



universität
wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

Conceptualization And Implementation Of A
Constraint-Modeling-Language

verfasst von / submitted by

Christoph Puhr

angestrebter akademischer Grad / in partial fulfillment of the requirements
for the degree of

Diplom-Ingenieur (Dipl.-Ing.)

Wien, 2016 / Vienna 2016

Studienkennzahl lt. Studienblatt /
degree programme code as it appears on
the student record sheet:

A 066 926

Studienrichtung lt. Studienblatt /
degree programme as it appears on
the student record sheet:

Masterstudium Wirtschaftsinformatik

Betreut von / Supervisor:

Mag. Dr. Hans-Georg Fill, Privatdoz.

Acknowledgements

I would like to thank my supervisor Mag. Dr. Hans-Georg Fill, Privatdoz. for his helpfulness and valuable hints regarding the conceptualization of a constraint modeling language as well as suggestions about references.

Furthermore, I would like to thank my fellow student Michael Bueltmann BSc for initial advices concerning an AdoScript validation functionality for constraints modeled in the constraint language.

Abstract (Deutsch)

Durch Beschränkungen, welche beliebige Modellierungssprachen adressieren und erweitern, erhalten Modellierer/-innen ein flexibles Spektrum an Konzepten und Methoden, um die Qualität von Modellen zu verbessern. Diese Arbeit beschreibt eine Modellierungssprache für Beschränkungen, welche durch *OCL* inspiriert sind und auf Metamodell-Ebene zugewiesen werden. Dies ermöglicht es, sowohl einzelne spezifische Modellinstanzen, als auch eine breite Anzahl an denkbaren Modellen zu adressieren und mit Beschränkungen anzureichern. Weiters beschreibt die Arbeit, wie eine solche Sprache in der *ADOxx meta modeling platform* konzeptionell umgesetzt und implementiert werden kann. Schlussendlich zeigt eine Funktionalität zur Validierung, dass zugewiesene Beschränkungen im Rahmen von praktisch orientierten Szenarien auch überprüft und ausgewertet werden können. Die Implementierung in *ADOxx* wurde auf einer CD, welche auf der letzten Seite der Arbeit zu finden ist, gespeichert.

Abstract (English)

By assigning constraints to arbitrary modeling languages, a modeller acquires a flexible spectrum of concepts and methods for improving the quality of models. This thesis introduces a Constraint-Modeling-Language which deposits *OCL* inspired restrictions on meta model layer affecting particular model instances or the entire amount of potentially instanced models. As a result, the language can administer very specific cases as well as quantitative scenarios regarding lots of imaginable model instances. In addition, the thesis describes how the above mentioned approach can be implemented in the *ADOxx meta modeling platform*. Last but not least, a validation functionality demonstrates the feasibility of constraint validations in practically orientated use case scenarios. The implementation in *ADOxx* is attached as CD which can be found at the last page of the thesis.

Contents

1.	Introduction	1
2.	Foundations	3
2.1.	Modeling Method	3
2.2.	Meta Meta-Model and Meta Model	5
2.3.	ADOxx	6
2.4.	UML and UML Class Diagrams	8
3.	Overview on existing Methods to design and visualize Constraints	11
3.1.	Introduction to OCL	11
3.2.	OCL Invariant Constraints	13
3.2.1.	Invariant Constraints for Attributes	13
3.2.1.1.	Invariant Constraints for Numeric Values	13
3.2.1.2.	Invariant Constraints for Strings and Enumerations	14
3.2.2.	Invariant Constraints for Associations and Objects	16
3.2.3.	Implications	17
3.3.	Visualization Approaches	18
3.3.1.	VOCL	18
3.3.2.	Constraint Diagrams	22
3.4.	Conclusion	27
4.	Conceptualization of the Constraint-Modeling-Language	29
4.1.	Modeling Process Integration	29
4.2.	Coherence between Constraint Model and Meta Model	31
4.3.	Existing Methods for Constraints in ADOxx	32
4.4.	Language Scope and Specification	32
4.5.	Notation	47
4.5.1.	Constraint Visualization	47

4.5.1.1.	Semantical Relationship between Compartments	48
4.5.1.2.	Header Compartment	49
4.5.1.3.	Model Type, Class, and Relation Class Compartment	49
4.5.1.4.	Attribute Compartment	51
4.5.1.5.	Connection Compartment	54
4.5.2.	Constraint Implications	55
4.6.	Conclusion	57
5.	Implementation of the Constraint-Modeling-Language	59
5.1.	Meta Model	59
5.1.1.	Constraint Class	60
5.1.1.1.	General Attributes	60
5.1.1.2.	First Compartment Attributes	62
5.1.1.3.	Second Compartment Attributes	63
5.1.1.4.	Third Compartment Attributes	64
5.1.1.5.	Forth Compartment Attributes	68
5.1.1.6.	Attrep	70
5.1.1.7.	Graphrep	70
5.1.2.	Implies Relation Class	71
5.2.	Validation	71
5.2.1.	Validation Scope	72
5.2.2.	Validation Implementation	72
5.3.	Preparation for Use Case Scenarios	75
5.4.	Constraint-Modeling-Language Application - Use Case Scenarios	77
5.5.	Conclusion	84
6.	Discussion	87
	References	89
A.	Attachment	93
A.1.	Validation in AdoScript	93

1. Introduction

By creating well-defined modeling languages it becomes necessary to think about the whole spectrum of potential instances to ensure semantically expressive models. But even if the range of possibilities is known, there is no guarantee for meaningful or correct models. Consequently, constraints can help to increase the quality of models by defining a frame of valid properties, constellations, and interactions between model elements. Therefore, by assigning constraints to whole models or specific model elements, invalid model conditions are going to be avoided and as a result, the original and semantical intention of modeling languages will be assured.

This work introduces a method for enriching pre-existing modeling languages, e.g. *BPMN* [14] or a modeling language for IT-architectures, with specific constraints restricting model characteristics. The objective behind the process of applying the Constraint-Modeling-Language to other modeling languages is the enhancement of arbitrary modeling languages and their semantical expressiveness. At this point, it is important to underline that the original modeling language itself stays unmodified. This means that there are no syntactical modifications necessary to make the Constraint-Modeling-Language able to work. This facts leads to a high degree of flexibility and a broad range of applicable modeling languages. This approach is realized by offering a parallel constraint model connected to models which should be enriched with constraints.

In technical context, constraints and techniques of the Constraint-Modeling-Language in general are settled and assigned on meta model level. This basically means that the constraint language is able to restrict syntactical elements e.g. attributes, relations, objects, and model types defined in the meta model of the assigned modeling language. The modeling procedure is structured as follows: The Constraint-Modeling-Language connects constraints to a meta model of a specific modeling language. A conceivable constraint might be: There must be at least one object of the class *Activity* in the model type *Activity Model*. The logical result of this process is a constraint enriched meta model which serves as basis for subsequent instances which are affected by previously determined constraints.

Constraints offered by the modeling language are inspired by the *Object Constraint Language* (OCL [15]). Although OCL is settled in the context of object orientated programming languages, methods and constraint types of OCL are going to serve as reference and in-

spiration. Therefore, the work analyses the variety of *OCL* constraints and the suitability apart the *UML* [16] specification.

The core of the thesis deals with three essential research questions. The questions are part of a progressive process implying that the first question discusses initial information and theoretical aspects while the third questions presents the implementation and final outcomes.

1. Which generic solutions regarding the implementation of constraints already exist? Are there specific approaches of visualizing constraints?

The first research question aims at pre-existing *OCL*-based solutions for designing constraints. The question also analyses existing methods for visualizing constraints. The outcomes of this research question serve as foundation for the second question.

2. How does the conceptualization of the Constraint-Modeling-Language look like?

The second research question deals with the process of creating a Constraint-Modeling-Language. After completing the language definition, an use case is going to test its practicability. The use case is conceptualized to cover many various situations and the modeling language has to deal with given scenarios. The final concept represents the initial situation of the third question.

3. How is the Constraint-Modeling-Method implementation realized?

The last question is geared towards the implementation in *ADOxx*. The implementation approach aims at a straightforward and convenient way of assigning constraints to referenced models. In addition, a validation functionality provided in *ADOscript* analyses and ensures the syntactical and semantical correctness of applied models.

2. Foundations

Since this work describes a process of building a modeling method, it is necessary to define some relevant terms to ensure a common understanding. First, as an initial point and the fundament for further work, it is important to define the term *modeling method* itself. Second, the term *meta model* has to be clarified. Third, the *ADOxx meta modeling platform* is going to be mentioned and explained. Last but not least, this chapter aims at describing the *UML* specification and models involved by this modeling language.

2.1. Modeling Method

According to a framework by Karagiannis and Kühn [22] (Fig. 2.1) a *modeling method* consists of a *modeling technique* and *mechanisms and algorithms*. In addition, the *modeling technique* could be further divided into a *modeling language* and a *modeling procedure*. Whereas the *modeling procedure* includes *steps*, for executing the *modeling language*, and *results*, the *modeling language* itself describes a *syntax* and *semantics*. While the *syntax* defines the grammar of the language, the *semantics* assigns a meaning to *syntax* elements. Consequently, the bridge between *syntax* and *semantics* is performed by a *semantic mapping* which gets semantical allocations to syntactical elements via a *semantic schema*. This schema could be described formally or informally e.g. with textual descriptions. Furthermore, the *notation* determines the visual appearance of the *modeling language* through the elements of the *syntax* and by assigning the specific *semantics*. As a last point, *mechanisms and algorithms* are used for the *modeling procedure* and its corresponding *modeling language*. They are divided into *generic*, *specific* and *hybrid mechanisms and algorithms*. The difference between the three approaches could be seen in their specific applicability. Whereas *generic mechanisms and algorithms* can deal with any kind of *modeling languages*, *specific mechanisms and algorithms* can only be applied to particular *modeling languages*. The *hybrid* approach can handle only specific *modeling languages* too, but by adapting *hybrid mechanisms and algorithms*, they fit for various scenarios [8].

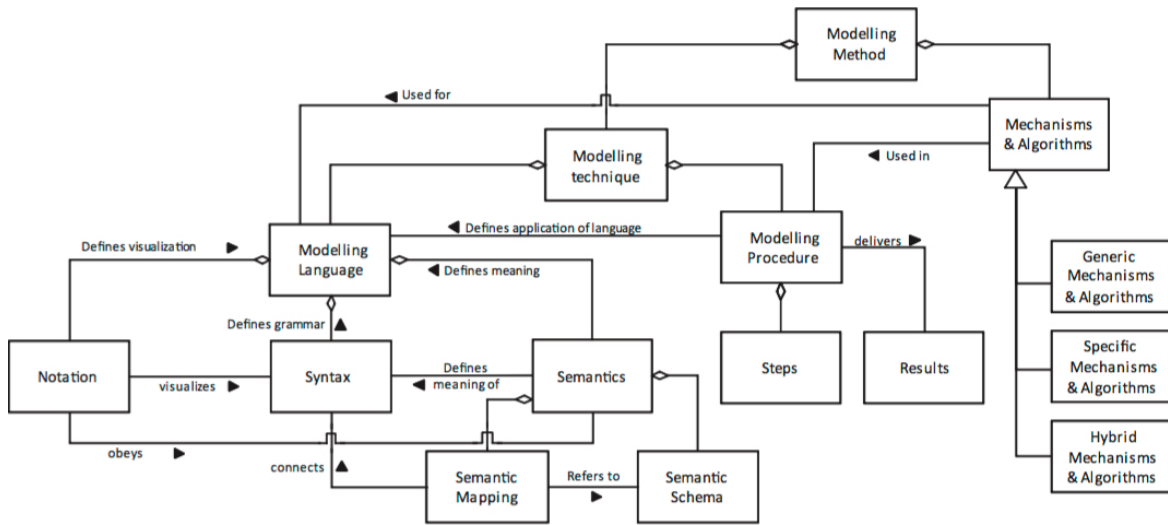


Figure 2.1.: Components of modeling methods [22]

Additionally, in reference to a framework established by the Open Model Initiative Laboratory¹, one iteration of creating a modeling method consists of five essential phases: (1) *create*, (2) *design*, (3) *formalize*, (4) *develop*, and (5) *deploy/validate*. Consequently, this process reiterates by gradually aggregating knowledge from a specific domain which goes hand in hand with the evolvement of modeling requirements [29]:

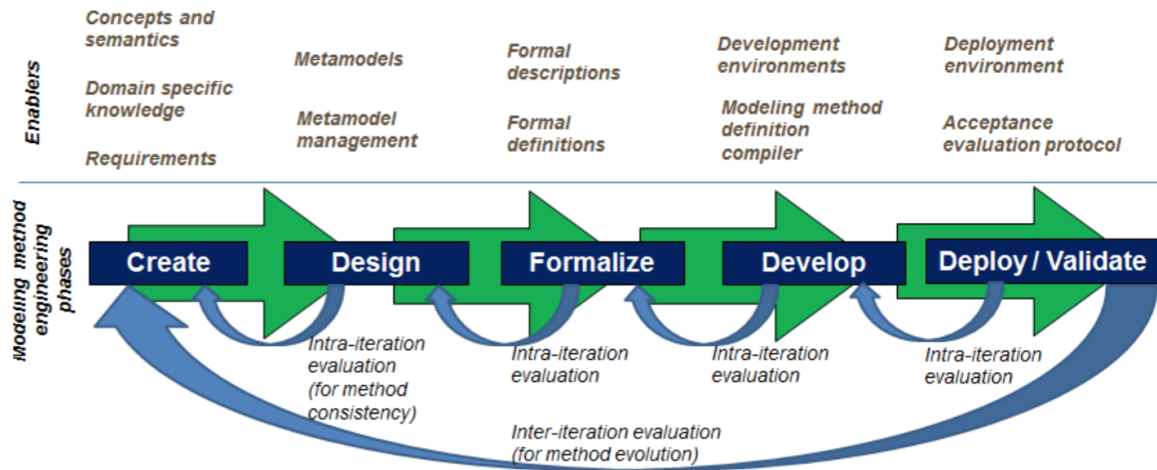


Figure 2.2.: Iterative process of generating a modeling method [29]

1. Creation:

The focus in this phase lies on knowledge aggregation and requirements specification. As a result, the output of this processes is the definition of *modeling language requirements* and *modeling functionality requirements*. Whereas *modeling language*

¹<http://www.omilab.org> (accessed June 22, 2016)

requirements deal with concepts and relations required in a specific modeling language, *modeling functionality requirements* consider competence questions which a model should be able to answer.

2. **Design:**

The *designing* process is the phase where core meta modeling efforts are required. In consequence, the result of this step is represented by a structured meta model including the language grammar, recommended visualization- and functionality approaches. The *Design* process can be supported by existing languages commonly used for domain modeling (e.g. class diagrams or ER diagrams).

3. **Formalization:**

This phase describes the output of previous phases in a non-ambiguously way. This step is essential for preparing the method implementation. In this thesis, the implementation is realized in *ADOxx*, which is going to be described in a following paragraph.

4. **Development:**

The *Development* step involves the selection of a specific and concrete meta-modeling platform to generate a modeling prototype. This process includes a compiler translating the abstract modeling language to technology-specific constructs of the chosen meta-modeling platform.

5. **Deployment/Validation:**

This phase deals with user acceptance tests and actions which are required to make such tests possible (e.g. packaging and installing the modeling prototype). The received feedback and potentially emerged additional requirements feed into the next iteration of the whole process.

2.2. Meta Meta-Model and Meta Model

In order to create a modeling language, it can be seen as state-of-the-art to apply meta meta-models [22], [24]. Furthermore, as discussed in section 2.1, a modeling language describes a *syntax* and *semantics*. The *syntax* can be divided into an *abstract syntax* represented by the *meta model* and a *concrete syntax* represented by a *model* [18, 19, 28]. In addition, an abstract *meta model* can be described by a *modeling language* too - the *meta modeling language*. Consequently, the *meta modeling language* and its syntax can also be divided into an *abstract syntax* and a *concrete syntax*. In this scenario, the *abstract syntax* embodies a *meta meta-model* whereas the *concrete syntax* represents a *meta model* [24, 9].

Important characteristics of *meta meta-models* are *inheritance* and *containment* mechanisms [28]. *Inheritance* refers to generalization and specialization relationships. This basically means that connected entities of a *meta model* inherit means for effecting polymorphic behaviors at execution or interpretation time. This is important for algorithms

applied on several, similar modeling languages because the algorithm can be bound automatically to entities which are inherited from general entities. On the other hand, *containment* refers to a inclusion of one or more entities into another entity on the *model* level. This makes it more comfortable to specify model types or aggregations involving a set of entities [9].

Fig. 2.3 visualizes an example for the relationship of a *meta meta model*, its corresponding *meta model* and the derived *model*. First, the *meta meta model* consists of the entities *Element* and *Attribute* which are both connected via the relationship *attached-to*. Second, on *meta model* layer, the entity E_1 is defined as *Element* while the two entities A_1 and A_2 are described as *Attribute* being attached to E_1 . Third, finally on *model* level, the entities ϵ_1 and ϵ_2 belong to the meta element E_1 while α_1 and α_2 are being assigned to A_1 and β_1 and β_2 to A_2 [9].

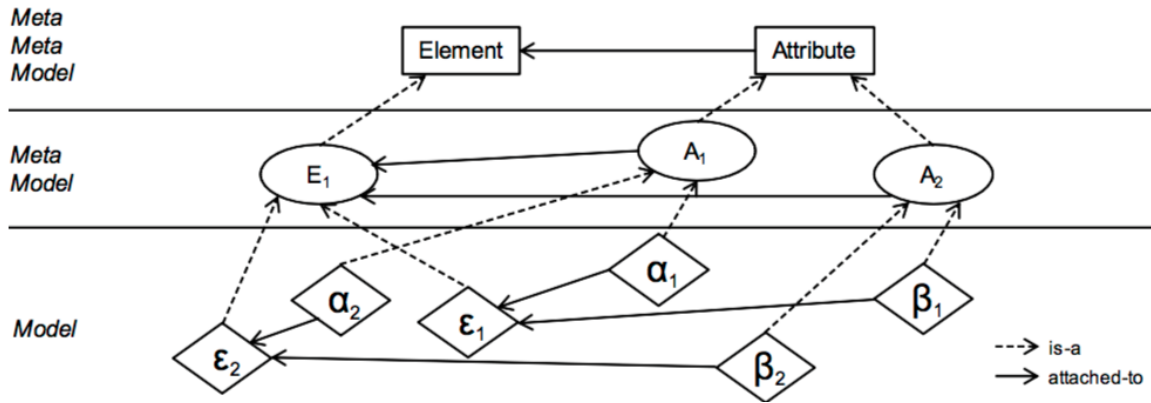


Figure 2.3.: Example for a Meta Meta Model, a Meta Model and a Model [9]

2.3. ADOxx

The ADOxx meta modelling approach has its original roots in the development of the ADO-NIS toolkit for business process management in 1995. Meanwhile, the ADOxx platform has become popular in a large number of academic and commercial projects worldwide [20, 21], [9]. Today, ADOxx is available as a commercial product or as an open use version for academic projects which can be requested by the Austrian section of the Open Models Initiative [8].

The ADOxx meta meta-model consists of *classes*, *relationships*, and *attributes* belonging to *classes* and *relationships* or *relation classes* [21]. Classes are organized in an *inheritance hierarchy* (well known from object orientated approaches) which means that sub

classes inherit attributes and characteristics from their assigned super classes. Furthermore, relationships are characterized by specific attributes which are named *from-class* and *to-class*. These attributes represent specifications for valid source and target classes. Relationships can be extended by cardinalities e.g. a specification to limit the amount of outgoing relationships of a class. Last but not least, *model types* represent a containment mechanism for classes and relationships. In addition, they are essential for the instantiation of classes and relationships, which consequently results in a *model* [9].

To make the ADOxx meta modeling approach more transparent, Fig. 2.4 visualizes the roles and languages in the process of generating a model in ADOxx.

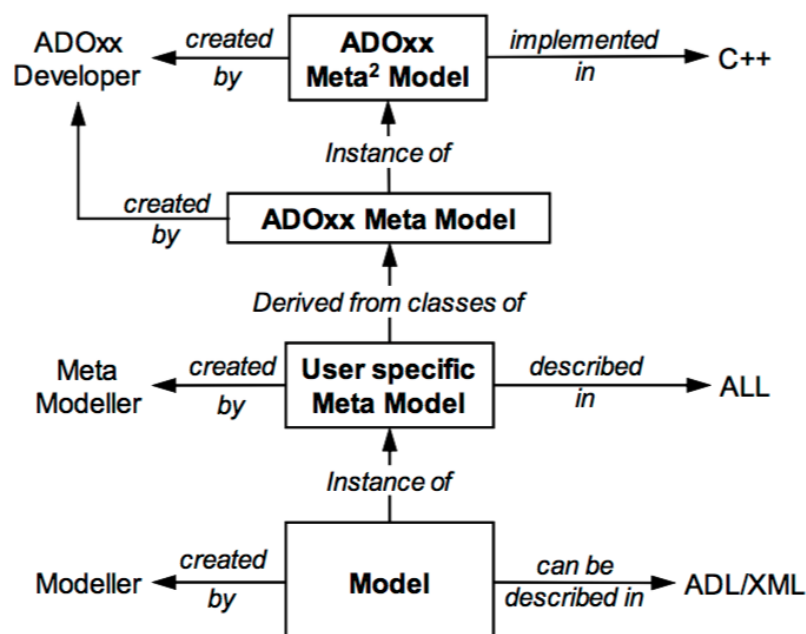


Figure 2.4.: Roles and languages in the modeling hierarchy of ADOxx [8]

The first level of Fig. 2.4 shows the *Meta² Model* (which stands for the meta meta-model). It is implemented in the programming language C++ and created by *ADOxx Developers*. The *ADOxx Meta Model*, in reference to section 2.3, represents an instance of the meta meta-model and is also generated by a *ADOxx Developer*. A *User specific Meta Model* is derived from classes of the *ADOxx Meta Model*. Its description is made in the proprietary language *ALL*, which stands for *ADOxx Library Language*. This specific language offers concepts for describing meta models. It bases on constructs defined in the *Meta² Model*. The *User specific Meta Model* represents the first time in the process of meta modeling in ADOxx, when a non-developing user creates input, in this case the *Meta Modeller*. Last but not least, a *Model* is an instance of the *User specific Meta Model*. It is created by a *Modeller* and described in the proprietary ADOxx export format *ADL* or in *XML* [8].

2.4. UML and UML Class Diagrams

Since *OCL* is going to have a significant weight for the conceptualization of a Constraint-Modeling-Language, it is essential to briefly describe *UML* and its specification first, as *UML* represents the environment in which *OCL* operates.

The *Unified Modeling Language* can be seen as "de facto standard" [3] of software engineering. Nevertheless, there are arguments implying that *UML* does not fulfill this role because of aspects like size, complexity, semantics, consistence and model transformation (e.g. [26, 4]). In 1994, *UML* was initially introduced by the *Object Management Group (OMG)* which also manages the standard of the *UML* specification. Basically, *UML* provides a framework to integrate various different kinds of diagrams [27]. The *OMG* itself defines *UML* as tool which helps to specify, visualize, and document models of software systems including structural and design aspects. Although *UML* has its focus in software systems and object orientated programming languages, it can also be used for business process modeling and modeling of other non-software systems [17].

The latest *UML* milestone specification 2.0 defines thirteen types of diagrams which can be categorized into three distinct approaches [17]:

- **Structure Diagrams** represent static application structure and includes the following diagrams: Class diagrams, Object Diagrams, Component Diagrams, Composite Structure Diagrams, Package Diagrams, and Deployment Diagrams.
- **Behavior Diagrams** characterize general types of behavior. Involved types of diagrams are: Use Case Diagrams, Activity Diagrams, and State Machine Diagrams.
- **Interaction Diagrams** are derived from the more general Behavior Diagrams and describe different aspects of interactions. Included diagrams are: Sequence Diagrams, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

Although *OCL* can be theoretically applied to any *UML* model [15], the *Object Constraint Language* is characterized by a strong focus on *UML* class diagrams. As a result, *UML* diagrams abroad the type of class diagrams are not going to be described more comprehensive in this thesis. As mentioned before, *UML* class diagrams visualize static structure of systems being modeled. The focus lies on a system and its specific elements, time aspects do not play a role. However, the static structure of a system is represented by *types* and their *instances* in the system. In most *UML* diagrams *types* include *classes*, *interfaces*, *data types*, or *components*. In this context, *UML* defines the term *classifier* which can describe any of the previously mentioned *types*, although it is usually used to define *classes* [1]. The notation, which defines the visual appearance of a modeling language, represents a class in form of a rectangle containing three compartments stacked vertically. The first compartment on top of the rectangle displays the name of the relevant class. The existence of the top compartment is essential for modeling a class diagram whereas the bottom two

compartments are optional. Furthermore, the middle part represents a list of attributes and corresponding data types. Also the last compartment visualizes a list, but in this case assigned operations belonging to the class [1]. An exemplary visualization of a *car* class might be:

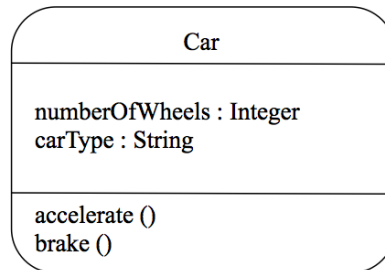


Figure 2.5.: Class Car modeled in an *UML* class diagram

Class diagrams also provide a notation for relationships named associations in the *UML* specification. Although class diagrams support five different types of associations, only one of them is going to be presented here, since this section aims at providing relevant basic techniques of modeling class diagrams for the further application of *OCL* constraints. The one selected type of relationships is a *bi-directional association*.

Bi-directional associations represent the standard assumption for relationships because they indicate that both classes are aware of each other and their relationship. They are visualized as solid line between two involved classes. At the either end of the line, a role name and multiplicities are assigned [1]. An example for a bi-directional association is shown in Fig. 2.6.

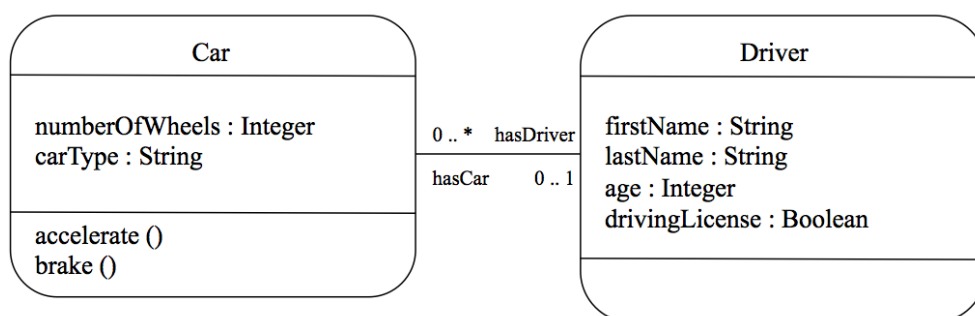


Figure 2.6.: Bi-directional association between the classes Car and Driver

The relationship between a *Car* class and a *Driver* class as well as assigned role names and multiplicities. Consequently, a car can have no driver or exactly one driver which is implicated by the multiplicity *0 .. 1*. On the other side, a driver can have (or better own)

no car or any number of cars shown by the multiplicity $0 \dots *$.

As a result and preparation for *OCL*, this section has given a basic overview on the procedure how classes and relationships are modeled in an *UML* class diagram. Although *UML* and class diagrams are settled in the object orientated environment, this modeling language and its specific characteristics is going to serve as inspiration for the later Constraint-Modeling-Language.

3. Overview on existing Methods to design and visualize Constraints

Since *OCL* represents the reference language for constraints in the context of *UML*, the Constraint-Modeling-Language is going to seize on concepts and methods of *OCL*. In addition, the thesis presents two distinct approaches for visualizing *OCL* constraints. This is going to enrich and positively influence the subsequent conceptualization of the Constraint-Modeling-Language.

3.1. Introduction to OCL

The Object Constraint Language is a formal language to enrich *UML* diagrams with specific constraints. Besides the generation of constraints, *OCL* functions as query language too, which is not relevant in the context of this thesis. First, as a starting point, the term *formal language* has to be clarified. Constraints are often described in *natural languages* which means that they are specified in a way how people would naturally communicate with each other. Although *natural languages* represent a straightforward and convenient method for characterizing constraints, they lead to potential ambiguities. To avoid this problem, *formal languages* have been developed. Consequently, by defining constraints with traditional formal languages, ambiguities can be eliminated, but this kind of language leads to a new disadvantage: The usability is very restricted and only people with a strong mathematical background are able to use the language in practice. *OCL* was developed to avoid this disadvantage by offering a structure which remains easy to read and write. *OCL* was originally designed as a practical business modeling language at an IBM Insurance division.

