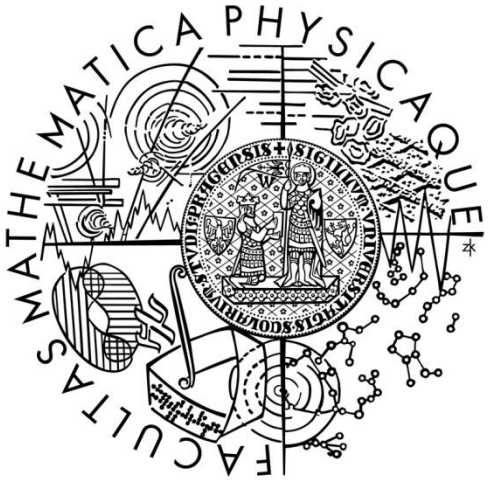
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Peter Piják

Universal Constraint Language

Department of Software Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Computer Science

Specialization: Software Systems

Prague 2011

I would like to thank to my advisor Mgr. Martin Nečaský, Ph.D. for his comments and advices, my thanks also belongs to RNDr. Irena Mlýnková, Ph.D., Martin Chytil, Karel Jakubec, Vladimír Kudelas and Marek Polák. Last but not least I want to thank to my family.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague 03.08.2011

Názov práce: Univerzálny jazyk pre integritné obmedzenia

Autor: Peter Piják

Katedra / Ústav: Katedra softwarového inženýrství

Vedúci diplomovej práce: Mgr. Martin Nečaský, Ph.D.

Abstrakt:

Dnešné softvérové systémy sú zvyčajne zložené zo systému viacerých komponent. Pri navrhovaní a modelovaní systému sa pri jednotlivých častiach modelu vyjadrujú integritné obmedzenia v rôznych jazykoch pre zápis integritných obmedzení (napríklad jazyk OCL pre UML diagramy tried, Schematron pre model XML alebo SQL triggery pre relačné databázy). Výrazy integritných obmedzení musia byť prekladané do výrazov nad inými meta-modelmi do iného jazyka, čo je netriviálna úloha.

V tejto práci je predstavený jazyk pre integritné obmedzenia Universal Constraint Language (UCL), ktorým je možné vyjadriť výrazy integritných obmedzení nad rôznymi dátovými meta-modelmi. Jazyk je formálne zadefinovaný a je implementovaný jeho analyzátor (parser). Ďalej popisujeme spôsob prevodu výrazov medzi jednotlivými meta-modelmi a odvodzovanie z výrazov v prezentovanom jazyku do existujúcich špecifických jazykov pre integritné obmedzenia.

Kľúčové slová: jazyk pre integritné obmedzenia, Modelom riadená architektúra (MDA), univerzálny formalizmus

Title: Universal Constraint Language

Author: Peter Piják

Department / Institute: Department of Software Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D.

Abstract:

Today’s software applications are typically compound of system of more application components. By modeling of software, various integrity constraint languages are used for particular parts of model (e.g. OCL for UML class diagrams, Schematron for XML or SQL triggers for relational databases). Constraint expressions need to be converted to expressions over different meta-models. These tasks are non-trivial.

In this thesis, a new common language Universal Constraint Language (UCL) for expressing integrity constraints over various data meta-models is introduced. It is formally defined and also its parser is implemented. We also present semi-automatic translating between constraints over various meta-models; and deriving constraints from the introduced language to constraints in specific constraint languages.

Keywords: constraint language, model-driven architecture, universal formalism

Contents

1. Introduction

1.1. Motivation

With the progress of advanced software architectures, such as Service-Oriented Architecture, there is a situation where today's software applications are no longer composed of consistent monolithic software. Current applications consist of a complex system of several simpler components. The individual components are responsible for the specified data and functionality parts of the entire system. These components communicate with each other. This decomposition of the software architecture brings many advantages but also problems. Design and maintenance of such a complex system is a nontrivial task.

By the design of each component, different meta-models are used. To model classes in a high-level object-oriented programming language, we use most frequently UML class diagrams [1]. To model relational databases, we use the relational model [4]. To design XML schemas, we can use conceptual model XSEM [3].

These data meta-models provide rich facilities for the description of functional and structural aspects of a software system or its sub-components. They can be expressed in a textual language as well as by a graphic notation. Direct expressive power of data models is limited for expressing complex integrity constraints. Often textual constraint languages are more suitable to solving this task.

Individual models are semantically related and interconnected. For example UML class in PIM diagram models a runtime object. These objects are in a relational database stored as columns of a table; or in XML document, they are represented by elements with attributes. A table in an ER diagram and an XSEM class are interconnected with a related UML class in a PIM diagram.

Integrity constraints are expressions which are used to express the consistency and the accuracy of modeled data. Simpler integrity constraints are applied to express the reference or the domain data integrity. They can be expressed directly using the data model. For example, the restriction that age of a person cannot be a negative number. More complex constraints must be expressed using more complex languages. For example, the condition that to a person whose age is less than 18 years cannot be assigned a car.

By a modeling of a software system, various integrity constraint languages are used for particular parts of model. By a designing of a relational database, simple domain constraints or referential constraints can be expressed directly using SQL statements by definition of columns of the tables. More complex constraints must be expressed using SQL *check* expressions or SQL triggers. Restrictions of the column's domain data can be easily visualized in a graphical annotation. By UML class diagrams, we use to express complex restrictions expressions in *Object Constraint Language* (OCL) [2]. By a modeling of XML data, domain and reference restrictions can be expressed directly by schema languages *DTD* [5] or *XML Schema* [6]. Complex constraints can be noted using languages *Schematron* [7] or *XQuery* [9].

By a modeling of a software system, we often solve such a situation that an integrity constraint need to be noted duplicated in different data meta-models. We have to express constraints it in different integrity constraints languages. These integrity constraints must be translated into constraints over another meta-model to another language. This is a nontrivial task. Such a translation must be handled manually. In addition, system designers must thoroughly understand various integrity constraints languages for each meta-model. And they must be aware of their expressive power. More often, there is currently no solution.

1.2. Aim of the thesis

The aim of this thesis is to introduce a new common language for expressing integrity constraints. The new language will be named *Universal Constraint Language* (UCL) and it will be based on OCL.

- The language must have a formally defined syntax and all kinds of its constructions, types, operations and expressions.
- In UCL, it must be able to express integrity constraints over different data models (of different meta-models). E.g. it must be possible to represent an integrity constraint over UML class diagrams, relational databases or XML documents in UCL expressions.
- To achieve the application of UCL over different meta-models, UCL will be based on a general data meta-model. We must propound and formally define this meta-model.

- It is necessary to formalize also a meta-model of UCL expressions and types. According this meta-model, it must be able to derive UCL constraints to different specific constraint languages for different meta-models. E.g. automatically translate a UCL constraint to an SQL trigger according defined rules.

We may express a UCL expression over a model of one meta-mode) and a UCL expression over a model of another meta-model. These two expressions may represent semantically the same condition of allowed data; but these expressions must not be equal. Names and relationship between elements in different models and meta-models can be different. Then two UCL expressions over various meta-models which reflect the same integrity constraint can be profoundly different too.

- The aim of this thesis is also to define deriving of UCL constraints over one data model to UCL constraints over another data model. If elements in separate models and parts of a complex software system are related and interconnected then it must be possible (according this interconnection of models) to derive constraints over one model to constraints over another related model.

The contribution of this thesis is to reduce the work by designing of complex software systems. We define UCL as a constraint language for various meta-models. The aim is that software architects will not have to express integrity constraints manually for each individual model in several different constraint languages. The vision is to create a pivot constraint language. It ensures that it will be sufficient to express a constraint only in one language (the created UCL). It will be able to automatically translate constraints over one meta-model to constraints over other meta-models and in different constraint languages.

Beside the theoretical work, the aim of this thesis is also to implement a software module which implements and demonstrates contributions of UCL. It must support and demonstrate:

- the application of UCL over different meta-models,
- the derivation of UCL constraints between different related meta-models,
- the derivation from UCL constraints to specific constraint languages.

1.3. Structure of the thesis

This thesis is organized as follows. In chapter 2, we provide a summary of several popular data meta-models. We also introduce constraint languages over these meta-models with their abilities and their expressive power.

In next chapter 3, we present best-known existing frameworks and studies for OCL. Dome of these projects extend the usage of OCL to other meta-models. Some analyze the derivation of OCL constraints to other constraint languages.

In chapter 4, we analyze ideas of the modeling and the meta-modeling. And we describe parts of the four-layer meta-modeling architecture (*Meta-Object Facility*). Then we introduce the architecture of our framework for modeling over various meta-models. This architecture supports UCL for integrity constraints over various data models. In this chapter, we define *UCL Data meta-model*. It is created to apply for different data meta-models. UCL expressions are based on this model.

In chapter 5, an informal description of UCL is presented. All expression types, operations and kinds of navigation expressions over UCL Data meta-model are explained. Chapter 6 describes meta-model of UCL expressions. According this meta-model, it is possible to implement a parser of UCL expressions over UCL Data meta-model. Chapters 7 and 8 define the mapping of meta-models of UML class diagrams and XSEM diagrams to UCL Data meta-model. They analyze the using of UCL expressions over these two meta-models. Using these mappings, we can express integrity constraints in UCL over UML and XSEM models. Chapter 9 continues in a utilization of UCL constraints. It describes the derivation of UCL constraints to Schematron rules (language for XML). And it presents the translation of UCL constraints over one meta-model to UCL constraints over another related meta-model which is interconnected by a mapping.

The last chapter 10 introduces the implemented software module in framework *DaemonX*. This software module was developed as a part of this thesis. In this software component, it is able to express UCL constraints over UML class diagrams, derive them to related XSEM models and translate them to Schematron rules. This software system is also extensible for the support of other data meta-models constraint using plug-ins.

2. Data models introduction

In this chapter, a short introduction to used data meta-models *UML class diagrams*, *relational model* and *XML* is presented. There are also introduced languages to express integrity constraints over models of these meta-models. Meta-models for UML class diagrams and for XML will be formally defined in chapters 7 and 8. Also the mappings from these meta-models to common data model of UCL will be defined.

Universal Constraint Language which is defined in this thesis will be applied to these presented meta-models thereinafter in this document. It will be possible to express integrity constraints in UCL over models of these meta-models.

2.1. UML class diagrams

UML class diagrams [1] are part of *Unified Modeling Language* (UML). Class diagrams belong between the static structure diagrams. They represent the structure of a system; they describe system's classes, attributes, methods and relationships between the classes. They are used for conceptual modeling of the system and for translating the model into the classes in a high-level object oriented programming language.

2.1.1. Constructs

The main building element is a *class*; it represents an interaction or an object to be programmed. A class has its *name* and other additional information like a *stereotype*. It can contain inner *attributes* and *operations*. An attribute is an element of data which is maintained by the class. An attribute has its *name*, *data type*, *multiplicity* (and other information like a *visibility* or a *default value*). An operation is specification of a service provided by the class. It represents what the class can do. Operation has a *name*, *parameters* (with *names* and *data types*), *return data type* (and other information like a *visibility*).

