedOCL as the common format that would be used for the comparison of the sources. They motivated their choice as follows:

- ‘UnifiedOCL can represent all constraints whereas the Java and SQL will have some problems with certain constraints’
- this transformation is easier and more obvious
- they benefit from syntax highlighting (UDQAM markups) – better perception of constraints
- ‘to ensure unification (abstract from details)’

All those reasons prove that the UnifiedOCL has a user-friendly representations and a convenient code editor. Moreover, it was a great pleasure for us to observe users selecting the UnifiedOCL representation after just a short introduction to it in Task II, which ensures us about a good learnability of this DSL and confirms its intuitiveness. Most of the participants that selected the conversion to Java or to SQL justified their decisions by the fact that they are used to some technology (more familiar with). One of them declared that after some experience with UnifiedOCL it would be the preferred choice.

95% of the participants confirmed that our system really helps in a comparison of various constraint transformations (70% totally agree, 25% agree). The great majority of them also admitted that they believe that transformations would be helpful to solve various software engineering tasks (85%) and they would be helpful in system modeling (90%). Such a result
will justify further research in that topic, as it turned out to be considered as very useful in facilitating software engineering tasks.

Every participant agreed that the tool was efficient. And also every participant agreed that it was convenient to use. The great majority of them was satisfied with the results (85%). Those who were not satisfied were missing some sort of order of the attributes in the target representation. We decided to maintain the same order of attributes as in the source representation, but nevertheless it is one of the extensions that should be implemented as it was mentioned by 7 participants (35%).

8.5.4 Task IV

Figure 8.7 presents the overview of the results for this task, which are discussed in details below.

The first three questions were correctly answered by all the users, except one user that provided an incorrect solution (but this result could be justified by the external conditions – we were running of time booked for that task). The fourth and fifth questions turned out to be more complicated. The former were correctly answered by 75% of the participants whereas the latter one by 45%. Only one participant used the raw data report to answer some questions. All other participants preferred to use mostly bar plots instead. The spider web plots were used mostly by advanced users (those that declared their good knowledge of data quality or constraint modeling – at least one of them higher than 4).

The snapshot feature turned out to be understood very well by the users – all of them (except two) used it to answer the fourth question. This was the most natural and simple way to find the answer to this question.

Interestingly we got very contradicting results regarding the overall assessment of the Data Quality Visualizer between users with strong industrial background (and no current academic background) – 40% and all the other participants – 60%. All the participants from the second group found our tool easy to use and all except one (neutral answer) easy to learn. While the members of the much smaller, strongly industrial group had slightly different opinions – 62.5% of that group think our tool is easy to learn and 75% think it is easy to use. This could be justified by the fact that for some reason also those people found it more difficult to get the answers. From all the other participants only 1 of them (8%) encountered troubles in finding answers, 17% were not decided (neutral answer). Finally nobody from both groups questioned the statement that our tool informs well about data quality. Also almost all participants (except two that answered ‘neutral’) confirmed that they understood the plots.

To sum up, the Data Quality Visualizer informs very well about the data quality and is understandable for the users. Most of the participants found it easy to use and easy to learn. For some reason, for only the small group of exclusively and strong industrial background, the user interface turned out not be easy to use. Therefore, the future work on our Data Quality
Figure 8.7: Task IV – results
Visualizer should be focused on user interface optimization for required by an industry.

8.6 User feedback

Besides what was discussed above, we got from users also a general feedback about what they think about our system, what additional features and extensions they would appreciate and what they would change. In this section we are presenting users’ comments grouped into several categories and in some cases our opinions how those features could be implemented.

Constraint transformations

- currently every transformation between two specific technologies consists of two steps. To make the transformation even simpler for the user it would be a good idea to hide the conversion to UnifiedOCL from users. We cannot completely hide the UnifiedOCL representation as this is used for multiple purposes but we can add a custom action for a direct transformation between two representations.

- for private fields of the classes, in order to make them accessible for Java Bean validation, one needs to provide getters. Those getters are cluttering the code. It is possible to use the Project Lombok\(^2\), which allows to use annotations @Getter to automatically generate the getter in the byte code. The source file is not cluttered by the boiler-plate code.

- relationships between UML classes should be represented in Java by more general concepts, e.g. a Collection instead of a Set

- in case of the transformation to SQL it would be a good idea to show a summary dialog with the linking tables, linking columns that were generated, recognized foreign keys etc. Thus, the user would not have to check the sources manually after the generation, whether they meet their expectations. We also recognized this problem and we also consider this as a very reasonable extension which is described in Section 9.3.

- currently we hardcoded how types from various representation are translated (using dictionaries – Subsection 6.17.6) – participants mentioned that they would like to customize also those dictionaries

Development environment

- every developer/company may use different code formatting rules, thus, it would be a good idea to use the Eclipse code formatter (customized by a particular user) to format the generated code

- various users may also want to organize imports in the Java code according to their preferences. It would be a better idea to use also Eclipse Save Action for organizing imports after the code has been generated.

• instead of showing a confirmation window after the successful code generation, for some users it would be better to open the generated source. We think it would make more sense to make it configurable via Eclipse Preferences whether the successful transformation should be confirmed by the dialog window or opening the generated source file (in some cases more than just one source file is generated, e.g. Java classes are stored in separate files).

• keyboard shortcuts would make the transformation process even faster for advanced users

• various transformations available in the context menu could be grouped in a submenu

Data Quality Visualizer

• there should be an indicator whether there has been any snapshot already recorded, and whether the Make snapshot action was successful

• use Color Brewer 2.0 for color schemes for charts

8.7 Summary

Our system was welcomed very warmly by our participants. They appreciated its utility when it comes to solve software engineering tasks, especially in the context of the model driven development. According to the users’ opinion it really helps with a complex and difficult code transformations in the software, and they are satisfied with the results it produces. Our system has met with a positive reception from the industry. We believe that the feedback we received from those people would allow us to polish the development environment for unified constraint representations and transformations to turn it into a commonly used product.

Moreover, after gathering the results from particular tasks we may say that the usability of our system is at a very high level. Users considered our system as easy to use and easy to learn and very efficient.

Finally, the observation of the way how people use the system and what they expect out of it would help us to set the future goals and research directions in the domains of data quality and constraint modeling.

9

Future Work

In this chapter we are presenting ways how our system could be enhanced. We are discussing
some limitations and ways how to overcome them. Future work and the possible extensions
discussed here, cover both conceptual and implementational issues.

9.1 More representations, more constructs, more constraints

The first and the most obvious extension of our system is to increase the number of supported
technology specific representations. E.g. we may implement a new plugin to support
constraints defined in Ruby on Rails (Active Record Validations\(^1\)).

We may also increase the number of supported constructs in the technologies that our system
already covers (Java, SQL, Drools). E.g. at present constraints for methods (parameters
and return types) are transformed only for Java Bean Validation representation. But they
can also be expressed in SQL – either as triggers or just as normal functions/procedures.
Implementing this feature for a *UnifiedOCL* to SQL transformation would make the system
more complete, allowing for constraint definitions also for the dynamic aspects of particular
technologies (static is related to the data structure).

Finally we can increase the number of supported constraints (list of currently supported
constraints is available in Appendix B and in Section 6.9). So far we have been focused on
implementing in our system almost all the constraints that are defined by the standards in
technologies that we had chosen i.e. Bean Validation + Hibernate Validator extension for
Java, PostgreSQL table schema definition constraints. The constraints that are not covered
yet is for instance the Brazilian extension of Hibernate Validator. Moreover, we can create
constraints that are not part of standards but in our opinion could be useful, e.g. *equals*
constraint that would try to ensure that the value of an attribute may take either the specified

\(^{1}\text{http://guides.rubyonrails.org/active_record_validations.html [Online, accessed 19-August-2015]}\)
value or null.

We believe that all the enhancements described above would move our system closer to the industrial standards that could be adopted in various software engineering projects.

9.2 UDQAM extensions

As we stated in Subsection 3.7.2 currently we are supporting the UDQAM constraint representation just for structural features i.e. attributes and references. UDQAM was designed to be powerful and extensible, keeping at the same time the representation very compact. Therefore, it could be used as well in every location where the constraints may occur. Thus, we may extend the current syntax of UnifiedOCL by introducing also the UDQAM markups in various locations: they may refer to the whole class, they may constraint method parameters or returned values, etc. This would allow a user to take benefit from the very simple and validator independent constraint definitions.