Regarding the technical background behind *OCL* expressions, it is essential to clarify *OCL*s functionality as modeling and specification language. *OCL* extends linked constructs in other programming languages. Therefore, it is not possible to write program logic or flow control in *OCL*. Furthermore, *OCL* expressions are not by definition directly executable. In addition, *OCL* constructs are without side effects which means that by evaluating *OCL* expressions, they just return a simple value. As a result, *OCL* expressions are called

instantaneous because they can not influence the state of a system even though *OCL* expressions could be used to specify a state change. Last but not least, *OCL* could be seen as typed language implying that each expression has a corresponding and specific type. Building proper *OCL* expressions means to conform to the type conformance rules of the *OCL* language e.g. Integer values and String values are not comparable [?].

Although *OCL* can be used for various purposes (besides its function as query language), the thesis is going to deal only with a specific purpose of *OCL*, named *specifying invariants on classes and types in the class model* [?] (the term *invariant* will be defined in a further section). The reason for this restriction is the object orientated environment in which *OCL* is settled. For example, specifying *pre- and post conditions on Operations and Methods* or *target (sets) for messages and actions* would make no sense for the modeling language presented in this work. Nevertheless, the partial area of *OCL* regarding invariants is going to embody a solid foundation for designing constraints involved in the Constraint-Modeling-Language.

OCL invariants are expressions which must be true for all instances (of the assigned type) at any time. To check the correctness and validity of *OCL* invariants, the returned value of invariants is of type boolean. The syntax is determined as follows:

```
context <classifier> inv [<constraint name>]:  
  <boolean expression>
```

An example for a simple *OCL* invariant constraint (with reference to Fig. 2.5) can be:

```
context Car inv aCarHas4Wheels:  
  self.numberOfWheels = 4
```

The *context*-keyword refers to the contextual instance of the *OCL* expression which is part of an invariant implicated by the keyword *inv*. It is followed by the name of the *type*, in this case *Car*. The keyword *self* stands for an instance of the type *Car* and represents the beginning of an evaluation. Regarding the keyword *self*, it could be optionally dropped and replaced by a different name playing the part of *self*. By replacing *self*, the previous constraint would have the following structure:

```
context c:Car inv aCarHas4Wheels:  
  c.numberOfWheels = 4
```

The most important characteristic of the invariant constraint could be seen in the fact that it holds for every instance of the type *Car*. The constraint restricts the attribute *numberOfWheels* to a logically assumption implying that a car always has four wheels to be a car. Attributes are written behind a dot and followed by operators and specific values. The data type of the attribute *numberOfWheels*, e.g. Integer, is determined

in the corresponding *UML* class diagram (Fig. 2.5) and not in the *OCL* expression itself.

3.2. OCL Invariant Constraints

This section is going to offer an overview on relevant kinds of *OCL* constraints. Constraints are divided into different types in order to clarify the involved elements which are restricted by a specific constraint. The identified types are categorized as follows:

- Invariant constraints for attributes
- Invariant constraints for associations and objects
- Implications

3.2.1. Invariant Constraints for Attributes

This type of *OCL* constraints deals exclusively with attributes. Furthermore, it is important to underline that constraints for attributes do not leave the rectangle of a modeled class in a class diagram. Basically, this means that constraints for attributes only affect a specific class and its own attributes (and no other attributes from other classes). Although section 3.1 has already shown an example for an attribute constraint aiming at the *Car* class, further examples with reference to Fig. 2.7 are going to enforce a deeper understanding.

3.2.1.1. Invariant Constraints for Numeric Values

First, we would like to say that a driver has to be at least 18 years old. The attribute *age* is defined as *Integer* data type in the class diagram. Operators for relational comparison and equality / inequality for *Integer* or other numeric data types are: $<$, $<=$, $>=$, $>$, $=$, and $<>$. The relevant *OCL* constraint is:

```
context Driver inv aDriverMustBe18:  
self.age >= 18
```

Furthermore, *OCL* provides possibilities to derive numeric values. For example, the restriction value 18 can also be expressed in alternative ways:

```
self.age >= 9 + 9
```

```
self.age >= 9 * 2
self.age >= 18 mod(19)
self.age >= 3 max(18)
self.age >= 18 max(3)
```

Please note: Possible operations differ between *Integer* and *Real* values. The above mentioned *modulo* operation can only be applied to *Integer* values whereas operations like *floor()* or *round()* work exclusively with *Real* values:

```
self.age >= (18.8).floor()
self.age >= (17.7).round()
```

As each *OC*L expression results in a *Boolean* value, it is also possible to combine several expressions with logical operators *and*, *or*, and *xor*. In this case, we build a frame of valid values with the *and* operator:

```
self.age >= 18 and self.age <= 100
```

Additionally, *OC*L provides a functionality to secure non-empty attribute values. This functionality can also be applied to *String* attributes or enumerations. The avoidance of empty values for attributes can makes sense in case of important attributes which are essential for the success of a process:

```
context Driver inv driverLastName:
self.lastName->notEmpty()
```

On the contrary, in some cases, it is also imaginable that an attribute's value must be empty:

```
self.lastName->isEmpty()
```

3.2.1.2. Invariant Constraints for Strings and Enumerations

In a next step, we would like to restrict the *first name String* attribute of the *Driver* class in a way that a driver's first name can not be *Max* or *Tim*:

```
context Driver inv driverFirstName:
self.firstName <> 'Max' or self.firstName <> 'Tim'
```

As the *first name* attribute has the data type *String*, we are logically restricted by the comparison operators *=* and *<>*. Additionally, *OC*L offers various other operators for the

type *String*. Examples are:

```
self.firstName.size() > 3
```

... implies that the length of a first name has to consist of more than three characters.

```
self.firstName.concat(' ').concat(self.lastName) <> 'Max Mueller'
```

... implies that the concatenation of the first name and last name attribute can not result in *Max Mueller*.

Regarding constraints for enumerations, it has to be clarified that *OCL* per se does not provide a specific functionality. This is the result of the fact that enumerations are already defined by a class diagram. For this reason, we adapt the attribute *carType* of the class *Car* presented in Fig. 2.5, 2.6, and 2.7 from data type *String* to a special data type *CarType*. Furthermore, we have to add a specification of the enumeration *CarType*:

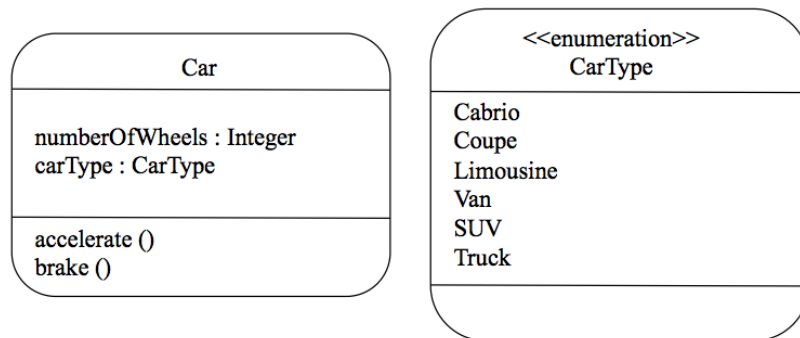


Figure 3.1.: Adapted class *Car* with new enumeration attribute *CarType*

Enumeration values are retrievable by a double colon. For example, we would like to restrict the *carType* enumeration attribute by saying that all values except *Cabrio* are permissible:

```
context Car inv carTypeCantBeCabrio:
self.carType <> self.carType::Cabrio
```

In cases if enumerations are not defined in the class diagram, enumeration constraints can also be created in an alternative (inconvenient) way. The previous example with no prior defined enumeration attributes in a class diagram would look like this:

```
context Car inv carTypeEnumeration:
self.carType = 'Coupe' or
self.carType = 'Limousine' or
self.carType = 'Van' or
```

```
self.carType = 'SUV' or  
self.carType = 'Truck'
```

3.2.2. Invariant Constraints for Associations and Objects

Constraints for associations allow the creation of inter-class constraints. This means that it is possible to leave a specific class and specify constraints for relationships to other classes. For example, we would like to specify that a car can only has a driver who is at least 18 years old and has a driving license (Fig. 2.7):

```
context Car inv driverAgeAndDrivingLicense:  
self.hasDriver.age >= 18 and  
self.hasDriver.drivingLicense = True
```

In contrast to previously mentioned examples regarding constraints for attributes, we access externally defined attributes of other classes. With the association *hasDriver* connected to the *Driver* class we can apply constraints to the *Driver* attributes *age* and *drivingLicense* by staying in the *Car* context.

Furthermore, *OCL* constraints allow restrictions for the multiplicity of associations. The multiplicity is originally set in the class diagram (Fig. 2.7). Within an *OCL* expression, it is possible to adapt the original multiplicity. For example, we would like to define that a driver has at least two and not more than five cars (the original multiplicity was *0 ... **):

```
context Driver inv hasCarGreaterAndLower:  
self.hasCar->size() >= 2 and  
self.hasCar->size() <= 5
```

As a result, we have changed the multiplicity from *0 ... ** to *2 ... 5*. However, this section combines constraints for associations and constraints for objects because they are directly connected in *OCL*. In the previous example, the multiplicity has direct impact on the creation of assigned objects because the *OCL* statement postulates minimum (set to three) and maximum (set to five) existences for *Car* objects related to a *Driver* object. We can also restrict multiplicities in a way, that a *Driver* class can not has any corresponding *Car* objects:

```
context Driver inv noHasCar:  
self.hasCar->isEmpty()
```

Or alternatively, that a *Driver* class must has at least one corresponding *Car* object:

```
context Driver inv hasCar:  
self.hasCar->notEmpty()
```

There are various possibilities for constraints which affect exclusively objects (without the way over associations). As a starting point, we would like to define a *Driver* object, with an age of 52 years, which has to exist:

```
context Driver inv thereIsADriverWith52Years:  
Driver.allInstances()->exists(d | d.age = 52)
```

The `->exists()` expression postulates that there is at least one element that makes the condition in the brackets true. On the contrary, the expression `->forAll()` states that all elements fulfill the condition.

If we want to quantify a specific amount of existences e.g. there must be at least three objects of the class *Driver* with an age of 52, the OCL expression is defined via a selection executed with `->select()` and followed by `->size()` restrictions:

```
context Driver inv thereAreAtLeast3DriversWith52Years:  
Driver.allInstances()->size(d | d.age = 52) >= 3
```

3.2.3. Implications

To make OCL constraints more case-specific, implications serve as additional functionality to allow a broader spectrum of restrictions. First, it has to be underlined that OCL implications represent an alternative way how constraints are going to be understood and evaluated. Until now, each constraint was per definition *active* as soon as it was written down. Henceforth, implications allow triggering mechanisms which force *latent* constraints being only activated if specific conditions arise. Generally in OCL, implications are indicated with the keyword *implies*. The following OCL constraint shows an example for an implication. Again, it is essential to note that the implication will only be activated if the association *hasCar* has a size greater than zero:

```
context Driver inv implicationToAgeAndDrivingLicense:  
Driver.hasCar->size() > 0 implies  
(Driver.age >= 18 and  
Driver.drivingLicense = True)
```

For the further work and the upcoming process of creating a Constraint-Modeling-Language, implications are going to be divided into a left- and right-hand-side schema. The schema follows the logic that the LHS has to be fulfilled to activate the constraint on the RHS:

LHS:

```
Driver.hasCar->size() > 0
```

IMPLICATION

RHS:

```
(Driver.age >= 18) AND (Driver.drivingLicense = True)
```

3.3. Visualization Approaches

This section provides a presentation about pre-existing approaches to visualize constraints. The focus does not lie on well prepared implementations and their applicability in praxis, but on theoretical concepts of constraint visualizations in the context of *UML* and *OCL*. Basically, this section is intended to get thought-provoking impulses regarding graphical methods of constraint modeling. With reference to [10], there are two different approaches discussed in the work: *VOCL* [25] and *Constraint Diagrams* [23].

3.3.1. VOCL

VOCL, which stands for *Visual OCL*, represents a graphical solution to create *OCL* constraints. The language was developed at the University of Berlin in 2002. It is conceptually based on the language description of Bottoni, Koch, Parisi-Presicce, and Taentzer [2]. By using *VOCL*, it strikes users of the language that the graphical notation generally follows the *UML* graphical representation for class diagrams. This analogy is quite intended to create a consistent language framework [25].

In *VOCL*, a constraint is visualized as a rounded rectangle. Like *UML* class diagrams, the rectangle is segmented into three vertically compartments: The first compartment on top of the rectangle indicates the *context* and can be seen as a header. The bottom two compartments represent the body and contain a condition [25].

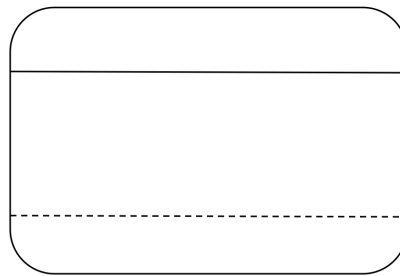
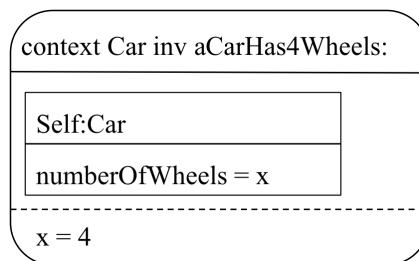


Figure 3.2.: Representation of a constraint in VOCL

In section 3.1, the thesis has presented the constraint *aCarHas4Wheels*. Instead of writing the constraint in textual form, it can also be modeled with VOCL:

Figure 3.3.: *aCarHas4Wheels*

context Car **inv**:
self.numberOfWheels = 4

At a first glance, the VOCL visualization seems to be very similar to its OCL textual equivalent. Nevertheless, there are some points in which the two approaches differ from each other. First, as a primary distinction, the condition in VOCL is divided into two parts whereas OCL provides only one conditioning segment:

OCL:

```
self.numberOfWheels = 4
```

VOCL:

```
self.numberOfWheels = x  
x = 4
```

By separating the affected attribute (*numberOfWheels*) and the specific attribute's value (4), the diagram becomes (subjectively) better readable because the effective restriction expressed in a value is always located at the bottom of the rectangle. In cases of calculations for the restriction value, this approach represents an advantage too because the diagram stays quite assessable.

In addition, the syntax is not absolutely identical: In the *OCL* example, the keyword *self* is followed by a specific attribute to be restricted. In *VOCL*, on the other side, *self* is followed by a colon (instead of a simple point) and the *context* class.

Furthermore, *VOCL* offers a notation for logical operators indicated by a vertical bar separating at least two conditions in the second compartment. In reference to the constraint *driverFirstName* (Section 3.2), the third compartment at the bottom assigns values to the variables *x* and *y* which are logically connected via an *or* operator. The unequal operator \neq in *OCL* is visualized as \neq in *VOCL*.

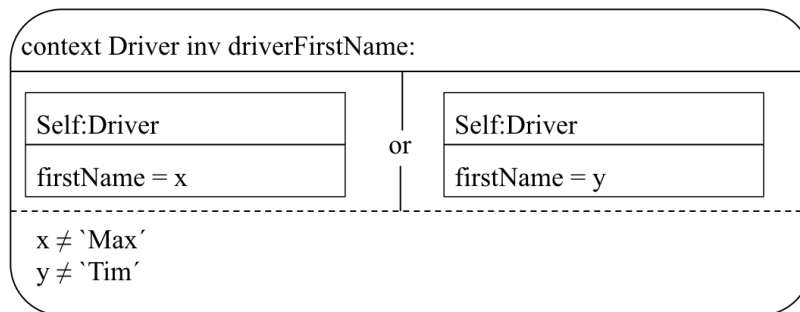


Figure 3.4.: *driverFirstName*

```
context Driver inv:
self.firstName <> 'Max' or
self.firstName <> 'Tim'
```

Associations are modeled very similar to *UML* class diagrams by connecting two relevant classes with a line indicating a relationship between them. To provide an example, the constraint *hasCar* (Section 3.2) can be modeled with two classes *Driver* and *Car* whereas *Car* and the association *hasCar* has the restriction to be $\rightarrow \text{notEmpty}()$. This is shown with the statement $\neq \emptyset$ and a dotted rectangle extending *Car*. On the contrary, $\rightarrow \text{Empty}()$ would be visualized as \emptyset .

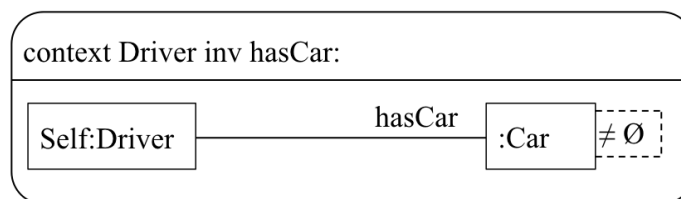


Figure 3.5.: *hasCar*

```
context Driver inv:
self.hasCar->notEmpty()
```

Constraints regarding the existence of objects were also previously described in Section 3.2. **Please note:** As far as *VOCL* is described in [25], the language does not provide

an equivalent to OCLs $\rightarrow allInstances()$. As a result, we assume that the VOCL notation includes a functionality for all instances by the keyword *allInstances*. On the contrary, the $\rightarrow exists$ operation is implemented in the language. It is indicated with the mathematical character \exists and added in a dotted rectangle to the condition. If the OCL constraint has been $\rightarrow forAll()$ instead of $\rightarrow exists()$, the VOCL notation would be \forall . First, we would like to model the constraint *thereIsADriverWith52Years*:

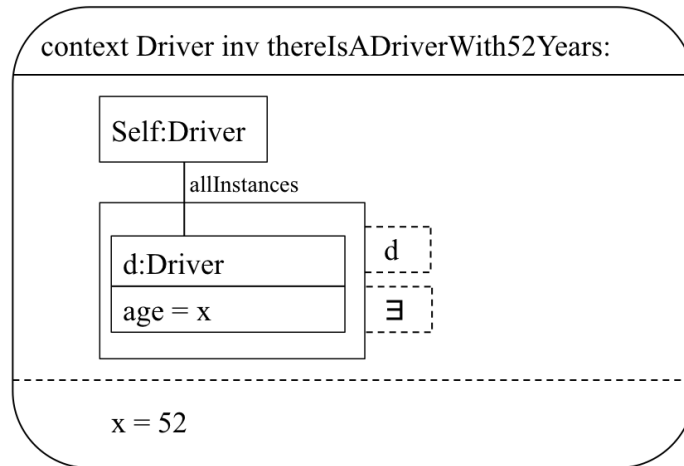


Figure 3.6.: *thereIsADriverWith52Years*

```
context Driver inv:
Driver.allInstances()
->exists(d | d.age = 52)
```

Next, we would like to visualize the constraint *thereAreAtLeast3DriversWith52Years*:

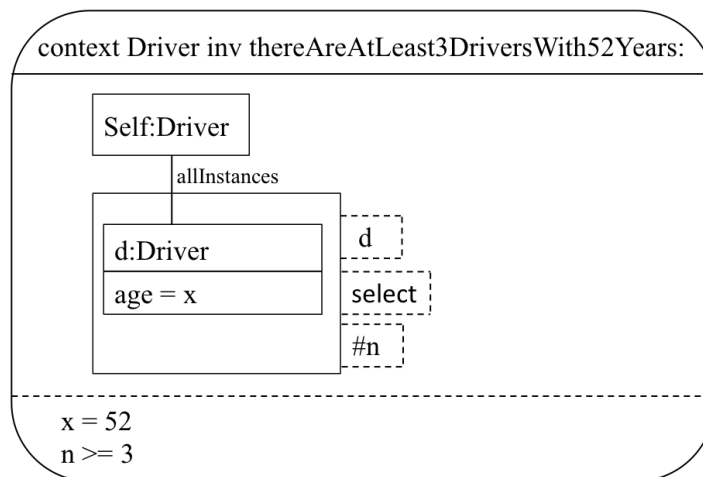


Figure 3.7.: *thereAreAtLeast3DriversWith52Years*

```
context Driver inv:
Driver.allInstances()
->size(d | d.age = 52) >= 3
```

The selection in *OCL* is visualized with an additional dotted rectangle in *VOCL*. The `->size()` expression is modeled as a `#` in combination with a variable (in this case `n`).

Last but not least, implications are also implemented in *VOCL*. They are shown as vertical bar separating the LHS and RHS. The keyword *id* does not exist in the *OCL* specification and is added in *VOCL* to graphically underline that the drivers above and below the keyword *implies* are the same [25]. The constraint *implicationToAgeAndDrivingLicense* from Section 3.2 in *VOCL*:

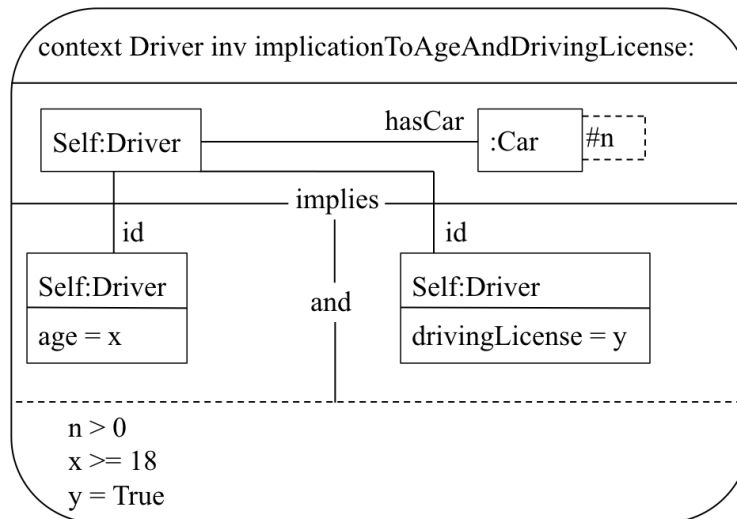


Figure 3.8.: *implicationToAgeAndDrivingLicense*

```
context Driver inv:
Driver.hasCar->size() > 0 implies
(Driver.age >= 18 and
Driver.drivingLicense = True)
```

3.3.2. Constraint Diagrams

Constraint Diagrams represent a method to replace mathematical formalizations to describe constraints in a more intuitive and practical way. The target group are software engineers who develop software without having a strong mathematical background. *Constraint Diagrams* are inspired by Venn diagrams and informal diagrams used by mathematicians for describing properties of functions and relations [23].

Constraint Diagrams are spider diagrams augmented by *arrows* and *wildcards*. While *arrows* are used for determining relationships between sets, *wildcards* implicate universal quantification. Arrows have a *label*, a *source* and a *target*. The *source* represents the set or element from which the navigation begins. In Fig. 3.9 the source of the relation arrow *r* is spider *x* whereas the target is represented by the circle *B*. The semantics of the diagram is $x.r = B$, where $x.r$ stands for $\{y:r(x, y)\}$ [11].

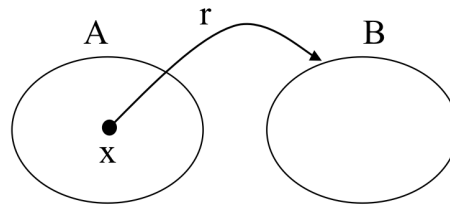


Figure 3.9.: Constraint Diagram with one arrow

The next example underlines the connection of *Constraint Diagrams* to Venn Diagrams and Euler Circles. In Fig. 3.10 the target circle is a circle contained in B . The semantics is $A.r \subseteq B$, where $A.r$ is shorthand for applying r to each element included in A followed by taking the *union* of the resulting sets. The target circle of r is a *derived set* and it is defined by the arrow r [11].

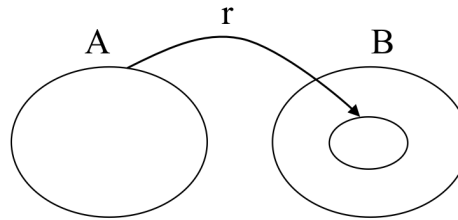


Figure 3.10.: Constraint Diagram with a derived circle

Regarding quantification, wildcards serve as method for describing all elements of a specific set. As an example, Fig 3.11 visualizes a diagram where for each element x in A $x.r$ and $x.h$ are disjoint. This can be formalized as: $\forall x \in A \bullet x.r \cap x.h = \{\}$ [11].

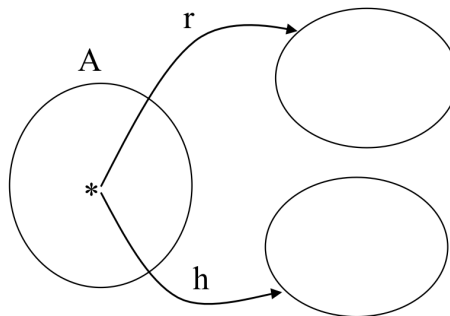


Figure 3.11.: Constraint Diagram with a derived circle

In a further step, we would like to visualize constraints which were previously modeled in VOCL with *Constraint Diagrams*. As a first preparation and in reference to [10], we make minor adaptations to the notation of *Constraint Diagrams*:

3. Overview on existing Methods to design and visualize Constraints

- In conformance with the *UML* notation, classes are modeled as rectangles instead of circles.
- For aspects of clarity and exact definition, particular constraints are surrounded by a rectangle.

At first glance, we model an initial constraint shown in Figure 3.3. *OCls* keyword *self* does not appear in the notation of *Constraint Diagrams* because it is indicated, in this case, by element *c* which in turn implies *for all c:Car*. Besides the class *Car*, the attribute *numberOfWheels* is modeled as rectangle too because it belongs to the class *Integer* for data types.

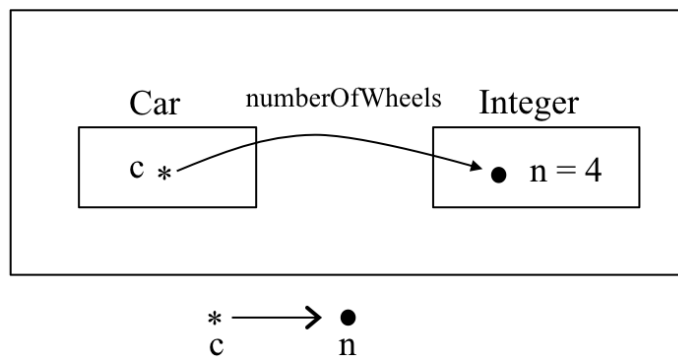


Figure 3.12.: *aCarHas4Wheels*

context Car **inv**:
self.numberOfWheels = 4

In Fig. 3.4, the thesis has shown the *VOCL* visualization for a constraint restricting the *firstName* attribute of the class *Driver*. Compared to *VOCL*, the same constraint can be modeled with *Constraint Diagrams*.

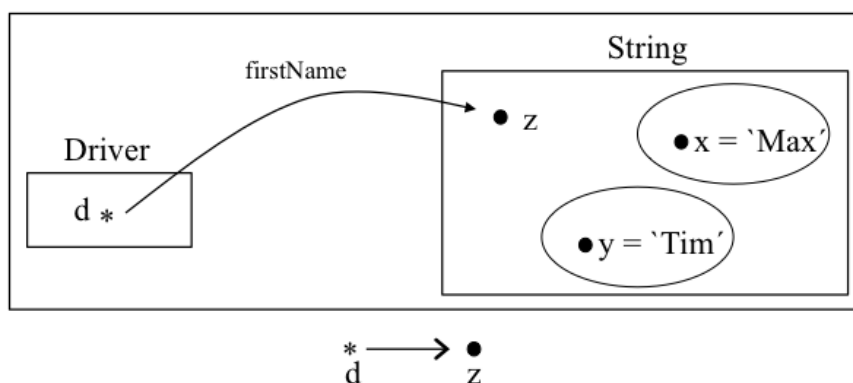


Figure 3.13.: *driverFirstName*

context Driver **inv**:
self.firstName <> 'Max' **or**
self.firstName <> 'Tim'

Fig. 3.13 underlines the fundamental distinction between the approaches of *VOCL* and *Constraint Diagrams*: While the notation of *VOCL* appears like a graphical list of conditions, *Constraint Diagrams* pursues a mathematical approach inspired by Venn diagrams and Euler Circles. As a result, *Constraint Diagrams* distinguish conditions in a way how they visualization is specifically done. In Fig. 3.13, the class *String* contains elements with the manifestation *Max* or *Tim*, but the relevant element *z* is located outside the circles of *x* and *y* which implies that the *firstName* attribute can not be *Max* or *Tim*.

In a next step, we deal with *UML* associations and how *OCL* constraints for associations can be modeled with *Constraint Diagrams*. As a reference, we select Fig. 3.5 and the corresponding *OCL* constraint. In contrast to *UML* or *VOCL*, associations can be drawn as circle indicating elements with a specific association e.g. *hasCar*. The next diagram describes the condition that a driver must has an association to a car. This is done by placing wildcard *d* in the intersection between *hasCar* and *Car*.