Relationships between two or more classes are represented by connecting lines elements: an *association*, an *aggregation* and a *composition*. They all can have a *name*. Ends of these lines are in connections to the classes adorned with *role names* and *roles multiciplities*.

A *generalization* is a relationship between two classes (the *parent class* and the *child class*), it indicates that the child class is a specialized form of the parent class and it has all inner elements and connected links of the parent class.

Figure 2.1 shows an example of UML class diagram with the classes with attributes. Classes are connected by an association with filled role names and role multiciplities.

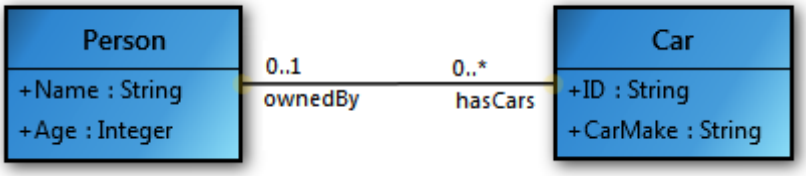


Figure 2.1: Example of a UML class diagram

2.1.2. Object Constraint Language

Object Constraint Language (OCL) [2] is a language which enables to describe expressions and constraints on UML object-oriented and state models. It was developed in IBM. Its current version (Jule 2011) is 2.2; or 2.3 – Beta 2. OCL is a part of the specification of UML.

By a modeling of an UML diagram, we typically need to describe additional information about the model, integrity constraints. Such constraints are typically specified in a natural language; but we need to describe them formally and unambiguously. OCL is a formal language based on the first-order predicate logic. Its syntax is very similar to programming languages. So it is usable for average system modeler without strong mathematical background.

OCL is a pure declarative language; its expressions are evaluated without side effect changes in the model; its expressions just return a value. OCL is not a programming language; it is designated to execution but only to express constraint invariants. OCL expressions are only a specification for programs that will determine accomplishments of the constraints.

OCL is a strong typed language. Each expression has a type which is defined statically before the interpretations. Each correct expression has to satisfy the type conformance language rules. The language has a predefined set of primitive types (*integer*, *boolean*, *string* and *real*); then each entity in the UML model represents a special type in OCL. Also structured types from the primitive types can

be created. *Tuple* as a structure of several values and collections *Set*, *Bag*, *Collection* and *Sequence*.

There are four types of constraints in OCL:

- *Invariant* is a constraint expressing a condition which must be satisfied (evaluated to *true*) all the time for all instances of defined context.
- *Pre-condition* of an operation is a condition which must be true at the time before execution of the operation.
- *Post-condition* of an operation is a condition which must be true at the time when the operation is ending its execution.
- *Guard* is a constraint which restricts the *Transition* in *UML state machine diagram*; it is evaluated and it must be true in the time when the transition fires.

2.1.3. OCL constraints example

This chapter does not specify in detail all possibilities of OCL. The language is used as the ground of our created language UCL which is defined in this thesis.

This subchapter shows only a short example of constraints in OCL in the context of the UML class diagram in Figure 2.1.

```
Context Person
inv: self.Age > 0
inv: self.Age < 18 implies self.hasCars->isEmpty()
```

Figure 2.2: Example of OCL constraints

The first line in Figure 2.2 defines the context of constraint expressions; constraints describe restrictions over the class *Person*. In the second line, there is a constraint expression which specifies the domain of the attribute *Age* of the class *Person*; its value cannot be negative. In the last line, there is a complex expression which defines the condition that instances of the class *Person* cannot be connected to any instance of the class *Car* through the association.

2.2. Relational model of databases

Relational model for databases [4] was invented by Edgar Frank Codd in IBM in 1969. The relation model is based on the first-order logic. The relation model provides a declarative way to specify data and queries over the database.

The Relational model uses concept of *relations* and *tables*. Tables in a database schema are represented by predicate variables; the data of tables and results of queries are represented by n-ary relations; constraints and queries are represented by predicates. A *tuple* is an ordered set of *attributes values*; an *attribute* is an ordered pair of its *name* and its *type*. A relation consists of a set of the attributes and of a set of n-ary tuples. A *table* is a graphical representation of the relation; a tuple is similar to the concept of table's rows. Tables contain definitions of *columns* which correspond to the relation's attributes.

If there are columns of the same type in different tables then these columns can create links between the tables. Other way to create links between tables is a *foreign key*. The foreign key is a reference to a key in other relation; the referencing table has the attribute with the value of the key in the referenced table.

Figure 2.3 shows an example of a graphic notation of tables in the relational model. The table *Cars* contains the column *PersonName* which is the foreign key to the table *Persons* on its key *Name*.

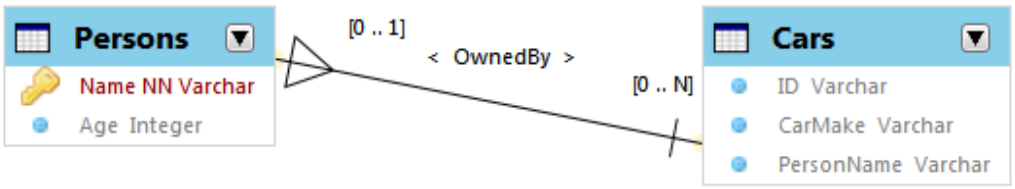


Figure 2.3: Example of the relation model

2.2.1. SQL Constraints

The usual way (to formal expression of integrity constraints over the relational model; at the implementation level of the design of database tables) is to use *SQL expressions* or *SQL triggers*. *Structured Query Language* [57] (SQL) provides standard opportunities to express different kinds of integrity constraints in relational databases. For the *referential integrity*, there are SQL statements `FOREIGN KEY` and `REFERENCES`, for the *entity integrity*, (e.g. a primary key cannot be null), for the *domain* and for the *user defined integrity* there are the statements `NOT NULL`, `UNIQUE` and `CHECK`.

```
CREATE TABLE Persons
(
  Name varchar(100) NOT NULL,
  Age int DEFAULT NULL,
  PRIMARY KEY (Name),
  CONSTRAINT AgeIsNotNegative_ck CHECK (Age > 0)
);
CREATE TABLE Cars
(
  ID varchar(100) NULL,
  CarMake varchar(100) NULL,
  PersonName varchar(100) NULL
  REFERENCES Persons(PersonName)
);
ALTER TABLE Persons
ADD CHECK PersonsUnder18WithoutCar
(
  NOT EXISTS
  (
    SELECT *
    FROM Persons
    JOIN Cars ON (Cars.PersonName = PersonName.Name)
    WHERE Persons.Age < 18
  )
);
```

Figure 2.4: Example of constraints in relational databases

Figure 2.4 demonstrates SQL expressions to create the tables *Persons* and *Cars*. By the definition of the table *Persons*, there is constraint `AgeIsNotNegative_ck` to set that *Age* value cannot be negative. There is also added `CHECK` constraint `PersonsUnder18WithoutCar` to the table *Persons* which checks that a person younger than 18 cannot have any car.

2.3. XML technologies

2.3.1. XML documents

XML (*Extensible Markup Language*) [5] is a markup language designed to store and carry structured data. It was designed by international standards organization *World Wide Web Consortium* (*W3C*). First XML was designated to store documents; now it is also used for representation of wide data structures in databases or Web services.

An example of an XML document is shown in Figure 2.5. An XML document mainly consists of *elements* and *attributes*. An element is build from two markup *tags* (the *beginning-tag* and the *ending-tag*) or from one *empty-element tag*. Other elements can be nested in an element. There is a root element which contains all other elements. Cause these rules elements of an XML document create a tree structure. An attribute is a markup construct consisting of a *name* and a *value* in quotes. Attributes are placed within a beginning-tag or within an empty-element tag.

```
<?xml version="1.0" encoding="utf-8"?>
<persons>
   <person>
      <name>George</name>
      <age>12</age>
   </person>
   <person>
      <name>Alice</name>
      <age>30</age>
      <cars>
         <car id="00123"><carMake>ABC123</carMake></car>
         <car id="00234"><carMake>ABC234</carMake></car>
      </cars>
   </person>
   <person>
      <name>Bob</name>
      <age>16</age>
      <cars>
         <car id="00789"><carMake>BCD789</carMake></car>
      </cars>
   </person>
</persons>
```

Figure 2.5: Example of an XML document

2.3.2. Levels of the correctness in XML

We name that an XML document is *well-formed* if it satisfies the common syntax rules introduced in the specification. Such rules are that a document contains only legal Unicode [58] characters, the beginning-tag and the end-tag of elements are equal (case-sensitive equal) and elements are correctly nested without interaction overlapping; and next. These rules are common for all possible XML documents.

Next definition is if an XML document is *valid* in respect of a specific *schema of an XML document*. Such a schema of an XML document defines the document's structure, content and integrity constraints. There are various languages to express

the schema of XML. They have different expressive power and they use different way to specify the schema of XML.

Document Type Definition (*DTD*) [5] defines the formal syntax of elements in XML documents, data types of elements and attributes and references between special attributes (with *ID* data type).

2.3.3. XML Schema

Language *XML Schema* (*XSD*, *XML Schema Definition*) [6] also defines the formal syntax like DTD but it is more expressive. It provides the greater support for data types, domain integrity constraints and the referential integrity between elements. Figure 2.6 shows an example of a schema of XML documents in language *XML Schema*. It is a definition of the schema of the XML document in Figure 2.5. It defines the syntax of a document and it defines the domain integrity constraints of element *age* in element *person*; its domain is a non-negative integer value. XML Schema has not the expressive power to define more complex integrity constraints.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="persons">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="person"
                    minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="age"
                          type="xs:nonNegativeInteger"/>
              <xs:element ref="cars" minOccurs="0"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

```
  <xs:element name="cars">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="car" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="carMake" type="xs:string"/>
            </xs:sequence>
            <xs:attribute name="id"
                          use="required" type="xs:string"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 2.6: Example of a schema in XML Schema

2.3.4. Schematron

Another XML schema language is *Schematron* [7]. It is based on rules about a presence or an absence of patterns in XML trees. Its expressive power is very strong. It can express complex integrity constraints of XML. It is able to define the syntax of an XML document using the rules of a presence of an XML element in other XML elements. But such a schema in Schematron is too verbose. In this way complex constraints in Schematron are often combined with other languages like *XML Schema*.