9.3 GUI extensions

To make our system more user friendly we can also extend the existing GUI with various features, e.g. wizards or graphical configurators. For instance a wizard for the SQL to UnifiedOCL transformation could be created. The transformation from SQL to UnifiedOCL involves execution of a lot of heuristics for discovering the underlying model. For instance, linking tables should be discovered. Instead of generating special classes that would represent linking tables, they should be mapped into references in the classes corresponding to the linked relations. Another example are the foreign keys. When composed foreign keys from one table correspond to all parts of the primary keys for the other tables, they should be recognized as object references in UnifiedOCL. Unfortunately, there might exist cases when the automatic generation of UnifiedOCL from SQL would bring undesired results. Usually, a user is aware that those situations are the exceptions when the heuristic is incorrect. For instance the linking table is not a linking table but a simple structure containing references to two entities and nothing else. The process of transformation consists of two steps. First the model is discovered and second the target model is generated using the intermediary models, converted constraints etc. In order to allow a user to fix potential mistakes in the generated UnifiedOCL representation, in between of those two steps, the discovered model should be displayed to the user. Moreover, it should be allowed for a user to edit conflicting/incorrect relations. Luckily, the discovered model is stored as small objects that could be easily mapped into e.g. Java Swing JTables\(^2\) which a user can edit. This would be a great improvement increasing the flexibility of our system.

9.4 Backend extensions

Presently we are storing UDQAM dictionaries (pluggable into the UnifiedOCL grammar) as JSON files. This approach could be also extended into other types of dictionaries discussed in Subsection 6.17.6. Moreover, we may consider storing them in the database. The database storage would enhance the flexibility of configuration and separate the dictionaries from the transformation implementation. Furthermore, adding new constraints would not involve changes in the code and can be performed with the support of graphical RDBMS clients. Due to portability issues it would be desired not to rely on any external heavyweight RDBMSs, therefore either SQLite\(^3\) or possibly a NoSQL\(^4\) database such as MongoDB\(^5\) usage would be recommended.

9.5 Editor extension

Currently the UnifiedOCL editor supports various features making the code creation more convenient, e.g. an outline view or code validation. We stated that the UnifiedOCL is the core part of our system, therefore it should be really polished. One of the improvements would be e.g. a code auto-formatting. Xtext generates the stub for the formatter that is recognized by the Eclipse editor. A user pressing CTRL+SHIFT+F can get the source formatted. This would be a nice feature making the work with UnifiedOCL within the Eclipse editor much more comfortable.

9.6 Partial generation

Presently in a single action we are generating all the target representation artifacts for the selected source representation. The newly generated files override the existing ones. Therefore, the generation process adheres to the rule all or nothing. However, it might be useful to regenerate just parts of the target representation, preserving some existing pieces of code in the target representation. This could be achieved by annotating parts of the source code specification or the target specification. It should also be possible to annotate the intermediary representation – UnifiedOCL (all transformations consist of two steps: a transformation from the source representation into UnifiedOCL and then next from the UnifiedOCL into the target representation). We may also display a wizard allowing to select what should be transformed or what should be preserved (remained untouched).

9.7 Not only constraints

We may also extend our system by handling features that are not really constraints but are very similar to them from the representation point of view. For example we may consider

\(^3\)https://www.sqlite.org/ [Online, accessed 21-August-2015]
supporting Java Persistence API (JPA)\(^6\) annotations in our system. The essential constraints that one can define using SQL are integrity constraints such as PRIMARY KEY or UNIQUE. However, in the object oriented languages they make no sense. The UnifiedOCL language is designed to be versatile, supporting various different programming paradigms including object-oriented and relational. Therefore, a user should be able to specify those constraints to be able to generate a correct SQL database schema. On the other hand, currently those constraints are not supported during conversions to Java as they have no correspondence in Bean Validation. But what one may want to do is the mapping from Java Beans to a relational schema using JPA. Therefore, UnifiedOCL may be used to generate at least a subset of JPA specification annotations along with the Java classes. This may include:

- No argument constructor
- Marking persistent classes with @Entity
- @Id
- @GeneratedValue (corresponds to SERIAL type from PostgreSQL)
- @UniqueConstraint
- @Embedded (contained attribute)
- @Temporal

9.8 New paradigms

Future research in the field of unified constraint representations should naturally tend to elaborate the term of constraint, and study it in various contexts. Following this idea we believe that it would be interesting to expand the UnifiedOCL transformation/representation capabilities to new paradigms – e.g. functional programming. Another possibility is to combine the concept of unified constraint representation with design by contract programming \(^6\), which in essence uses constraints internally as preconditions/postconditions or invariants. For instance Clojure\(^7\) language has a native support for the contract programming\(^8\) and at the same time it is a functional language. Therefore, it would be a significant contribution to explore the UnifiedOCL relevance for such a language or, in general, for design by contract concepts. The significant difference is the fact that instead of defining constraints over data (values) or data schema, one can define constraints over the code execution flow.

9.9 Integration with external tools

DresdenOCL is an example of a toolkit that can do some work for us regarding the OCL transformation. We believe that there is no need to reinvent the wheel. Therefore, wherever


\(^7\)http://clojure.org/ [Online, accessed 19-August-2015]

possible we would like to include into our project the results of research of other authors. For instance a tool described in [10], which is publicly available\(^9\), can be used to transform complex SQL expressions into the OCL expressions. It could be integrated with our system enriching its capabilities of constraint transformations.

9.10 Data quality mapping vs schema mapping

In our system values of data quality dimensions are computed using violations of constraints by some entities and a mapping file. In other words we are providing a feedback about the runtime system data quality – or we can say at the instance level. 

Another approach would be to provide a data quality feedback using just the data schema. We can call this approach the offline data quality validation. The idea is to define kind of a meta-schema with meta-constraints which would be the generalizations of the data structures and constraint concepts and could be used to validate a schema. We believe that a schema with defined constraints may lead to achieve a better data quality. But what we are implicitly assuming is the fact that constraints are correctly specified in the schema. The schema validation would make this point explicit. Below we are providing an example that clearly justifies the need of schema validation:

The European Aviation Safety Agency (EASA) is the agency of the European Union (EU) that carries out the surveillance of all the airlines that conduct their businesses in the EU member states. It is responsible for supervising the implementation of safety regulations by the airlines, certifying particular aircrafts or their types and monitoring airlines’ activities within its jurisdiction area. For the sake of the following example we assume that it might also collect more confidential/detailed data that it is in fact entitled to.

In order to effectively carry out its mission, EASA demands that each air carrier has to periodically (on a regular basis) send various data sets which are then processed and analyzed by EASA. Imagine that EASA requires each airline to report information about various aspects of their operational activities. These may include data about financial conditions, detailed technical data about their fleets or flight connections. As each air service provider is different, in order to be able to extract some information from the reports, the data should conform to the same rules regarding the data quality. E.g. any information about the aircraft mileage should be represented with an accuracy to 3 decimal places. Let’s assume that besides processing the data, the agency also stores the data as part of the Online Analytical Processing (OLAP) system. Therefore, the most desired format of the reported data sets is the SQL (or any other database) script that allows to import the data. EASA does not enforce each airline to use the same data schema (this might be inappropriate due to their complex and unique structures and business models), but imposes restrictions on the particular data characteristics. Moreover, EASA demands from all the airlines to provide the

annual summary in the elaborated, compiled and printable form. Also here there
are no limitations with regards to the form but the presented content has to be
consistent with the standardized data quality specifications. Below are samples
of the requirements of the data quality:

• Each entity type should be uniquely identified by an integer identifier
• Any time data should be represented with an accuracy of 3 decimal places
• Any speed should be represented with an accuracy of 2 decimal places
• A report should include only past flights (e.g. not include the flights that
  are in progress at the time of the report generation)
• Personal information about pilots must be complete i.e. include a birth
date, a full name and surname, the unique license number, an address
• Any description cannot be longer than 250 signs
• Flight numbers (identifiers) must conform to the IATA (International Air
  Transport Association) format
• Financial data has to include only fully settled transactions
• Naming conventions adhere to common sense rules – e.g. the time of the
  beginning of the flight should be named startTime or beginTime but not
  AGryh?tIK

In order to enforce those data quality requirements, EASA may provide a kind
of a meta-schema with meta-constraints that could be used by various airlines to
validate their schemas or the generated reports that would be passed on to EASA.
During this thesis we managed to explore various aspects related to the problems of data quality in software engineering. We were primarily focused on the model driven development approach in constraint representation and transformation, that would lead to achieve the unified data quality, independent of the chosen specific technology. Moreover, we wanted to obtain some general information about data quality based on constraint violations.