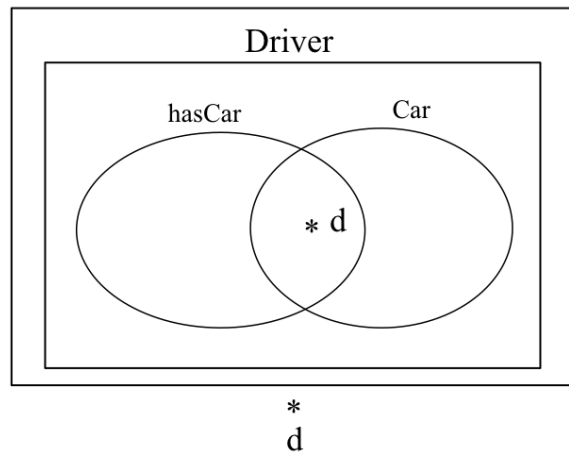


Figure 3.14.: *hasCar*
context Driver **inv:**
 self.hasCar->notEmpty()

The next diagram is very similar to Fig. 3.12. However, by replacing the wildcard *** to a black dot, we change the semantics from all elements of a set to a specific element of a set. This procedure meets *OCls* expression *->exists*. Furthermore, there is no need for *OCls* expression *allInstances()* because the dot implies both *->exists* and *allInstances()* which means $\exists d \in \text{Driver}$.

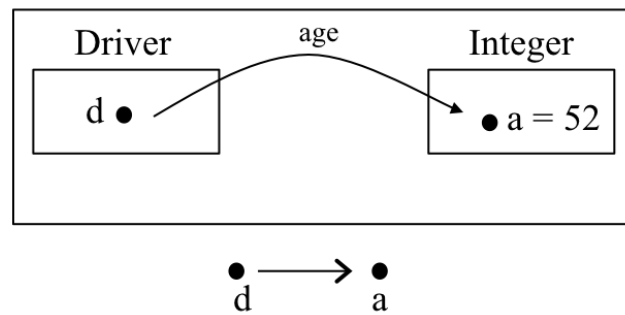


Figure 3.15.: *thereIsADriverWith52Years*

```
context Driver inv:
Driver.allInstances()
->exists(d | d.age = 52)
```

The next example, in reference to Fig. 3.7, deals with *OCls* expression *->size()* in order to describe a restriction that there must be at least three drivers with an age of 52 years. This constraint can be modeled with *Constraint Diagrams* by using three dots *d*, *e*, and *f* where $\{d, e, f\} \in \text{Driver}$. **Please note:** The proper meaning of the constraint describing a *->size* ≥ 3 can not be modeled exactly with *Constraint Diagrams*. As a result, we have to adapt the operator \geq to $=$.

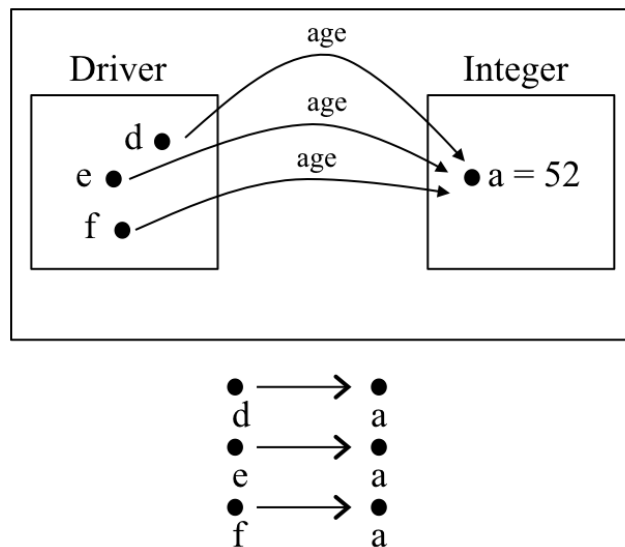


Figure 3.16.: *thereAreAtLeast3DriverWith52Years*

```
context Driver inv:
Driver.allInstances()
->size(d | d.age = 52) >= 3
```

Last but not least, we attend to the example from Fig. 3.8 which is about an implication between the size of the association *hasCar* and the driver's *age* as well as the *drivingLicense* attribute. While modeling Fig. 3.16, we have experienced that we can not visualize

->size() with relational operators $<$, $<=$, $>$, $>=$. This issue is relevant for the next example too because Fig. 3.8 states that the size of the association *hasCar* must be at least three. Consequently, we have to adapt the ->size() from > 0 to $= 1$.

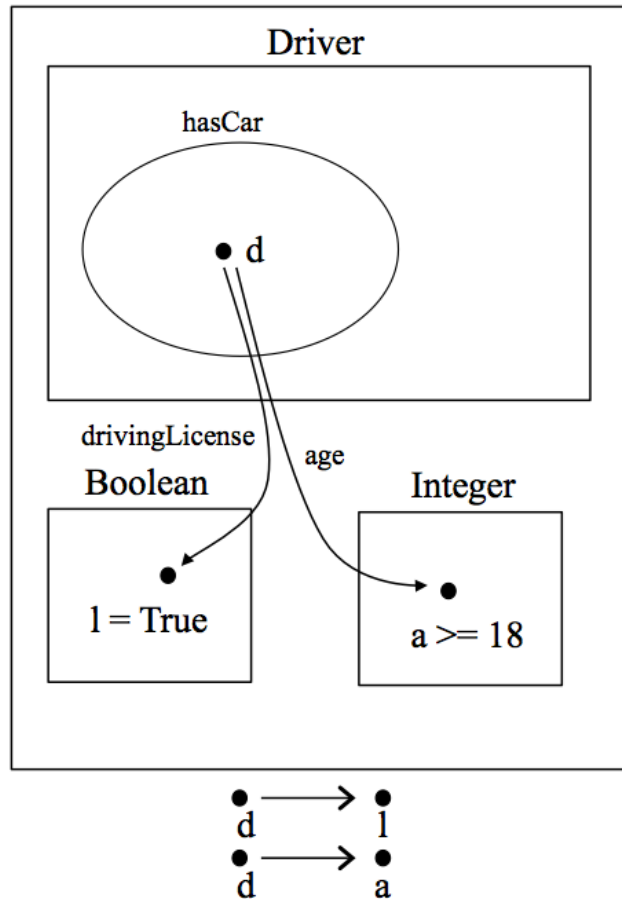


Figure 3.17.: *implicationToAgeAndDrivingLicense*
context Driver **inv**:
 Driver.hasCar->size() = 1 implies
 (Driver.age >= 18 **and**
 Driver.drivingLicense = True)

3.4. Conclusion

The first research question acts as fundamental basis for a further process of designing a Constraint-Modeling-Language and implementing a modeling method. The first essential point was identifying the spectrum of *OCL* constraints. Further on, we have made distinctions between various kinds of *OCL* constraints: Constraints affecting attributes, constraints for associations and constraints regarding objects. In addition, we have shown that *OCL* constraints are able to leave a statical environment and have dynamical charac-

teristics by triggering conditions via implications. By separating implications to a LHS and RHS, we have laid the foundation for processing implicated constraints in the Constraint-Modeling-Language.

Next, we have introduced two distinct visualizing approaches for *OCL* constraints: *VOCL* and *Constraint Diagrams*. While *VOCL* represents a notation inspired by *UML* class diagrams, *Constraint Diagrams* and their notation are settled within an obvious mathematical environment including Venn Diagrams, Euler Circles and the visualization of set theory. Besides the fact that both approaches are very different, both of them are an inspiration for the generation of an own modeling language for constraints. Nevertheless, we were confronted with some aspects of *VOCL* and *Constraint Diagrams* which have restricted the applicability for arbitrary *OCL* expressions. On the one hand, regarding *VOCL*, we have made assumptions how to model *OCLs allInstances*. On the other hand, *Class Diagrams* were not able to model *OCLs* $\rightarrow size()$ expression combined with operators $<$, $<=$, $>$, $>=$.

4. Conceptualization of the Constraint-Modeling-Language

The following two chapters are closely related to each other in regards to Figure 2.2 and the process of generating a modeling method. This means that both, the conceptualization and implementation, are part of an iterating process for modeling method evolution. While this chapter deals with the theoretical concept of the Constraint-Modeling-Language, the third research question is going to simultaneously evaluate the feasibility of the theory. This procedure serves the purpose of avoiding cases in which the concept is not able to be implemented in *ADOxx*.

4.1. Modeling Process Integration

At first glance, it has to be defined in which step of a modeling process the Constraint-Modeling-Language steps in. For this approach, Figure 2.4 serves as foundation as it defines four layers of modeling in *ADOxx*:

1. *The ADOxx Meta² Model-Layer*
2. *The ADOxx Meta Model-Layer*
3. *The User Specific Meta Model-Layer*
4. *The Model-Layer*

Since the first and second layers are created by *ADOxx Developers*, these layers are not relevant for the concept of the modeling language. Moreover, the Constraint-Modeling-Language is intended to extend an *User Specific Meta Model* on meta model level. This extension is going to be realized with a specific *Constraint Model*. As a result, the *Model* can be validated against the *Constraint Model* to check if all defined constraints are fulfilled.

4. Conceptualization of the Constraint-Modeling-Language

The previously mentioned approach of extending a meta model with a constraint model leads to the following procedure of modeling:

1. Definition of a meta model
2. Definition of a constraint model
3. Creation of a specific model

In addition, this procedure implies that also the *Constraint Model* is created by a *Meta Modeller* (in contrast to the *Model* which is created by a *Modeller*). As a consequence, the definition of the *Constraint Model* is made before a specific model is generated. This means that the *Meta Modeller* must have knowledge about useful constraints as the *Meta Modeller* does not know the specific model generated by the *Modeller*.

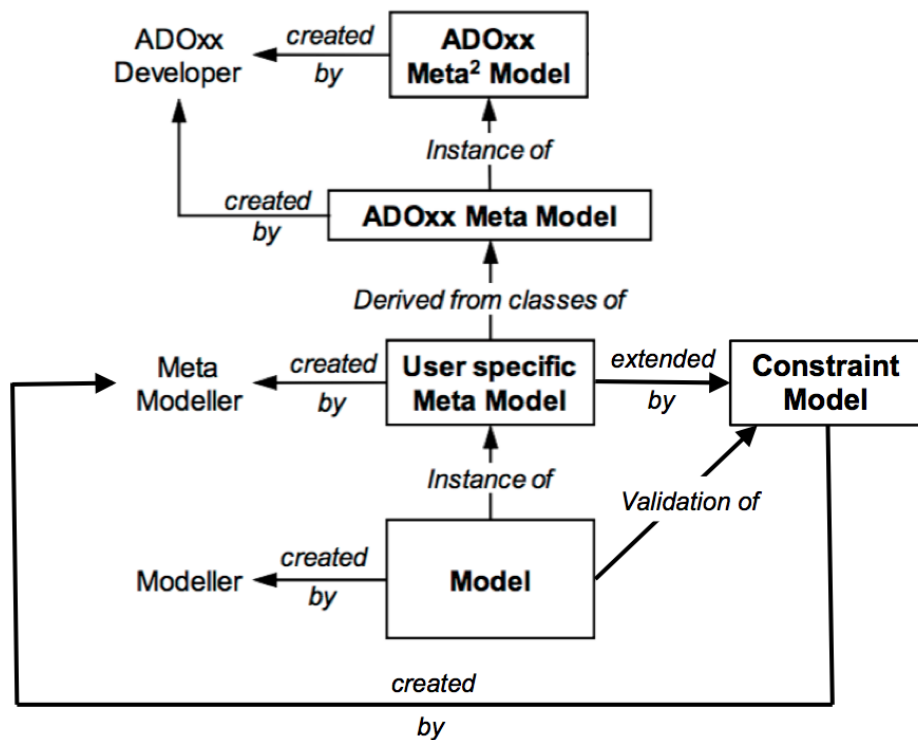


Figure 4.1.: Constraint model extension of the figure in [8]

4.2. Coherence between Constraint Model and Meta Model

For the further work, it is essential to describe a constraint meta model and interactions between constraint models and common meta models. As already mentioned in Fig 4.1, a *Constraint Model* extends an *User specific Meta Model* while it can be validated against a simple *Model*. But how is the meta constraint model organized? And furthermore, how does a constraint model assigns constraints to models? To provide a fundamental answer to these questions, a schematic visualization of the constraint assignment process is going to be used. It is exemplary described in context of a *Car Meta Model*.

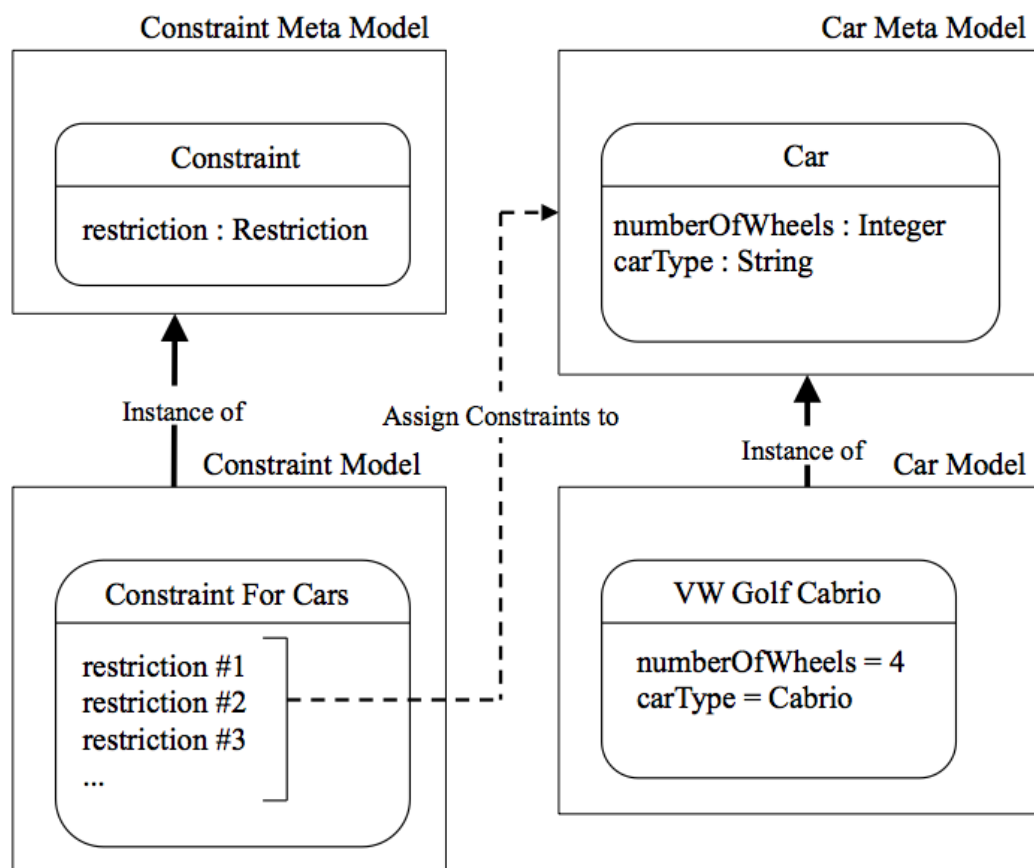


Figure 4.2.: Coherence between Constraint Meta Model, Constraint Model, Meta Model, and Model

The *Constraint Meta Model* consists of a *Constraint Class* which includes one or more *restrictions* of special type *Restriction*. The explanation of this special type is extensively done in section 4.4 - in simplified terms a *restriction* embodies a constraint for a broad spectrum of application scenarios e.g. a restriction for a class, a model type, an attribute value etcetera. The instance of a *Constraint Meta Model* is represented by a *Con-*

straint Model defining explicit restrictions in a corresponding *Meta Model* implicated by `<<assignConstraintsTo>>`. As a result, the *Meta Model* is enriched with constraints by the *Constraint Model*. By creating instances from the *Meta Model* e.g. a *Car Model*, the constraints are going to be inherited by the *Model*.

4.3. Existing Methods for Constraints in ADOxx

For the sake of completeness, it has to be mentioned that *ADOxx* already provides functionalities to implement rudimentary constraints and restrictions. These constraints are based on commands of the class attribute *Class cardinality*. This attribute contains a cardinality definition of a selected class and describes (1) the minimal/maximal number of objects of this class per model and (2) the minimal/maximal number of relations of a specific type, incoming or outgoing from an object [13].

For example, we can describe the following case with the *Class cardinality* attribute in *ADOxx*:

There has to be at least one object of the class Car up to a maximum of five objects. Furthermore, there has to be at least one outgoing relation of relation class hasDriver from objects of class Car. In addition, there can not be an incoming relation of class hasDriver to objects of the class Car.

In the class Car, the following statement has to be entered:

```
CARDINALITIES min-objects:1 max-objects:5  
RELATION hasDriver min-outgoing:1 max-incoming:0
```

As a result, we have seen that basic constraints can already be generated in *ADOxx* with the *Class cardinality* attribute. The Constraint-Modeling-Language is going to cover a much broader spectrum of possible constraints to enforce a profound definition of restrictions for arbitrary models. Moreover, it is going to skip the functionality of *Class cardinality* because it will offer an equivalent in order to represent a holistic modeling language for constraints. Consequently, it can also be implemented in alternative modeling platforms.

4.4. Language Scope and Specification

The Constraint-Modeling-Language is designed to assign constraints to model types, classes, relation classes and combinations of these. Furthermore, there are various kinds of dif-

ferent restriction which can be applied. This chapter is going to describe the complete scope of the language sorted by particular restriction types. The combination of multiple restrictions to one single constraint will be textually and visually discussed in the next section.

As Fig. 4.2 has shown, a *Constraint* class includes one or more *restrictions* which can be divided into four meta restriction types:

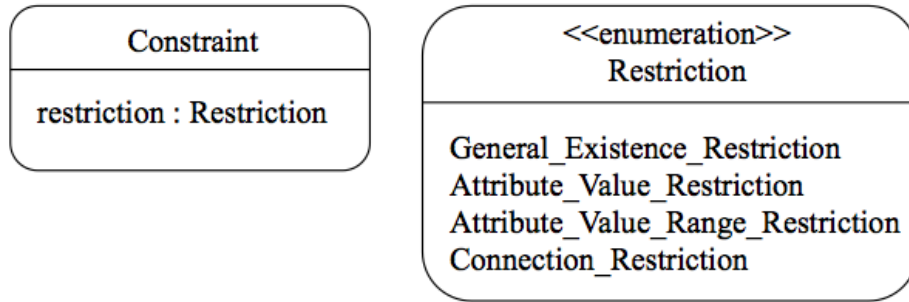


Figure 4.3.: Meta restriction types defined in a constraint

Furthermore, the four meta restriction types can be partitioned into concrete restriction types:

- **General Existence Restriction:**
 - Instance Existence / Non-Existence
 - Cardinality Existence / Non-Existence
 - Attribute Existence / Non-Existence
- **Attribute Value Restriction:**
 - Attribute Value Existence / Non-Existence
- **Attribute Value Range Restriction:**
 - Attribute Value Range Existence / Non-Existence
- **Connection Restriction:**
 - Connection Existence / Non-Existence

In order to give a specific description of each concrete restriction type, *OC*L equivalents are used. These *OC*L statements enforce a quick understanding implicated by the fact that *OC*L by itself is very self-explanatory. In addition, the exemplary *OC*L equivalents are, in some cases, not conform with the *OC*L specification but serve as basis for explanation.

1. **Instance Existence:** This type postulates the existence of a determined instance. On the one hand, in cases of model types, an instance is represented by a model. On the other hand, instances of classes and relation classes appear as objects. The con-

straint semantically expresses a condition in which a least one instance of a model type, class or relation class has to exist. As a result, the lower bound of quantitative instances is > 0 while the constraint does not care about the upper bound.

Applicable to: model types, classes, relation classes

OCL equivalent: context $\langle \text{MT} \mid \text{Class} \mid \text{Relation Class} \rangle$ inv:
allInstances() ->exists()

Example: There must be at least one instance of the model type *Car Model*.

2. **Instance Non-Existence:** This type postulates the non-existence of a specific instance. In contrast to *Instance Existence*, this constraint represents the opposite. In particular, it implies that a defined instance of a model type, class or relation class can not exist. Consequently, the amount of existing instances must be < 1 .

Applicable to: model types, classes, relation classes

OCL equivalent: context $\langle \text{MT} \mid \text{Class} \mid \text{Relation Class} \rangle$ inv:
allInstances() ->isEmpty()

Example: Objects of the class *Car* can not exist.

3. **Cardinality Existence / Non-Existence:** This type describes the existence or non-existence of specific instances in a range of values. The constraint can be seen as extension and combination of the previous two mentioned types. It is possible to fully cover the semantics of *Instance Existence* and *Instance Non-Existence* but it offers additional functionality. When explicitly compared to *Instance Existence*, it offers the possibility to set an upper bound for the amount of instances e.g. the amount of modeled instances must be > 0 and < 10 . Moreover, the restriction < 10 is towards *Instance Non-Existence* as it states a valid upper bound for the appearance of instances.

Applicable to: model types, classes, relation classes

OCL equivalent: context $\langle \text{MT} \mid \text{Class} \mid \text{Relation Class} \rangle$ inv:
allInstances() ->size() $\langle \text{numeric_operator} \rangle \langle \text{integer} \rangle$
[
AND inv:
allInstances() ->size() $\langle \text{numeric_operator} \rangle \langle \text{integer} \rangle$
]

Explantation:

- $\langle \text{numeric_operator} \rangle$ refers to operators $<$, \leq , \geq , $>$, $=$, and $<>$
- Square brackets [] indicate an additional and optional statement

- Cardinality existence provides an AND logic operator to connect statements

Example: There must exist at least three to a maximum of five objects of the class *Car*.

4. **Attribute Existence:** This type postulates the existence of a specific attribute. It can be seen as predecessor of the following constraint types which restrict particular attribute values. In contrast to them, this type only states the existence of an attribute which means that a defined attribute has to appear in a model type, class or relation class. Furthermore, it has to be clarified that if constraint types for particular attribute values are used, this constraint type is obsolete because a specification of an attribute value implies the existence of the correlative attribute.

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<attribute> ->exists()

Example: Objects of the class *Car* must have the attribute *numberOfWheels*.

5. **Attribute Non-Existence:** This type postulates the non-existence of a specific attribute. Again, the constraint implies the opposite of the *Attribute Existence* type. In consequence, it states that a defined attribute can not be part of a model type, class or relation class.

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<attribute> ->isEmpty()

Example: Objects of the class *Car* can not have the attribute *First Name*.

6. **Attribute Value Existence:** This constraint type implies the existence of a specific attribute value. It could be seen as abstract super class of constraint types regarding attribute values. As a result, it can not be practically applied because there are multiple different attribute types in *ADOxx* which must be individually handled and specified.

- 6.1. **Integer Value Existence:** This type postulates the existence of a particular value for an Integer attribute. In *ADOxx* Integers are defined as an integer from -1,999,999,999 to 1,999,999,999. The amount of digits is limited to 10 plus an optional + or – sign. The standard value is 0 or an alternatively determined value

[12].

Applicable to: model types, classes, relation classes

OCLEquivalent: context <MT | Class | Relation Class> inv:
self.<Integer_Attribute> = <Integer>

Example: The attribute *numberOfWheels* of objects from class *Car* must be 4.

- 6.2. **Double Value Existence:** This type postulates the existence of a defined value for a Double attribute. In ADOxx, a Double attribute is defined for a float within +/- 999,999,999,999,999 for an integer or +/- 999,999,999.999999 for figures with 6 decimals. The standard value for Double is 0.000000. It should not exceed 15 significant digits with at last 6 decimal digits [12].

Applicable to: model types, classes, relation classes

OCLEquivalent: context <MT | Class | Relation Class> inv:
self.<Double_Attribute> = <Double>

Example: The attribute *Oto100* of objects from class *Sports Car* must be 7.567.

- 6.3. **String Value Existence:** This type postulates the existence of a specific value for a String attribute. Attributes of type String are defined for texts up to 3.700 characters of any type. For the String attribute *name*, the maximum amount of characters is reduced by 250.

Applicable to: model types, classes, relation classes

OCLEquivalent: context <MT | Class | Relation Class> inv:
self.<String_Attribute> = <String>

Example: The attribute *firstName* of objects from class *Driver* must be *Mueller*.

- 6.4. **Longstring Value Existence:** This type implies the existence of a specific value for a Longstring attribute. In contrast to String attributes, which are generally limited by 3.700 characters, Longstring Attributes can cover texts up to 32.000 characters [12].

Applicable to: model types, classes, relation classes

OCLEquivalent: context <MT | Class | Relation Class> inv:
self.<Longstring_Attribute> = <Longstring>

Example: The attribute *carDetails* of objects from class *Car* must be *<here stands long text>*.

- 6.5. **Time Value Existence:** This type postulates the existence of a determined value for a Time attribute. In *ADOxx*, the Time format is defined as YY:MM:DDD:HH:MM:SS [12].

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<Time.Attribute> = <Time>

Example: The attribute *dateBuilt_time* of objects from class *Car* must be *02:001:00:00:01*.

- 6.6. **Date Value Existence:** This type postulates the existence of a specific value for a Date attribute. The Date format is described in the form YYYY:MM:DD [12].

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<Date.Attribute> = <Date>

Example: The attribute *dateBuilt_date* of objects from class *Car* must be *2002:01:01*.

- 6.7. **Datetime Value Existence:** This type implies the existence of a specific value for a Datetime attribute. In *ADOxx*, the Datetime format is has the form YYYY:MM:DD HH:MM:SS [12].

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<Datetime.Attribute> = <Datetime>

Example: The attribute *dateBuilt_datetime* of objects from class *Car* must be *2002:01:01 00:00:01*.

- 6.8. **Enumeration Value Existence:** This type postulates the existence of a specific selected value for an Enumeration attribute. In *ADOxx*, an Enumeration attribute is characterized by a set of values. An Enumeration attribute has exactly one value of this set [12]. In the meta model, Enumeration values are defined in the *EnumerationDomain* which can be found in the *Facets* chapter of the Enumeration attribute. The method of defining enumerations includes writ-

ing multiple selection values separated by a @ e.g. with reference to Fig. 3.1: Cabrio@Coupe@Limousine@Van@SUV@Truck.

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<EnumerationAttribute> = <Enumeration>

Example: The attribute *carType* of objects from class *Car* must be *Coupe*.

- 6.9. **Enumerationlist Value Existence:** This type implies the existence of one or more specifically selected values for an Enumerationlist attribute. In contrast to Enumeration attributes, an Enumerationlist attribute can have more than one selected value of a defined set [12]. The way of describing an Enumeration list in the meta model is the same as for Enumeration attributes.

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<EnumerationlistAttribute> = <Enumerationlist>

Example: The attribute *carFeatures* of objects from class *Car* must be *Xenon*.

- 6.10. **Programmcall Value Existence:** This type implies the existence of specific values (items and parameters) for a Programmcall attribute. Items are related to *AdoScripts* which can be called and executed over the user interface [12].

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<ProgrammcallAttribute> = <Programmcall>

Example: The attribute *callProgramm* of objects from class *Car* must have the Program arguments "C:\Programme\Test\test.exe".

- 6.11. **Expression Value Existence:** This type postulates the existence of a specific value for an Expression attribute. Expressions are formulas which can not be longer than 3.600 characters [12]. For example, they are used to calculate results of attribute values.

Applicable to: model types, classes, relation classes

OCL equivalent: context <MT | Class | Relation Class> inv:


```
self.<Expression.Attribute> = <Expression>
```

Example: The attribute *deriveCarAge* of objects from class *Car* must have the expression *<expression>*.

- 6.12. **Interref Value Existence:** This type describes the existence of a intermodel reference. The restriction refers to the *Refdomain* which can include references to Model Types and Model Types plus appearing Objects.

Applicable to: model types, classes

OCL equivalent: context <MT | Class> inv:
 self.<Interref.Attribute>.<Refdomain> = <Modref | Objref>

Example: Objects from the class *Car* must have an Interref attribute pointing at Model Type *Garage Model*.

- 6.13. **Record Value Existence:** This type postulates the existence of a specific row in a table. The table columns are built with attributes assigned in the table class.

Applicable to: model types, classes

OCL equivalent: context <MT | Class> inv:
 self.<Record.Class>.<Record.Attribute1> = <value>
 AND
 self.<Record.Class>.<Record.Attribute2> = <value>

Example: The table *Car Table* must include a row with attributes *a1* = 3 and *a2* = 10.

7. **Attribute Value Non-Existence:** This constraint type implies the non-existence of a specific attribute value. As it follows the same logic and procedure as in 6.1 till 6.13, a further explanation is going to be left out, since there is practically only the operator changing from = to <>. Regarding semantics, this constraint type states conditions to exclude particular defined values for attributes.
8. **Attribute Value Range Existence / Non-Existence:** The following constraint types restrict attribute values with a frame of valid properties. E.g. for numeric values, operators = and <> get extended by <, <=, >=, and >. In addition, logical operators AND as well as OR are added to formulate more expressive restrictions. **Please note:** In the context of the Constraint-Modeling-Language, the proper meaning of the OR operator corresponds to XOR. As a result, OR means that only one condition of a set of various conditions can appear (which is strictly spoken the meaning of XOR).