```
<?xml version="1.0" encoding="utf-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
  <pattern name="Persons constraints">
    <rule context="persons/person">
      <assert test="age &gt;= 0">Person's age can not be
negative.
      </assert>
      <report test="age &lt; 18 and cars/car">Person younger
than 18 cannot have assigned any car.
      </report>
    </rule>
  </pattern>
</schema>
```

Figure 2.7: Example of constraints in Schematron

Figure 2.7 demonstrates constraints in Schematron. A schema in Schematron is an XML document that consists of some elements *pattern* with inner

elements *rule*. Elements *rule* define the context of constraints. In elements *rule* are placed elements *assert* and *report* with constraints. Elements *assert* define conditions which must be satisfied; and elements *report* define conditions which cannot occur. Both elements *assert* and *report* contain a constraint expression in language *XPath* [8]. This expression is evaluated by the checking of validity of the XML document.

XPath (XML Path Language) is a query language used to selecting nodes of an XML document. It is based on the tree structure of XML documents. It provides the navigation and the selection queries over XML trees. It can be also used to compute values from XML. Using of XPath provides the expressive power of Schematron.

2.3.5. XQuery

XQuery [9] is a functional query language over XML data. It is used to extract data of XML documents or transform documents. XQuery uses expressions in XPath to specify the source data in XML and *FLWOR* expressions (`FOR`, `LET`, `WHERE`, `ORDER BY` and `RETURN`) in a syntax similar to *SQL* to processing of the data. This syntax induces that the expressive power of XQuery is similar to the power of SQL.

XQuery was designed for the creation of queries on XML but it can also be used for the definitions of integrity constraints. The integrity constraints are defined as queries that return if XML content contains only allowed data or if it contains any data restricted by the constraint. Figure 2.8 represents an integrity constraint expressed by XQuery. The *FLWOR* expression selects persons under 18 which have assigned any car; this is not allowed. If the result of this query is empty then the expression returns the content `<result>true</result>`. Else it returns `<result>false</result>` that indicates disallowed data by the constraint.