First of all, we managed to create the unified data quality requirements (constraints) representation – UnifiedOCL. It can capture concepts related to the data structure (model) and constraints defined on it. The model, based on OCLinEcore, was designed to be expressive enough to cover most concepts from the UML class diagram. UnifiedOCL is powerful enough to express a multitude of constraints by making their specification possible in a traditional way by using OCL, but also in an newly introduced compact form of UDQAM labels. In that way, a developer can benefit from powerful OCL constructs to define constraints on complex interrelated objects but also to specify constraints that cannot be expressed using this language due to the limitations of its available constructs. Moreover, developers can specify also constraints that are too difficult/complex to be expressed using OCL, but where some validators/algorithms are capable of evaluating them (validator independence). The UDQAM representation can be extended by a set of predefined constraints grouped into dictionaries. As a consequence, a developer may benefit from theoretically unlimited capabilities to specify software constraints. UnifiedOCL together with UDQAM is powerful enough to cover concepts from various programming paradigms, e.g. imperative or declarative. A user can use our representation when working within object-oriented domain (Java), relational databases (PostgreSQL) or business rules (Drools). Our proof-of-concept implementation for those three sample technologies was very warmly welcomed by users which considered our representation as very convenient and intuitive. The human-readable syntax of our representation, implemented as DSL, allows for defining constraints in a single
location. Afterwards, this single, central constraint definition could be used to generate various technology specific representations of constraints, ensuring their consistency. In that way, the development of multi-layer, distributed information systems is getting a bit easier. In that sense we may consider our representation as a constraint repository, or a framework for technology independent constraint definition. Our system conforms to the model driven development approach, making the programming of new systems much faster and more convenient. Moreover, our toolkit is not limited to the three mentioned representations, but can be easily extended, by means of plugins, with new representations, e.g. JavaScript.

Second, we investigated technical aspects of constraint transformations between various representations. We investigated available approaches, technologies and tools to find a good combination that will allow for a straightforward implementation of various transformations. Moreover, we discussed a general architecture of transformations, elaborating common steps required to construct a code generator transforming a source representation into a target one. We implemented all the transformations (along with the UnifiedOCL representation) as a set of Eclipse plugins. The toolkit, evaluated also during the user study conducted by us, turned out to be easy to use and easy to learn. Our participants assessed its usability as very good. The results produced by our system satisfied expectations of users. According to their opinion it really facilitates various software engineering tasks, therefore, a further investigation of that topic is absolutely desired. Moreover, the conducted user study confirmed that the problems, that our system is trying to solve, are of high importance and our current efforts constitute a good basis for turning such a tool into a widely used product.

Finally, we extended the concept of data quality indicators by merging it with the concept of data quality dimensions. We defined a way how to map constraints into data quality dimensions using their names and locations within the model. In that way, the results of a model validation (constraint violations) together with the mapping defined by a user can be processed to provide a user with feedback about various aspects of data quality of the considered data instances. Moreover, we created a Data Quality Visualizer, which presents the computed values for various data quality dimensions in a graphical way. Therefore, we made a step forward towards a better management of data quality problems in information systems – a user can get insight into data that is stored or processed by a particular system. The conducted user study proved that potential users can obtain the general overview of the quality of data used by their system. Moreover, they may find answers for detailed, real life data quality issues using our Data Quality Visualizer. This all together may help them to minimize operational costs and increase reliability of their systems by properly taking care of the quality of data they are using/processing.

The promising results of our user study are encouraging to further investigation of the topics described in this thesis. We are convinced that our conceptual and technological investigation together with a valuable feedback obtained from the participants will contribute to the progress in the data quality research activities. Moreover, we believe that the findings of this thesis would help to solve various data quality issues, which in turn, would increase the reliability and usefulness of various information systems.
A.1 UnifiedOCL grammar

```plaintext
// The UnifiedOCL grammar is based on OCLInEcore.xtext:
//
// <copyright>
//
// Copyright (c) 2010, 2011 E.D. Willink and others.
// All rights reserved. This program and the accompanying materials
// are made available under the terms of the Eclipse Public License v1.0
// which accompanies this distribution, and is available at
//
// Contributors:
// E.D. Willink - initial API and implementation
// E.D. Willink (Obeo) - Bug 416287 - tuple-valued constraints
//
// <copyright>
//
grammar ch.ethz.unifiedocl.UnifiedOCL with org.eclipse.ocl.examples.xtext.essentialocl.EssentialOCL
import "http://www.eclipse.org/emf/2002/Ecore" as.ecore
```
import "http://www.eclipse.org/ocl/3.1.0/BaseCST" as base
import "http://www.eclipse.org/ocl/3.1.0/EssentialOCLCST" as essentialocl
import "http://www.eclipse.org/ocl/3.1.0/Pivot" as pivot
import "http://www.ethz.ch/unifiedocl/UnifiedOCL"
// generate unifiedOCL "http://www.ethz.ch/unifiedocl/
 UnifiedOCL"

UniOCLModule:
{UniOCLModule} (ownedNestedPackage+=UniOCLPackage)*
;

PrimitiveTypeIdentifier:
  'Boolean'
  | 'Integer'
  | 'Real'
  | 'String'
  | 'UnlimitedNatural'
  | 'Date'
  | 'void'
  | 'OclAny'
  | 'OclInvalid'
  | 'OclVoid'
;

UniOCLVisibility returns UniOCLVisibility:
  'public'
  | 'protected'
  | 'private'
  | 'package'
;

UniOCLAggregation returns UniOCLAggregation:
  'none'
  | 'composite'
  | 'shared'
;

UniOCLConcurrency returns UniOCLConcurrency:
  'sequential'
  | 'guarded'
  | 'concurrent'
;

UniOCLParameterDirection returns UniOCLParameterDirection:
  'in'
  | 'out'
  | 'inout'
;

UniOCLOperationDirection returns UniOCLOperationDirection:
  'in'
  | 'out'
  | 'inout'
;

UniOCLAggregationType returns UniOCLAggregationType:
  'none'
  | 'composite'
  | 'shared'
;

UniOCLParameter returns UniOCLParameter:
  'in'
  | 'out'
  | 'inout'
;

UniOCLParameterDirection returns UniOCLParameterDirection:
  'in'
  | 'out'
  | 'inout'
;

UniOCLProperty returns UniOCLProperty:
  'NOT'
  | 'true'
  | 'false'
;

UniOCLParameterKeyword returns UniOCLParameterKeyword:
  'in'
  | 'out'
  | 'inout'
;

UniOCLTypeKeyword returns UniOCLTypeKeyword:
  'void'
  | 'OclAny'
  | 'OclInvalid'
  | 'OclVoid'
;

UniOCLTypeIdentifier returns UniOCLTypeIdentifier:
  'Boolean'
  | 'Integer'
  | 'Real'
  | 'String'
  | 'UnlimitedNatural'
  | 'Date'
  | 'void'
  | 'OclAny'
  | 'OclInvalid'
  | 'OclVoid'
;

UniOCLVisibility returns UniOCLVisibility:
  'public'
  | 'protected'
  | 'private'
  | 'package'
;

UniOCLAggregation returns UniOCLAggregation:
  'none'
  | 'composite'
  | 'shared'
;

UniOCLConcurrency returns UniOCLConcurrency:
  'sequential'
  | 'guarded'
  | 'concurrent'
;

UniOCLParameterDirection returns UniOCLParameterDirection:
  'in'

// generate unifiedOCL "http://www.ethz.ch/unifiedocl/UnifiedOCL"
UniOCLTypeQualifier:
   'abstract'
   | 'final'
   | 'static'

UniOCLAttributeQualifier:
   'static'
   | 'const'

UniOCLBuiltInConstraintForUMLAttribute:
   'derived'
   | 'ordered'
   | 'readonly'
   | 'unique'