- 8.1. **Integer Value Range Existence / Non-Existence:** This type postulates a frame of valid values for an Integer attribute. It is an extension of a basic Integer value Existence or Non-Existence constraint. The frame is built with numeric operators and logic operators *AND* as well as *OR*.

Applicable to: model types, classes, relation classes

Numeric operators: <, <=, >=, >, =, and <>

Logic operators: AND, OR

OCL equivalent: context <MT | Class | Relation Class> inv:
 self.<Integer_Attribute><numeric_operator><Integer>
 [
 <AND | OR>
 self.<Integer_Attribute><numeric_operator><Integer>
]

Example: The attribute *numberOfSeats* of objects from class *Car* must be greater than 0 and less than 10.

- 8.2. **Double Value Range Existence / Non-Existence:** This type postulates a frame of valid values for a Double attribute. The frame follows the same logic as *Integer Value Range Existence / Non-Existence* constraints.

Applicable to: model types, classes, relation classes

Numeric operators: <, <=, >=, >, =, and <>

Logic operators: AND, OR

OCL equivalent: context <MT | Class | Relation Class> inv:
 self.<Double_Attribute><numeric_operator><Double>
 [
 <AND | OR>
 self.<Double_Attribute><numeric_operator><Double>
]

Example: The attribute *0to100* of objects from class *Sports Car* must be greater than 3.5 and less than 9.5.

- 8.3. **String Value Range Existence / Non-Existence:** This type postulates a frame of valid values for a String attribute. In contrast to the previous mentioned two con-

straint types which are dealing with a frame for numeric values, this constraint type is about a frame for String values. As a result, comparison operators are set to String operators which offer =, and <>. Regarding logic operators, only OR is applicable.

Applicable to: model types, classes, relation classes

String operators: =, and <>

Logic operators: OR

OCL equivalent:

```
context <MT | Class | Relation Class> inv:
    self.<String.Attribute><string_operator><String>
    [
        OR
        self.<String.Attribute><string_operator><String>
    ]
```

Example: The attribute *First Name* of objects from class *Driver* must be *Mueller* or *Mustermann*.

- 8.4. **Longstring Value Range Existence / Non-Existence:** This type postulates a frame of valid values for a Longstring attribute. It follows the same logic as the *String Value Range Existence / Non-Existence* constraint.

Applicable to: model types, classes, relation classes

String operators: =, and <>

Logic operators: OR

OCL equivalent:

```
context <MT | Class | Relation Class> inv:
    self.<Longstring.Attribute><string_operator><Longstring>
    [
        OR
        self.<Longstring.Attribute><string_operator><Longstring>
    ]
```

Example: The attribute *carDetails* of objects from class *Car* can not be <here stands long text> or <here stands alternative long text>.

- 8.5. **Time Value Range Existence / Non-Existence:** This type postulates a frame of valid values for a Time attribute. As it is about the restriction of numeric values,

numeric operators are used and the full spectrum of logic operators is available.

Applicable to: model types, classes, relation classes

Numeric operators: <, <=, >=, >, =, and <>

Logic operators: AND, OR

OCL equivalent:

```
context <MT | Class | Relation Class> inv:
    self.<Time_Attribute><numeric_operator><Time>
    [
        <AND | OR >
        self.<Time_Attribute><numeric_operator><Time>
    ]
```

Example: The attribute *dateBuilt_time* of objects from class *Car* must be between 02:001:00:00:01 and 16:001:00:00:01.

- 8.6. **Date Value Range Existence / Non-Existence:** This type postulates a frame of valid values for a Date attribute. Again, Date attributes are of type numerical.

Applicable to: model types, classes, relation classes

Numeric operators: <, <=, >=, >, =, and <>

Logic operators: AND, OR

OCL equivalent:

```
context <MT | Class | Relation Class> inv:
    self.<Date_Attribute><numeric_operator><Date>
    [
        <AND | OR >
        self.<Date_Attribute><numeric_operator><Date>
    ]
```

Example: The attribute *dateBuilt_date* of objects from class *Car* must be greater than 2002:01:01 and less than 2016:01:01 .

- 8.7. **Datetime Value Range Existence / Non-Existence:** This type postulates a frame of valid values for a Datetime attribute with a potential restriction stated by numeric operators.

Applicable to: model types, classes, relation classes

Numeric operators: <, <=, >=, >, =, and <>

Logic operators: AND, OR

OCL equivalent: context <MT | Class | Relation Class> inv:
 self.<Datetime.Attribute><numeric_operator><Datetime>
 [
 <AND |OR >
 self.<Datetime.Attribute><numeric_operator><Datetime>
]

Example: The attribute *dateBuilt.datetime* of objects from class *Car* must be between *2002:01:01 00:00:01* and *2016:01:01 00:00:01*.

- 8.8. **Enumeration Value Range Existence / Non-Existence:** This type deals with a frame of valid values for an Enumeration attribute. As it was mentioned in *Enumeration Value Existence*, this kind of attribute allows only a single value selection made by the modeler. In order to adapt a value range for this attributes, an OR logic operator can be used.

Applicable to: model types, classes, relation classes

String operators: =, and <>

Logic operators: OR

OCL equivalent: context <MT | Class | Relation Class> inv:
 self.<Enumeration.Attribute><string_operator>
 <Value of enumeration>
 [
 OR
 self.<Enumeration.Attribute><string_operator>
 <Value of enumeration>
]

Example: The attribute *carType* of objects from class *Car* must be *Coupe* or *SUV*.

- 8.9. **Enumerationlist Value Range Existence / Non-Existence:** This type defines a frame of valid values for an Enumerationlist attribute. In contrast to Enumeration attributes, a range of possible values can be additionally realized with an AND logic operator.

Applicable to: model types, classes, relation classes

String operators: =, and <>

Logic operators: AND, OR

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<Enumerationlist_Attribute><string-operator>
<Value of enumeration>
[
<AND | OR>
self.<Enumerationlist_Attribute><string-operator>
<Value of enumeration>
]

Example: The attribute *carFeatures* of objects from class *Car* must be *Xenon* and *Navigation Sytem*.

- 8.10. **Programmcall Value Range Existence / Non-Existence:** This type postulates a frame of valid values for a Programmcall attribute. This type is quite similar to the type of Enumeration attributes since it exclusively permits a value range created with OR logic operators.

Applicable to: model types, classes, relation classes

String operators: =, and <>

Logic operators: OR

OCL equivalent: context <MT | Class | Relation Class> inv:
self.<Programmcall_Attribute><string-operator>
<Programmcall>
[
OR
self.<Programmcall_Attribute><string-operator>
<Programmcall>
]

Example: The attribute *callProgramm* of objects from class *Car* must have the *Program arguments* "C:\Programme\Test\test.exe" or "C:\Programme\Test\test2.exe".

- 8.11. **Expression Value Range Existence / Non-Existence:** This type states a frame of valid values for an Expression attribute. It follows the same logic as the type for Programmcall attributes.

Applicable to: model types, classes, relation classes

String operators: =, and <>

Logic operators: OR

OCL equivalent: context <MT | Class | Relation Class> inv:
 self.<Expression.Attribute><string.operator>
 <Expression>
 [
 OR
 self.<Expression.Attribute><string.operator>
 <Expression>
]

Example: The attribute *deriveCarAge* of objects from class *Car* must have the expression <expression1> or <expression2>.

- 8.12. **Interref Value Range Existence / Non-Existence:** This type deals with a frame of valid values for an Interref attribute. Again, multiple different values can be logically connected via *OR*.

Applicable to: model types, classes

String operators: =, and <>

Logic operators: OR

OCL equivalent: context <MT | Class> inv:
 self.<Interref.Attribute><Refdomain><string.operator>
 <Modref | Objref>
 [
 OR
 self.<Interref.Attribute><Refdomain><string.operator>
 <Modref | Objref>
]

Example: Objects from the class *Car* must have an Interref attribute pointing at Model Type *Garage Model* or *Driver Model*.

- 8.13. **Record Value Range Existence / Non-Existence:** This type postulates a frame of valid rows in a table. As a result, an *AND* logic operator has to be available to

enable multiple simultaneously existing rows.

Applicable to: model types, classes

String operators: =, and <>

Logic operators: AND, OR

OCL equivalent: context <MT | Class> inv:
(
 self.<RecordClass><RecordAttribute1><string_operator>
 <value>
 AND
 self.<RecordClass><RecordAttribute2><string_operator>
 <value>
)
<AND | OR>
(
 self.<RecordClass><RecordAttribute1><string_operator>
 <value>
 AND
 self.<RecordClass><RecordAttribute2><string_operator>
 <value>
)

Example: The table *Car Table* must include a row with $a1 = 3$ and $a2 = 10$. Furthermore, there has to be a row with $a1 = 12$ and $a2 = 0$.

9. **Connection Existence:** This constraint type states the existence of a relationship between objects of two classes. This allows the modeler the creation of predefined patterns e.g. objects of class *Car* must always have a relationship to objects of class *Driver*. Furthermore, it will be possible to set constraints regarding attributes for this previously mentioned and specific relationship.

Applicable to: Classes

OCL equivalent: context <Class> inv:
 self.<relation.to.other.class> ->exists()
 [
 AND context <Relation Class> inv:
 self.<attribute><operator><value>
]

10. **Connection Non-Existence:** This constraint type postulates the non-existence of a relationship between objects of two classes. As a result, it represents the opposite of the type Connection Existence. Moreover, in contrast to Connection Existence, constraints about the specific relationship are not possible because there can not be a relationship.

Applicable to: Classes

OCL equivalent: context <Class> inv:
self.<relation.to.other.class> ->isEmpty()

Example: Objects of class *Car* can not be connected with objects of class *Insurance personnel*.

4.5. Notation

Although the notation determines the visual appearance of a modeling language, this section is additionally dealing with an explanation how to connect multiple particular types of constraints to a discrete and single one. The visual appearance is inspired by *VOCL* and *UML* class diagrams. This implies the assumption of a rectangle partitioned in several compartments.

4.5.1. Constraint Visualization

In consequence, each compartment provides defined functionalities. Furthermore, the lanes which separate the third and forth compartment indicate an *AND* statement to connect expressive conditions with the second compartment. The content inserted in each compartment is very similar to *OCL*, although there are some distinctions compared to the original *OCL* specification. Generally spoken, the notation bases on an extension of *OCL* to make *OCL* constraints meaningfully applicable in the context of *ADOxx* and the *ADOxx* modeling platform.

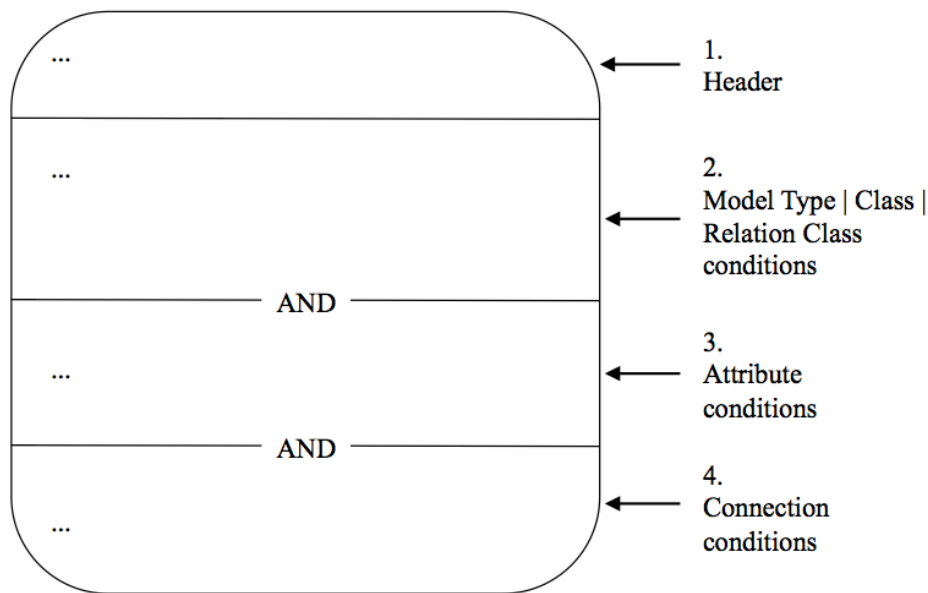


Figure 4.4.: Basic notation concept for the Constraint-Modeling-Language

4.5.1.1. Semantical Relationship between Compartments

Before we can continue with a description of the four compartments, it is essential to clarify their semantical relationship to each other first. Fig. 4.5 visualizes three different kinds of compartment dependencies: (1) *Selection Dependency*, (2) *Attribute Dependency*, and (3) *Connection Dependency*. The first one is quite straight-forward and implies a dependency between the top two compartments. It states that the selection made in the header field has to be continued in the same context in the second compartment e.g. the header defines a context for class *Car*. Consequently, the second compartment has to deal with objects of class *Car* too.

The next dependency regarding attributes states that a condition made in the second compartment influences the third compartment e.g. the second compartment defines a restriction for objects of class *Car*, e.g. there has to at least one object of class *Car* in the model. In addition, the following third compartment implies attribute restrictions for the condition built in compartment two. E.g. There has to be at least one object of class *Car* with the attribute *carFeatures* = "Xenon".

Last but not least, the third kind of dependencies regarding connections has its dependency origin in the second compartment too. It implies restrictions for the connection of class objects (defined in the second compartment) to other class objects. E.g. there has to be at least one object of class *Car* connected with one object of class *Driver* via the relationship *hasDriver*.

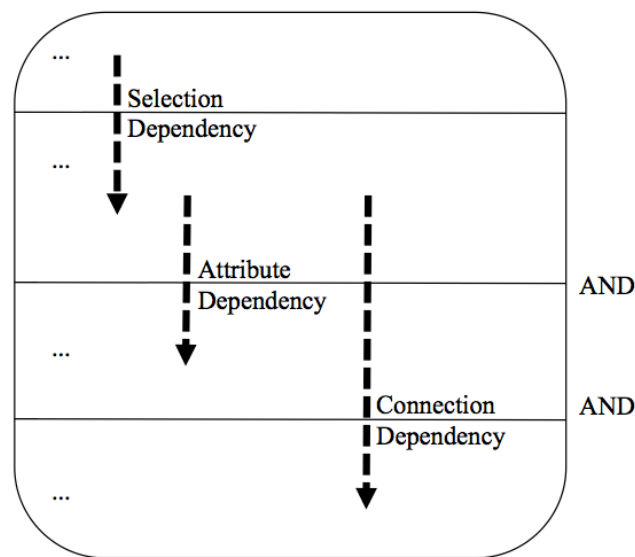
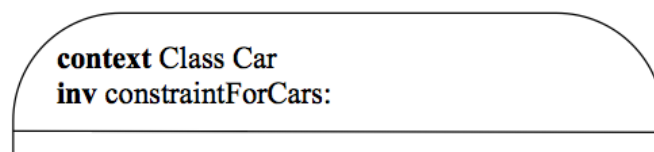


Figure 4.5.: Dependencies of compartments

4.5.1.2. Header Compartment

The Header compartment represents first and general information regarding affected modeling elements (model types, classes, or relation classes), their names, and a constraint name to foster an easy understanding and differentiation from other constraints. The first keyword in the header line is *context* followed by *Model Type*, *Class*, or *Relation Class*. For the completion of a context specification, a name of the model type, class, or relation class has to be stated. The next line in the header considers the keyword *inv* and a following constraint name. Except for the declaration of the modeling element, this is pretty much related to the *OCL* procedure of constraint definitions.

Figure 4.6.: Header for a class *Car*

4.5.1.3. Model Type, Class, and Relation Class Compartment

The second compartment specifies conditions for model types, classes, and relation classes. There are four different types of restrictions which can be entered in this section:

1. \exists **instance of** <**Model Type** | **Class** | **Relation Class**>: This type is related to the

constraint type 1. *Instance Existence* in section 4.4. It postulates that there must be at least one instance of a specific modeling element.

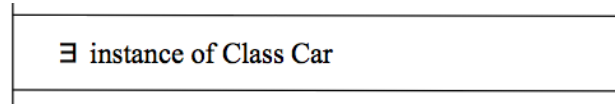


Figure 4.7.: Instance existence

2. **# instance of <Model Type | Class | Relation Class>**: This type refers to the constraint type 2. *Instance Non-Existence*. It states that there can not be at least one instance of a modeling element.

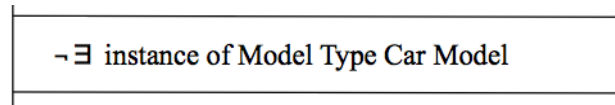


Figure 4.8.: Instance non-existence

3. **∀ instances of <Model Type | Class | Relation Class>**: This statement is not described in the chapter before as it does not represent a restriction. It embodies the standard selection and is used if no specific restriction about the existence of instances is favored.

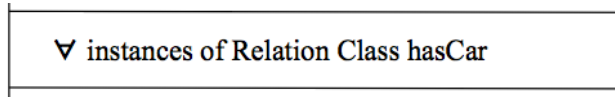


Figure 4.9.: Statement for all instances

4. **Card existence <Restriction> instances of <Model Type | Class | Relation Class>**: The last option in the second compartment is related to the constraint type 3. *Cardinality Existence / Non-Existence*. It describes quantitative existences of instances with numeric operators <, <=, >=, >, =, and <> and logic operators AND respectively OR.

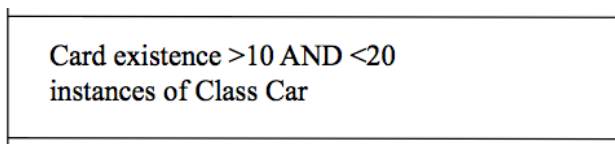


Figure 4.10.: Card existence statement

4.5.1.4. Attribute Compartment

In contrast to the other compartments, the spectrum of potential conditions and restrictions made in the attribute section is much broader. First, before dealing with concrete attribute values, we would like to visualize constraint types regarding the general existence or non-existence of attributes as described in 4. *Attribute Existence* and 5. *Attribute Non-Existence* in section 4.4. Similar to OCL, attribute conditions are implicated by the keyword *self*. Besides *self*, there are additional keywords added in the Constraint-Modeling-Language. In case of constraints for general attribute existence or non-existence, the keyword *Attribute* followed by a concrete attribute's name is added after *self*:

```
self.Attribute.numberOfWheels ->exists ()
```

Figure 4.11.: Constraint stating the existence of the attribute *numberOfWheels*

```
self.Attribute.numberOfWheels ->isEmpty ()
```

Figure 4.12.: Constraint stating the non-existence of the attribute *numberOfWheels*

For attribute values, the keyword *Value* is added after the keyword *Attribute*. There are also functionalities to add *AND* respectively *OR* logic operators according to the attribute's type. By building several restriction constructs, each construct gets a frame to enforce a sophisticated overview.

Important: As described before, a frame represents a constraint construct. Constructs can always (independent from the attribute types) be connected with logic operators *AND*, alternatively *OR*. **In case of more than two constructs linked with different logic operators, AND operators are having priority against OR operators which means that AND binds stronger than OR (e.g. Fig. 4.13)**

Please note: A constraint regarding the value of an attribute simultaneously implies the existence of the attribute. As a result, there is no additional need to state the existence of an attribute first.

For a clear visualization and presentation of the different attribute constraints, we will summarize various attribute data types into three categories (1) Numeric Category, (2) String Category, and (3) Specific Category.

1. **Numeric Category:** Covered constraint data types are *Integer*, *Double*, *Time*, *Date*, and *Datetime*. Attributes of this category can be compared with numeric operators and supplemented with logic operators *AND*, alternatively *OR*.

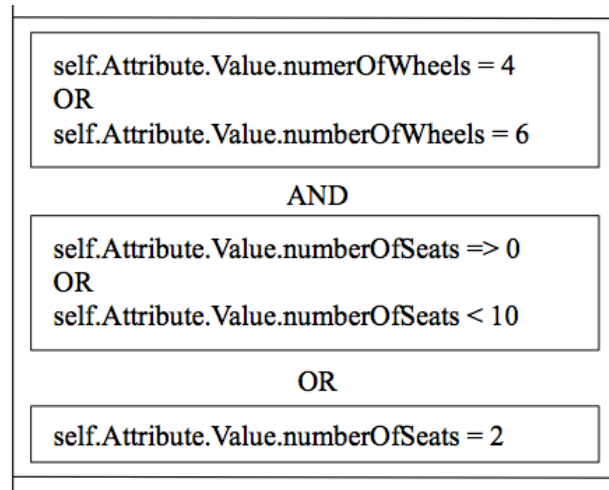


Figure 4.13.: Restriction for different numeric attributes and values

2. **String Category:** Involved data types are *String*, *Longstring*, *Enumeration* and *Enumerationlist* types. Attributes of the *String Category* can be compared with String operators and supplemented within the frame with logic operator *OR*. Nevertheless *Enumerationlist* represents a special case and supports additionally *AND* logic operators.

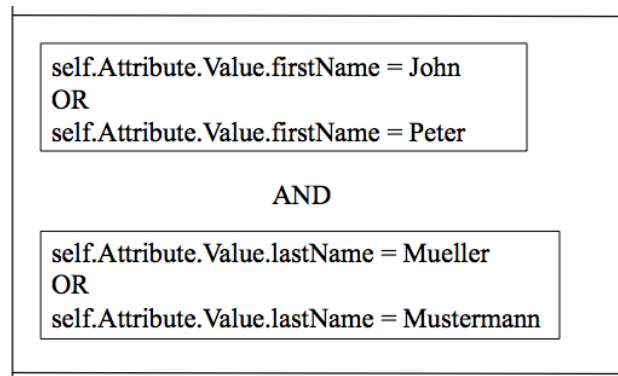


Figure 4.14.: Restriction for different String attributes and values

```

self.Attribute.Value.carType = Coupe
OR
self.Attribute.Value.carType = SUV

```

Figure 4.15.: Restriction for Enumeration attribute and values

```

self.Attribute.Value.carFeatures = Xenon
AND
self.Attribute.Value.carFeatures = Navigation System
AND
self.Attribute.Value.carFeatures = Leather Seats

```

Figure 4.16.: Restriction for Enumerationlist attribute and values

3. **Specific category:** The notation of specific data types can not be summarized that clear as it was possible for numeric and String categories. This is a result of the occurring heterogeneity requiring an isolated visualization for each data type. Special data types are: *Programmcall*, *Expression*, *Interref*, and *Record* -types.

3.1. **Programmcall type:** Constraints for attributes with type *Programmcall* are introduced by the keyword *Prcall* after the keyword *Attribute*. In contrast to the previously mentioned attribute types, restriction values for Programmcalls are not explicitly visualized. To check the concrete constraint, the *ADOxx* notebook has to be opened. This is going to be shown in chapter 6.

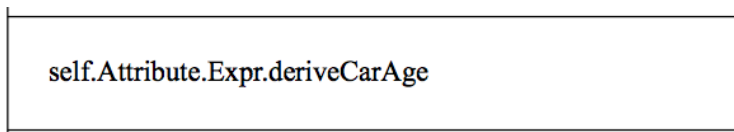
```

self.Attribute.Prcall.callExternalCarProgramm

```

Figure 4.17.: Restriction for Programmcall attribute. The concrete restriction is shown in the *ADOxx* notebook.

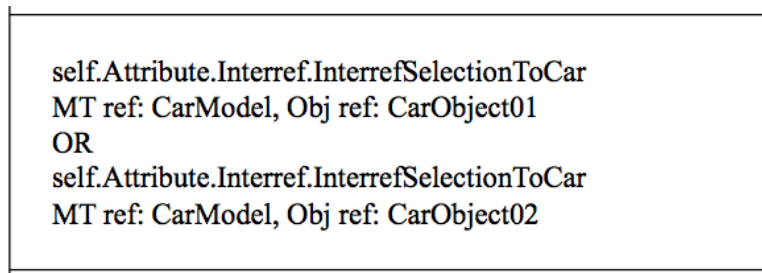
3.2. **Expression type:** The keyword for constraints about expressions is *Expr*. Apart from that, it follows the same procedure as *Programmcall*.



```
self.Attribute.Expr.deriveCarAge
```

Figure 4.18.: Restriction for Expression attribute. The concrete restriction is shown in the *ADOxx* notebook.

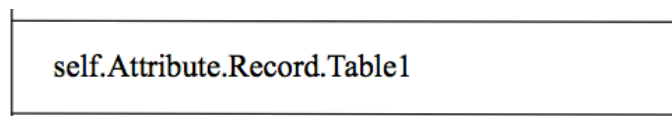
- 3.3. **Interref type:** Conditions for Interref attributes are visualized in the notation of the Constraint-Modeling-Language. The relevant references pointing at model types or concrete objects are shown in an additional row. The keyword implying constraints for *Interref* types is simply *Interref*. Furthermore, a reference to model types or model types and concrete objects is visualized.



```
self.Attribute.Interref.InterrefSelectionToCar  
MT ref: CarModel, Obj ref: CarObject01  
OR  
self.Attribute.Interref.InterrefSelectionToCar  
MT ref: CarModel, Obj ref: CarObject02
```

Figure 4.19.: Restriction for Interref attribute

- 3.4. **Record type:** As well as *Programmcall* and *Expression* types, the *Record* data type is also not concretely visualized in the notation. By accessing the *ADOxx* notebook, the various constraints for *Record* attributes can be set. The keyword is *Record*.



```
self.Attribute.Record.Table1
```

Figure 4.20.: Restriction for Record attribute

4.5.1.5. Connection Compartment

The last compartment at the bottom of the rectangle is exclusively reserved for class objects. It deals with constraints for the connection to other class objects. In addition, attributes regarding this specific relationship can be restricted too. Keywords for *Connection Existence* and *Connection Non-Existence* are *mustBeConnectedWith* and *canNotBeConnectedWith*. An optional row can state a constraint for the relationship e.g. Objects of class *Car* have to be connected with objects of class *Driver* with the relationship *hasDriver* while the

relationship *hasDriver* has a defined attribute with a particular value. In addition, it also possible to assign a name of the object which has to be connected.

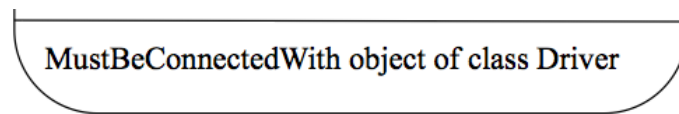


Figure 4.21.: Connection constraint

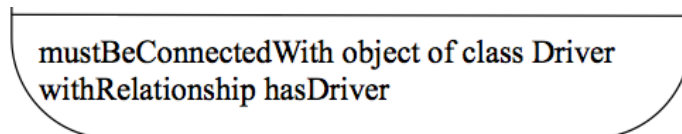


Figure 4.22.: Connection constraint specifying the relationship

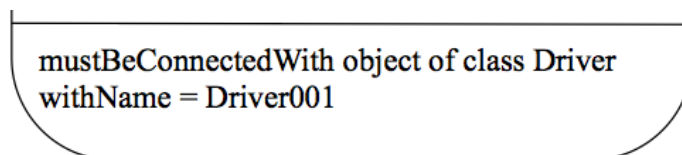


Figure 4.23.: Connection constraint specifying the name of the connected object

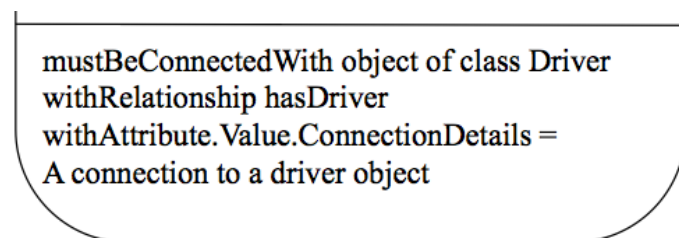


Figure 4.24.: Connection constraint specifying the relationship and a relationship attribute

4.5.2. Constraint Implications

First, it is essential to describe the semantical mechanics of constraint implications. If a specific condition implies another condition, this situation states that the first condition is not valid in general but rather in just a particular case. Section 3.2.3 has already provided a short implication example, and for the further work, we would like to visualize this constraint.