```
let $bad_persons :=
  for $person in fn:doc("persons-cars.xml")/persons/person
  where $person[age < 18 and cars/car]
  return $person
return <result>{(fn:empty($bad_persons))}</result>
```

Figure 2.8: Example of a constraint in XQuery

2.3.6. XSEM

XSEM [3] is a conceptual model for XML data. It is divided into the two layers; *platform-independent model* (PIM) and *platform-specific model* (PSM). PIM layer models real elements and relations between them without their concrete representation. It uses the standard elements of UML class diagrams: *classes*, *attributes* and *binary associations*. PSM layer models a concrete XML schema where PSM's model elements represent XML elements and XML attributes with concrete types. The PIM diagram is only one. Usually there are more PSM diagrams which represent different views on the same model. PSM's elements are semantic interconnected with PIM's elements.

For purposes of this thesis, we will use the XSEM PSM layer to model schemas of XML documents. XSEM diagram consist of *classes*, *attributes*, *associations* and *content models*. A *class* has a text *name*; a *class* contains *attributes*. An *attribute* has a *name*, a *data type*, cardinality and an *XML form*. The XML form of an attribute models its XML representation; there are two forms: a *simple element* (an XML element with simple content) or an *attribute* (an XML attribute of an XML element). An *association* is an oriented directed binary relation between two *classes* or *content models*; we call them the *parent* and the *child*. An association has two *association ends*, a *name* and *cardinality*. A *content model* element has a *content model type*. There are three possible types: a *set*, a *choice* and a *sequence*. A *class* which does not have any *parent element* connected through an *association* is called a *top class*. An XSEM PSM diagram can create graphs. In this graph *classes* and *content models* are vertices and *associations* are edges. The whole XSEM PSM diagram must represent a directed forest of graphs.

A *class* models an XML complex content element (a sequence of XML elements and attributes); an *attribute* models an XML attribute or an XML element of a simple content. An *association* models an XML complex content with the *association's name* (or if the *association's name* is empty then with *child class's name*) of complex type modeled by *association's child* and *association's cardinality*. A *content model* models a complex XML content which is a set, a choice or a sequence of XML elements; this complex XML content is modeled by *associations* connected to the *content model*.

In Figure 2.9, there is an example of an XSEM PSM diagram with the two top classes *purchase* and *persons*.

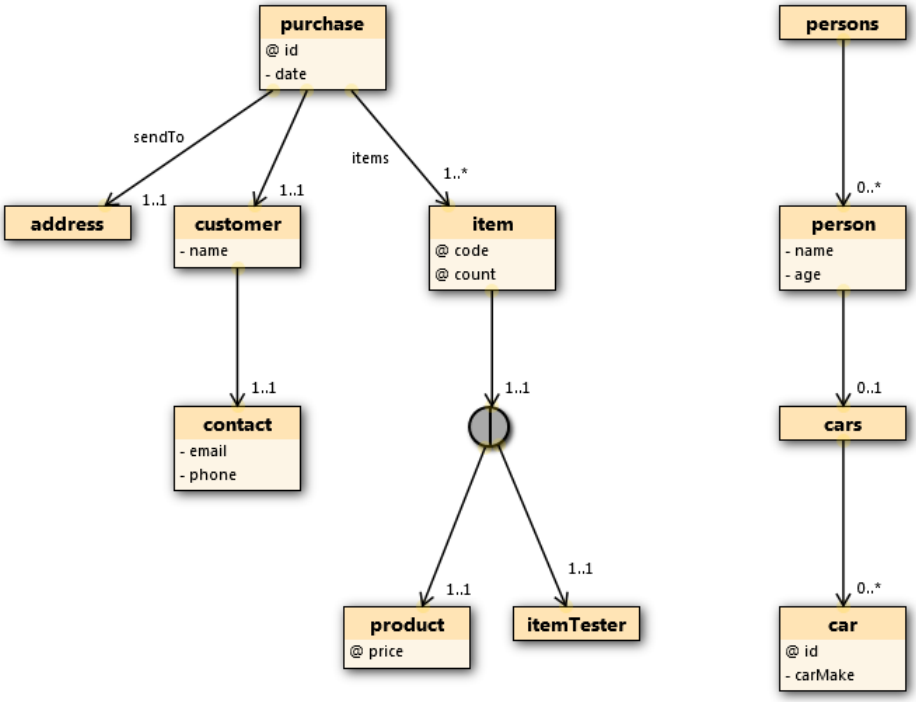


Figure 2.9: Example of an XSEM PSM diagram

3. Related work

Object Constraint Language (OCL) [2] was developed in 1995 at IBM. OCL is from version 1.1 (from 1997) a part of the official standard for UML. First, there was no significant support in existing case tools. Only several academic projects were created.

On the present, there are many modeling CASE tools [10] which support creating of data models, first of all in UML class diagrams. They can create and edit diagrams following the UML graphical notation. They implement features like the code generation or the reverse engineering (creating models from a source code) [11] [12] [13] [14].

Some of today's UML tools support the specification of OCL constraints on created models. However, most of them do not support the syntax and the semantic analysis of created OCL expressions over data models.

In this chapter, we study existing tools with the support for OCL. And we compare their usage and their features. We compare the tools according the implemented degree of the analysis of constraint expressions. The simplest tools enable only to write and save OCL expressions without any syntax analysis. In the next level, more tools implement the syntax analysis and type checking of OCL expressions. But they cannot control the semantics; OCL expressions are not interconnected with models. Some advanced tools implement also the derivation of OCL constraints to a source code of programming languages. Some tools implement the dynamic validation of constraints over objects which are modeled in the tool.

3.1. Tools

3.1.1. IBM OCL Parser

Probably, the first OCL available tool was IBM's *OCL Parser* [15]. It was developed by the OCL authors at IBM [12]. It was implemented by the automatically generation from OCL's grammar using *JavaCC* parser generator. Is functionality includes only a syntax checking and partial checking of the basic and collection types. Input models must be inserted in a special format. The application was not user friendly.

3.1.2. ModelRun

ModelRun [16] was one of the first commercial tools with the support of OCL [13]. It was creates by company *BoldSoft* to validate models. ModelRun enables the OCL navigation over created models.

In ModelRun, a user can define a model; set of classes and associations. Then he can create prototypes of the created model. He can create objects as instances of the classes and instances of the associations by drag-and-drop operations between the created objects. Then it is possible to execute OCL queries over the objects. And it is able to validate snapshots of the object model according expressed OCL invariants.

ModelRun supports the syntactic, semantic analysis, type checking over the basic types and the models. It also supports the dynamic validation of invariants.

3.1.3. OCTOPUS

Octopus (OCL Tool for Precise UML Specifications) [18] was developed by company *Klasse Objecten*. It is an Eclipse [19] plug-in. It enables the syntactic and semantic analysis of OCL over UML data models. Octopus implements the transformation of UML class diagrams including OCL constraints into Java code.

The component of the generating Java source code was initially developed as a study of *Model-driven architecture* (MDA) [20]. It demonstrates that it is possible to create a *platform-independent model* (including integrity constraints) and transform it to a *platform-specific model* and to *platform-dependent source code*.

3.1.4. USE

USE (UML-based Specification Environment) [12] [21] [22] is a system for the specification of information systems. It was developed at the *University of Bremen, Germany*. The current version USE 3.0.0 RC2 is from December 2010.

A specification of an information system in USE is a UML model. It contains textual descriptions of classes, attributes, associations, operations and constraints. This textual description is not a standard. System requirements are represented as OCL constraints. USE has a feature for animating snapshots of a system of objects. These animations should provide to architect a feedback to achieve better design before implementation starts.

USE supports the full semantic analysis of constraints. It supports also validation of OCL invariants, pre- and post-conditions.

3.1.5. Enterprise Architect

Enterprise Architect (EA) [23] from *Sparx Systems* is a popular complex commercial visual modeling case tool. It supports modeling of software systems, business processes or industry based domains. It supports many industry modeling standards. It also supports the reverse engineering over source code in many programming languages. It is able to integrate *EA* with *Eclipse* [19] or *Visual Studio* [25].

A few years ago, EA did not have any support for OCL. In 2010 it supported the syntactic and semantic analysis of OCL over created models [11]. But it cannot derive OCL constraints to programming languages or other constraint languages.

In the current version 9 Build 907 (June 2011), it is possible [24] to validate a UML element, relationship or attribute against an OCL constraint. Constraints are added individually for each element in a model by properties dialogs; the context of OCL constraints is defined implicitly by the property dialog.

3.1.6. Dresden OCL

Dresden OCL [26] is a software platform for OCL. It was developed at *Technische Universität in Dresden, Germany*. It is probably the most complex tool with the support for OCL. It provides the full evolution of OCL expressions. It is an open source platform and it is designed to extensible modularity. It enables to integrate it to existing UML tools in Java platform.

There are 3 major versions of Dresden OCL:

- *Dresden OCL Toolkit*, from 1999 which supports OCL 1.3;
- *Dresden OCL2 Toolkit*, from 2005 which supports OCL 2.0;
- *Dresden OCL for Eclipse*, which supports the current latest version which support OCL 2.2.

The version *Dresden OCL2 Toolkit* is based on *NetBeans Metadata Repository* (MRD). MDR is a standalone implementation of MOF. It consists of several libraries and it is a part of NetBeans [28].

The whole application of version *Dresden OCL for Eclipse* is based on Eclipse SDK. Its architecture is based on *Pivot model* [28] [32] (but it is a meta-

model; next *Pivot meta-model*). It is an exchange format for meta-models. It provides an abstraction for the evaluation of OCL expressions. It is possible to adapt different meta-models to Pivot meta-model.

This Pivot meta-model is based on *Eclipse Modeling Framework* [30] (EMF). EMF is a modeling framework for building tools and other applications which are based on structured data models. EMF provides a set of tools which generate a set of Java classes for the created model. EMF includes a meta-model *ECore* [31] for describing models. ECore is a meta-meta-model (M3 level). Elements of ECore are in Figure 3.1. ECore E.g. ECore allows defining elements:

- *EPackage* (contains *EPackages* and *EClasses*), *EClass* (contains *EAttributes*, *EReferences*, *EOperations*), *EAttribute* (has *EDataType*), *EReference* (a relation end), *EDataType*, *EEnum*, *EOperation*, *EParamater*.

ECore use the class *EClass* also for *UML associations* – elements which connect other elements. Pivot meta-model is based on ECore. It extends ECore with Java classes in the package *tudresden.ocl20.pivot.pivotmodel*. It is demonstrated in Figure 3.2. *Dresden OCL* use OCL expressions which are based [35] over Pivot meta-model.

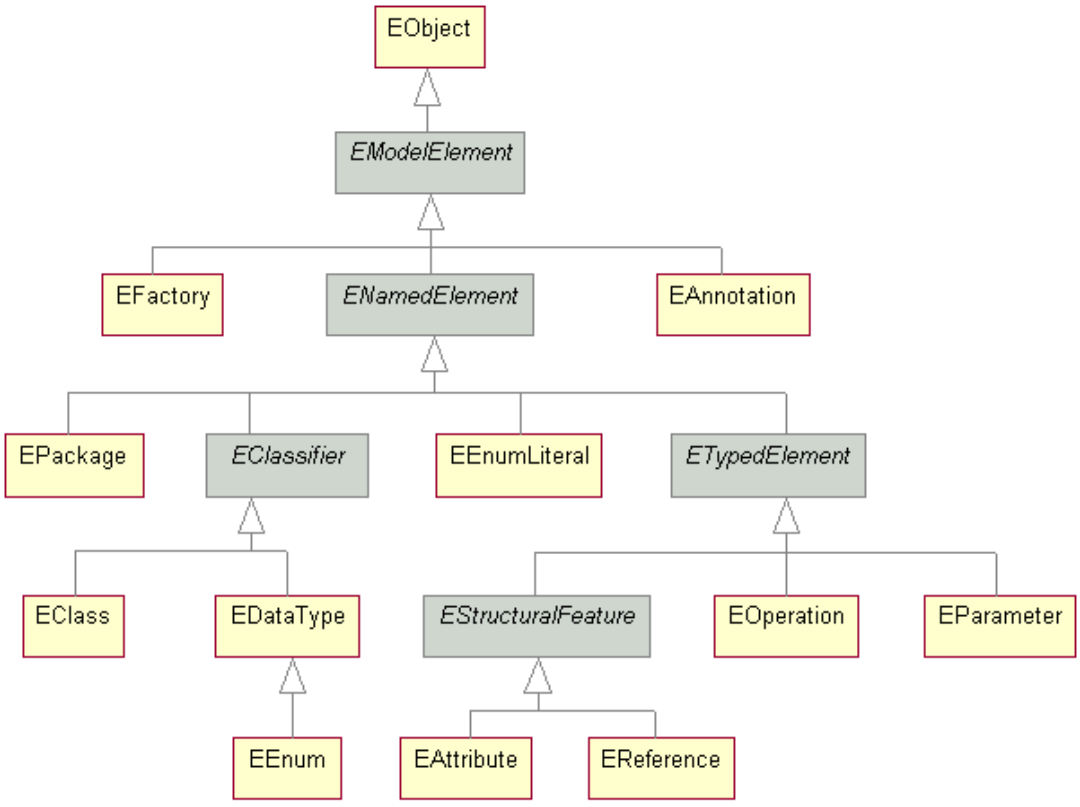


Figure 3.1: ECore meta-model

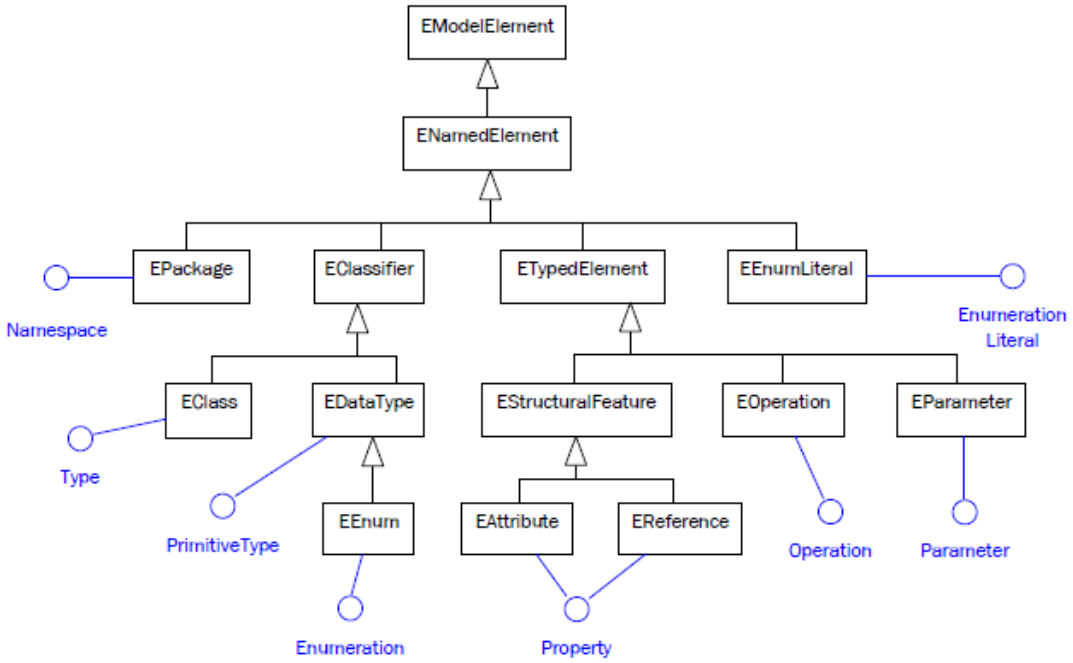


Figure 3.2: ECore meta-model adapted to Dresden Pivot meta-model

Other features of Dresden OCL are:

- It is possible to translate a UML class diagram model with OCL constraints to Java code [34];
- It is possible to translate OCL constraints over UML to SQL expressions;
- It is possible to express OCL constraints over a XML model which is adapted to model in Pivot meta-model.

Dresden OCL is a complex universal framework. It is able to use OCL over each meta-model which can be adapted to Pivot meta-model based on ECore. It is possible to translate OCL expressions to another language. But it is not possible to interconnect different meta-models and their models.

3.1.7. Eclipse OCL

Eclipse OCL Project [36] is an independent isolated component of Eclipse. It is a library; set of Java packages. It provides a parser and an interpreter for OCL constraints over any meta-model which is instance of ECore meta-meta-model. It defines API for parsing of OCL expressions. It defines *OCL abstract syntax model* as a meta-model of OCL expression. It implements [37] the *visitor* design pattern [38] for analyzing or transforming the model of OCL expressions.

3.1.8. Kent OCL

Kent Object Constraint Language Library [39] [40] (Kent OCL, KOCL) is an OCL implementation at *University of Kent at Canterbury, England*. It consists of an OCL parser, analyser and code generator. It defines *Bridge* [39] meta-model over various meta-models. It provides its libraries for *Kent Modeling Framework project* (KMF), EMF and for Java. Bridge meta-model is represented by Figure 3.3. Bridge is a more reduced meta-model than ECore.

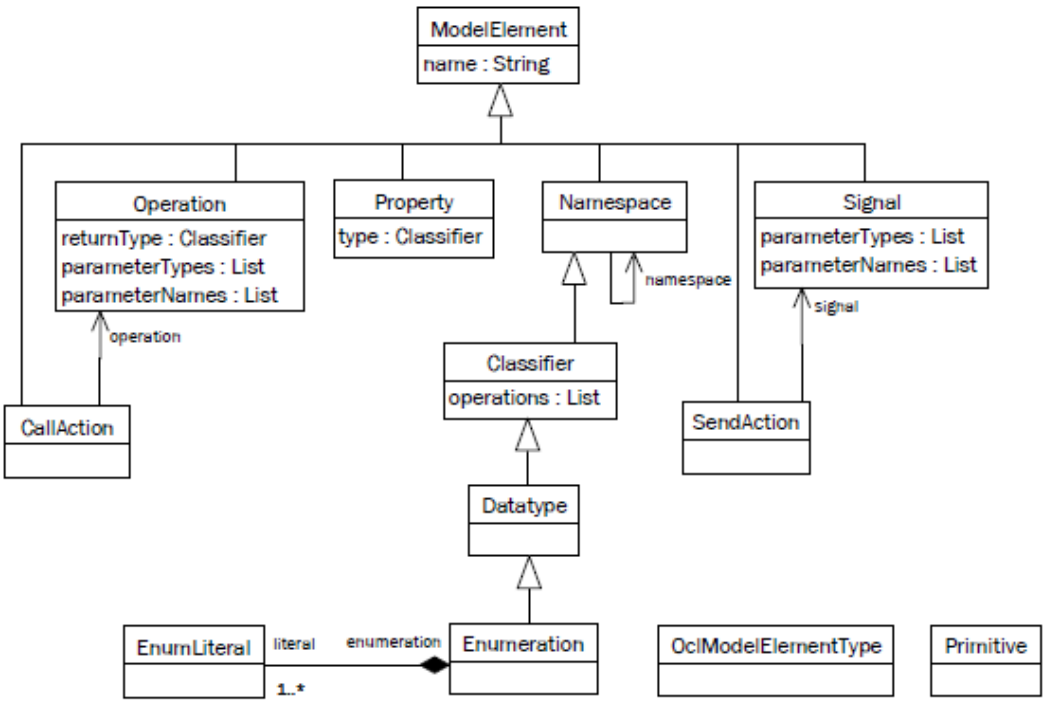


Figure 3.3: Kent Bridge meta-model

3.2. Deriving OCL to other constraint languages

There are also some studies which analyze derivation OCL constraints over UML models to other constraint languages for other meta-models, e.g. to relational databases [41] [11]. We have not found any study about derivation of OCL to XML constraint languages, Schematron or XQuery.

3.2.1. Demuth and Hussmann

In the paper “Using UML/OCL Constraints for Relational Database Design” [41] the mapping of OCL expressions to SQL code is presented. OCL is based on the first-order predicate logic. The result of the study is that almost all constructions of OCL (except iterate construct) can be derived to SQL queries.

Authors present mapping from construct of OCL to types of SQL queries according a family of patterns. Each pattern represents an idea of translating an OCL construct to an SQL query. Authors consider only the translation of OCL invariants to SQL *CHECK* constructs.

The study first analyse the mapping of a relational database schema from a UML model. UML classes are mapped to relational tables. Associations which represent the relationship many to many or if they are connected to an association class are derived to database tables, too. Associations one to many or one to one even if they are recursive associations are mapped to foreign keys.

Each OCL invariant in a context block:

```
Context className
inv invName: expression
```

is mapped to a SQL predicate where variable *self* represent the row of the table:

```
CREATE ASSERTION invName
CHECK (
  NOT EXIST (
    SELECT *
    FROM className AS self
    WHERE not (expression)
))
```

A pattern over basic column can not result in a collection:

```
[Context Person]
self.age > 18 and self.isUnemployed = true
```

is mapped to:

```
[from Person self]
self.Age > 18 and self.IsUnemployed = true
```

An OCL sub-expression which refers to an association:

```
[Context Company]
self.employee
```

is mapped to a SQL query:

```
[from Company]
SELCT ID FROM Person
WHERE PID in (
  SELECT PID FROM Job
  WHERE CID in (
    SELECT CID FROM Company
    WHERE CID = self.CID))
```

The study map all OCL collection expressions and collection operations to SQL queries.

3.3. Conclusion

Many modeling CASE tools support creating and drawing models. Some of them enable the code generation and the reverse engineering. However, the support for integrity constraints in OCL is rare and infrequent. The most complex tool for OCL is *Dresden OCL*. It enables use OCL for different meta-model by mapping to Pivot meta-model. And it enables derive OCL to other constraint languages (SQL and Java code).

The aim of our thesis is to express constraints in our language for different meta-models, too. As well, we want to support mapping between different meta-models. And we plan to derive constraints between different interconnected meta-models. We will use a constraint language based on OCL. It will be based on a different meta-model then ECore or Dresden Pivot meta-model. The meta-model will be not so common. But it will provide more possibilities for relationships between model elements; and it will provide more possibilities for navigation expressions for the created constraint language. But not all language constructs, structures and libraries of OCL will be used and implemented in our language and in our framework.

4. Architecture, UCL Data meta-model

The task of this thesis is to introduce and define UCL. It is a constraint language which will be usable to express integrity constraints for different data meta-models. Expressions and types of UCL must be based on a specific meta-model. We will name it *UCL Data meta-model*. E.g. OCL is based on the meta-model of UML; Schematron is based on a meta-model of XML. The aim is to use constraints in language UCL for various meta-model. For two different interconnected models of different meta-models, it must be able to interconnect also the two mapped models of UCL meta-model, too. Our aim is also to derive constraints over different meta-models. UCL Data meta-model should be consisted of the widest possible types of data constructs and relationships between these constructs.

In this chapter, we introduce and also formally define UCL Data meta-model. But first we explain the terms modeling, metamodeling and four-layer meta-modeling architecture.

4.1. Meta-Object Facility

4.1.1. Modeling

In general, a *model* is anything that is used in any way to describe anything else. It is a way to capture ideas, relationships, decisions and requirements in a defined notation that can be used for different domains. Modeling allows a better understanding of the system. Data model in software engineering is a model which represents the structure of data. It enables storing and exchanging of data.

4.1.2. Meta-modeling

The model is a description of the system or part thereof expressed in a defined language or in a graphical representation. The mechanism of a creation and of a definition of such a modeling system is called a *meta-modeling*.

It is important to distinguish between the concepts of the *model* and the *meta-model*. A model defines what elements and their properties may exist in the system; a model is an abstraction of the real world. A meta-model is an abstraction at the higher layer. A meta-model is the result of a process of the abstraction of the

model (or of the modeling language). A meta-model is methodologically a model of the model; it defines types of the elements of the model with type of their properties.

Figure 4.1 shows the relationship between the model and the meta-model; it consists of two layers. The layer *Model* describes in UML class diagrams persons, cars and relationships between them. Further it models for persons their name and age; and for cars their ID and their mark. UML class diagrams created by software designers exist in the layer *Model*. The Layer *Meta-model* is an abstraction of the layer *Model*. It describes (models) all elements, properties of elements and relationships between elements in the layer *Model*. The meta-model that is used in the figure defines the element *UML class*, *UML association* and *UML attribute*. The element *UML Class* in the meta-model is the *meta-class*; the elements *Person* and *Car* in the model are its instances. Similarly, the attributes *Name*, *Age*, *ID* and *CarMake* are instances of the meta-class *UML Attribute* and the association in the model is an instance of the meta-class *UML Association*.

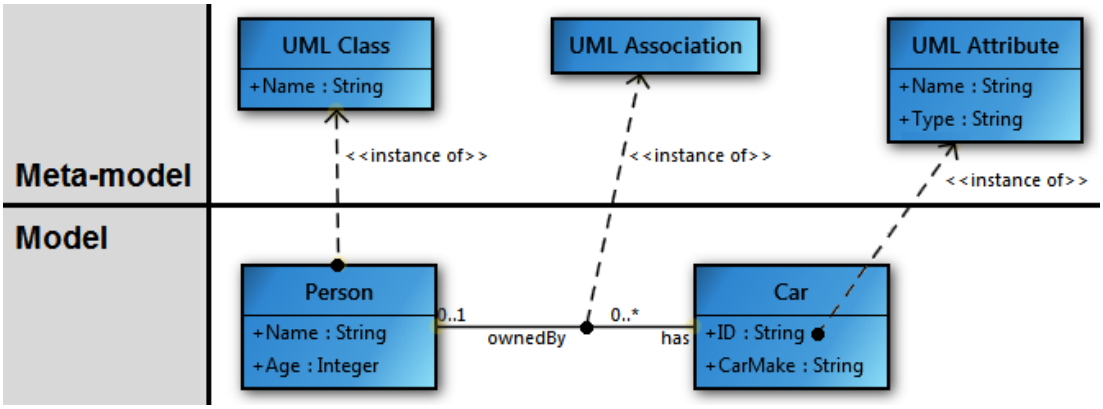


Figure 4.1: Relation between the model and the meta-model

4.1.3. Four-layer architecture

Object Management Group defines the four-layer meta-modeling architecture *Meta-Object Facility* (MOF) [42] to define UML. The constituent layers are *instances* (M0), *user model of the system* (M1), *meta-model* (M2) and *meta-meta-model* (M3).

In the layer M0, there are real instances of the modeled system. For example, for an object-oriented information system there are instances of classes (e.g. instances of class *Person*: *Alice* or *Bob*) in the case of relational databases there are rows of the corresponding tables and in the case of an XML model there are the elements with their content. The layer M0 is an instance of the layer M1 (model) which is located on one layer upper. By a modeling of the data in the layer

M0, there are computer representations of real entities; such as representations of people or cars. By a modeling of processes, there are the real elements; it means real persons, real objects and real processes.

The layer M1 (model) represents models – diagrams that model rules of the modeled domain. It is for example a schema of XML documents or a UML diagram with concrete classes (*Person*, *Car*…), attributes and associations. Elements in this layer are the classifiers for the in the layer M0.

The layer M2 (meta-model) defines constituent languages for specifying models. For example, in the case of UML class diagrams, the layer defines what a class, an attribute, an association is; and what their properties and rules are. In the case of relational databases, the layer defines what a table, a column and a foreign key is. Elements in this layer are the classifiers for the elements in the layer M1.

Each element in the layer M2 (meta-model) is an instance of an element in the layer M3 (meta-meta-model). Layer M3 defines concepts that can be used in defining of modeling languages. This level of abstraction supports the creation of different models from the same set of basic concepts.

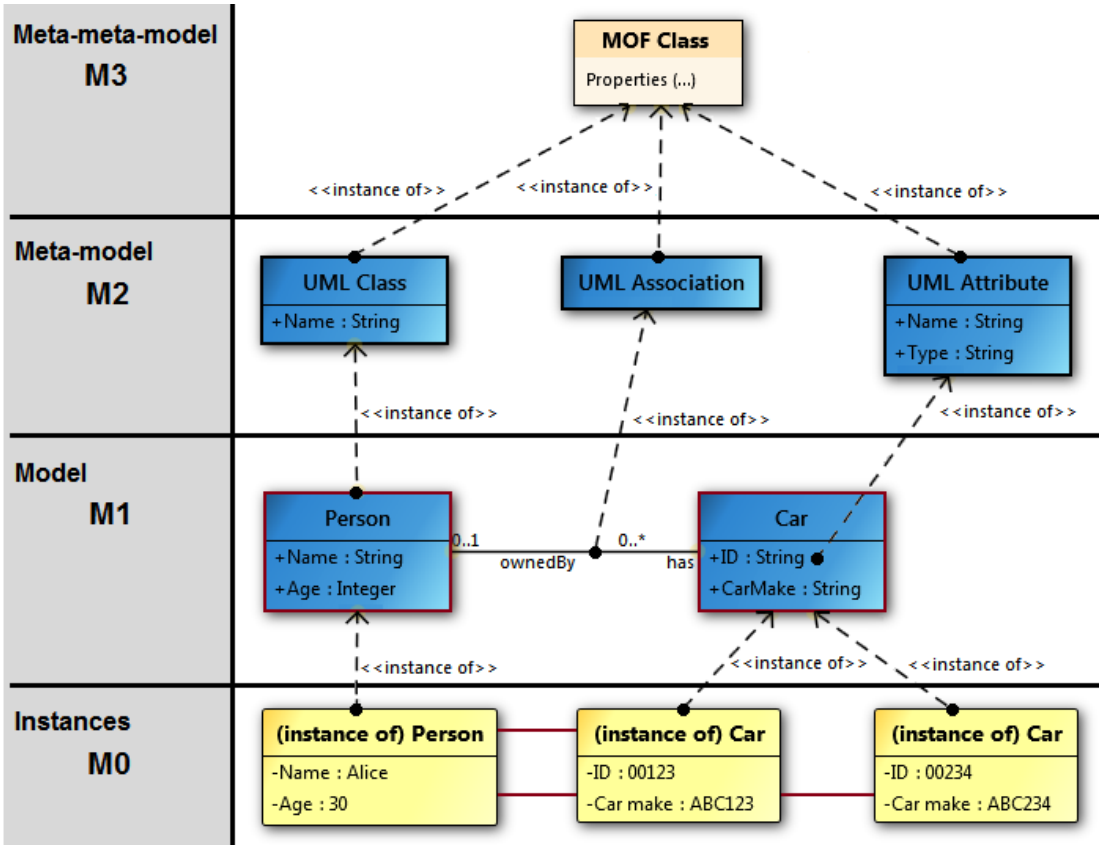


Figure 4.2: Example of the four-layer meta-modeling architecture

Figure 4.2 illustrates an example of the four-layer meta-modeling architecture for UML class diagram inclusive the layer M0 with one instance of class *Person* and two instances of class *Car*. As an illustration, there is one class *MOFClass* in the layer M3. It is the classifier for all elements in the layer M2 which describe the UML meta-model.

MOF is a standard for *Model-driven engineering* (MDE). MDE is a software development methodology to creating and exploiting domain models. MOF describes creating and manipulating of meta-models by definitions interfaces that describe those operations. An example of a supporting standard of MOF is *XML Metadata Interchange* (XMI) [59] which defines an exchanging format for models in the MOF layers.