UniOCLBuiltInConstraintForUMLOperation:
   'query'
   | 'ordered'
   | 'unique'

UniOCLBuiltInConstraintForUMLParameter:
   'ordered'
   | 'unique'

StringLiteral:
   SINGLE_QUOTED_STRING | 'now'

UniOCLCustomConstraintParameterLiteral:
   Identifier '=' ( StringLiteral | NUMBER_LITERAL | 'true' | 'false' | Identifier )

UniOCLCustomConstraint returns UniOCLCustomConstraint:
   name=UniOCLCustomConstraintIdentifier ('(' ( value+= UniOCLCustomConstraintParameterLiteral ',' ')'+')' )? ( dqm= DQConstraintMapping )?
UniOCLCustomConstraintIdentifier:
  Identifier | 'null' | 'default'
;
UniOCLPackage returns UniOCLPackage:
  (visibility=UniOCLVisibility)?
  'package' name=Identifier
  '{ (ownedType+=ClassifierCS)* }'
;
ClassifierCS returns base::ClassifierCS:
  UniOCLClass
  | UniOCLInterface
  | UniOCLEnumeration
  | UniOCLPrimitive
  | UniOCLException
;
UniOCLClass returns UniOCLClass:
  (visibility=UniOCLVisibility)?
  (qualifier+=UniOCLTypeQualifier)*
  'class' name=Identifier
  (specializes ownedSuperType+=TypedRefCS (',')
   ownedSuperType+=TypedRefCS)*)?
  (realizes implementedInterfaces+=TypedRefCS (',')
   implementedInterfaces+=TypedRefCS)*)?
  '{
   (ownedOperation+=UniOCLOperation
    | ownedProperty+=StructuralFeatureCS
    | ownedConstraint+=UniOCLInvariant)*
  }'
;
UniOCLInterface returns UniOCLInterface:
  (visibility=UniOCLVisibility)?
  (qualifier+=UniOCLTypeQualifier)*
  'interface' name=Identifier
  (specializes ownedSuperType+=TypedRefCS (',')
   ownedSuperType+=TypedRefCS)*)?
  (realizes implementedInterfaces+=TypedRefCS (',')
   implementedInterfaces+=TypedRefCS)*)?
  '{
   (ownedOperation+=UniOCLOperation
    | ownedConstraint+=UniOCLInvariant)*
  }'
UniOCLEnumeration returns UniOCLEnumeration:
   (visibility = UniOCLVisibility)?
   'enum' name = Identifier
   '{'
   (ownedLiterals += EnumerationLiteralCS)*
   '}'
   ;

EnumerationLiteralCS returns base::EnumerationLiteralCS:
   'literal' name = Identifier ';' ;

UniOCLPrimitive returns UniOCLPrimitive:
   (visibility = UniOCLVisibility)?
   'type' name = Identifier
   ;

UniOCLException returns UniOCLException:
   'exception' name = Identifier
   ;

StructuralFeatureCS returns base::StructuralFeatureCS:
   UniOCLAttribute | UniOCLReference
   ;

UniOCLAttribute returns UniOCLAttribute:
   (visibility = UniOCLVisibility)?
   (qualifier += UniOCLAttributeQualifier)*
   'attribute' name = Identifier ';' ownedType =
   TypedMultiplicityRefCS
   ('{'
   (constraints += UniOCLCustomConstraint ',', '?') |
   (labels += UniOCLBuiltInConstraintForUMLAttribute ',', '?') |
   (aggregation = UniOCLAggregation ',', '?')
   )+ '
   (''
   'default' Identifier '?';' (ownedDefaultValueExpression=ExpSpecificationCS '?';')* '}' )? ';
   ;

UniOCLReference returns UniOCLReference:
   (visibility = UniOCLVisibility)?
   (qualifier += UniOCLAttributeQualifier)*
UniOCLOperation returns UniOCLOperation:
  (visibility=UniOCLVisibility)?
  (qualifier+=UniOCLTypeQualifier)*
  'operation' name=Identifier
  (ownedParameter+=UniOCLParameter (',', ownedParameter+=UniOCLParameter)*)?
  ('throws' ownedException+=TypedRefCS (',', ownedException+=TypedRefCS)*)?
  ('default' Identifier? ':': (ownedDefaultExpression+=ExpSpecificationCS ';')*)?

UniOCLParameter returns UniOCLParameter:
  name=Identifier
  (direction=UniOCLParameterDirection)? (ownedType=TypedMultiplicityRefCS)?
  ('{(' (types+=UniOCLBuiltInConstraintForUMLParameter (',', types+=UniOCLBuiltInConstraintForUMLParameter)*)
      (label=UniOCLBuiltInConstraintForUMLParameter 'default' Identifier? ':': (ownedDefaultExpression+=ExpSpecificationCS ';')*)
      ('{(' (concurrent=UniOCLConcurrency (',', concurrent=UniOCLConcurrency)*)
      (constraints+=UniOCLCustomConstraint (',', constraints+=UniOCLCustomConstraint)*)
      )+ '})')

UniOCLInvariant returns UniOCLInvariant:
  stereotype='invariant' (name=Identifier)? '': specification
  =ExpSpecificationCS (dqm=DQConstraintMapping)? ';'
UniOCLPrecondition returns base::ConstraintCS:
  stereotype='precondition' (name=Identifier)? ':'
  specification=ExpSpecificationCS ';'
;
UniOCLPostcondition returns base::ConstraintCS:
  stereotype='postcondition' (name=Identifier)? ':'
  specification=ExpSpecificationCS ';'
;
ExpSpecificationCS returns essentialocl::ExpSpecificationCS:
  ownedExpression=ExpCS
;
TypedRefCS returns base::TypedRefCS:
  TypeLiteralCS | TypedTypeRefCS;

TypedMultiplicityRefCS returns base::TypedRefCS:
  TypedRefCS (multiplicity=MultiplicityCS)?;

TypedTypeRefCS returns base::TypedTypeRefCS:
  pathName=PathNameCS;

DQConstraintMapping returns DQConstraintMapping:
  '' mappings+=DQDimensionMapping '' mappings+=DQDimensionMapping) * '' ]'';

DQDimensionMapping returns DQDimensionMapping:
  name=(DQBuiltInDimensions|ID) weight=DQWeight
;

DQBuiltInDimensions:
  'Accuracy'
  | 'Completeness'
  | 'Currency'
  | 'Timeliness'
  | 'Volatility'
  | 'Consistency'
  | 'Integrity'
  | 'Interpretability'
  | 'Accessibility'
  | 'Reliability'
  | 'Portability'
  | 'Appropriateness'
A.2 Data Quality Mapping grammar

```plaintext
grammar ch.ethz.unifedocl.dqm.DQM with org.eclipse.xtext.
    common.Terminals

generate DQM "http://www.ethz.ch/unifedocl/dqm/DQM"

DataQualityMapping:
    definitions+=ConstraintDefinition*;

ConstraintDefinition:
    location=ID name=ID ']' mappings+=ConstraintMapping (',',
    mappings+=ConstraintMapping)* ']' ;

ConstraintMapping:
    name=(BuiltInDQDimensions|ID) weight=WEIGHT
;

BuiltInDQDimensions:
    'Accuracy'
    | 'Completeness'
    | 'Correctness'
    | 'Currency'
    | 'Timeliness'
    | 'Volatility'
    | 'Consistency'
    | 'Integrity'
    | 'Interpretability'
    | 'Accessibility'
    | 'Reliability'
    | 'Portability'
    | ' Appropriateness'
;

terminal WEIGHT:
    ('.0.(INT)+)|('.1.0')
;
```

DQWeight returns ecore::EDouble:
    NUMBER_LITERAL
;
List of supported constraints

In this appendix we are presenting the list of all constraints supported by our system originating from Java Bean Validation, Hibernate Validator and PostgreSQL.
Table B.1: Bean Validation – supported constraints