On the one hand, the LHS is given as `Driver.hasCar->size () > 0` which can be modeled as following:

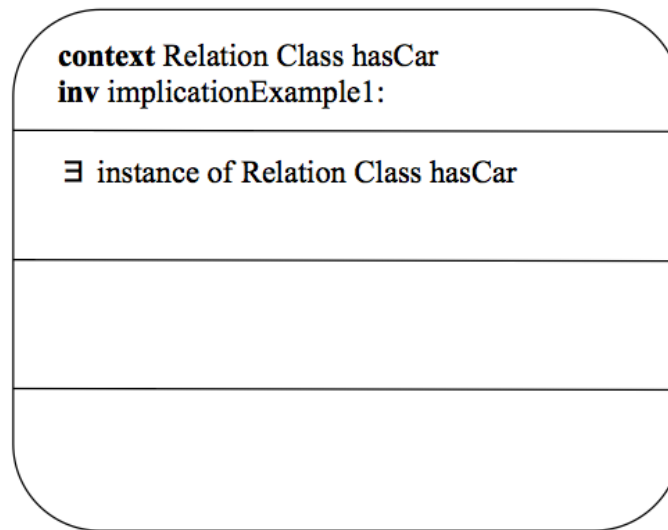


Figure 4.25.: LHS of the implication shown in Section 3.2.3

On the other hand, the RHS is: (Driver.age \geq 18) AND (Driver.drivingLicense = True):

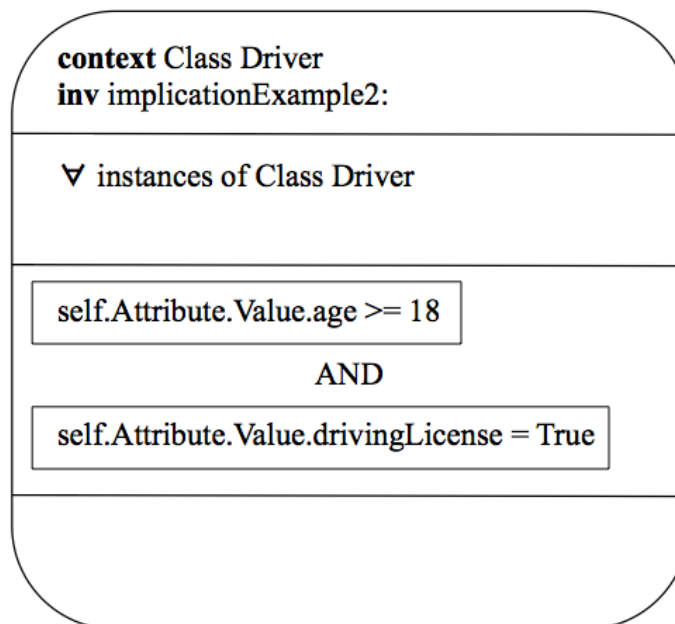


Figure 4.26.: RHS of the implication shown in Section 3.2.3

Considered as isolated and independent constraints, both of them have an general and invariant meaning. As a result, they are valid at each point of time in the modeling process. By adding an implication relationship visualized as directed vector, we can

postulate an implicational constraint which will be only triggered if the LHS was activated:

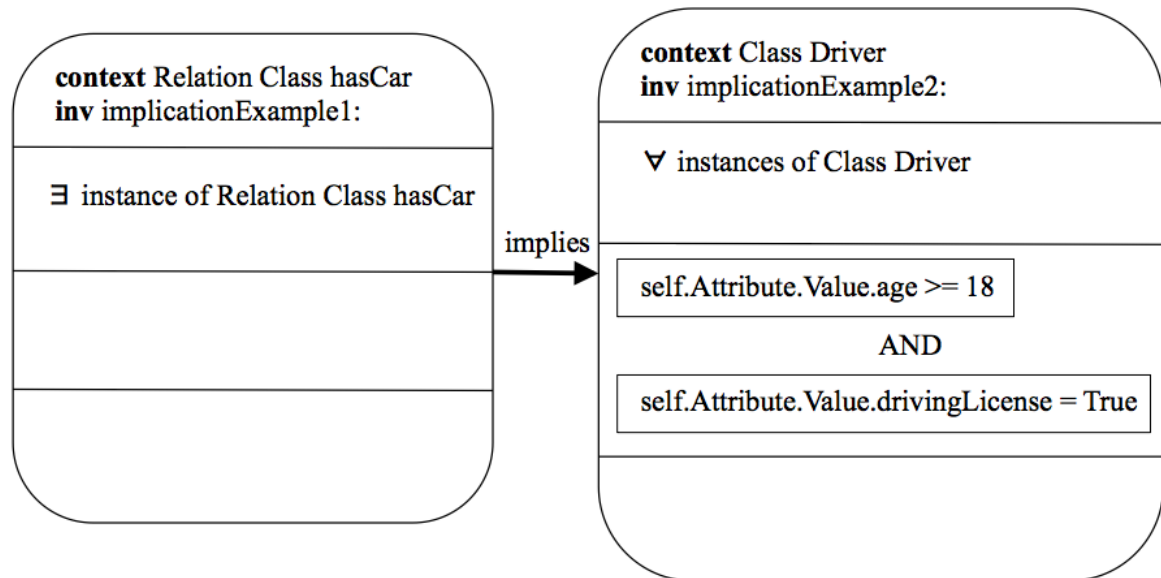


Figure 4.27.: Combining LHS and RHS to an implication

4.6. Conclusion

By recalling the outcomes of the first research question, the second one specifies various essential points of the Constraint-Modeling-Language. First, it is important to mention and structure the modeling process including the definition of a constraint model before the actual modeler creates final models.

Moreover, it could be seen as fundamental to understand how constraint meta models, constraint models, meta models and models are connected together. At this point, the thesis has outlined that a constraint object includes one or more restrictions which address entire corresponding meta models.

Pre-existing methods for constraints in *ADOxx* are described as they represent a subset of the constraint language scope. Furthermore, the definition and formalization of language constructs is necessary to specify the exact scope of the language. Formalizations are done with *OCL* statements to create a common understanding of constraint semantics.

In order to visualize the language constructs of the Constraint-Modeling-Language, the notation defines the visual appearance in reference to *UML* class diagrams and the *VOCL*

approach. The understanding of dependencies between the various compartments is explained as it embodies an essential help to be able to relate to the semantics of a constraint object.

5. Implementation of the Constraint-Modeling-Language

The modeling language is implemented in the meta modeling platform *ADOxx* v1.5¹, a product of the *BOC group*². Although *ADOxx* offers the possibility to create entirely new modeling libraries, the Constraint-Modeling-Language will be attached to a pre-existing modeling library named *SeMFIS Library 0.42 for ADOxx 1.3 and 1.5*³ [5, 7, 6]. *SeMFIS* is originally designed for engineering semantic annotations of conceptual models but nevertheless, it provides multiple modelling languages to implement various different models (e.g. BPMN models). As a result, *SeMFIS* models are going to be used as foundation enriched by constraints provided in the Constraint-Modeling-Language. The implementation documented in the further work is exemplary for an implementation approach in order to show that the implementation is realizable with methods provided in *ADOxx*. As a result, the implementation does not cover the whole spectrum of the Constraint-Modeling-Language.

5.1. Meta Model

The meta model defined in *ADOxx* is quite simplistic since the language is designed to consist of only one object class (*Constraint*) and one relation class (*implies*). First, a superclass *__ConstraintLanguage__* was defined under the top node *__D-construct__* which represents the highest superclass in the meta class hierarchy. The creation of *__ConstraintLanguage__* was done in order to generate a distinct isolation of the constraint language from constructs of the *SeMFIS* approach. Under the node of *__ConstraintLanguage__*, the class *Constraint* is placed which finally generates *Constraint* objects in a constraint model. In consequence, the meta model hierarchy is: *__D-construct__* → *__ConstraintLanguage__* → *Constraint*. The relation class *implies*, on the other hand, is settled under the folder *Relation classes*.

Although the approach of having only one distinct class seems to be quite uncommon

¹<https://www.adoxx.org/live/download-15> (accessed June 22, 2016)

²<https://at.boc-group.com> (accessed June 22, 2016)

³<http://www.omilab.org/web/semfis/> (accessed July 4, 2016)

prima facie, it offers three advantages for the concept of the constraint language:

- The validation of constraints against other models is going to be better realizable without unnecessary overhead. This overhead will certainly occur by using multiple classes and relation classes. This circumstance is a result of a more complicated validation because information from many class objects and relation class objects would have to be aggregated first. Moreover, there would be a need for the validation to consider a broad spectrum of potential constellations of class objects which makes it more difficult to extract the meaning of a modeled constraint.
- The concept of the Constraint-Modeling-Language is inspired by *UML* class diagrams which are explicitly known by a huge amount of people. This fact leads to comfortable first steps into this approach and the meaning of a constraint can be understood quite fast.
- The constraint language aims at providing a simple and straightforward method for assigning constraints to models. The readability and the understanding of a constraint visualized as a table, which has to be read top-down, follows an approach which is similar to the standard procedure of reading text in books for example.

5.1.1. Constraint Class

Because of the fact the modeling language uses only one particular class object, there are many attributes stored in the *Constraint* class. As a result, requirements regarding the creation of a clear overview about the attribute distribution to the four compartments have to be fulfilled.

5.1.1.1. General Attributes

General attributes do not belong to any compartments and are used for a necessary initial specification of a constraint. This specification describes the extend of applying a specific constraint.

- **Layer:** This attribute defines the layer on which constraints are settled. There are two options: (1) *Object Layer*, which implies that the constraint is going to restrict particular objects and (2) *Model Layer* which defines restrictions on an upper hierarchy affecting whole models.
- **Constraint Application:** This offers a selection if the constraint has to be applied on (1) *All models of model type* or on a (2) *Model with specific model name of model type*. This attribute is important to enable model specific restrictions with constraints e.g. *Model 1* has to be restricted by *Constraint 1* but *Model 2* has to be independent from *Constraint 1*.

- **Model_Type_Name:** A definition of the model type affected by constraints. The enumeration is filled with model types belonging to the *SeMFIS* library: *Company map*, *Business process model*, *Business process diagram (BPMN 2.0)*, *Service pool*, *Document pool*, *Frames Ontology Model*, *Ontology Model*, *Term Model*, *Semantic Annotation Model*, *Class / Object Diagram* and *Application Architecture (Diagram)* which will be added in the thesis to show various use cases in an IT-architecture environment.
- **Model_Name:** This attribute is activated if *Constraint_Application = Model with specific name of model type*. It defines the name of the model which is exclusively restricted.

Finally, the general attributes are structured in the *ADOxx Notebook* defined in the attribute *Attrep*. This generates an user interface where the attributes are shown and attribute values can be modified by the user.

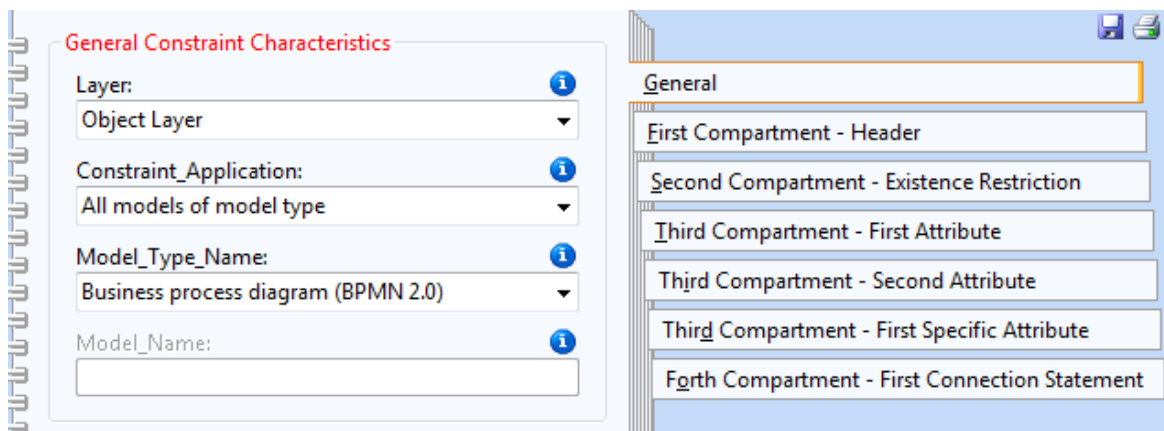


Figure 5.1.: General attributes shown in the *ADOxx Notebook*

Fig. 5.2 visualizes how these attributes address a *SeMFIS* meta model. The arrow *Assign Constraints to* which stops at the border of the *SeMFIS* meta model means that the whole meta model is affected and *Layer* is set to *Model Layer*. The other arrow stopping at a class (or relation class) implicates that the *Layer* is set to *Object Layer*.

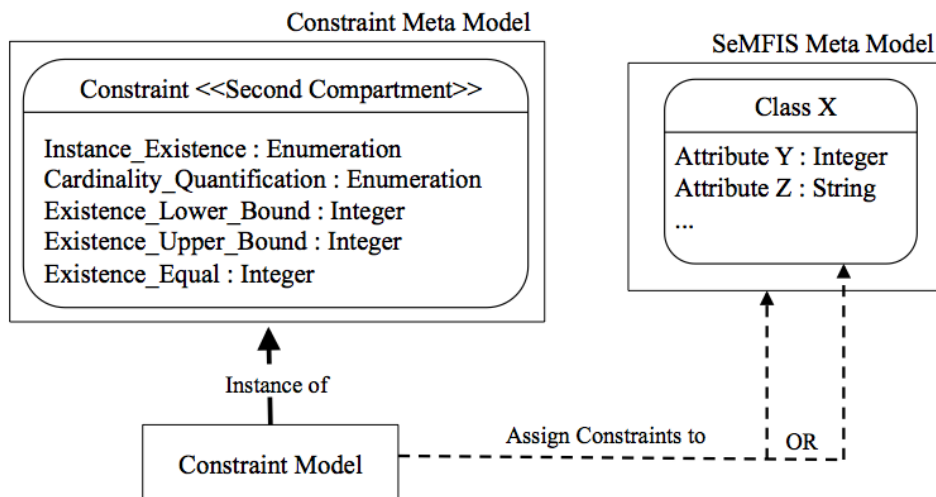


Figure 5.2.: General attributes in a constraint meta model and how they affect a *SeMFIS* meta model

5.1.1.2. First Compartment Attributes

Attributes belonging to the first compartment specify the header characteristics.

- **Constraint Name:** This attribute defines an arbitrary constraint name to distinguish the constraint from others. An expressive constraint name does also imply the meaning of the constraint.
- **Context Selection:** If the *Layer* attribute = *Object Layer*, *Context_Selection* can specify whether the constraint deals with classes or relation classes.
- **Context Name:** A specification of the previously selection made in *Context_Selection*. As a result, the name of the class or relation class is defined here.

In case of an *Object Layer* selection, the header refers to a particular class name or relation class name in a *SeMFIS* meta model:

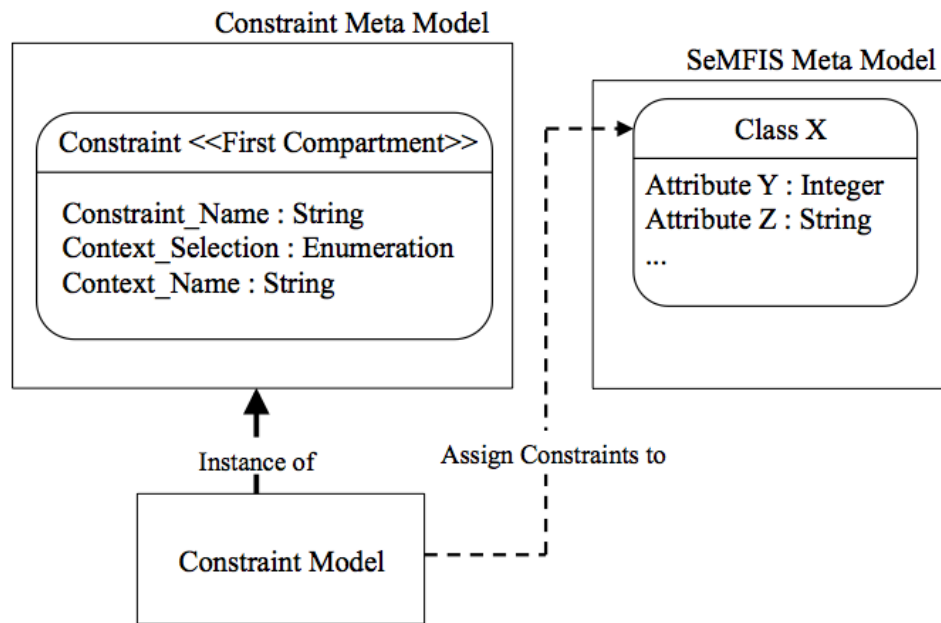


Figure 5.3.: First compartment attributes in a constraint meta model and how they affect a *SeMFIS* meta model

5.1.1.3. Second Compartment Attributes

As already described, the second compartment provides options regarding existence restrictions.

- **Instance Existence:** This is the main attribute in the notebook chapter of the second compartment as it determines the existence of instances. The enumeration attribute offers the following selection: (1) *Instance Existence*, (2) *Instance Non-Existence*, (3) *Cardinality Existence*, and (4) *No Restriction* which implies a \forall statement.
- **Cardinality Quantification:** A specification which must be made if *Instance Existence* = *Cardinality Existence*. This attribute determines lower- and upper bounds for instance existences e.g. there must be > 3 and < 9 instances of a class or it defines a particular value for instance existences e.g. there must be exactly four instances of a class.
- **Existence Lower Bound:** An Integer value which defines the lower bound for instance existences. The attribute is activated if *Cardinality Quantification* = *Lower-/Upper-Bounds for Existence*.
- **Existence Upper Bound:** Integer value determining an upper bound for instance existences. The activation of this attribute follows the same procedure as the lower bound attribute.
- **Existence Equal:** Integer value implying the specific amount of instances which have to exist. It is activated if *Cardinality Quantification* = *Existence equal Value*.

Second compartment attributes affect the instance existences of entire *SeMFIS* meta models or the instance existences of particular classes / relation classes.

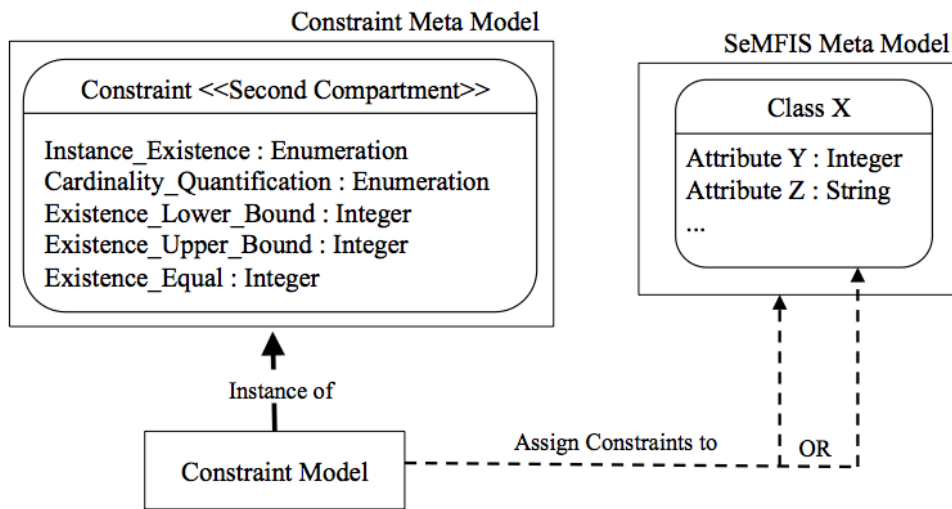


Figure 5.4.: Second compartment attributes in a constraint meta model and how they affect a *SeMFIS* meta model

5.1.1.4. Third Compartment Attributes

The description of third compartment attributes is a way more complex than the previously mentioned procedure for attributes shown in the notebook. This is a result of the fact that attributes are divided into numeric, string and specific categories. Each category requires individual attributes in order to create adequate attribute restrictions. For the visualization in the Notebook, numeric and string attributes are combined together to a *standard* attribute category. Specific attributes are treated isolated by a separated Notebook chapters.

Standard attributes: The implementation provides functionalities to restrict a maximum amount of two standard attributes. Attributes for attribute restrictions follow a particular pattern regarding attribute names: The actually attribute name comes after an integer implying the attribute chronology e.g. *Attribute_Name* requires *1_Attribute_Name* to underline that it is the name of the first restricted attribute.

- **Attribute Restriction Activate:** This attribute activates the restriction for an attribute. If it is set to *Disabled*, all other attributes in this chapter are inactive.
- **Attribute General Restriction:** With the help of this attribute, the user is able to set a restriction for an attribute existence or an attribute value.
- **Attribute Name:** The name of the attribute which should be restricted.
- **Attribute Data Type:** The data type of the chosen attribute. As we deal with stan-

dard attributes in this section, the enumeration contains the following options: INTEGER, DOUBLE, STRING, LONGSTRING, TIME, DATE, DATETIME, ENUMERATION, ENUMERATIONLIST.

- **Attribute Existence:** This attribute specifies the existence of an attribute. It offers the options (1) *must exist* or (2) *Can not exist*. The attribute in general is activated if *Attribute_General_Restriction = Restrict Attribute Existence*.

The following attributes are related to a restriction for attribute values:

- **Attribute Restriction Type:** Defines a constraint for a particular attribute value or an attribute value range.
- **Attribute Numeric Operator:** This attribute provides numeric operators for attributes belonging to data types INTEGER, DOUBLE, TIME, DATE, or DATETIME.
- **Attribute String Operator:** String operators for attributes assigned to data type STRING, LONGSTRING, ENUMERATION, or ENUMERATIONLIST.
- **Attribute Value:** The final attribute value which has to be restricted by a constraint.

The now following attributes extend the previously ones by stating restrictions for an attribute value range:

- **Attribute Value Range Logic.Operator AND_OR:** Provides an *AND* respectively an *OR* logic operator to connect two numeric conditions e.g. integer attribute *execution_time* > 50 AND < 100. In addition to numeric attributes, ENUMERATIONLIST attributes can also be restricted by this attribute.
- **Attribute Value Range Logic.Operator OR:** This attribute addresses a value range for string attributes by offering an *OR* logic operator e.g. string attribute *name* = *Mueller* OR *Mustermann*.
- **Attribute Second Numeric Operator:** Same attribute as *Attribute_Numeric_Operator* but it restricts the second attribute value of the range.
- **Attribute Second String Operator:** This attribute follows the same procedure as *Attribute_Second_Numeric_Operator*.
- **Attribute Second Value:** The second attribute value of the range.

To logically connect restrictions for multiple attributes, the attribute

Logic.Operator Between First And Second Attribute is used. It provides an *AND* respectively an *OR* operator. Third compartment attributes restrict particular attribute characteristics of a class / relation class or attribute characteristics of a meta model.

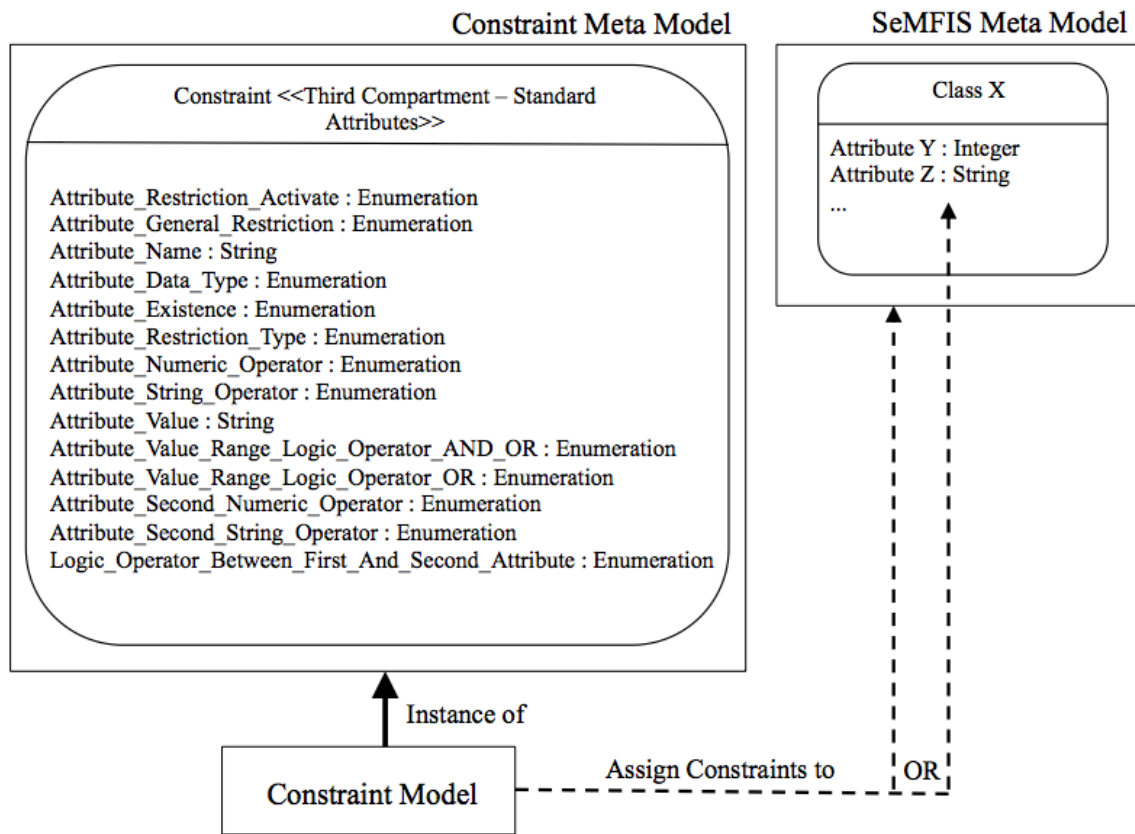


Figure 5.5.: Third compartment attributes for restricting a standard attribute in a *SeMFIS* meta model

Specific attributes: In contrast to the amount of restrict-able standard attributes, the implementation only supports the generation of a constraint for one specific attribute: INTERREF attributes. Apart from that, the attribute structure broadly follows the same principle as for standard attributes.

- **Specific Attribute Restriction Activate:** Attribute for the activation of an attribute constraint (same principle as for standard attributes).
- **Logic Operator To First Attribute:** This attribute logically connects a first standard attribute to a first specific attribute. The attribute is activated if the second standard attribute is disabled.
- **Logic Operator To Second Attribute:** This attribute logically connects a second standard attribute to a first specific attribute. As described before, the logical connection between the first and second standard attribute is already processed with **Logic Operator Between First And Second Attribute**.
- **Specific Attribute General Restriction:** Same principle as for standard attributes.
- **Specific Attribute Name:** Same principle as for standard attributes.
- **Specific Attribute Data Type:** Data types for specific attributes are: PROGRAMM-CALL, EXPRESSION, INTERREF, RECORD.

- **Specific Attribute Existence:** Same principle as for standard attributes.
- **Specific Attribute Restriction Type:** Same principle as for standard attributes.

The next group in the Notebook deals with INTERREF specific attributes for restriction:

- **Specific Attribute Interref Restriction:** A definition of the restrict-able Refdomain explained in chapter 4.4 section 8.13. The enumeration offers the options (1) *Restrict reference to model type* or (2) *Restrict reference to object*.
- **Specific Attribute Interref Operator:** A string operator reserved for interref attributes.
- **Specific Attribute Interref Model Type Reference:** The model type which has to be inserted in a restricted interref attribute.
- **Specific Attribute Interref Object Reference:** The particular object which has to be part of the Refdomain. This attribute is activated if *Specific Attribute Interref Restriction* = *Restrict reference to object*.
- **Specific Attribute Value Range Interref Logic Operator OR:** An OR operator which is activated if *Specific Attribute Restriction Type* = *Restrict Attribute Value Range*.
- **Specific Attribute Second Interref Operator:** A string operator reserved for a second value of a value range regarding interref attributes.
- **Specific Attribute Second Interref Model Type Reference:** An alternative model type which has to be inserted in a restricted interref attribute.
- **Specific Attribute Second Interref Object Reference:** An alternative particular object which has to be part of the Refdomain.