4.1.4. Layer of UCL Data meta-model

UCL Data meta-model must be defined in one of the MOF layers. Various data meta-models (e.g. meta-models for UML class diagrams, for relational databases and for schemas of XML) are placed in the layer M2. For an integration of UCL Data meta-model there are two possibilities. The first is to place it at the same level and use the layer M3 only for a definition of a meta-meta-model. Then all elements in the layer M1 of different meta-models will be mapped to elements in M1 of UCL Data meta-model. Else we can lift UCL Data meta-model to the layer M3 to direct applying to different meta-models. Then all elements in the layer M2 of different meta-models will be mapped to elements in M2 of UCL Data meta-model. There are no differences in the meta-model of UCL expressions accruing from these two possibilities for the placing in the layers. Since we want to integrate UCL for existing different data models, we place the UCL Data meta-model into the layer M3.

4.2. Architecture

Figure 4.3 represent the architecture of a typically modeling software system. A software architect wants to model a complex software system. He creates models in the level M1. First he models the whole system in UML class diagrams in the PIM level. UML class diagram models also the runtime application. Then he creates some XSEM diagrams which model XML documents, messages or databases in M0 level (grey arrows). And the architect can create diagrams in ER models which model storing of data in a database.

PIM diagram represents the whole system. PSM diagrams represent individual parts of the complex system. Elements from PSM diagrams are semantically related with elements in PIM diagram. The architect can create mapping between elements from the PIM diagram to elements in PSM diagrams (the interrupted red arrows).

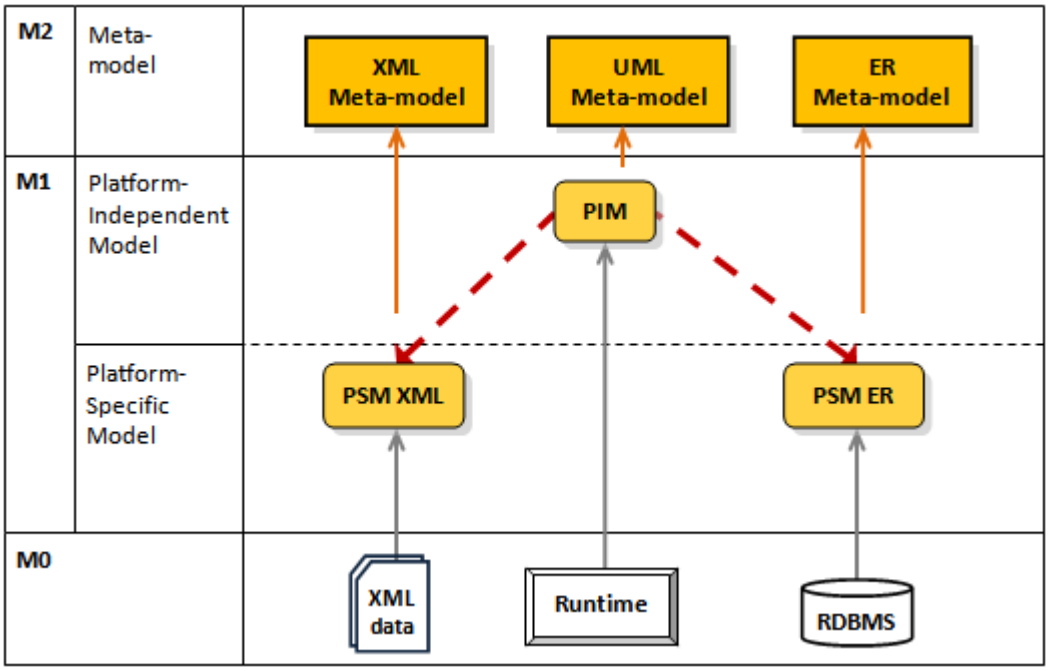


Figure 4.3: Architecture of a modeling framework with PIM, PSM XML and PSM ER

A modelling software system must define meta-models. E.g. to achieve it is able to create an UML class diagram in the software system, the system must define the UML meta-model. Created UML / XML / ER diagrams are instances of UML meta-model / XML meta-model / ER meta-model (the orange arrows).

Figure 4.4 introduces the architecture of a system with the support for UCL. In the layer M3, we have defined UCL Data meta-model, the common data meta-model for purposes of UCL.

In the layer M2, there are also UCL meta-models. For each meta-model, we must define concrete UCL Data meta-model. E.g. we have UML meta-model; it defines what a class or an attribute is. Then in the layer M2, we must define UML-UCL-meta-model; it defines what is a class or an attribute in a view of UCL Data meta-model.

In the layer M1, we must for each model create its UCL model. E.g. for an UML model (the orange rectangle in M1) we must create a UML UCL model (the blue rectangle in M1). The creation (the big blue arrows) of a UML UCL model from

an UML model is automatic; it is defined by rules in the UML-UCL-meta-model (in M2). UCL models (in M1) are instances of UCL meta-models (in M2) (the small blue arrows).

UCL constraints are based on individual UCL models (in M1). Let we have an UML model and we want to express UCL constraints over it. We do not express these constraints over the UML model but over the created UML UCL model of this UML model. UCL constraints (the green rectangles) are based over the UML UCL model (the blue line).

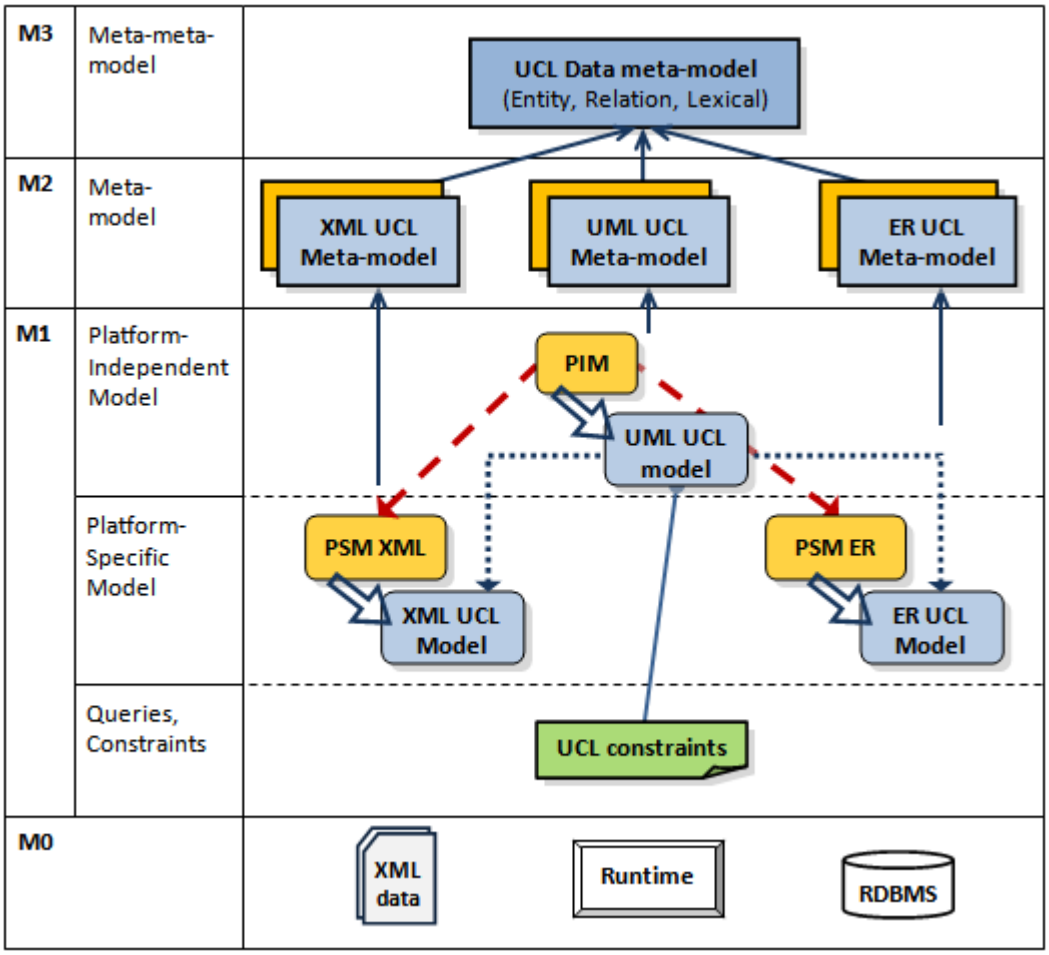


Figure 4.4: Architecture of a framework with UCL data models

Figure 4.5 illustrates the derivation of UCL constraints over one model to UCL constraints over another model and derivation of UCL constraints to other constraint languages (the big blue arrows).

Individual models (in M1) represent parts of the complex software system. These models are semantically related. A software architect must create the

mapping (interconnections) between elements in the individual models (the interrupted red arrows).

The individual UCL models are interconnected, too (the blue lines). It is possible to deduce the mapping between the individual UCL models (the blue rectangles in M1) for the mapping between the models.

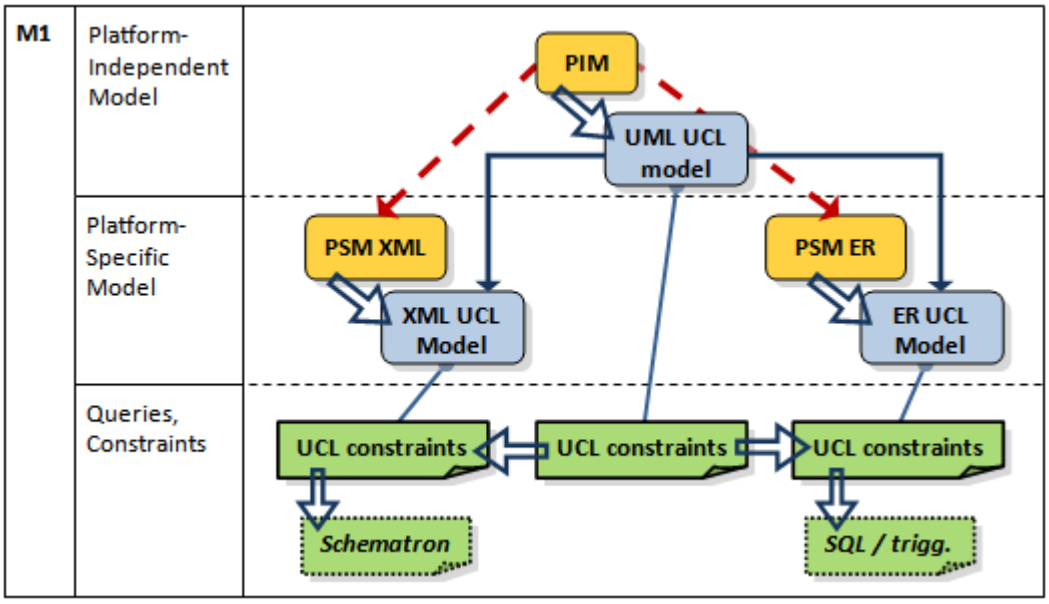


Figure 4.5: Architecture of deriving UCL to other constraint languages

If we have expressed UCL constraints over one model (e.g. over UML) then we can automatically derive the UCL constraints over another models (the big blue arrows). It is also possible to derive UCL constraints to specific constraint languages; e.g. derive UCL constraints over XML to Schematron rules.

4.3. Analysis and elements in UCL Data meta-model

UCL Data meta-model must be sufficiently general to be able to model many different data models (meta-models). There are two ways how to proceed with the analysis of what and how many elements will contain UCL Data meta-model. The first option is that it will contain only a small count of elements that will be universal but it will define different kinds of relationships and connections between these elements. The second option is that UCL Data meta-model will include a wider count of elements which will be more concrete and they will fit easily specific elements in some selected modeling data models. The aim of this thesis is to create a framework which also enables the derivation of constraints (in our language)

between different meta-models. To define this derivation, we must first define a mapping between two models. The derivation between two models of meta-models will be based on this mapping. The mapping will map elements of one model (of UCL Data meta-model) to elements of another model (of UCL Data meta-model). Different meta-models can define different relationships between their elements. But we need map different kinds of elements to a general kind of element; and we need map different kinds of relationship to general kinds of relationships; because we must define the mapping between different meta-models. We have chosen the first option to achieve UCL Data meta-model more general, thinner and easier to formalize its concept and the mapping between different meta-models.

Data meta-models typically contain simple elements that directly model the data values; e.g. a class's attribute in UML, a column in relational databases, an attribute or a simple element in XML. These kinds of elements from M2 will be represented in M3 by the element *UCL lexical*. A UCL lexical will contain the properties *Name* (string) and *Type* (enumeration from the basic data types *Integer*, *Real*, *Boolean* and *String*).

Data meta-models typically contain complex elements that contain simple elements (represented by UCL lexicals); e.g. a UML class, an XML complex element or a database table. These kinds of elements from M2 will be represented in M3 by the element *UCL entity*. A UCL entity has the property *Name*. A UCL entity can contain inner UCL lexicals. Some data meta-models contain elements to group other elements; e.g. a package in UML. These elements will be also represented by a UCL entity. In some data meta-models complex elements can contain other complex elements. So UCL entity can contain its inner UCL entities. UCL Data meta-model defines next relationships between UCL entities. It defines the neighborhood on an equivalent level and the generalization. The neighborhood between UCL entities can represent e.g. the relationship between UML classes in the same UML package.