<table>
<thead>
<tr>
<th>Bean Validation Annotation</th>
<th>UDQAM label</th>
<th>UDQAM Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>@AssertTrue</td>
<td>asserttrue</td>
<td></td>
</tr>
<tr>
<td>@AssertFalse</td>
<td>assertfalse</td>
<td></td>
</tr>
<tr>
<td>@DecimalMin</td>
<td>decimalmin</td>
<td>value (STRING)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inclusive (BOOLEAN)</td>
</tr>
<tr>
<td>@DecimalMax</td>
<td>decimalmax</td>
<td>value (STRING)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inclusive (BOOLEAN)</td>
</tr>
<tr>
<td>@Min</td>
<td>decimalmin</td>
<td>value (INTEGER)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inclusive (BOOLEAN)</td>
</tr>
<tr>
<td>@Max</td>
<td>decimalmin</td>
<td>value (INTEGER)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inclusive (BOOLEAN)</td>
</tr>
<tr>
<td>@Size</td>
<td>size</td>
<td>min (INTEGER)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>max (INTEGER)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inclusive (BOOLEAN)</td>
</tr>
<tr>
<td>@Future</td>
<td>future</td>
<td></td>
</tr>
<tr>
<td>@Past</td>
<td>past</td>
<td></td>
</tr>
<tr>
<td>@Digits</td>
<td>digits</td>
<td>integer (INTEGER)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fraction (INTEGER)</td>
</tr>
<tr>
<td>@Pattern</td>
<td>pattern</td>
<td>regexp (STRING)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>flag (STRING) – partially supported</td>
</tr>
<tr>
<td>@Null</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>@NotNull</td>
<td>notnull</td>
<td></td>
</tr>
</tbody>
</table>
### Table B.2: Hibernate Validator – supported constraints

<table>
<thead>
<tr>
<th>Hibernate Validator Annotation</th>
<th>UDQAM label</th>
<th>UDQAM Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>@NotEmpty</td>
<td>notempty</td>
<td></td>
</tr>
<tr>
<td>@NotBlank</td>
<td>notblank</td>
<td></td>
</tr>
<tr>
<td>@Length</td>
<td>length</td>
<td>min (INTEGER) max (INTEGER)</td>
</tr>
<tr>
<td>@Range</td>
<td>range</td>
<td>min (INTEGER) max (INTEGER)</td>
</tr>
<tr>
<td>@Email</td>
<td>email</td>
<td></td>
</tr>
<tr>
<td>@CreditCardNumber</td>
<td>creditcard</td>
<td>ignoreNonDigitCharacters (BOOLEAN) – partially supported</td>
</tr>
<tr>
<td>@EAN</td>
<td>can</td>
<td>type (INTEGER)</td>
</tr>
<tr>
<td>@URL</td>
<td>url</td>
<td>protocol (STRING) host (STRING) port (STRING) regexp (STRING) – partially supported flags (STRING) – partially supported</td>
</tr>
</tbody>
</table>

### Table B.3: SQL (PostgreSQL) – supported constraints

<table>
<thead>
<tr>
<th>SQL constraint</th>
<th>UDQAM label or UnifiedOCL concept</th>
<th>UDQAM Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIMARY KEY</td>
<td>primarykey</td>
<td></td>
</tr>
<tr>
<td>FOREIGN KEY REFERENCES</td>
<td>foreignkey</td>
<td>table (IDENTIFIER) column (IDENTIFIER)</td>
</tr>
<tr>
<td>NOT NULL</td>
<td>notnull</td>
<td></td>
</tr>
<tr>
<td>NULL</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>UNIQUE</td>
<td>unique</td>
<td></td>
</tr>
<tr>
<td>SERIAL</td>
<td>autoincrement</td>
<td></td>
</tr>
<tr>
<td>column type NUMBER with defined precision of numbers</td>
<td>digits</td>
<td>integer (INTEGER)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>fraction (INTEGER)</td>
</tr>
<tr>
<td>column type VARCHAR with defined max length</td>
<td>length</td>
<td>max (INTEGER)</td>
</tr>
<tr>
<td>CHECK</td>
<td>constraints that covers most of the corresponding concepts from Bean Validation and Hibernate Validator</td>
<td>see UDQAM labels from Table B.2 and Table B.1</td>
</tr>
</tbody>
</table>
C.1 Task I

Figure C.1: Task I – UML class diagram with data model
C.2 Task II

Using the *UnifiedOCL* representation add to the class Student a field to reflect the number of courses he/she can attend and define a constraints to denote that the student cannot attend more than 10 courses.

```java
package evaluation{

    class Student
    {
        attribute name : String { notnull length(max=20)};
        attribute surname : String { notnull length(max=30) } ;
        attribute birthDate : Date { past } ;
        attribute legiValid : Boolean { asserttrue } ;
        attribute legiNumber : Integer { unique primarykey} ;
        attribute gpa : Real { decimalmin(value=4) decimalmax(value=6)} ;
    }

    class Study
    {
        attribute description : String { notblank };
        reference participants : Student[0..*];
    }
}
```

Figure C.3: Task II – UnifiedOCL code to be enriched
C.3 Task III

```java
package task3a;

import java.util.Date;
import javax.validation.constraints.*;
import org.hibernate.validator.constraints.Length;

class Flight {
    public Integer id;
    @Future
    public Date startTime;
    @AssertTrue
    public Boolean isCivil;
    @Past
    public Date endTime;
    public Integer number;
    @Pattern(regexp = "^[X\w]+$")
    public String name;
    @Length(max = 100)
    public String description;
    @DecimalMax(value = "125.75")
    public Double cost;
    public Pilot pilot;
}

class Pilot {
    public Integer licenceNumber;
}
```

Figure C.4: Task III – Java code

```sql
CREATE TABLE Flight (  
id INTEGER,  
startTime DATE CHECK (startTime>NOW()),  
isCivil BOOLEAN CHECK (isCivil=true),  
endTime DATE CHECK (endTime<NOW()),  
number INTEGER,  
name VARCHAR CHECK (name LIKE('^[X\w]+$')),  
description VARCHAR(100) CHECK (length(description)<100),  
cost REAL CHECK (cost<125.75),  
pilot INTEGER,  
UNIQUE (pilot),  
FOREIGN KEY (pilot) REFERENCES Pilot(licenceNumber)
);
CREATE TABLE Pilot (  
licenceNumber INTEGER UNIQUE
);
```

Figure C.5: Task III – SQL code
C.4 Forms
Achieving Unified Data Quality Representation by Constraints Transformation - Background Information

*Required

1. **Gender** *
   
   *Mark only one oval.*
   
   - female
   - male

2. **Year of birth** *
   
   *e.g. 1989

3. **Are you enrolled in a study programme at the moment?** *
   
   *If you are enrolled in more than one please check the highest level of studies
   *Mark only one oval.*
   
   - Yes, Bachelor in Computer Science
   - Yes, Master in Computer Science
   - Yes, Doctorate in Computer Science
   - No
   - Other: .................................................................

4. **Are you working?** *
   
   *Mark only one oval.*
   
   - Yes, Full time
   - Yes, Part time
   - No
   - Other: .................................................................

5. **What is the highest level of education you have achieved?** *
   
   *Mark only one oval.*
   
   - Bachelor in Computer Science
   - Master in Computer Science
   - Doctorate in Computer Science
   - Other: .................................................................
6. **How would you rate your skills in the following topics?** *

*Mark only one oval per row.*

<table>
<thead>
<tr>
<th>Topic</th>
<th>1 - never heard, never tried</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7 - expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraints modeling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Quality in Information Systems</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model Driven Engineering (MDE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UML (only class diagram)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object Constraint Language (OCL)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java - Bean Validation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL - Queries</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL - DDL (table schema definition etc.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Drools</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eclipse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7. **What IDE do you use most regularly?** *

*Put the name of the IDE such as Eclipse, NetBeans, IntelliJ IDEA, ...*
Achieving Unified Data Quality Representation by Constraints Transformation - Task 1

*Required

1. **Chosen technology**
   
   *Mark only one oval.*
   
   - [ ] Java
   - [ ] SQL
   - [ ] Drools

**Task solved manually**

2. **Were you able to solve the task?** *
   
   *Mark only one oval.*
   
   - [ ] yes
   - [ ] partially
   - [ ] no

3. **What part weren't you able to solve?** (If the answer to the previous question was not yes)

   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………

4. **What was the most time consuming?**
   
   e.g. specification of all properties of the class

   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
   …………………………………………………………………………………………………………………
5. **What was the most difficult?**
   e.g. relationships mapping

6. **How would you simplify the process?**

---

**Task solved using the tool**

**Task solved using the tool**

7. **Do you agree with the following statements?** *
   *Mark only one oval per row.*

<table>
<thead>
<tr>
<th>Statement</th>
<th>1 - totally disagree</th>
<th>2 - disagree</th>
<th>3 - neutral</th>
<th>4 - agree</th>
<th>5 - totally agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>The obtained result is what I expected</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The tool was easy to use</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The tool was easy to learn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The tool simplifies the process of technology specific code generation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The tool makes the process of technology specific code generation much faster</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I am satisfied with the results</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Using the tool was efficient</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8. **Did you need to adjust the generated code to meet your expectations?** *
   *Mark only one oval.*

   - [ ] yes
   - [ ] no
9. What did you change in the generated code? (If the answer to the previous question was yes)

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

10. Did the tool correctly solve the difficulties that you encountered when solving this task manually? *

Mark only one oval.