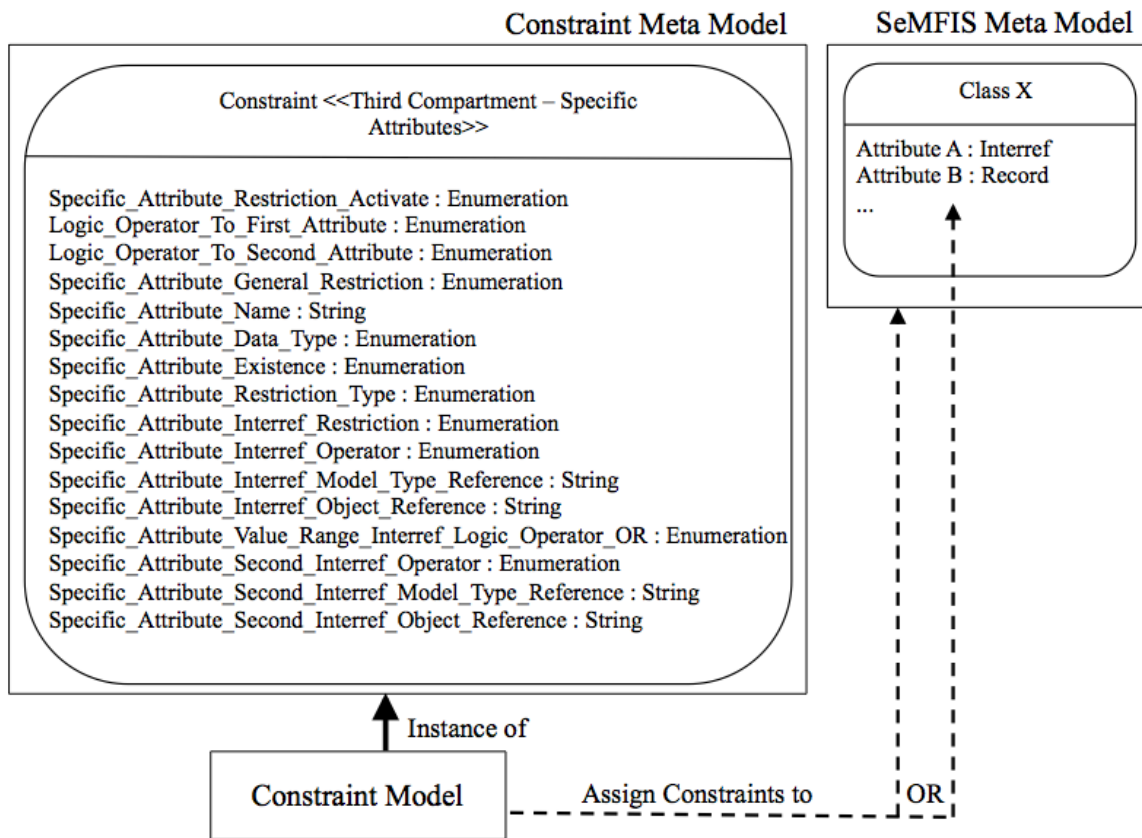


Figure 5.6.: Third compartment attributes for restricting a specific attribute in a *SeMFIS* meta model

5.1.1.5. Forth Compartment Attributes

Finally, the last compartment sets restrictions for relationships of class objects. In addition, the implementation includes the opportunity to assign a constraint to one standard attribute belonging to a specified relationship. Because of the fact that the constraint language provides support for multiple relationship constraints, the nomenclature is related to the third compartment. That means that attributes for restriction begin with an integer value implying the chronology. In the further work, a constraint regarding the relationship of class objects will be named *connection statement*.

- **Connection Statement Activate:** This attribute initially activates a constraint for a relationship.
- **Connection Statement Kind Of Restriction:** This attribute offers the enumeration (1) *Must be connected with* and (2) *Can not be connected with*.
- **Connection Statement To Object Of Class:** A definition of the class name to which a relationship has to exist or can not exist.
- **Connection Statement With Object Name:** A definition of the object name to which

a relationship has to exist or can not exist. This attribute represents a more precise specification and bases on the value entered in the previously described attribute. If the value of this attribute is empty, there will be no restriction regarding the object name.

- **Connection_Statement_With_Relationship:** A specification of the relationship name which must be used to connect two class objects. If the value of the attribute is empty, all kinds of relationships are valid.
- **Connection_Statement_Relationship_Attribute_Restriction_Activate:** This attribute enables a restriction for an attribute of the previously defined relationship.

The following attributes refer to an attribute restriction of defined relationships which are specified before. At this point, the implementation is stripped-down in context of the spectrum how attribute value restrictions can be described. As a result, the implementation only provides the functionality to assign constraints to attribute values (and not attribute existences or value ranges).

- **Relationship_Attribute_Name:** The name of the attribute which appears in the relationship.
- **Relationship_Attribute_Data_Type:** In this case, the implementation supports only standard attribute data types.
- **Relationship_Attribute_Numeric_Operator:** A numeric operator which is activated if the data type is numeric.
- **Relationship_Attribute_String_Operator:** A string operator which is activated if the data type is string.
- **Relationship_Attribute_Value:** The final value of the attribute restricted by a constraint.

As Fig. 5.7 visualizes, forth compartment attributes address exclusively classes in a *SeMFIS* meta model and optionally corresponding standard class attributes in relationship classes.

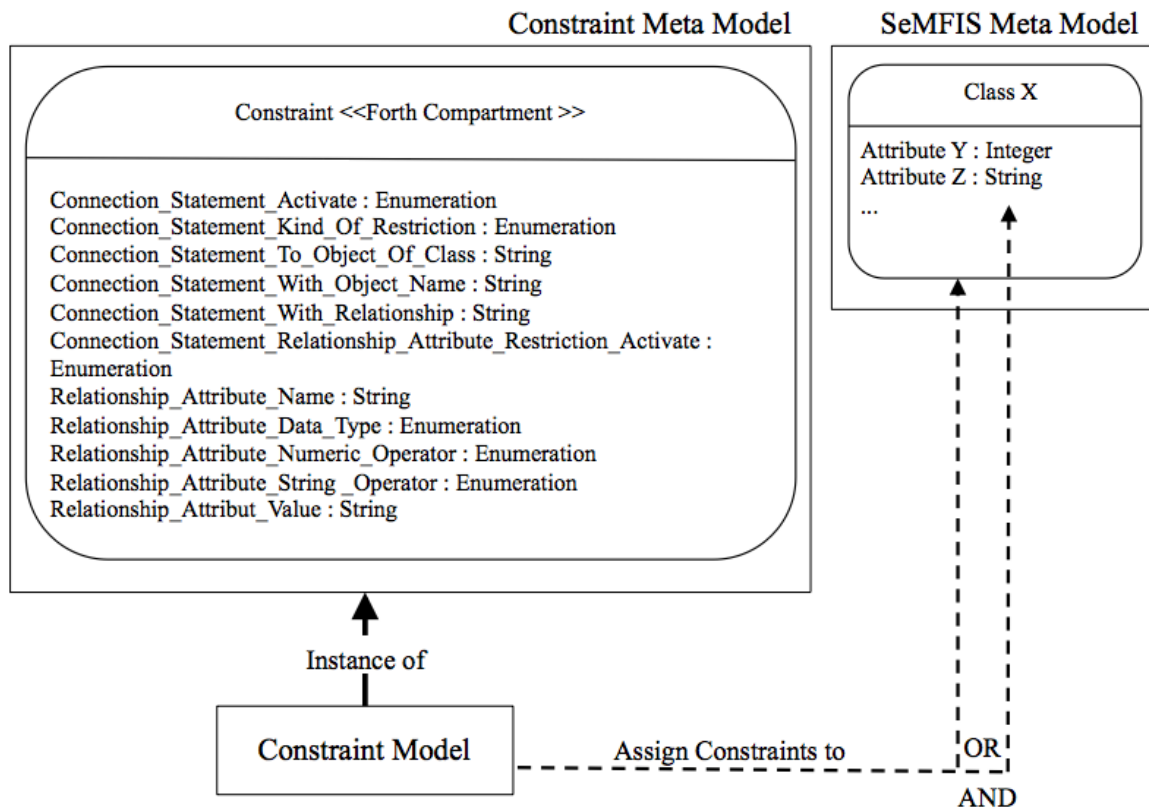


Figure 5.7.: Forth compartment attributes for restricting a connection statement and a relationship attribute in a *SeMFIS* meta model

5.1.1.6. Attrep

The *ADOxx* Notebook is defined in an attribute named *Attrep* in the constraint class. The complete *Attrep* code can be found on a CD attached to the thesis.

5.1.1.7. Graphrep

To define the notation of constraint objects, *ADOxx* offers the attribute *Graphrep* in the constraint class. In analogy to *Attrep*, the *Graphrep* code is located on the attached CD too.

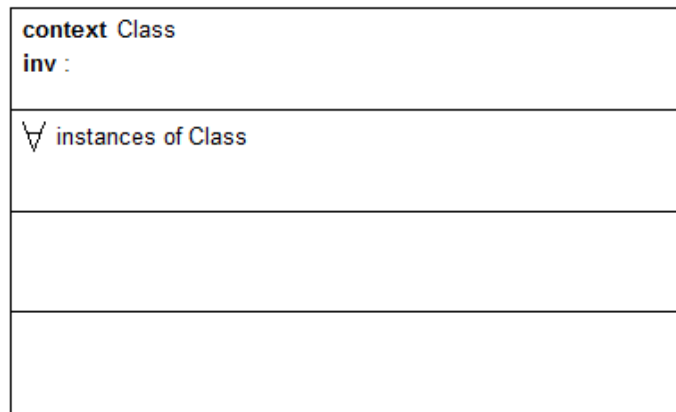
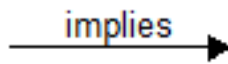


Figure 5.8.: Standard notation by initially creating a constraint object

5.1.2. Implies Relation Class

There is only one relation class in the meta model: *implies*. In contrast to constraint objects, the relation class has a very simple structure. Due to the fact that *implies* does not support any relevant attributes which can be set in the Notebook, there is no *Attrep* defined in the meta model. The notation determined in *Graphrep* is straight forward as it is simply visualized as black arrow.

Figure 5.9.: Notation of relation class *implies*

5.2. Validation

As it was shown in the previous sections and chapters, the Constraint-Modeling-Language is designed to connect at least two different models: On the one hand, a distinct productive model which is used to visualize an IT-architecture or a business process for example, and on the other hand, a constraint model which creates place for defining constraints in reference with the productive model. In consequence, a validation functionality is needed to merge these models to ensure that the productive model semantically corresponds to constraints defined in a constraint model.

5.2.1. Validation Scope

Because of the fact that an entire implementation of a validation functionality would be disproportionate work in programming for this thesis, it offers few implemented segments of a holistic validation method. The implemented validation covers the following parts:

- **Model Layer:**
 - Second Compartment (isolated):
 - * Instance Existence: A check if at least one model of a defined model type does exist.
 - * Instance Non-Existence: Validation if there are no models of a defined model type.
 - * Cardinality Existence: A verification if the number of model existences is in a given spectrum or is equal a particular value.
- **Object Layer:**
 - Second Compartment (isolated):
 - * Instance Existence: A functionality which verifies if at least one instance of a class exists in a specific model or in all models of a model type. If all models of a model type are selected, the validation will stop if one instance was found in an arbitrary model.
 - * Instance Non-Existence: A validation which checks if there is no instance of a class in a particular model or in all models of a model type.
 - * Cardinality Existence: This verifies if the amount of instances corresponds to a defined range or to a single value. Again, the validation is possible for all models of a model type or a specific model.
 - Second Compartment + Third Compartment:
 - * Instance Existence + Value Restriction for a Standard Attribute: A check if there is an instance of a class having a specific attribute value in all models or a particular model.
 - * For all Instances + Existence of a Standard Attribute: Verification which checks if all instances of a class are having a specific attribute.
 - * For all Instances + Value Restriction for a Standard Attribute: A check if all instances of a class are having a particular attribute value.

5.2.2. Validation Implementation

The validation is implemented in *AdoScript*⁴, a scripting language provided in *ADOxx*. The code for the functionality could be found in an external file named *validation.asc*, which will be loaded and executed in a constraint model if the modeller clicks on *Validation*

⁴<https://www.adoxx.org/live/adoscript-language-constructs/> (accessed June 29, 2016)

→ *Execute Validation*. Generally, the validation script bases on constraint models and concrete constraints which are modeled in these models. The script is not capable of validating more than one constraint model at the same time. In contrast to the meta model / model visualizations in section 5.1, the validation operates exclusively on model layer:

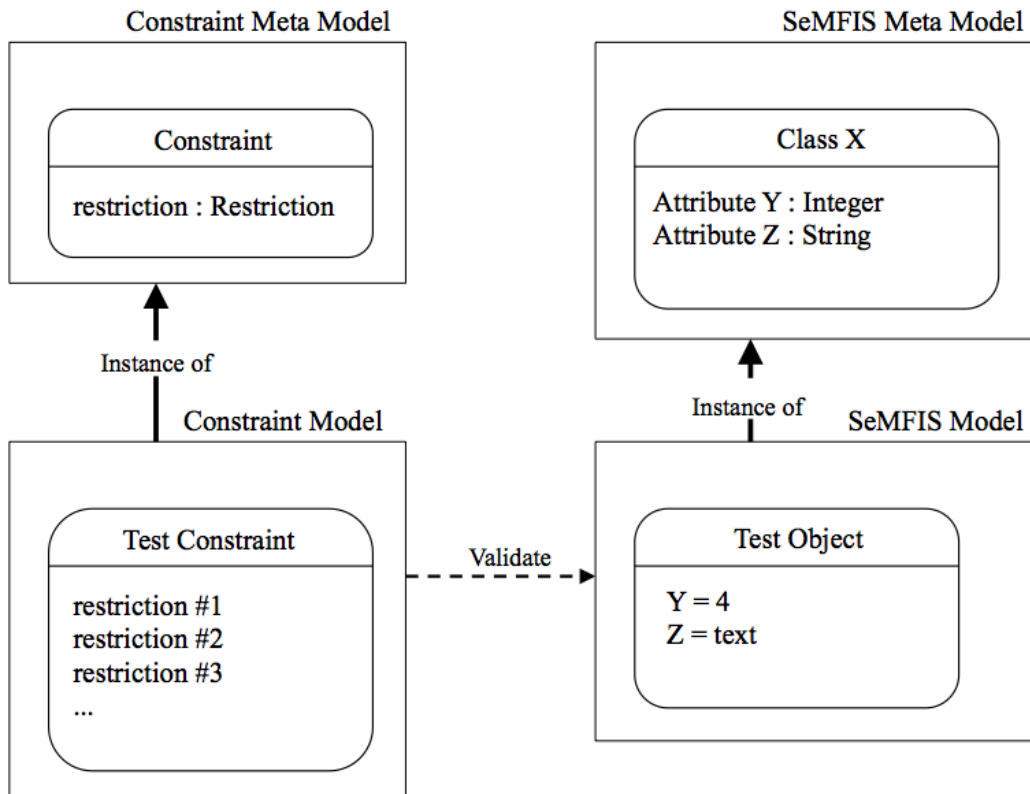


Figure 5.10.: The validation accesses restrictions made in constraint model and validates restrictions against a *SeMFIS* model

In further consequence, this means that the validation checks particular constraints which are settled on meta model layer but can only be evaluated on instances of the corresponding meta model, which refers to the model layer.

The *AdoScript* validation can be simplified modeled as an iterative process in a *SeMFIS* activity diagram:

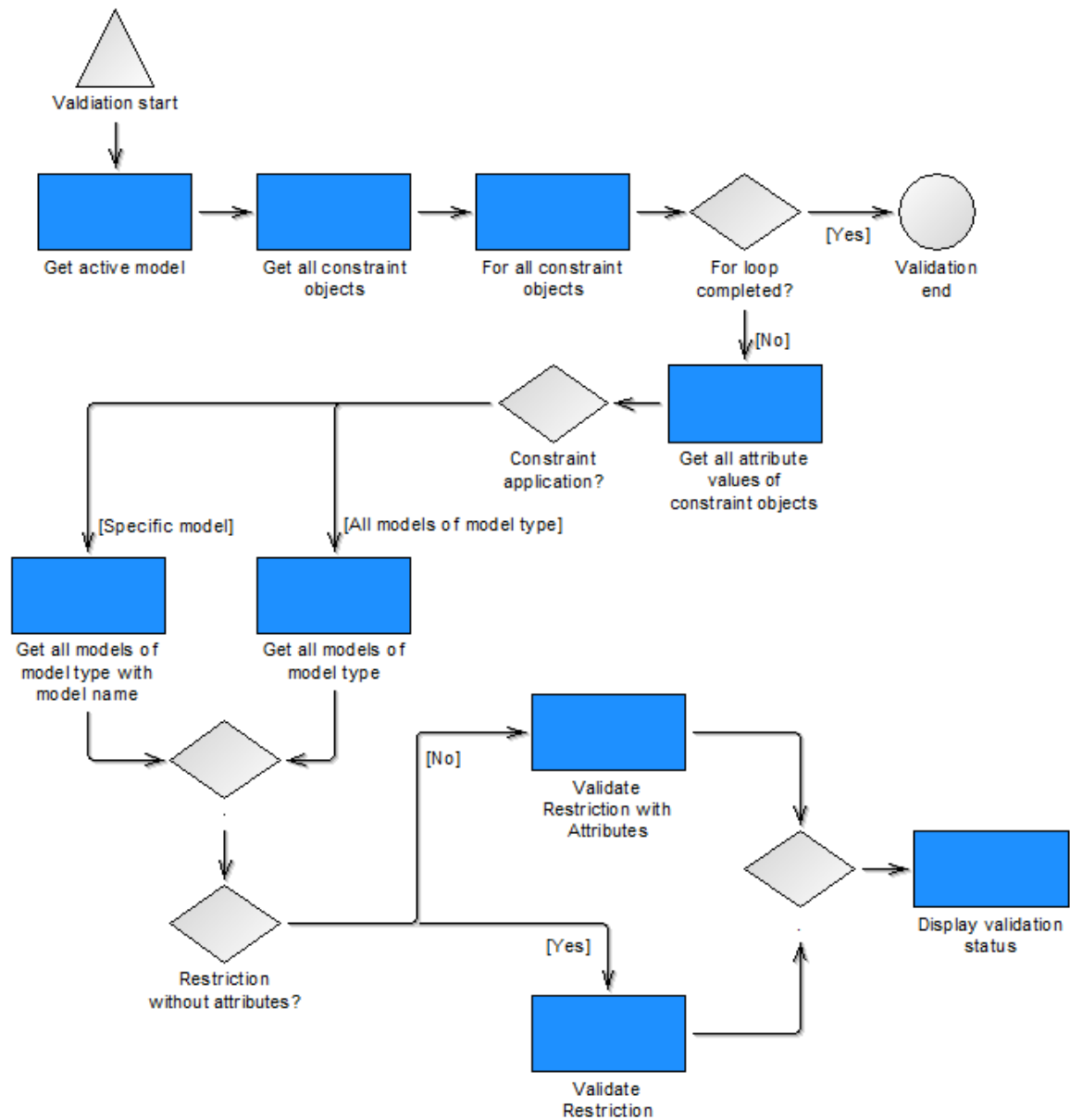


Figure 5.11.: Process of validation

The process is structured as follows:

1. The script collects all constraint objects in a currently opened constraint model.
2. For each constraint object, the script extracts attribute values which are set by default or modified by the modeller. The attributes which get read by the script are exactly the attributes which were characterized in section 5.1.1.

3. In a next step, the script checks the value of attribute *Constraint.Application*. Depending on the value, the script executes a specific *AQL*⁵ query to get the object IDs of all models of a particular model type or only a specific model of a model type. *AQL* was chosen in this context, because the evaluation with the query is a quite more comfortable than with the *AdoScript* equivalents.
4. The further sections in the code deal with various scenarios and constellations of second and third compartment attributes. The validation of these scenarios takes place in multiple nested IF clauses.
5. In case of *Instance Existence* was chosen in the second compartment, the validation stops if a condition was found that meets the constraint requirements (instead of iterating over the whole number of referenced models and objects). This is because *Instance Existence* implies that there is at least one object fulfilling a constraint.
6. Successful validations are displayed as *viewboxes* with a short explanation of the validation.
7. Non-successful validations are shown as *warningboxes* with a short description too.

5.3. Preparation for Use Case Scenarios

Before the thesis finally shows the outcomes of the implementation and the practical application of the Constraint-Modeling-Language, we add a *Application Architecture* meta model to the *SeMFIS* library. As a result, this enables the modeling of some use cases settled in an IT-architecture environment with *ADOit*⁶ inspired models. The spectrum of added classes, relation classes as well as corresponding attributes is intensively reduced as the model is used to show the fundamental functionality of the constraint language.

In order to create a sample model for an application architecture, we have to define the meta model first. The meta model consists of the classes *Application Component* and *Interface*. Relation classes are implemented as *Used Interfaces*, *Provided Interfaces*, and *Replaced Application Components*. To describe the meta model, *Application Component* and *Interface* can be expressed as class diagram:

⁵<https://www.adoxx.org/live/adoxx-query-language-aql/> (accessed June 28, 2016)

⁶<https://de.boc-group.com/adoit/> (accessed June 22, 2016)

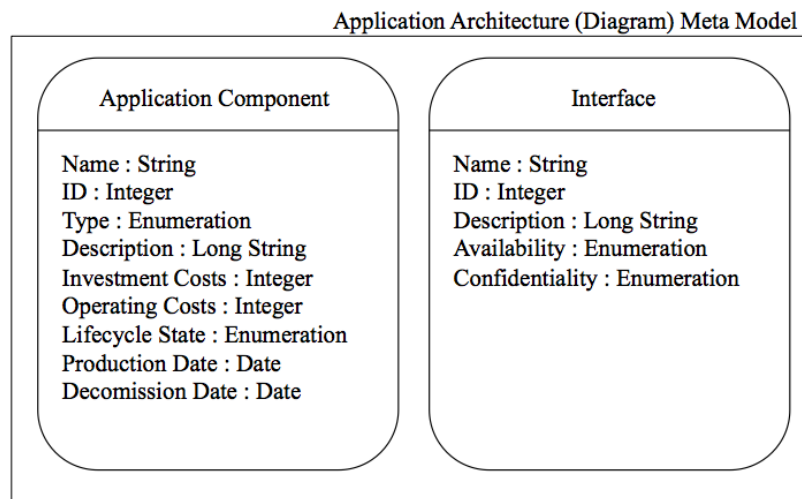


Figure 5.12.: Meta model for the classes *Application Component* and *Interfaces*

Relation classes, on the other hand, do not contain any attributes which are relevant for the use cases:

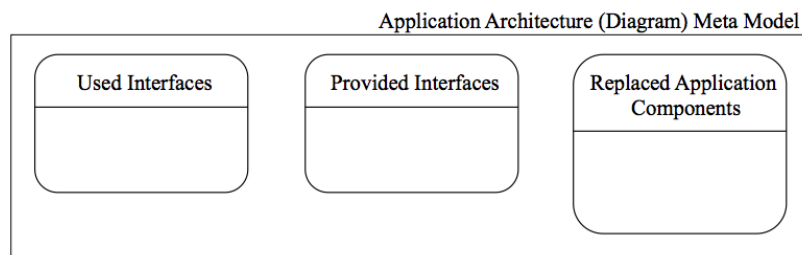


Figure 5.13.: Meta model for relation classes *Used Interfaces*, *Provided Interfaces*, and *Replaced Application Components*

After defining the notation for classes and relation classes as well as aggregating them to the model type *Application Architecture (Diagram)*, we can generate an application architecture model named *Use Case Application Architecture Model*. This model is inspired by a sample model included in the *ADOit* platform.

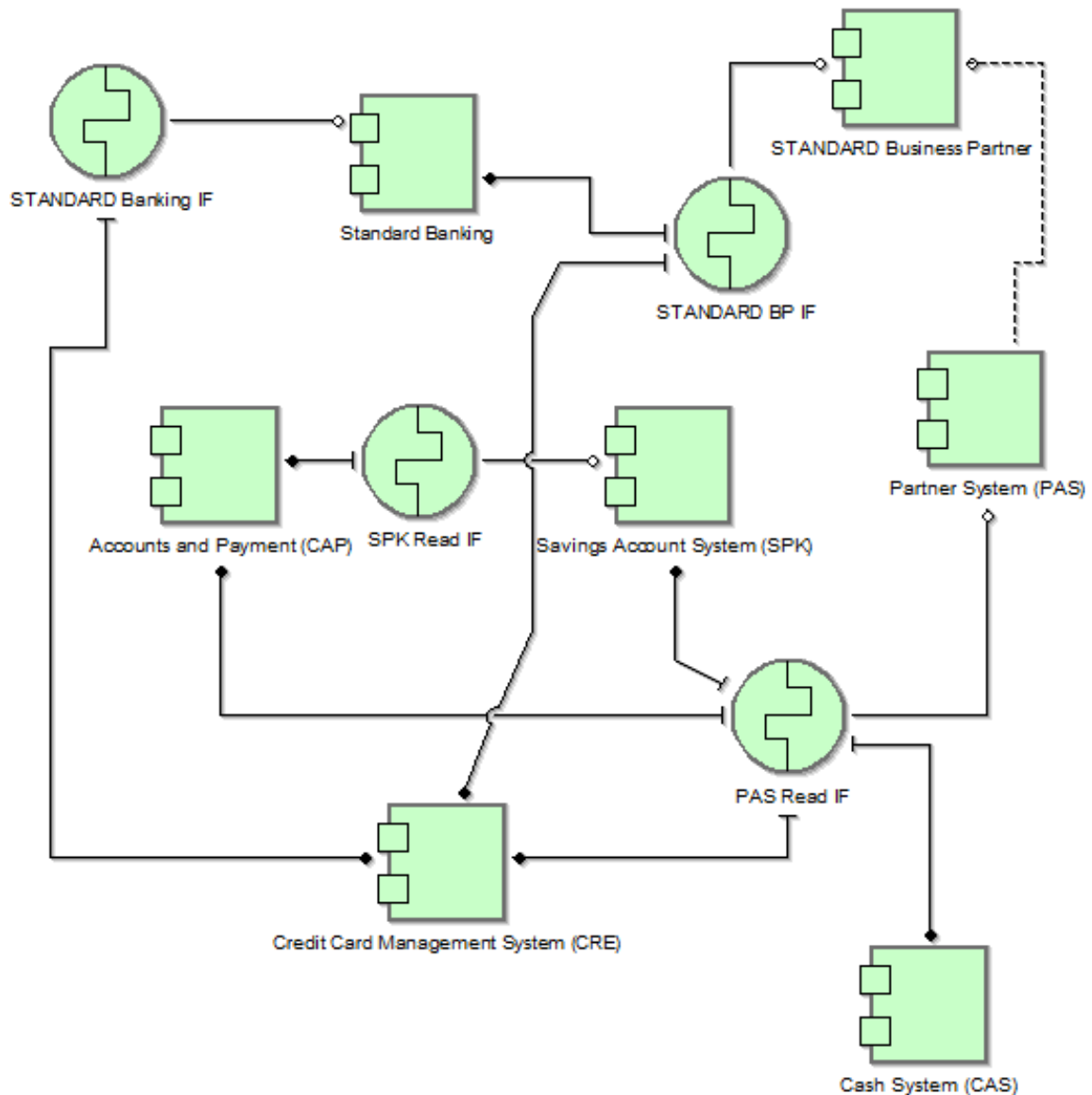


Figure 5.14.: *Use Case Application Architecture Model* modeled within the *SeMFIS* library

5.4. Constraint-Modeling-Language Application - Use Case Scenarios

In a first attempt, the thesis deals with two scenarios affecting the model layer (in contrast to the object layer). First, we would like to state, that there has to exist at least one up to a maximum of three models of model type *Application Architecture (Diagram)*:

context Model Type Application Architecture (Diagram) inv Cardinality existence for Application Architecture models:
Card existence ≥ 1 AND ≤ 3 instances of Application Architecture (Diagram)

Figure 5.15.: Cardinality existence constraint for models of model type *Application Architecture (Diagram)*

The validation is successful, since we have exclusively created the model *Use Case Application Architecture Model*. As a result, the constraint stating a cardinality existence of ≥ 1 and ≤ 3 is fulfilled because there is exactly one model of this model type. In consequence, the script displays a successful validation message:

```
Validation for Cardinality existence for Application
Architecture models successful!
There are  $\geq 1$  and  $\leq 3$  models of type Application Architecture
(Diagram) .
```

In a reverse case, a constraint stating the non-existence of instances of model type *Application Architecture (Diagram)* would not validate successfully:

context Model Type Application Architecture (Diagram) inv Application Architecture model can not exist:
$\neg \exists$ instance of Application Architecture (Diagram)

Figure 5.16.: Non-existence constraint for models of model type *Application Architecture (Diagram)*

The validation is coherently not successful, since there exists a model of model type *Application Architecture (Diagram)*:

```
Validation for Cardinality existence for Application
Architecture models not successful!
```

In further steps, we would like to deal with constraints on object layer. An initial constraint could be for example: There must exist at least four objects of class *Application Component* up to a maximum of twenty. Furthermore, there has to be at least one object of class *Interface*.

context Class Application Component inv Cardinality existence for Application Components:	context Class Interface inv Interface must exist:
Card existence ≥ 4 AND ≤ 20 instances of Class Application Component	\exists instance of Class Interface

Figure 5.17.: Existence constraints for classes *Application Component* and *Interface*

By validating the two constraints against the *Use Case Application Architecture Model*, both constraints can be validated with success. On the one hand, the constraint *Cardinality existence for Application Components* is fulfilled because there are seven *Application Component* objects in the model. As the lower bound is four and the upper bound twenty, the validation of this constraint is positive. On the other hand, constraint *Interface must exist* is fulfilled too, since there are four *Interface* objects in the relevant model. Consequently, this validation is also successful.

```
Validation for Cardinality existence for Application
Components in Use Case Application Architecture Model
successful!
There are  $\geq 4$  and  $\leq 20$  objects of class Application
Component.
```

```
Validation for Interface must exist successful!
There is at least one object of class Interface in a
model of model type Application Architecture (Diagram).
```

To assign and validate constraints for attributes, we have to add attribute values for some sample objects in the *Use Case Application Architecture Model* first. Paradigmatically, we

have chosen two objects which are going to be validated: *Cash System (CAS)* of class *Application Component* and *PAS Read IF* of class *Interface*. In succession, we attach meaningful attribute values to both objects.