Many data meta-models define binary and n-ary links between their elements; e.g. a UML association or a foreign key in databases. These links are in UCL Data meta-model represented by the element *UCL relation*. A UCL relation is an n-ary link between UCL entities. A UCL relation has the property *Name* and it has two or more *UCL relation ends*. A relation end is connected to an UCL entity and it is adorned with the properties *Name* and *Cardinality*.

A UCL entity can contain other inner UCL entities. Then a UCL entity can contain also inner UCL relations; e.g. when an UML association is inserted in a UML

package. IN UCL Data meta-model, there is also the neighbouring between a UCL entity and a UCL relation; e.g. when a UML association class is connected to a UML association.

We consider the defined links (UCL relations) and the individual relationships between UCL entities so general that we do not define additional meta-model elements. All elements of data meta-models and relationships between them must be mapped to the presented elements in UCL Data meta-model and their relationships.

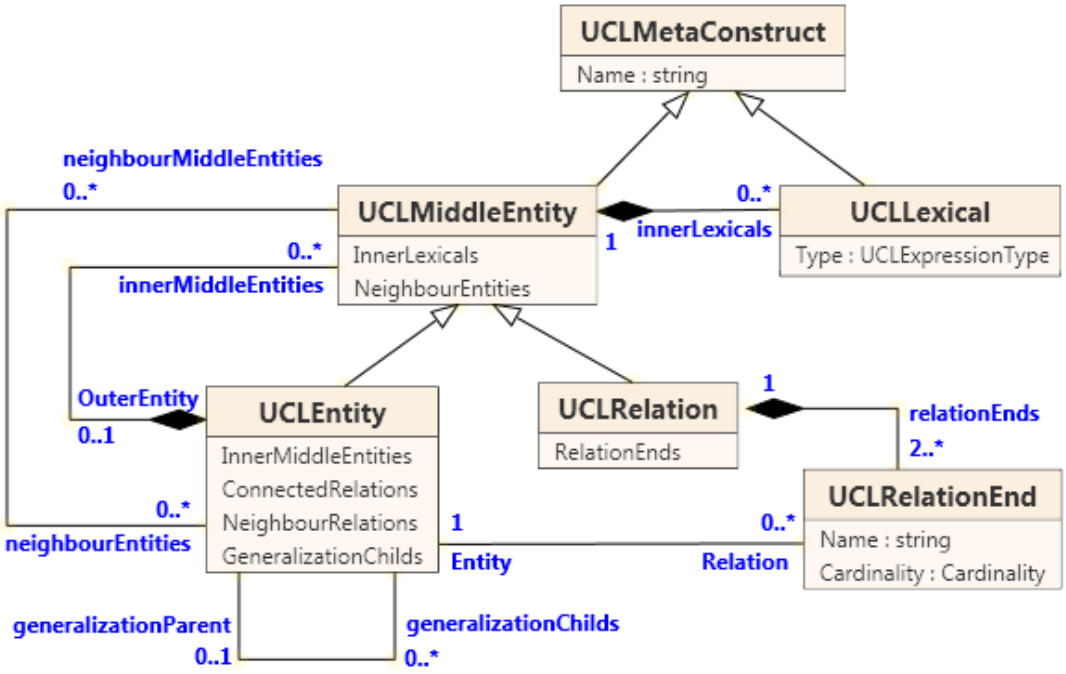


Figure 4.6: Class diagram of UCL Data meta-model

4.4. UCL Data meta-model description

Figure 4.6 shows the class diagram of UCL Data meta-model elements. This meta-model contains the nonabstract elements: *UCL entity*, *UCL relation*, *UCL lexical* and *UCL relation end* (which belongs to an instance of *UCL relation*). Concerning the common properties and the relationships between the elements there are defined abstract classes *UCLMetaConstruct* and *UCLMiddleEntity*. *UCLMetaConstruct* is the base class for the other (three) elements (entity, relation and lexical). It has the attribute *Name*, because all elements have a name. The class *UCLLexical* contains the attribute *Type*, and the inherited *Name*. *UCLMiddleEntity* is the base class for the classes *UCLEntity* and *UCLRelation*; because of their common characteristics. The both elements may contain

UCLLexicals (the role *InnerLexicals*), they both can be inserted in another *UCLEntity* (the relationship *OuterEntity*) and they both can neighbour with other *UCLEntity* (the relationship *NeighbourEntities*).

The class *UCLEntity* can contain *UCLEntities* and *UCLRelations* (the relationship *InnerMiddleEntities*). Then it can be connected to *UCLRelations* (the relation *ConnectedRelations*); it can neighbour with *UCLRelations* (the relation *NeighbourRelations*); it can be the child *UCL entity* by the generalization from one *UCLEntity* and it can be the generalization base *UCL entity* for mote other *UCLEntities*. The class *UCLEntity* inherits from the class *UCLMiddleEntity*, so inherits the attribute *Name*, it can contain *UCLLexicals* and it can neighbour with other *UCLEntities*, too.

UCLRelation is a n-ary link between *UCLEntities*; it contains collection of *UCLRelationEnds*. A *UCLRelationEnd* represents a connection of the *UCLRelation* to a *UCLEntity*; it is adorned with the *Name* and the *Cardinality* of the connection. The class *UCLRelation* inherits from the class *UCLMiddleEntity*, so it has the *Name*, it can contain *UCLLexicals* and it can neighbour with other *UCLEntities*, too.

4.5. Concept definition

In this sub-chapter, the introduced UCL Data meta-model is formally defined. The presented formal definition gives the context for UCL expressions and it is also required for the later formal definition of the meta-model of UCL expressions.

UCL Data meta-model has these components: the set of all UCL entities with their properties, inner elements, relationships and the generalization hierarchy, the set of UCL entities that has no outer UCL entity; the set of all UCL lexicals with their properties; the set of all UCL relations with their properties, inner elements and relationships; and the set of UCL relations that has no outer Entity.

In the following sections, individual components are defined in detail. Then they will be combined to build the complete definition of UCL Data meta-model.

Definition 4.1: For the naming of particular elements, we need assume an alphabet. Let Σ be the alphabet of letters, then $\mathcal{L} = \Sigma +$ is the set of non-empty finite words for the naming of elements. And let set $\Sigma *$ is the set of all finite words inclusive λ (the empty word).

Definition 4.2 (Cardinality):

The set $\mathcal{C} = \{(min, max) \in \mathbb{N}_0 \times (\mathbb{N} \cup \{*\}) \mid min \leq max \vee max = *\}$ is the set of all cardinalities. A cardinality is a pair $(min, max) \in \mathcal{C}$. The values min and max determine the possible count of instances covered by the cardinality.

4.5.1. Basic types

Data types will be defined in depth later in the meta-model of UCL. But now, we must define the set of the predefined basic types *Integer*, *Real*, *Boolean* and *String*.

Definition 4.3 (Basic types): Set $\mathcal{T}_B = \{Integer, Real, Boolean, String\}$ is the set of the predefined basic types.

4.5.2. Elements

Definition 4.4 (Elements):

- E is the finite set of all entities (*UCLEntities*).
- R is the finite set of all relations (*UCLRelations*).
- L is the finite set of all lexicals (*UCLLexicals*).
- Re is the finite set of all relation ends (*UCLRelationEnds*) of all relations.

4.5.3. Entities

Definition 4.5 (Entities): A structure of entities is the tuple *Entities*:

$$Entities = \\ \left(E, E.name, E.lex, E.inEn, E.inRel, E.conRel, E.accEnds, E.neEn, E.neRel, \prec_{E.gen}\right),$$

- $E.name : E \longrightarrow \mathcal{L}$ assigns a name to each entity.
- $E.lex : E \longrightarrow \mathcal{P}(L)$ assigns a set of lexicals to each entity.
- $E.inEn : E \longrightarrow \mathcal{P}(E)$ assigns a set of inner entities to each entity.
- $E.inRel : E \longrightarrow \mathcal{P}(R)$ assigns a set of inner relations to each entity.
- $E.conRel : E \longrightarrow \mathcal{P}(R)$ assigns to each entity a set of relations which are connected to the entity.
- $E.accEnds : E \longrightarrow \mathcal{P}(Re)$ assigns to each entity a set of relations ends which are accessible (by a navigation) from the entity.
- $E.neEn : E \longrightarrow \mathcal{P}(E)$ assigns a set of neighbouring entities to each entity.
- $E.neRel : E \longrightarrow \mathcal{P}(R)$ assigns a set of neighbouring relations to each entity.
- The partial order $\prec_{E.gen} \subseteq E \times E$ is a generalization hierarchy on entities.

4.5.4. Relations

Relations describe structural relationships between entities. A relation can connect two or more entities. Relations connect entities through their relations ends. The count *n* of all relation ends of the relation is called the degree of the relation. According the degree value, the relation is called the *n*-ary relation. An entitiy can be connected to any count of relations. Binary associations where both ends are connected to the same entity are allowed; they are called the recursive or the self-relations.

Definition 4.6 (Relations): A structure of relations is the tuple *Relations* and a structure of relation ends is the tuple *RelEnds*:

$Relations = (R, R.name, R.lex, R.neEn, R.ends)$,

$RelEnds = (Re, Re.relation, Re.name, Re.card, Re.entity)$, where:

- $R.name : R \longrightarrow (\mathcal{L} \cup \lambda)$ assigns a name to each relation, the empty name is allowed.
- $R.lex : R \longrightarrow \mathcal{P}(L)$ assigns a set of lexicals to each relation.
- $R.neEn : R \longrightarrow \mathcal{P}(E)$ assigns a set of neighbouring entities to each relation.
- $R.ends : R \longrightarrow \mathcal{P}(Re)$ assigns a set of relation ends to each relation. Each relation has at least two relation ends. $\forall r \in R;\ |R.ends\ (r)| \geq 2$.
- $Re.name : Re \longrightarrow \mathcal{L}$ assigns a role name to each relation end.
- $Re.relation : Re \longrightarrow R$ assigns to each relation end its (outer) relation.
- $Re.entity : Re \longrightarrow E$ assigns a connected entity to each relation end.
- $Re.card : Re \longrightarrow \mathcal{C}$ assigns to each relation end the cardinality of the connection to the connected entity.