☐ yes

☐ partially

☐ no

11. Which difficulty was not solved? (If the answer to the previous question was not yes)

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

12. Do you have any other comments?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
1. **Do you agree with the following statements?**
   Please note that the correct name of the constraint or the attribute type is not relevant!
   *Mark only one oval per row.*

<table>
<thead>
<tr>
<th>Statement</th>
<th>1 - totally disagree</th>
<th>2 - disagree</th>
<th>3 - neutral</th>
<th>4 - agree</th>
<th>5 - totally agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>I knew how to define the new field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I knew how to define the constraint</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The UnifiedOCL syntax is intuitive</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I believe I can learn it quickly</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I believe in the future I would not have trouble recalling how to do it again</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. **Do you have any other comments?**

   ...........................................................................................................................
   ...........................................................................................................................
   ...........................................................................................................................
   ...........................................................................................................................
   ...........................................................................................................................
   ...........................................................................................................................
1. Which transformation did you use to solve the task? *
   
   Mark only one oval.
   
   - [ ] Java to SQL
   - [ ] SQL to Java
   - [x] Java to UnifiedOCL, SQL to UnifiedOCL

2. Why? *
3. **Do you agree with the following statements?** *  
Mark only one oval per row.

<table>
<thead>
<tr>
<th>Statement</th>
<th>1 - totally disagree</th>
<th>2 - disagree</th>
<th>3 - neutral</th>
<th>4 - agree</th>
<th>5 - totally agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>It is easy to compare manually two different representations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>It is a time consuming task to manually compare the two different</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>representations</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>It would be difficult to manually transform one representation into the</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>It would be time consuming to manually transform one representation into</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>the other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I understand the concept of transformations and unified data quality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>representation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The tool was helpful to solve this task</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I knew how to use the tool to transform one representation into the other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The tool was convenient to use</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Using the tool was efficient</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I am satisfied with the results</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformations between various constraint representations are useful</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformations between various constraint representations would be</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>helpful in system modeling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transformations between various constraint representations would be</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>helpful to solve software engineering issues</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4. **Do you have any other comments?**

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
Achieving Unified Data Quality Representation by Constraints Transformation - Task 4 (Data Quality Visualizer)

Based on the running instance of the DQ Visualizer with the pre-loaded data, answer the following questions.

*Required

**Question 1: Which instance of the class Flight is of the worst quality?**

Provide the information about the visualization parameters that helped you answer this question (final configuration used)

1. **Answer**

2. **Feedback form**
   *Mark only one oval.*
   
   - Spider web plot
   - Bar plot
   - Report

3. **Scope**
   *Mark only one oval.*
   
   - MULTIPLE CLASSES
   - model.Airport
   - model.Flight
   - model.Machine
   - model.Passenger
   - model.Person
   - model.Pilot
   - model.Plane

4. **Grouping mode**
   *Mark only one oval.*
   
   - Combined
   - Separately
5. **Display mode** *
   *Mark only one oval.*
   - By Class
   - By Dimension

6. **Difference tracking used?** *
   *Mark only one oval.*
   - yes
   - no

**Question 2: Which data quality dimension of the class Flight is the worst?**

Provide the information about the visualization parameters that helped you answer this question (final configuration used)

7. **Answer** *

8. **Feedback form** *
   *Mark only one oval.*
   - Spider web plot
   - Bar plot
   - Report

9. **Scope** *
   *Mark only one oval.*
   - MULTIPLE CLASSES
   - model.Airport
   - model.Flight
   - model.Machine
   - model.Passenger
   - model.Person
   - model.Pilot
   - model.Plane

10. **Grouping mode** *
    *Mark only one oval.*
    - Combined
    - Separately
11. **Display mode** *
   *Mark only one oval.*
   - By Class
   - By Dimension

12. **Difference tracking used?** *
    *Mark only one oval.*
    - yes
    - no

**Question 3: Which data quality dimension of the whole system is the worst?**

Provide the information about the visualization parameters that helped you answer this question (final configuration used)

13. **Answer** *

14. **Feedback form** *
    *Mark only one oval.*
    - Spider web plot
    - Bar plot
    - Report

15. **Scope** *
    *Mark only one oval.*
    - MULTIPLE CLASSES
    - model.Airport
    - model.Flight
    - model.Machine
    - model.Passenger
    - model.Person
    - model.Pilot
    - model.Plane

16. **Grouping mode** *
    *Mark only one oval.*
    - Combined
    - Separately
Question 4: Which data quality aspect (dimension) is affected the most when considering the class Person and including only pilots instead of only passengers?

Provide the information about the visualization parameters that helped you answer this question (final configuration used)

20. Feedback form *
   Mark only one oval.
   ○ Spider web plot
   ○ Bar plot
   ○ Report

21. Scope *
    Mark only one oval.
    ○ MULTIPLE CLASSES
    ○ model.Airport
    ○ model.Flight
    ○ model.Machine
    ○ model.Passenger
    ○ model.Person
    ○ model.Pilot
    ○ model.Plane

22. Grouping mode *
    Mark only one oval.
    ○ Combined
    ○ Separately
23. **Display mode** *
   
   *Mark only one oval.*
   
   - [ ] By Class
   - [ ] By Dimension

24. **Difference tracking used?** *
   
   *Mark only one oval.*
   
   - [ ] yes
   - [ ] no

**Question 5: Which classes contain inconsistent data?**

Provide the information about the visualization parameters that helped you answer this question (final configuration used)

25. **Answer** *

   
   

26. **Feedback form** *
   
   *Mark only one oval.*
   
   - [ ] Spider web plot
   - [ ] Bar plot
   - [ ] Report

27. **Scope** *
   
   *Mark only one oval.*
   
   - [ ] MULTIPLE CLASSES
   - [ ] model.Airport
   - [ ] model.Flight
   - [ ] model.Machine
   - [ ] model.Passenger
   - [ ] model.Person
   - [ ] model.Pilot
   - [ ] model.Plane

28. **Grouping mode** *
   
   *Mark only one oval.*
   
   - [ ] Combined
   - [ ] Separately
29. **Display mode**  * 
   Mark only one oval.
   
   ☐ By Class
   ☐ By Dimension

30. **Difference tracking used?**  *
   Mark only one oval.
   
   ☐ yes
   ☐ no

**Summary**

31. **Do you agree with the following statements?**  *
   Mark only one oval per row.

<table>
<thead>
<tr>
<th></th>
<th>1 - totally disagree</th>
<th>2 - disagree</th>
<th>3 - neutral</th>
<th>4 - agree</th>
<th>5 - totally agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>The tool was easy to use</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I had no problem finding the answers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The tool informs well about the data quality</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I understood the plots</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>It was easy to learn the tool</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

32. **Do you have any other comments?**

........................................................................................................................................
........................................................................................................................................
........................................................................................................................................
........................................................................................................................................
........................................................................................................................................
D.1 Entity – Type

This file contains pairs of entity identifier (name) and entity type. By entity we mean a single unit of data, e.g. a table row, an object. The entity identifier may be for instance the result of toString method called on the object or a hash computed using this entity, etc.

Below we present a JSON file structure:

```
{
    entity_name : entity_type,
    ...
}
```

where:

- **entity_name** – the name or identifier of an entity
- **entity_type** – the type of an entity.