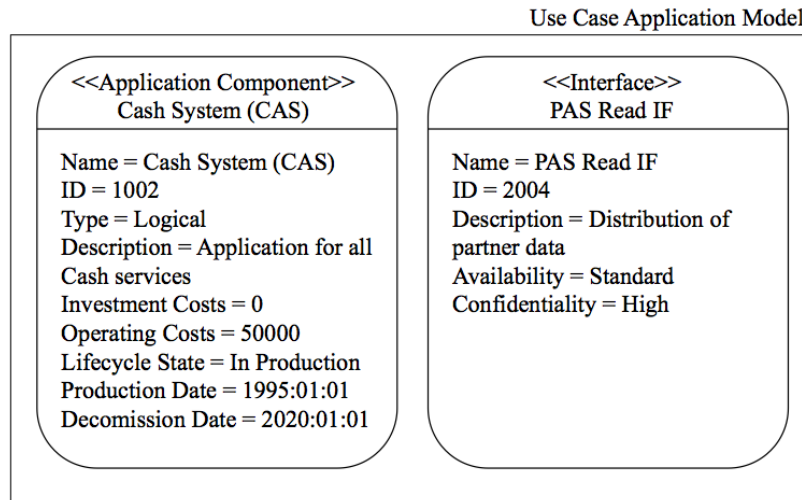


Figure 5.18.: Objects *Cash System (CAS)* and *PAS Read IF* in the *Use Case Application Architecture Model*

An initial basic constraint might be: There must be an object with name *Cash System (CAS)* in the *Use Case Application Architecture Model*:

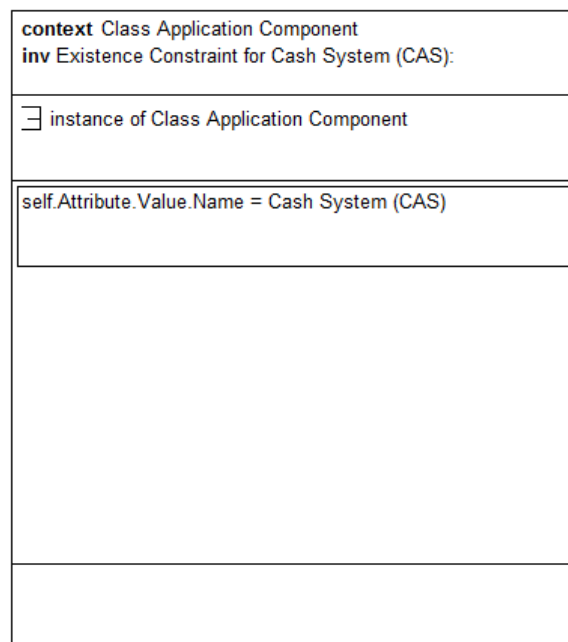


Figure 5.19.: Existence constraints for object *Cash System (CAS)*

The validation delivers a positive answer since there is an object in the *Use Case Application Architecture Model* having an attribute *Name* with value *Cash System (CAS)*.

```
Validation for Existence Constraint for Cash System
(CAS) successful!
There is at least one object of class Application
Component with an attribute Name = Cash System (CAS)
of type STRING found in a model Use Case Application
Architecture Model.
```

As it is shown in script notification above, the validation process for attributes evaluates additionally attribute data types to increase the validation scope.

To successfully evaluate further attributes for objects named *Cash System (CAS)* and *PAS Read IF*, we have to shorten the extent of the original *Use Case Application Architecture Model*. This is necessary because the validation script only includes the functionality to exclusively validate a value for one attribute and the name of an attribute already fills this role. As a result, we can not evaluate a constraint exemplary shown in Fig. 5.20 addressing two attribute values to restrict the *ID* value for object *Cash System (CAS)*.

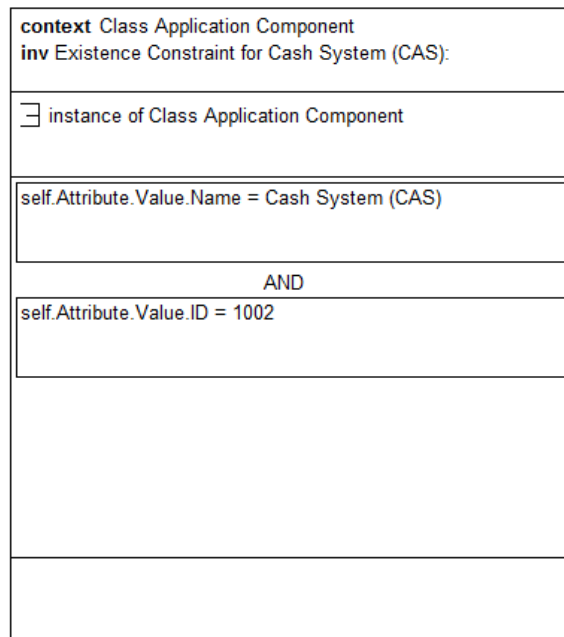


Figure 5.20.: Existence constraints for object *Cash System (CAS)* addressing two attribute values

In order to demonstrate that a validation for attributes of objects *Cash System (CAS)* and *PAS Read IF* is possible, we modify the *Use Case Application Architecture Model* in a way,

that it only consists of the two objects. In consequence, it is clear that the script addresses only the two given specific objects. We model a new application architecture model named *Shortened Use Case Application Architecture Diagram*:

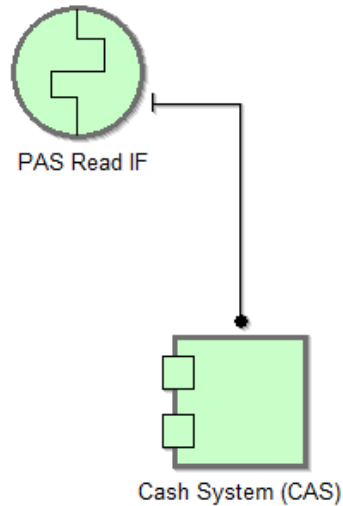


Figure 5.21.: *Shortened Use Case Application Architecture Model* modeled within the *SeM-FIS* library

First, we would like to verify two initial constraints for the objects *Cash System (CAS)* and *PAS Read IF*: (1) For the cash system, the INTEGER attribute *Operating Costs* must be lower than 100.000. (2) The attribute value of ENUMERATION attribute *Availability* findable in object *PAS Read IF* can not be *No Entry*:

context Class Application Component inv Attribute Constraint for Cash System (CAS):	context Class Interface inv Attribute Constraint for PAS Read IF:
\forall instances of Class Application Component	\forall instances of Class Interface
self.Attribute.Value.Operating Costs < 100000	self.Attribute.Value.Availability != No Entry

Figure 5.22.: First attribute constraints for objects *Cash System (CAS)* and *PAS Read IF*

By validating these two constraints against attribute values shown in Fig. 5.18, both validations deliver a positive notification:

```
Validation for Attribute Constraint for Cash System
(CAS) successful!
```

```
Object 59227 appearing in Shortened Use Case
Application Architecture Model has an attribute
Operating Costs < 100000 of data type INTEGER.
```

```
Validation for Attribute Constraint for PAS Read IF
successful!
```

```
Object 59000 appearing in Shortened Use Case
Application Architecture Model has an attribute
Availability != No Entry of data type ENUMERATION.
```

In a next step, we would like to validate the following constraints: (1) The attribute *Decommission Date* of data type INTEGER must exist and (2) the attribute *Description* of object *PAS Read IF* can not be empty:

context Class Application Component inv Attribute Constraint for Cash System (CAS):	context Class Interface inv Attribute Constraint for PAS Read IF:
\forall instances of Class Application Component	\forall instances of Class Interface
<div>self.Attribute.Decommission Date ->exists()</div>	<div>self.Attribute.Value.Description != ""</div>

Figure 5.23.: Second attribute constraints for objects *Cash System (CAS)* and *PAS Read IF*

In a last step, the validation displays the following positive results, since the constraints are both fulfilled:

```
Validation for Attribute Constraint for Cash System
(CAS) successful!
```

```
All objects of class Application Component have
an attribute named Decommission Date of
data type DATE.
```

```
Validation for Attribute Constraint for PAS Read IF
successful!
```

```
Object 59000 appearing in Shortened Use Case
Application Architecture Model has an attribute
Description != "" of data type LONGSTRING.
```

5.5. Conclusion

The third research question combines the theory aggregated in the first research question and methods of the second research question to create a implementation prototype of the Constraint-Modeling-Language.

The most complex work in this chapter was of unanticipated theoretical nature as it is represented by the definition and conceptualization of the meta model. Although the meta model consists of only one class and one relation class, the attribute hierarchy of a constraint object is many-layered and it was not that easy to specify it in a way that a visualization in *Graphrep* and a validation in *AdoScript* can follow it.

In the beginning, the implementation of the validation itself was quite complex, due to the high amount of attributes which trigger distinct validation scenarios. Especially in this case, it was essential to get a clear overview on the multiple scenarios and attribute constellations. In addition, it has to be mentioned that due to the simple concept of having only one class for objects, the implementation of the validation was obviously easier than it would have been by having various classes and relation classes.

Recapitulatory, the result of the implementation approach is a running prototype which shows that the Constraint-Modeling-Language can be implemented in *ADOxx* and offers interactive functionalities to verify and instantiate concepts of the second research question.

6. Discussion

The last chapter of the master thesis deals with two questions: (1) *How far is the goal of the thesis described in the introduction reached?* and (2) *What is the primary benefit of the Constraint-Modeling-Language in comparison to the ADOxx query language AQL?*

Regarding the first question and the aim of enhancing the semantical expressiveness of arbitrary modeling languages, we can assert that the Constraint-Modeling-Language shown in this thesis fulfills this goal. By restricting existence characteristics, attribute values or relationship constellations, the language is able to improve pre-existing modeling languages and their expressiveness. This happens, for example, by excluding particular values for attributes which make no sense in the context a model is settled. In addition, it should be also mentioned that the language does not include the entire spectrum of imaginable constraints as there are quite countless possibilities of restricting model characteristics. But in fact, the language offers a tradeoff of feasible constraints which appear to be useful in common modeling processes.

The answer to the second question is quite more complex: As already mentioned during the implementation of the validation functionality, ADOxx provides a query language named AQL. With the help of this language information about particular models, model content and dependencies of models can be extracted. Consequently, the question which appears now in this context is: How far does the Constraint-Modeling-Language offer additional benefits against basically using AQL for checking model characteristics? As a result, the advantages of the constraint language against AQL can be summarized as follows:

- **Usability:** The constraint language provides a graphical notation for constraints. The visualization follows a simple logical schema and users of the Constraint-Modeling-Language can understand the semantical meaning of a visualized constraint quite fast.
- **Scalability:** In cases of many different constraints for various models, the execution of AQL queries implies a lot of code and obviously time exposure. On the contrary, the Constraint-Modeling-Language can be divided and structured into multiple constraint models enabling a good overview and a swift assignment of constraints.
- **Extensibility:** In contrast to AQL, the constraint language allows additional functionality triggered by particular conditions in constraint models e.g. if the attribute

6. Discussion

Execution_Time exceeds the value 90, the script has to send an e-mail notification to the process responsible.

Bibliography

- [1] Donald Bell. UML Basics: The class diagram. *IBM.[Online] IBM*, 15(09), 2004. http://www.softwareresearch.net/fileadmin/src/docs/teaching/WS13/SE/UML_basics-_The_class_diagram.pdf [Online, accessed June 22, 2016].
- [2] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Gabriele Taentzer. *UML 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools: 4th International Conference Toronto, Canada, October 1–5, 2001 Proceedings*, chapter A Visualization of OCL Using Collaborations, pages 257–271. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.
- [3] D. Budgen, A. J. Burn, O. P. Brereton, B. A. Kitchenham, and R. Pretorius. Empirical evidence about the UML: a systematic literature review. *Software: Practice and Experience*, 41(4):363–392, 2011. https://www.researchgate.net/publication/220280714_Empirical_evidence_about_the_UML_a_systematic_literature_review [Online, accessed June 22, 2016].
- [4] Dov Dori. Why Significant UML Change is Unlikely. *Commun. ACM*, 45(11):82–85, November 2002. https://www.researchgate.net/publication/27292971_Why_significant_UML_change_is_unlikely [Online, accessed June 21, 2016].
- [5] Hans-Georg Fill. SeMFIS: A Tool for Managing Semantic Conceptual Models. In *Workshop on Graphical Modeling Language Development*, Lyngby, Denmark, July 2012. http://homepage.dke.univie.ac.at/fill/papers/Fill_SeMFIS_GMLD_Workshop.pdf [Online, accessed July 2, 2016].
- [6] Hans-Georg Fill. Semantic-based Modeling for Information Systems using the SeMFIS Platform. *Tutorial for Informatik'2016*, 2016. <http://www.informatik2016.de/1187.html> [Online, accessed July 2, 2016].
- [7] Hans-Georg Fill. SeMFIS: A Flexible Engineering Platform for Semantic Annotations of Conceptual Models. *Semantic Web(SWJ)*, 2016. <http://www.semantic-web-journal.net/content/semfis-flexible-engineering-platform-semantic-annotations-conceptual-models-0> [Online, accessed July 2, 2016].

- [8] Hans-Georg Fill and Dimitris Karagiannis. On the conceptualisation of modelling methods using the ADOxx meta modelling platform. *Enterprise Modelling and Information Systems Architectures-An International Journal*, 8(1), 2013. eprints.cs.univie.ac.at/3657/1/Fill_Karagiannis_EMISA_2013.pdf [Online, accessed June 22, 2016].
- [9] Hans-Georg Fill, Timothy Redmond, and Dimitris Karagiannis. Formalizing Meta Models with FDMM: The ADOxx Case. In J. Cordeiro, L. Maciaszek, and J. Filipe, editors, *Enterprise Information Systems - 14th International Conference, ICEIS 2012, Wroclaw, Poland, June 28 - July 1, 2012, Revised Selected Papers*, volume 141 of *LNBIP*. Springer, 2013. http://homepage.dke.univie.ac.at/fill/papers/Fill_etal_FDMM_ADOxx_2013.pdf [Online, accessed June 21, 2016].
- [10] Andrew Fish, John Howse, Gabriele Taentzer, and Jessica Winkelmann. Two Visualizations of OCL: A comparison. 2005. <http://www.mathematik.uni-marburg.de/~swt/Publikationen-Taentzer/VOCLTR.pdf> [Online, accessed June 22, 2016].
- [11] Joseph Yossi Gil, John Howse, and Stuart Kent. Constraint diagrams: a step beyond uml. In *tools*, page 453. IEEE, 1999. https://kar.kent.ac.uk/21740/1/constraint_diagrams_a_step_gil.pdf [Online, accessed June 22, 2016].
- [12] BOC group. ADOxx Documentation - class Attribute and Attribute Types. <https://www.adoxx.org/live/class-attribute-and-attribute-types> [Online, accessed June 22, 2016].
- [13] BOC group. ADOxx Documentation - Class Cardinality. <https://www.adoxx.org/live/class-cardinalities> [Online, accessed June 22, 2016].
- [14] Object Management Group. Business Process Model and Notation. Version 2.0.2, <http://www.omg.org/spec/BPMN/2.0.2/> [Online, accessed June 21, 2016].
- [15] Object Management Group. Object Constraint Language. Version 2.4, <http://www.omg.org/spec/OCL/2.4> [Online, accessed June 22, 2016].
- [16] Object Management Group. Unified Modeling Language. Version 2.5, <http://www.omg.org/spec/UML/2.5/> [Online, accessed June 22, 2016].
- [17] Object Management Group. Introduction to OMG's Unified Modeling Language (UML), June 2005. http://www.omg.org/gettingstarted/what_is_uml.htm [Online, accessed June 21, 2016].
- [18] D. Harel and B. Rumpe. Modeling Languages: Syntax, Semantics and All That Stuff,

- Part I: The Basic Stuff. Technical report, Jerusalem, Israel, Israel, 2000. <http://www4.in.tum.de/publ/papers/HR00.pdf> [Online, accessed June 22, 2016].
- [19] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, October 2004. <http://www.wisdom.weizmann.ac.il/~harel/papers/ModSemantics.pdf> [Online, accessed June 22, 2016].
- [20] Paul Harmon. The BPTrends 2010 BPM Software Tools Report on BOC's Adonis Version 4.0, 2015. <http://www.bptrends.com/publicationfiles/2010\%20BPM\%20Tools\%20Report-BOCph.pdf> [Online, accessed June 23, 2016].
- [21] S. Junginger, H. Kuehn, R. Strobl, and D. Karagiannis. ADONIS: A next generation business process management tool - Concepts and Applications. *Wirtschaftsinformatik*, 42(5):292–401, 2010.
- [22] Dimitris Karagiannis and Harald Kühn. Metamodelling platforms. In *EC-Web*, volume 2455, page 182, 2002. http://www.openmodels.at/c/document_library/get_file?uuid=08ab1053-ebfc-4c65-9120-3eb419ea5090&groupId=268312, [Online, accessed June 22, 2016].
- [23] Stuart Kent. Constraint Diagrams: Visualizing Invariants in Object-oriented Models. *SIGPLAN Not.*, 32(10):327–341, October 1997. https://kar.kent.ac.uk/21444/1/Visualizing_Invariants_in_Object-Oriented_Models.pdf [Online, accessed June 24, 2016].
- [24] Heiko Kern, Axel Hummel, and Stefan Kühne. Towards a Comparative Analysis of Meta-metamodels. In *Proceedings of the Compilation of the Co-located Workshops on DSM'11, TMC'11, AGERE! 2011, AOPES'11, NEAT'11, & VML'11, SPLASH '11 Workshops*, pages 7–12, New York, NY, USA, 2011. ACM. <http://www.dsmforum.org/events/dsm11/papers/kern.pdf> [Online, accessed June 22, 2016].
- [25] Christiane Kiesner, Gabriele Taentzer, and Jessica Winkelmann. Visual OCL: A Visual Notation of the Object Constraint Language. *Technische Universität Berlin*, 2002. <http://www.user.tu-berlin.de/o.runge/tfs/projekte/vocl/gKTW02.pdf> [Online, accessed June 24, 2016].
- [26] Cris Kobryn. Will UML 2.0 Be Agile or Awkward? *Commun. ACM*, 45(1):107–110, January 2002. <http://www.itu.int/itudoc/itu-t/workshop/framewrk/004/78054.pdf> [Online, accessed June 22, 2016].
- [27] Marian Petre. UML in Practice. In *Proceedings of the 2013 International Conference on Software Engineering, ICSE '13*, pages 722–731, Piscataway, NJ, USA, 2013.

- IEEE Press. <http://oro.open.ac.uk/35805/8/UML\%20in\%20practice\%20.pdf> [Online, accessed June 22, 2016].
- [28] Jonathan Sprinkle, Bernhard Rumpe, Hans Vangheluwe, and Gabor Karsai. Meta-modelling: State of the Art and Research Challenges. *CoRR*, abs/1409.2359, 2014. <http://arxiv.org/pdf/1409.2359.pdf> [Online, accessed June 22, 2016].
- [29] Niksa Visic, Hans-Georg Fill, Robert Andrei Buchmann, and Dimitris Karagiannis. A Domain-specific Language for Modeling Method Definition: from Requirements to Grammar. In *IEEE Ninth International Conference on Research Challenges in Information Science 2015*, May 2015. http://homepage.dke.univie.ac.at/fill/papers/Visic_etal_2015_RCIS_Online_Accepted_Version.pdf [Online, accessed June 22, 2016].

A. Attachment

A.1. Validation in AdoScript

```
1 CC "Modeling" GET_ACT_MODEL
2
3 CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:"Constraint"
4 SETL bModelName:1
5 SETL lConstraints:(objids)
6 SETL triggeredModelExistence:0
7 SETL triggeredObjectExistence:0
8
9 #Get all relevant attribute values for all constraint objs
10 FOR sObj in:(lConstraints)
11 {
12 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Constraint_Name"
13 SETL sConstraint_Name:(val)
14
15 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Model_Type_Name"
16 SETL sModelType:(val)
17
18 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Model_Name"
19 SETL sModelName:(val)
20
21 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Constraint_Application"
22 SETL sCon_App:(val)
23
24 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Context_Selection"
25 SETL sContext_Selection:(val)
26
27 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Context_Name"
28 SETL sContext_Name:(val)
29
30 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Instance_Existence"
31 SETL sInstance_Existence:(val)
32
33 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Layer"
```

A. Attachment

```
34 SETL sLayer:(val)
35
36 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Existence_Lower_Bound"
37 SETL nExistenceLowerBound:(val)
38
39 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Existence_Upper_Bound"
40 SETL nExistenceUpperBound:(val)
41
42 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Existence_Equal"
43 SETL nExistenceEqual:(val)
44
45 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"Cardinality_Quantification"
46 SETL sCardinalityQuantification:(val)
47
48 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_Restriction_Activate"
49 SETL sFirstAttributeRestrictionActivate:(val)
50
51 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_General_Restriction"
52 SETL sFirstAttributeGeneralRestriction:(val)
53
54 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_Name"
55 SETL sFirstAttributeName:(val)
56
57 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_Data_Type"
58 SETL sFirstAttributeDataType:(val)
59
60 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_Existence"
61 SETL sFirstAttributeExistence:(val)
62
63 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_Restriction_Type"
64 SETL sFirstAttributeRestrictionType:(val)
65
66 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_Numeric_Operator"
67 SETL sFirstAttributeNumericOperator:(val)
68
69 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_String_Operator"
70 SETL sFirstAttributeStringOperator:(val)
71
72 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:"l_Attribute_Value"
73 SETL sFirstAttributeValue:(val)
74
75
76 IF (sCon_App = "All models of model type")
77 {
78 CC "AQL" EVAL_AQL_EXPRESSION expr:("<\""+sModelType+"\">") modelscope
79 }
80 ELSE
```



```
81 {
82 CC "AQL" EVAL_AQL_EXPRESSION expr:("<" + sModelType + "\">[?\"Name\" =
83 \"\" + sModelName + "\" ]") modelscope
84 }
85
86 SETL noModels:0
87
88
89 IF (tokcnt(objids) = 0)
90 {
91 SETL noModels:1
92 }
93
94 SETL triggeredModelNonExistence:0
95
96 IF (sLayer = "Model Layer" AND noModels = 1 AND sInstance_Existence
97 = "Instance Non-Existence")
98 {
99 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
100 \nThere are no models of type " + sModelType + ".") title:("Validation")
101 }
102 ELSIF (sLayer = "Model Layer" AND noModels = 0 AND sInstance_Existence
103 = "Instance Non-Existence")
104 {
105 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name +
106 " not successful!")
107 SETL triggeredModelNonExistence:1
108 }
109 ELSIF (sLayer = "Model Layer" AND noModels = 1 AND sInstance_Existence
110 = "Instance Existence")
111 {
112 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name
113 + " not successful!")
114 }
115 ELSIF (sLayer = "Model Layer" AND noModels = 1 AND sInstance_Existence
116 = "Cardinality Existence" AND
117 sCardinalityQuantification = "Lower-/Upper-Bounds for Existence"
118 AND nExistenceLowerBound > 0)
119 {
120 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name
121 + " not successful!")
122 }
123 ELSIF (sLayer = "Model Layer" AND noModels = 1 AND sInstance_Existence
124 = "Cardinality Existence" AND
125 sCardinalityQuantification = "Existence equal Value" AND nExistenceEqual > 0)
126 {
127 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name
```

A. Attachment

```
128 + " not successful!")
129 }
130
131 FOR sModel in: (objids)
132 {
133   SETL nModelID:(VAL sModel)
134
135   CC "Core" IS_MODEL_LOADED modelid:(nModelID) # check if model is loaded
136   IF (ecode)
137   {
138     SET nEcode:(ecode) SET sErrText:(errtext)
139     EXIT
140   }
141
142   IF (NOT isloaded)
143   {
144     CC "Core" LOAD_MODEL modelid:(nModelID)
145     SETL bDiscardModel:1
146   }
147
148
149   IF (sLayer = "Model Layer" AND triggeredModelNonExistence = 0
150   AND triggeredModelExistence = 0)
151
152   {
153     CHECK_MODEL_EXISTENCE sModelType:(sModelType)
154     sReturn:sReturn
155
156     IF (tokcnt(sReturn) > 0 AND sInstance_Existence = "Instance Existence")
157     {
158       CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
159 \nThere is at least one model of type " + sModelType + ".") title:("Validation")
160       SETL triggeredModelExistence:1
161     }
162     ELSIF (tokcnt(sReturn) >= nExistenceLowerBound AND tokcnt(sReturn)
163 <= nExistenceUpperBound AND sInstance_Existence = "Cardinality Existence" AND
164 sCardinalityQuantification = "Lower-/Upper-Bounds for Existence") title:("Validation")
165 {
166   CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name +
167 " successful!\nThere are >= "
168 + STR nExistenceLowerBound + " and <= " + STR nExistenceUpperBound +
169 " models of type " + sModelType + ".") title:("Validation")
170   SETL triggeredModelExistence:1
171 }
172   ELSIF (tokcnt(sReturn) = nExistenceEqual AND sInstance_Existence
173 = "Cardinality Existence" AND sCardinalityQuantification =
174 "Existence equal Value") title:("Validation")
```

```
175 {
176 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
177 \nThere are " + STR nExistenceEqual + " models of type " + sModelType + ".")
178 title:("Validation")
179 SETL triggeredModelExistence:1
180 }
181 ELSE
182 {
183 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name
184 + " not successful!")
185 }
186 }
187
188 IF (sContext_Selection = "Class" AND sLayer = "Object Layer")
189 {
190 CHECK_CLASS_EXISTENCE nModelID:(nModelID)
191 sContext_Name:(sContext_Name)
192 sContext_Selection:(sContext_Selection)
193 sReturn:sReturn
194
195 #Instance existence w/o attribute
196 IF (tokcnt(sReturn) > 0 AND sInstance_Existence = "Instance Existence" AND
197 sFirstAttributeRestrictionActivate = "Disabled" AND triggeredObjectExistence = 0)
198 {
199 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
200 \nThere is at least one object of class " + sContext_Name
201 + " in a model of model type " + sModelType + ".") title:("Validation")
202 SETL triggeredObjectExistence:1
203 }
204
205 #instance existence with attribute
206 IF (tokcnt(sReturn) > 0 AND sInstance_Existence = "Instance Existence"
207 AND sFirstAttributeRestrictionActivate = "Enabled")
208 {
209 #First Attribute Value Restriction String
210 IF (sFirstAttributeGeneralRestriction = "Restrict Attribute Value"
211 AND (sFirstAttributeDataType != "INTEGER" AND sFirstAttributeDataType != "DOUBLE"
212 AND sFirstAttributeDataType != "TIME" AND sFirstAttributeDataType != "DATE" AND
213 sFirstAttributeDataType != "DATETIME"))
214 {
215 SETL checked:0
216
217 IF (sCon_App = "All models of model type")
218 {
219
220 CC "Core" GET_ALL_MODEL_VERSIONS modeltype:(sModelType)
221 SETL lVersionIds:(modelversionids)
```