Example:

```
{
    "Plane [ lifeJacketsNumber=null, age=21, weight=1500.287, seatsNumber=null, cabinCrewNumber=null ]": "model.Plane",
```
D.2 Types hierarchy

This file contains hierarchy (relationships subtype-supertype) for all types of data entities visualized by Data Quality Visualizer. The hierarchy is represented as map that matches type name with names of its supertypes (a collection). Below we present a JSON file structure:

```json
{
    entity_type: [entity_supertype, ...]
    ...
}
```

where:

- `entity_type` – the type of an entity
- `entity_supertype` – the type of an entity that is a supertype for another entity.

Example:

```json
{
    "model.Pilot": [
        "model.Person"
    ],
    "model.Plane": [
        "java.lang.Object",
        "model.Machine"
    ],
    ...
}
```
D.3 Constraint violations

This file contains descriptions of all constraint violations for the given set of data. This is a collection of objects that consists of three elements: the name of the entity, the name of the violated constraint and the constraint location (within the data structure model). Below we present a JSON file structure:

```
[
  {
    "entity": entity_name,
    "constraintName": constraint_name,
    "constraintLocation": constraint_location
  },
  ...
]
```

where:

- `entity_name` – the name of an entity, that violates the constraint
- `constraint_name` – the name of a violated constraint
- `constraint_location` – the location of a violated constraint in the data structure model

Example:

```
{
  {
    "entity": "Flight 6 [ startTime=Tue Jul 30 10:20:56 CEST 2013, endTime=Fri Jul 31 10:20:56 CEST 2015, number=null, name=LXmVcBSTOzEj, id=6, description=plEea3nqiodO, destinationAirport=null, passengers=null, sourceAirport=null, pilots=null, plane=null ]", "constraintName": "IsProfitable", "constraintLocation": "model_Flight"
  },
  {
    "entity": "Passenger [ seatNumber=null, creditCardNumber=4111111111111111, hasTicket=false, name=Bob, surname=Muster, birthDate=null, title=Sir/Madam, email=aaa@mail.com, id=4, address=aaa ]", "constraintName": "AssertTrue", "constraintLocation": "model_Passenger_hasTicket"
  }
}
```
D.3. CONSTRAINT VIOLATIONS
Airline – demonstration scenario

E.1 UML class diagram with constraints
Figure E.1: Airline scenario
E.2 UnifiedOCL

```java
package model {

    public interface Machine {
    }

    public abstract class Person {
    public attribute name: String;
    public attribute surname: String {nonnull};
    public attribute birthDate: Date {past};
    public attribute title: String {default: 'Sir/Madam'};
    public attribute email: String {email};
    public attribute id: Integer;
    public attribute address: String {notblank};
    invariant PersonId: not self.id.oclIsUndefined() and self.allInstances()->forall(a: Person | a<>self implies a.id<>self.id);
    invariant NameNotNull: self.name<>null;
    }

    public class Plane realizes Machine {
    public attribute lifeJacketsNumber: Integer;
    public attribute age: Integer;
    public attribute weight: Real {digits(integer=4, fraction=2) decimalMax(value='125.75')};
    public attribute seatsNumber: Integer;
    public attribute cabinCrewNumber: Integer;
    public operation takeOff(speed: in Integer, landingGear: inout Boolean): String {
        precondition SufficientSpeedToTakeOff: speed>350;
        precondition LandingGearPulledOutBeforeTakeOff: landingGear=true;
        body TakeOffBody: landingGear=false;
        postcondition LandingGearRetractedAfterTakeOff: landingGear=false;
        postcondition NoProblemsDuringTakeOff: result=null;
    };
    invariant ModernEnough: self.age<25;
    invariant EnoughSafetyEquipment: self.lifeJacketsNumber = self.seatsNumber+self.cabinCrewNumber*2+5;
    }
```
public class Pilot specializes Person
{
    public attribute licenceExpired : Date { future } ;
    public attribute licenceNumber : Integer { unique } ;
    public attribute salary : Real { range (min=2500, max =5300) } ;
}

public class Passenger specializes Person
{
    public attribute seatNumber : Integer { autoincrement unique } ;
    public attribute creditCardNumber : String { creditcard } ;
    public attribute hasTicket : Boolean { asserttrue } ;
}

public class Flight
{
    public attribute startTime : Date ;
    public attribute endTime : Date ;
    public attribute number : Integer { unique } ;
    public attribute name : String ;
    public attribute id : Integer { primarykey } ;
    public attribute description : String { length (max=180)} ;
    public reference destinationAirport#arrivals : Airport{
        unique } ;
    public reference sourceAirport#departures : Airport{
        unique } ;
    public reference pilots : Pilot[2..2]{ unique } ;
    public attribute passengers : Passenger[1..*]{ unique } ;
    public reference plane : Plane{ unique } ;
    invariant EnoughLifeJackets : self.plane.
lifeJacketsNumber = self.passengers ->size () ;
    invariant LufthansaGroup : self.name.matches( ‘^LX\\w+$’ ) ;
    invariant IsProfitable : self.passengers ->size () *
500–10000>12500 ;
    invariant CorrectTime : self.startTime<self.endTime ;
}

public class Airport
{
    public attribute IATACode : String { unique } ;
APPENDIX E. AIRLINE – DEMONSTRATION SCENARIO

74 75
76
77
78
79

public reference arrivals#destinationAirport : Flight [0..*]{size(min=3 max=9) unique};
public reference departures#sourceAirport : Flight [0..*]{unique};

invariant AvoidCongestion : self.arrivals ->size()+self.departures ->size()<12;
}


e.3 Data Quality Mapping

model_airport AvoidCongestion [Consistency 0.2]
model_airport_arrivals Size [Correctness 0.2, Consistency 0.2]

model_flight EnoughLifeJackets [Consistency 0.2]
model_flight CorrectTime [Consistency 0.2, Timeliness 0.5]
model_flight IsProfitable [Consistency 0.2]
model_flight_description Length [Appropriateness 0.2]
model_flight_name Pattern [Correctness 0.2]

model_passenger_creditcard_number CreditCardNumber [Correctness 0.2]
model_passenger_hasticket AssertTrue [Correctness 0.5]

model_person_address NotBlank [Completeness 0.2]
model_person_birthdate Past [Timeliness 0.2]
model_person_email Email [Correctness 0.2]
model_person_name NotNull [Completeness 0.2]
model_person_surname NotNull [Completeness 0.2]

model_pilot_licenceexpired Future [Timeliness 0.5]
model_pilot_salary Range [Correctness 0.2]

model_plane EnoughSafetyEquipment [Consistency 0.5]
model_plane_age Max [Correctness 0.2]
model_plane_weight Digits [Accuracy 0.5]
model_plane_weight DecimalMax [Correctness 0.2]
Acronyms

**API** Application Programming Interface. 10, 90, 92, 195

**ATL** ATL Transformation Language. 80, 83, 84, 195

**BNF** Backus-Naur Form. 21, 195

**CASE** Computer-aided software engineering. 28, 52, 85, 133, 195

**DQAF** Data Quality Analysis Feedback. 24, 195

**DQIF** Data Quality Input Feedback. 24, 195

**DQM** Data Quality Mapping. 101, 109, 114, 195

**DSL** Domain Specific Language. 3, 22, 27, 28, 49, 55, 79–83, 86, 89, 101, 104, 111, 114, 124, 142, 153, 195

**EMF** Eclipse Modeling Framework. 30, 82, 83, 85, 86, 90, 118, 195

**EMP** Eclipse Modeling Project. 82–84, 119, 195

**GPL** General Purpose Language. 22, 195

**GUI** Graphical User Interface. 28, 90, 118, 148, 195

**HUTN** UML Human-Usable Textual Notation. 30, 52, 80, 195

**IDE** Integrated Development Environments. 21, 81, 118, 119, 195

**IS** Information System. 1, 195

**JAR** Java Archive. 81, 92, 107, 195

**Java EE** Java Enterprise Edition. 10, 195

**JPA** Java Persistence API. 150, 195

**JSON** JavaScript Object Notation. 52, 77, 87, 92, 126, 149, 187–189, 195

**KM3** Kernel Meta Meta Model. 80, 83, 195
M2M  model-to-model. 18, 19, 53–55, 195
M2T  model-to-text. 18, 19, 50, 53–55, 195
MDA  Model Driven Architecture. 17, 18, 22, 24, 30, 84, 132, 195
MDE  Model Driven Engineering. 18, 22, 28, 50, 58, 195
MOF  Meta-Object Facility. 6, 18, 30, 84, 195
MOFM2T MOF Model To Text Transformation Language. 83, 195
MS   Microsoft. 81, 195
MSDK Modeling SDK for Visual Studio. 81, 195
MVP  Model View Presenter. 90, 91, 114, 195, 202
OCL  Object Constraint Language. 7–9, 18–21, 24, 27, 28, 30, 37, 40–44, 46–50, 55, 62, 85–90, 92, 94, 98, 100, 104–107, 111, 124, 126, 150, 151, 153, 195, 205
OLAP Online Analytical Processing. 151, 195
OMG  Object Management Group. 6, 9, 17, 18, 30, 44, 49, 52, 83–85, 195
OSGi Open Service Gateway Initiative. 81, 82, 107, 110, 115, 118, 195
PIM  Platform Independent Model. 18, 28, 50, 195
PSM  Platform Specific Model. 18, 28, 50, 195
QVT  Query/View/Transformation. 19, 83, 84, 89, 195
QVTD Query/View/Transformation Declarative. 55, 195
QVTO Query/View/Transformation Operational. 18, 107, 118, 195
RDBMS Relational Database Management System. 8, 11, 75, 89, 149, 195
SDK  Software Development Kit. 195
T4   Text Template Transformation Toolkit. 81, 195
TCS  Textual Concrete Syntax. 80, 195
**Acronyms**

**UML2** Eclipse implementation of UML. 52, 83, 85, 89, 107, 124, 195

**XMI** XML Metadata Interchange. 30, 52, 82, 85, 89, 90, 104, 195

**XML** Extensible Markup Language. 50, 55, 81, 82, 195
List of Figures

2.1 Class Person with defined constraints ........................................ 11
2.2 Class Plane with defined constraints .......................................... 12
2.3 Annotation EnoughSafetyEquipment definition ................................ 12
2.4 A validator for the annotation EnoughSafetyEquipment ................... 13
2.5 Table Person with defined constraints .......................................... 13
2.6 Table Plane with defined constraints ........................................... 14
2.7 Rules for class Person .................................................................. 15
2.8 Rules for class Plane ................................................................... 16
2.9 Model to model transformation overview ....................................... 20
2.10 Model to text transformation overview .......................................... 20
2.11 Data quality dimensions proposed by Batini and Scannapieco ............ 23
3.1 Overview of UnifiedOCL ................................................................. 29
3.2 Syntax of UnifiedOCL ................................................................... 32
4.1 System overview .......................................................................... 51
5.1 Definition of the penalty function for a specified dimension ............... 63
5.2 Definition of the penalty function for all dimensions .......................... 63
5.3 Definition of the violation function ............................................... 63
5.4 Value of data quality dimension for a single instance ........................ 65
5.5 Value of data quality dimension for multiple instances separately ....... 66
5.6 Value of data quality dimension for multiple instances combined ...... 66
5.7 Value of data quality dimension for a single class (all instances combined) 66
5.8 Constraints to data quality mapping for the classes Pilot, Person and Airport ............................................................... 67
5.9 Value of data quality dimension for multiple classes separately .......... 69
5.10 Value of data quality dimension for multiple classes combined ........ 69
5.11 Value of data quality dimension for the whole system (all instances of all classes) ..................................................... 69
5.12 Textual report – data quality feedback ............................................. 73
5.13 Spider web plot – data quality feedback ............................................. 74
5.14 Bar plot – data quality feedback ..................................................... 75
5.15 Data Quality Feedback Visualizer – system overview ...................... 76

6.1 Data Quality Visualizer – MVP architecture – UML class diagram ........ 91
6.2 Used technologies ........................................................................... 93
6.3 Transformation architecture – overview ........................................... 95
6.4 Structure of an Xtext project – DQM language .................................. 102
6.5 Structure of an Xtext user interface project – DQM language ............. 103
6.6 ch.ethz.unifiedocl.java – UML class diagram .................................. 108
6.7 ch.ethz.unifiedocl.drools – UML class diagram ................................. 112
6.8 Sample generator class: ch.ethz.unifiedocl.sql.generator.SQL2UnifiedOCLGenerator – UML class diagram ......................... 113
6.9 All bundles(plugins) – component diagram ..................................... 116

7.1 The UnifiedOCL installation – Add Eclipse Update Site ..................... 120
7.2 The UnifiedOCL installation – Select plugins to install ....................... 121
7.3 The Eclipse preferences for UnifiedOCL ........................................... 122
7.4 UnifiedOCL transformations in Eclipse ............................................ 123
7.5 A transformation confirmation window .......................................... 123
7.6 The Eclipse editor for UnifiedOCL .................................................. 125
7.7 The Load Model window ............................................................... 127
7.8 Data Quality Visualizer – a GUI overview ...................................... 128
7.9 A type hierarchy graph ............................................................... 129

8.1 Knowledge and experience of participants .................................... 134
8.2 Task I – results ............................................................................... 137
8.3 Task III – chosen technology ......................................................... 138
8.4 Task II – results ............................................................................... 140
8.5 Task III – results ............................................................................... 141
8.6 Task III – chosen transformation ..................................................... 142
8.7 Task IV – results ............................................................................... 144
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>Task I – UML class diagram with data model</td>
<td>167</td>
</tr>
<tr>
<td>C.2</td>
<td>Task I – UML class diagram with data model enriched with constraints</td>
<td>168</td>
</tr>
<tr>
<td>C.3</td>
<td>Task II – UnifiedOCL code to be enriched</td>
<td>168</td>
</tr>
<tr>
<td>C.4</td>
<td>Task III – Java code</td>
<td>169</td>
</tr>
<tr>
<td>C.5</td>
<td>Task III – SQL code</td>
<td>169</td>
</tr>
<tr>
<td>E.1</td>
<td>Airline scenario</td>
<td>192</td>
</tr>
</tbody>
</table>
List of Tables

3.1 UnifiedOCL – Class ................................................. 34
3.2 UnifiedOCL – Interface ............................................ 35
3.3 UnifiedOCL – Enumeration ...................................... 35
3.4 UnifiedOCL – Exception .......................................... 36
3.5 UnifiedOCL – Primitive type .................................... 37
3.6 UnifiedOCL – Method .............................................. 37
3.7 UnifiedOCL – Method parameters .............................. 38
3.8 UnifiedOCL – Attribute ........................................... 39
3.9 UnifiedOCL – Reference .......................................... 40
3.10 Constraint representation: OCL vs UDQAM ..................... 47

5.1 Symbol legend of equations for values of data quality dimensions .... 63
5.2 Constraint violations - class Pilot ................................. 66
5.3 Constraint violations - class Airport ............................. 67

7.1 Supported input file types ........................................ 124
7.2 Supported other file types ...................................... 126
7.3 Data Quality Visualizer – Used formulae ....................... 127

8.1 Participants - summary ........................................... 135

B.1 Bean Validation – supported constraints ....................... 164
B.2 Hibernate Validator – supported constraints .................. 165
B.3 SQL (PostgreSQL) – supported constraints .................... 165
Foremost, I would like to express my sincerest gratitude to my supervisor David Weber for his continuous support, patience and engagement. His detailed and relevant comments and suggestions were truly helpful at every stage of the work on this thesis. I am also indebted to him for all of the time he spent guiding me during this life-long experience.

Furthermore, I would like to thank Prof. Dr. Moira C. Norrie for allowing me to work on such an interesting topic in the GlobIS group. I wish to express my gratefulness for all the members of that group that were always available and willing to help.

Additionally, I want to thank all the participants of the user study who have shared their precious time and provided me with a valuable feedback.

I would also like to thank all my fellow students, sharing the same lab, for a friendly atmosphere and fruitful discussions. Special thanks is owed to all of my friends who were here with me during the whole course of this thesis.

Last but not least, I am indebted to my parents and my two sisters for their constant support and encouragement not only during my work on this thesis but throughout my whole life.
Bibliography


Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Achieving Unified Data Quality Representation by Constraints Transformations

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):
Szymanek

First name(s):
Jakub

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Zürich, 23.08.2015

Signature(s)

Jakub Szymanek

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.