A. Attachment

```
222
223 FOR sModel in:(lVersionIds)
224 {
225 CC "Core" GET_CLASS_ID classname:(sContext_Name)
226 CC "Core" GET_MODEL_BASENAME modelid:(VAL sModel)
227 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
228 CC "Core" GET_ATTR_TYPE attrid:(attrid)
229 CC "Core" GET_ALL_OBJS_WITH_ATTR_VAL modelid:(VAL sModel) classid:(classid)
230 attrid:(attrid) val:(sFirstAttributeValue)
231
232 #CC "AdoScript" INFOBOX (objids)
233 IF (tokcnt(objids) > 0 AND attrtype = sFirstAttributeDataType AND
234 sFirstAttributeStringOperator = "=")
235 {
236 SETL checked:1
237 SETL sBaseName:(basename)
238 }
239 ELSIF (tokcnt(objids) = 0 AND attrtype = sFirstAttributeDataType AND
240 sFirstAttributeStringOperator = "!=")
241 {
242 SETL checked:1
243 SETL sBaseName:(basename)
244 }
245 }
246
247 IF (checked = 1)
248 {
249 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
250 \nThere is at least one object of class " + sContext_Name +
251 " with an attribute " + sFirstAttributeName + " " + sFirstAttributeStringOperator
252 + " " + sFirstAttributeValue + " of type " + sFirstAttributeDataType
253 + " found in model " + sBaseName
254 + ".") title:("Validation")
255 SETL checked:0
256 }
257 ELSE
258 {
259 CC "AdoScript" WARNINGBOX ("Validation not successfull!")
260 }
261
262 }
263 IF (sCon_App != "All models of model type")
264 {
265 CC "Core" GET_MODEL_ID modelname:(sModelName) modeltype:(sModelType)
266 CC "Core" GET_CLASS_ID classname:(sContext_Name)
267 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
268 CC "Core" GET_ATTR_TYPE attrid:(attrid)
```

```
269 CC "Core" GET_ALL_OBJS_WITH_ATTR_VAL modelid:(modelid) classid:(classid)
270 attrid:(attrid) val:(sFirstAttributeValue)
271
272 IF (tokcnt(objids) > 0 AND attrtype = sFirstAttributeDataType AND
273 sFirstAttributeStringOperator = "=")
274 {
275 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
276 \nThere is at least one object of class " + sContext_Name +
277 " with an attribute " + sFirstAttributeName + " " + sFirstAttributeStringOperator
278 + " " + sFirstAttributeValue + " of type " + sFirstAttributeDataType
279 + " found in model " + sModelName
280 + ".") title:("Validation")
281 }
282 ELSIF (tokcnt(objids) = 0 AND attrtype = sFirstAttributeDataType AND
283 sFirstAttributeStringOperator = "!=")
284 {
285 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
286 \nThere is at least one object of class " + sContext_Name +
287 " with an attribute " + sFirstAttributeName + " " + sFirstAttributeStringOperator
288 + " " + sFirstAttributeValue + " of type " + sFirstAttributeDataType
289 + " found in model " + sModelName
290 + ".") title:("Validation")
291 }
292 ELSE
293 {
294 CC "AdoScript" WARNINGBOX ("Validation not successfull!")
295 }
296 }
297 }
298
299
300 #First Attribute Value Restriction Numeric
301 IF (sFirstAttributeGeneralRestriction = "Restrict Attribute Value"
302 AND (sFirstAttributeDataType = "INTEGER" OR sFirstAttributeDataType = "DOUBLE" OR
303 sFirstAttributeDataType = "TIME" OR sFirstAttributeDataType = "DATE"
304 OR sFirstAttributeDataType = "DATETIME"))
305 {
306 SETL checked:0
307
308 IF (sCon_App = "All models of model type")
309 {
310
311 CC "Core" GET_ALL_MODEL_VERSIONS modeltype:(sModelType)
312 SETL lVersionIds:(modelversionids)
313
314 FOR sModel in:(lVersionIds)
315 {
```

A. Attachment

```
316 CC "Core" GET_CLASS_ID classname:(sContext_Name)
317 CC "Core" GET_MODEL_BASENAME modelid:(VAL sModel)
318 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
319 CC "Core" GET_ATTR_TYPE attrid:(attrid)
320 CC "Core" GET_ALL_OBJS_WITH_ATTR_VAL modelid:(VAL sModel) classid:(classid)
321 attrid:(attrid) val:(sFirstAttributeValue)
322
323 #CC "AdoScript" INFOBOX (objids)
324 IF (tokcnt(objids) > 0 AND attrtype = sFirstAttributeDataType
325 AND sFirstAttributeNumericOperator = "=")
326 {
327 SETL checked:1
328 SETL sBaseName:(basename)
329 }
330 ELSIF (tokcnt(objids) = 0 AND attrtype = sFirstAttributeDataType
331 AND sFirstAttributeNumericOperator = "!=")
332 {
333 SETL checked:1
334 SETL sBaseName:(basename)
335 }
336 ELSIF ((sFirstAttributeNumericOperator != "=" OR
337 sFirstAttributeNumericOperator != "!=") AND attrtype = sFirstAttributeDataType)
338 {
339 CC "Core" GET_ALL_OBJS_OF_CLASSNAME classname:(sContext_Name) modelid:(VAL sModel)
340
341 #CC "AdoScript" INFOBOX (objids)
342
343 FOR sObj in:(objids)
344 {
345 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrid:(attrid)
346
347 IF (val < VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = "<")
348 {
349 SETL checked:1
350 SETL sBaseName:(basename)
351 }
352 ELSIF (val <= VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = "<=")
353 {
354 SETL checked:1
355 SETL sBaseName:(basename)
356 }
357 ELSIF (val > VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = ">")
358 {
359 SETL checked:1
360 SETL sBaseName:(basename)
361 }
362 ELSIF (val >= VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = ">=")
```

```
363 {
364 SETL checked:1
365 SETL sBaseName:(basename)
366 }
367 }
368 }
369 }
370 IF (checked = 1)
371 {
372 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
373 \nThere is at least one object of class " + sContext_Name +
374 " with an attribute " + sFirstAttributeName + " " + sFirstAttributeNumericOperator
375 + " " + sFirstAttributeValue + " of type " + sFirstAttributeDataType
376 + " found in model " + sBaseName
377 + ".") title:("Validation")
378 SETL checked:0
379 }
380 ELSE
381 {
382 CC "AdoScript" WARNINGBOX ("Validation not successfull!")
383 }
384
385 }
386 IF (sCon_App != "All models of model type")
387 {
388 CC "Core" GET_MODEL_ID modelname:(sModelName) modeltype:(sModelType)
389 CC "Core" GET_CLASS_ID classname:(sContext_Name)
390 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
391 CC "Core" GET_ATTR_TYPE attrid:(attrid)
392 CC "Core" GET_ALL_OBJS_WITH_ATTR_VAL modelid:(modelid) classid:(classid)
393 attrid:(attrid) val:(sFirstAttributeValue)
394
395 IF (tokcnt(objids) > 0 AND attrtype = sFirstAttributeDataType
396 AND sFirstAttributeStringOperator = "=")
397 {
398 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
399 \nThere is at least one object of class " + sContext_Name +
400 " with an attribute " + sFirstAttributeName + " " + sFirstAttributeStringOperator
401 + " " + sFirstAttributeValue + " of type " + sFirstAttributeDataType
402 + " found in model " + sModelName
403 + ".") title:("Validation")
404 }
405 ELSIF (tokcnt(objids) = 0 AND attrtype = sFirstAttributeDataType
406 AND sFirstAttributeStringOperator = "!=")
407 {
408 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
409 \nThere is at least one object of class " + sContext_Name +
```

A. Attachment

```
410 " with an attribute " + sFirstAttributeName + " " + sFirstAttributeNumericOperator
411 + " " + sFirstAttributeValue + " of type " + sFirstAttributeDataType
412 + " found in model " + sModelName
413 + ".") title:("Validation")
414 }
415 ELSIF ((sFirstAttributeNumericOperator != "=" OR
416 sFirstAttributeNumericOperator != "!=") AND attrtype = sFirstAttributeDataType)
417 {
418 SETL checked:0
419 CC "Core" GET_ALL_OBJS_OF_CLASSNAME classname:(sContext_Name) modelid:(modelid)
420
421 #CC "AdoScript" INFOBOX (objids)
422
423 FOR sObj in:(objids)
424 {
425 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrid:(attrid)
426
427 IF (val < VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = "<")
428 {
429 SETL checked:1
430 }
431 ELSIF (val <= VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = "<=")
432 {
433 SETL checked:1
434 }
435 ELSIF (val > VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = ">")
436 {
437 SETL checked:1
438 }
439 ELSIF (val >= VAL sFirstAttributeValue AND sFirstAttributeNumericOperator = ">=")
440 {
441 SETL checked:1
442 }
443
444 }
445 IF (checked = 1)
446 {
447 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
448 \nThere is at least one object of class " + sContext_Name +
449 " with an attribute " + sFirstAttributeName + " " + sFirstAttributeNumericOperator
450 + " " + sFirstAttributeValue + " of type " + sFirstAttributeDataType
451 + " found in model " + sModelName
452 + ".") title:("Validation")
453 SETL checked:0
454 }
455 }
456 ELSE
```



```
457 {
458 CC "AdoScript" WARNINGBOX ("Validation not successfull!")
459 }
460 }
461 }
462 }
463 #Instance non-existence
464 ELSIF (sInstance_Existence = "Instance Non-Existence" AND
465 triggeredObjectExistence = 0)
466 {
467 CC "Core" GET_MODEL_BASENAME modelid:(nModelID)
468 #CC "AdoScript" INFOBOX (basename)
469 IF (tokcnt(sReturn) = 0)
470 {
471 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " in " +
472 basename + " successful!\nThere are no objects of class " + sContext_Name + ".")
473 title:("Validation")
474 SETL triggeredObjectExistence:1
475 }
476 ELSE
477 {
478 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name + " in "
479 + basename + " not successfull!")
480 }
481 }
482
483 #Cardinality bounds
484 ELSIF (sInstance_Existence = "Cardinality Existence" AND sCardinalityQuantification
485 = "Lower-/Upper-Bounds for Existence")
486 {
487 CC "Core" GET_MODEL_BASENAME modelid:(nModelID)
488 IF (tokcnt(sReturn) >= nExistenceLowerBound AND tokcnt(sReturn)
489 <= nExistenceUpperBound)
490 {
491 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " in "
492 + basename + " successful!\nThere are >= " + STR nExistenceLowerBound + " and <= "
493 + STR nExistenceUpperBound
494 + " objects of class "
495 + sContext_Name + ".") title:("Validation")
496 }
497 ELSE
498 {
499 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name + " in "
500 + basename + " not successfull!")
501 }
502 }
503
```

```
504
505 #Cardinality equal
506 ELSIF (sInstance_Existence = "Cardinality Existence" AND
507 sCardinalityQuantification = "Existence equal Value")
508 {
509 CC "Core" GET_MODEL_BASENAME modelid:(nModelID)
510 IF (tokcnt(sReturn) = nExistenceEqual)
511 {
512 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " in "
513 + basename + " successful!\nThere are " + STR nExistenceEqual
514 + " objects of class " + sContext_Name
515 + ".") title:("Validation")
516 }
517 ELSE
518 {
519 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name
520 + " in " + basename + " not successful!")
521 }
522 }
523
524 #For All
525 ELSIF (sInstance_Existence = "No Restriction")
526 {
527 #First Attribute Restriction enabled
528 IF (sFirstAttributeRestrictionActivate = "Enabled")
529 {
530 #First Attribute Existence
531 IF (sFirstAttributeGeneralRestriction = "Restrict Attribute Existence")
532 {
533 IF (sFirstAttributeExistence = "Must exist")
534 {
535 CC "Core" GET_CLASS_ID classname:(sContext_Name)
536 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
537 IF (ecode = 0)
538 {
539 CC "Core" GET_ATTR_TYPE attrid:(attrid)
540 IF (attrtype = sFirstAttributeDataType)
541 {
542 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name
543 + " successful!\nAll objects of class " + sContext_Name
544 + " have an attribute named " +
545 sFirstAttributeName + " of data type " + attrtype + ".") title:("Validation")
546 }
547 ELSE
548 {
549 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name + " not successful!")
550 }
```

```

551 }
552 ELSE
553 {
554 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name + " not successful!")
555 }
556 }
557 ELSE #Can not exist
558 {
559 CC "Core" GET_CLASS_ID classname:(sContext_Name)
560 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
561 IF (ecode != 0)
562 {
563 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " successful!
564 \nAll objects of class " + sContext_Name + " have no attribute named " +
565 sFirstAttributeName + ".") title:("Validation")
566 }
567 ELSE
568 {
569 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name + " not successful!")
570 }
571 }
572 }
573 #First Attribute Value Restriction String for all models
574 IF (sFirstAttributeGeneralRestriction = "Restrict Attribute Value"
575 AND (sFirstAttributeDataType != "INTEGER" AND sFirstAttributeDataType != "DOUBLE"
576 AND sFirstAttributeDataType != "TIME" AND sFirstAttributeDataType != "DATE"
577 AND sFirstAttributeDataType != "DATETIME") AND sCon_App = "All models of model type")
578 {
579 CC "Core" GET_ALL_MODEL_VERSIONS modeltype:(sModelType)
580 SETL lVersionIds:(modelversionids)
581
582 FOR sModel in:(lVersionIds)
583 {
584 CC "Core" GET_MODEL_BASENAME modelid:(VAL sModel)
585 SETL sModelBaseName:(basename)
586
587 CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(VAL sModel) classname:(sContext_Name)
588
589 #CC "Core" ECODE_TO_ERRTEXT ecode:(ecode)
590 #CC "AdoScript" INFOBOX (errtext)
591
592 SETL lContextObjs:(objids)
593
594 FOR sObj in:(lContextObjs)
595 {
596 CC "Core" GET_CLASS_ID classname:(sContext_Name)
597 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)

```

A. Attachment

```
598 CC "Core" GET_ATTR_TYPE attrid:(attrid)
599 SETL sFirstAttributeType:(attrtype)
600
601 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:(sFirstAttributeName)
602 SETL sFirstAttributeModelValue:(val)
603
604 #CC "AdoScript" INFOBOX (sFirstAttributeType + " " + sFirstAttributeDataType)
605
606 IF (ecode != 0 OR sFirstAttributeType != sFirstAttributeDataType)
607 {
608 CC "AdoScript" WARNINGBOX ("Validation not successfull! There is no attribute "
609 + sFirstAttributeName + " of data type " + sFirstAttributeDataType + ".")
610 title:("Validation")
611 }
612 ELSIF (sFirstAttributeValue = sFirstAttributeModelValue AND
613 sFirstAttributeStringOperator = "=")
614 {
615 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
616 \nObject " + sObj + " appearing in " + sModelBaseName+" has an attribute "
617 + sFirstAttributeName + " = " + sFirstAttributeValue + " of data type "
618 + sFirstAttributeType + ".") title:("Validation")
619 }
620 ELSIF (sFirstAttributeValue != sFirstAttributeModelValue AND
621 sFirstAttributeStringOperator = "!=")
622 {
623 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
624 \nObject " + sObj + " appearing in " + sModelBaseName + " has an attribute "
625 +
626 sFirstAttributeName + " != " + sFirstAttributeValue + " of data type "
627 + sFirstAttributeType + ".") title:("Validation")
628 }
629 ELSE
630 {
631 CC "AdoScript" WARNINGBOX ("Validation for " + sModelBaseName + " not successful!")
632 }
633 }
634 }
635 }
636 #First Attribute Value Restriction String for a specific model
637 IF (sFirstAttributeGeneralRestriction = "Restrict Attribute Value"
638 AND (sFirstAttributeDataType != "INTEGER" AND sFirstAttributeDataType != "DOUBLE"
639 AND sFirstAttributeDataType != "TIME" AND sFirstAttributeDataType != "DATE"
640 AND sFirstAttributeDataType != "DATETIME") AND sCon_App
641 != "All models of model type")
642 {
643 CC "Core" GET_MODEL_ID modelname:(sModelName) modeltype:(sModelType)
644
```

```

645 CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:(sContext_Name)
646
647 SETL lContextObjs:(objids)
648
649 FOR sObj in:(lContextObjs)
650 {
651 CC "Core" GET_CLASS_ID classname:(sContext_Name)
652 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
653 CC "Core" GET_ATTR_TYPE attrid:(attrid)
654 SETL sFirstAttributeType:(attrtype)
655
656 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:(sFirstAttributeName)
657 SETL sFirstAttributeModelValue:(val)
658
659 #CC "AdoScript" INFOBOX (sFirstAttributeType + " " + sFirstAttributeDataType)
660
661 IF (ecode != 0 OR sFirstAttributeType != sFirstAttributeDataType)
662 {
663 CC "AdoScript" WARNINGBOX ("Validation not successfull! There is no attribute "
664 + sFirstAttributeName + " of data type " + sFirstAttributeDataType + ".")
665 title:("Validation")
666 }
667 ELSIF (sFirstAttributeValue = sFirstAttributeModelValue AND
668 sFirstAttributeStringOperator = "=")
669 {
670 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name +
671 " sucessfull!\nObject " + sObj + " appearing in " + sModelName+ " has an attribute "
672 +
673 sFirstAttributeName + " = " + sFirstAttributeValue + " of data type "
674 + sFirstAttributeType + ".") title:("Validation")
675 }
676 ELSIF (sFirstAttributeValue != sFirstAttributeModelValue AND
677 sFirstAttributeStringOperator = "!=")
678 {
679 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name
680 + " sucessfull!\nObject " + sObj + " appearing in " + sModelName
681 + " has an attribute "
682 +
683 sFirstAttributeName + " != " + sFirstAttributeValue + " of data type "
684 + sFirstAttributeType + ".") title:("Validation")
685 }
686 ELSE
687 {
688 CC "AdoScript" WARNINGBOX ("Validation for " + sModelName + " not successful!")
689 }
690 }
691

```

A. Attachment

```
692 }
693 #First Attribute Value Restriction Numeric for all models
694 IF (sFirstAttributeGeneralRestriction = "Restrict Attribute Value"
695 AND (sFirstAttributeDataType = "INTEGER" OR sFirstAttributeDataType = "DOUBLE" OR
696 sFirstAttributeDataType = "TIME" OR sFirstAttributeDataType = "DATE"
697 OR sFirstAttributeDataType = "DATETIME") AND sCon_App = "All models of model type")
698 {
699 CC "Core" GET_ALL_MODEL_VERSIONS modeltype:(sModelType)
700 SETL lVersionIds:(modelversionids)
701
702 FOR sModel in:(lVersionIds)
703 {
704 CC "Core" GET_MODEL_BASENAME modelid:(VAL sModel)
705 SETL sModelBaseName:(basename)
706
707 CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(VAL sModel) classname:(sContext_Name)
708
709 #CC "Core" ECODE_TO_ERRTEXT ecode:(ecode)
710 #CC "AdoScript" INFOBOX (errtext)
711
712 SETL lContextObjs:(objids)
713
714 FOR sObj in:(lContextObjs)
715 {
716 CC "Core" GET_CLASS_ID classname:(sContext_Name)
717 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
718 CC "Core" GET_ATTR_TYPE attrid:(attrid)
719 SETL sFirstAttributeType:(attrtype)
720
721 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:(sFirstAttributeName)
722 SETL sFirstAttributeModelValue:(val)
723
724 IF (ecode != 0 OR sFirstAttributeType != sFirstAttributeDataType)
725 {
726 CC "AdoScript" WARNINGBOX ("Validation not successfull! There is no attribute "
727 + sFirstAttributeName + " of data type " + sFirstAttributeDataType
728 + ".") title:("Validation")
729 }
730 ELSIF (VAL sFirstAttributeValue = sFirstAttributeModelValue
731 AND sFirstAttributeNumericOperator = "=")
732 {
733 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
734 \nObject " + sObj + " appearing in " + sModelBaseName+ " has an attribute "
735 + sFirstAttributeName + " = " + sFirstAttributeValue + " of data type "
736 + sFirstAttributeType + ".") title:("Validation")
737 }
738 ELSIF (VAL sFirstAttributeValue != sFirstAttributeModelValue
```

```

739 AND sFirstAttributeNumericOperator = "!=")
740 {
741 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name
742 + " sucessfull!\nObject " + sObj + " appearing in " + sModelBaseName
743 + " has an attribute "
744 + sFirstAttributeName + " != " + sFirstAttributeValue + " of data type "
745 + sFirstAttributeType + ".") title:("Validation")
746 }
747 ELSIF (VAL sFirstAttributeValue > sFirstAttributeModelValue
748 AND sFirstAttributeNumericOperator = "<")
749 {
750 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name
751 + " sucessfull!\nObject " + sObj + " appearing in " + sModelBaseName
752 + " has an attribute "
753 + sFirstAttributeName + " < " + sFirstAttributeValue + " of data type "
754 + sFirstAttributeType + ".") title:("Validation")
755 }
756 ELSIF (VAL sFirstAttributeValue >= sFirstAttributeModelValue AND
757 sFirstAttributeNumericOperator = "<=")
758 {
759 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
760 \nObject " + sObj + " appearing in " + sModelBaseName + " has an attribute "
761 + sFirstAttributeName + " <= " + sFirstAttributeValue + " of data type "
762 + sFirstAttributeType + ".") title:("Validation")
763 }
764 ELSIF (VAL sFirstAttributeValue < sFirstAttributeModelValue
765 AND sFirstAttributeNumericOperator = ">")
766 {
767 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name
768 + " sucessfull!\nObject " + sObj + " appearing in " + sModelBaseName
769 + " has an attribute "
770 + sFirstAttributeName + " > " + sFirstAttributeValue + " of data type "
771 + sFirstAttributeType + ".") title:("Validation")
772 }
773 ELSIF (VAL sFirstAttributeValue <= sFirstAttributeModelValue
774 AND sFirstAttributeNumericOperator = ">=")
775 {
776 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name
777 + " sucessfull!\nObject " + sObj + " appearing in " + sModelBaseName
778 + " has an attribute "
779 + sFirstAttributeName + " >= " + sFirstAttributeValue + " of data type "
780 + sFirstAttributeType + ".") title:("Validation")
781 }
782 ELSE
783 {
784 CC "AdoScript" WARNINGBOX ("Validation for " + sModelBaseName + " not successful!")
785 }

```

A. Attachment

```
786 }
787 }
788
789 }
790 #First Attribute Value Restriction Numeric for a specific model
791 IF (sFirstAttributeGeneralRestriction = "Restrict Attribute Value" AND
792 (sFirstAttributeDataType = "INTEGER" OR sFirstAttributeDataType = "DOUBLE" OR
793 sFirstAttributeDataType = "TIME" OR sFirstAttributeDataType = "DATE"
794 OR sFirstAttributeDataType = "DATETIME") AND sCon_App != "All models of model type")
795 {
796 CC "Core" GET_MODEL_ID modelname:(sModelName) modeltype:(sModelType)
797
798 CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(modelid) classname:(sContext_Name)
799
800 SETL lContextObjs:(objids)
801
802 FOR sObj in:(lContextObjs)
803 {
804 CC "Core" GET_CLASS_ID classname:(sContext_Name)
805 CC "Core" GET_ATTR_ID classid:(classid) attrname:(sFirstAttributeName)
806 CC "Core" GET_ATTR_TYPE attrid:(attrid)
807 SETL sFirstAttributeType:(attrtype)
808
809 CC "Core" GET_ATTR_VAL objid:(VAL sObj) attrname:(sFirstAttributeName)
810 SETL sFirstAttributeModelValue:(val)
811
812 IF (ecode != 0 OR sFirstAttributeType != sFirstAttributeDataType)
813 {
814 CC "AdoScript" WARNINGBOX ("Validation not successfull! There is no attribute "
815 + sFirstAttributeName + " of data type " + sFirstAttributeDataType
816 + ".") title:("Validation")
817 }
818 ELSIF (VAL sFirstAttributeValue = sFirstAttributeModelValue AND
819 sFirstAttributeNumericOperator = "=")
820 {
821 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
822 \nObject " + sObj + " appearing in " + sModelName+ " has an attribute "
823 + sFirstAttributeName + " = " + sFirstAttributeValue + " of data type " +
824 sFirstAttributeType + ".") title:("Validation")
825 }
826 ELSIF (VAL sFirstAttributeValue != sFirstAttributeModelValue AND
827 sFirstAttributeNumericOperator = "!=")
828 {
829 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
830 \nObject " + sObj + " appearing in " + sModelName + " has an attribute "
831 + sFirstAttributeName + " != " + sFirstAttributeValue + " of data type " +
832 sFirstAttributeType + ".") title:("Validation")
```



```
833 }
834 ELSIF (VAL sFirstAttributeValue > sFirstAttributeModelValue AND
835 sFirstAttributeNumericOperator = "<")
836 {
837 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
838 \nObject " + sObj + " appearing in " + sModelName + " has an attribute "
839 + sFirstAttributeName + " < " + sFirstAttributeValue + " of data type "
840 + sFirstAttributeType + ".") title:("Validation")
841 }
842 ELSIF (VAL sFirstAttributeValue >= sFirstAttributeModelValue AND
843 sFirstAttributeNumericOperator = "<=")
844 {
845 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
846 \nObject " + sObj + " appearing in " + sModelName + " has an attribute "
847 + sFirstAttributeName + " <= " + sFirstAttributeValue + " of data type "
848 + sFirstAttributeType + ".") title:("Validation")
849 }
850 ELSIF (VAL sFirstAttributeValue < sFirstAttributeModelValue AND
851 sFirstAttributeNumericOperator = ">")
852 {
853 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + " sucessfull!
854 \nObject " + sObj + " appearing in " + sModelName + " has an attribute "
855 + sFirstAttributeName + " > " + sFirstAttributeValue + " of data type "
856 + sFirstAttributeType + ".") title:("Validation")
857 }
858 ELSIF (VAL sFirstAttributeValue <= sFirstAttributeModelValue AND
859 sFirstAttributeNumericOperator = ">=")
860 {
861 CC "AdoScript" VIEWBOX text:("Validation for " + sConstraint_Name + "
862 sucessfull!\nObject " + sObj + " appearing in " + sModelName + " has an attribute "
863 + sFirstAttributeName + " >= " + sFirstAttributeValue + " of data type " +
864 sFirstAttributeType + ".") title:("Validation")
865 }
866 ELSE
867 {
868 CC "AdoScript" WARNINGBOX ("Validation for " + sModelName + " not successful!")
869 }
870 }
871 }
872 }
873 ELSE
874 {
875 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name + " not successful!")
876 }
877 }
878
879
```

A. Attachment

```
880 #relation class was chosen as constraint reference
881 IF (sContext_Selection = "Relation Class" AND sLayer = "Object Layer")
882 {
883 CHECK_RELATIONCLASS_EXISTENCE nModelID:(nModelID)
884 sContext_Name:(sContext_Name)
885 sContext_Selection:(sContext_Selection)
886 sReturn:sReturn
887
888 IF (tokcnt(sReturn) > 0 AND sInstance_Existence = "Instance Existence")
889 {
890 CC "AdoScript" INFOBOX ("Validation for " + sConstraint_Name + " successful!
891 \nThere are objects of relation class "+sContext_Name)
892 }
893 ELSIF (tokcnt(sReturn) = 0 AND sInstance_Existence = "Instance Non-Existence")
894 {
895 CC "AdoScript" INFOBOX ("Validation for " + sConstraint_Name + " successful!
896 \nThere are no objects of relation class"+sContext_Name)
897 }
898 ELSIF (tokcnt(sReturn) >= nExistenceLowerBound AND tokcnt(sReturn) <=
899 nExistenceUpperBound AND sInstance_Existence = "Cardinality Existence" AND
900 sCardinalityQuantification = "Lower-/Upper-Bounds for Existence")
901 {
902 CC "AdoScript" INFOBOX ("Validation for " + sConstraint_Name + " successful!")
903 }
904 ELSIF (tokcnt(sReturn) = nExistenceEqual AND sInstance_Existence
905 = "Cardinality Existence" AND sCardinalityQuantification
906 = "Existence equal Value")
907 {
908 CC "AdoScript" INFOBOX ("Validation for " + sConstraint_Name + " successful!")
909 }
910 ELSE
911 {
912 CC "AdoScript" WARNINGBOX ("Validation for " + sConstraint_Name + " not successful!")
913 }
914 }
915 IF (bDiscardModell) # discard when it was loaded
916 {
917 CC "Core" DISCARD_MODEL modelid:(nModelID)
918 }
919 }
920 }
921 }
922
923 PROCEDURE global CHECK_MODEL_EXISTENCE sModelType:string
924 sReturn:reference
925 {
926 CC "Core" GET_ALL_MODEL_VERSIONS modeltype:(sModelType)
```

```
927 SETL sReturn:(modelversionids)
928 }
929
930 PROCEDURE global CHECK_CLASS_EXISTENCE nModelID:integer
931     sContext_Name:string
932     sContext_Selection:string
933     sReturn:reference
934 {
935 CC "Core" GET_ALL_OBJS_OF_CLASSNAME modelid:(nModelID) classname:(sContext_Name)
936 SETL sReturn:(objids)
937 }
938
939 PROCEDURE global CHECK_RELATIONCLASS_EXISTENCE nModelID:integer
940     sContext_Name:string
941     sContext_Selection:string
942     sReturn:reference
943 {
944 SETL sReturn:""
945 CC "Core" GET_ALL_CONNECTORS modelid:(nModelID)
946 FOR sCon in:(objids)
947 {
948 CC "Core" GET_CLASS_ID objid:(VAL sCon)
949 CC "Core" GET_CLASS_NAME classid:(classid)
950
951 IF (classname = sContext_Name)
952 {
953 SETL sReturn:(tokcat(sReturn,sCon))
954 }
955 }
956 }
```