4.5.5. Lexicals

Lexicals describe properties of entities and relations. A lexical has a name and one of the basic types.

Definition 4.7 (Lexicals):
A structure of lexicals is the tuple: $Lexicals = (L, L.name, L.type)$, where:

- $L.name : L \longrightarrow \mathcal{L}$ assigns a name to each lexical.
- $L.type : L \longrightarrow \mathcal{T}_B$ assigns a basic type to each lexical.

4.5.6. Generalization

We define the generalization hierarchy over entities. The generalization is a partial order (a binary relation which is reflexive, transitive and antisymmetric) on the set of the entities E. We denoted it $\prec_{E.gen} \subseteq E \times E$. The generalization is a taxonomic relation between two entities. If entities $e_1, e_2 \in E$ are in the generalization relationship $e_1 \prec_{E.gen} e_2$ then e_1 is the child entity of the parent entity e_2.

A child entity inherits all lexicals, all inner entities, all neighbour entities and relations and all connected relations of the parent entity. We define an auxiliary function *parents* which returns all generalization parent entities of an entity.

- $parents : E \longrightarrow \mathcal{P}(E)$
- $parents(e) = \{f \in E | e \prec_{E.gen} f\}$

  Relation $\prec_{E.gen}$ is reflexive: $(\forall e \in E)$: $\left(e \prec_{E.gen} e\right) \wedge \left(e \in parents(e)\right)$.

4.5.7. Lexicals restrictions

Each lexical must be inserted into exactly one entity or a relation as the inner lexical: $(\forall l \in L)$: $|\{e \in E | l \in E.lex(e)\} \cup \{r \in R | l \in R.lex(r)\}| = 1$.

4.5.8. Relations restrictions

Function *Re.relation* assigns to each relation end its relation.

- $(\forall r \in R)(\forall a \in R.ends)$: $Re.relation(a) = r$

The role names of relation ends (*a*, *b*) for a relation (*r*) must be distinct. It is necessary because we use the role names to the navigation from UCL relations to the connected UCL entities. These names must be distinct.

- $(\forall r \in R)\left(\forall a, b \in R.ends(r)\right)$: $\left(Re.name(a) = Re.name(b)\right) \Rightarrow (a = b)$

All lexicals in a relation must have distinct names. We use the names of inner lexicals for UCL navigation expressions from the UCL relation to the inner UCL lexicals.

- $(\forall r \in R)\left(\forall l, m \in R.lex(r)\right)$: $\left(L.name(l) = L.name(m)\right) \Rightarrow (l = m)$

A relation can be inserted into an entity as the inner relation. It can be inserted into at most one entity or in no entity: $(\forall r \in R)$: $|\{e \in E | r \in E.inRel(e)\}| \leq 1$. Relations which are not inserted in any entity are called *root relations*:

- $R.root \subseteq R$

- $R.root = \{r \in R | (\forall e \in E): r \notin E.inRel(e)\}$

Root relations must have distinct names. We need the names of root relations to express the context of UCL expressions.

- $(\forall r, s \in R.root): \big(R.name(r) = R.name(s)\big) \Rightarrow (r = s)$

The binary relationship neighbourhood between an entity (*e*) and a relation (*r*) is symmetric.

- $(\forall e \in E)(\forall r \in R): \big(r \in E.neRel(e)\big) \Leftrightarrow \big(e \in R.neEn(r)\big)$

All neighbour entities must have distinct names. We need the names of the neighbour entities of a relation to UCL navigation expression which navigate from the relation the neighbour entities.

- $(\forall r \in R)\big(\forall e, f \in R.neEn(r)\big): \big(E.name(e) = E.name(f)\big) \Rightarrow (e = f)$

4.5.9. Entities restrictions

An entity can be inserted into other entity as the inner entity. It can be inserted into at most one entity or in no entity: $(\forall e \in E): |\{f \in E | e \in E.inEn(f)\}| \leq 1$. Entities which are not inserted in any other entity are called *root entities*:

- $E.root \subseteq E$
- $E.root = \{e \in E | (\forall f \in E): e \notin E.inEn(f)\}$

The root entities must have distinct names. We need the names of root entities to define the context of UCL constraints; they must be distinct.

- $(\forall e, f \in E.root): \big(E.name(e) = E.name(f)\big) \Rightarrow (e = f)$

All lexicals in an entity must have distinct names. We need the names of inner lexicals of an entity for UCL navigation expression from the entity to the inner lexicals.

- $(\forall e \in E)\ \big(\forall l, m \in E.lex(e)\big): \big(L.name(l) = L.name(m)\big)\ \Rightarrow\ (l = m)$

Entity can contain inner entities and inner relations; these inner entities and inner relations of an entity must have distinct names. We need names of inner elements (entities and relations) of an entity to navigation expressions and also for the definitions of the context of UCL expressions.

- $(\forall e \in E)\ \big(\forall f, g \in E.inEn(e)\big): \big(E.name(f) = E.name(g)\big) \Rightarrow (f = g)$
- $(\forall e \in E)\ \big(\forall r, s \in E.inRel(e)\big): \big(R.name(r) = R.name(s)\big) \Rightarrow (r = s)$

Function *E.conRel* gets relations (*r*) which are connected to the entity (e). These relations (*r*) must contain the relation end (*a*) which is connected to the entity (*e*).

- $(\forall e \in E)(\forall r \in R)\colon \big(r \in E.conRel(e)\big) \Leftrightarrow (\exists a \in R.ends(r)\colon Re.entity(a) = e)$

Function *E.accEnds* gets all relation ends which are accessible (reachable by a navigation expression) from the entity. Such relations ends are all relation ends (*a*) of all relations (*r*) where exists a relation end (*b*) that is connected to the source entity (*e*). Relation ends *a* and *b* cannot be the same.

- $(\forall e \in E)\colon E.accEnds(e) = \\ \cup_{r \in R.conRel(e)} \{a \in R.ends(r) \big| \big(\exists b \in R.ends(r)\big)\colon Re.entity(b) = e \wedge a \neq b\}$

Accessible relation ends from an entity must have distinct names. We need the names of the accessible relation ends from an entity to UCL *relation step* navigation *relation stop* navigation expressions.

- $(\forall e \in E)\ \big(\forall a, b \in E.accEnds(e)\big)\colon \big(Re.name(a) = Re.name(b)\big) \Rightarrow (a = b)$

The binary relationship neighbourhood between the entities (*e*, *f*) is symmetric.

- $(\forall e, f \in E)\colon \big(f \in E.neEn(e)\big) \Leftrightarrow \big(e \in E.neEn(f)\big)$

All neighbour entities must have distinct names. We need names of the neighbouring entities for the UCL navigation expressions from an entity to its neighbouring entities.

- $(\forall e \in E)\big(\forall f, g \in E.neEn(e)\big)\colon \big(E.name(f) = E.name(g)\big) \Rightarrow (f = g)$

The binary relationship neighbourhood between an entity (*e*) and a relation (*r*) is symmetric.

- $(\forall e \in E)(\forall r \in R)\colon \big(r \in E.neRel(e)\big) \Leftrightarrow \big(e \in R.neEn(r)\big)$

All neighbour relations must have distinct names. We need names of the neighbouring relation for the UCL navigation expressions from an entity to its neighbouring relations.

- $(\forall e \in E)\big(\forall r, s \in E.neRel(e)\big)\colon \big(R.name(r) = R.name(s)\big) \Rightarrow (r = s)$

4.5.10. Relations navigation names

Definition 4.8 (Simple steps from relations):

For relations, names of lexicals, names of neighbour entities and names of relation ends are used by UCL in the same notation (*navigation simple steps*). All names of

elements connected by this relationship from an entity must be distinct; because we use the names for the navigation.

- $ReNames_1 \stackrel{\text{def}}{=} Re.lexNames \stackrel{\text{def}}{=} \bigcup_{l \in R.lex}\{L.name(l)\}$ is set of lexical names.
- $ReNames_2 \stackrel{\text{def}}{=} Re.neEnNames \stackrel{\text{def}}{=} \bigcup_{e \in R.neEn}\{E.name(e)\}$ is set of neighbour entities names.
- $ReNames_3 \stackrel{\text{def}}{=} Re.endsNames \stackrel{\text{def}}{=} \bigcup_{a \in R.ends}\{Re.name(a)\}$ is set of relation ends names.

The collection of the sets $\{ReNames_i\}$ must be pairwise disjoint.

- $ReNames_i \cap ReNames_j = \emptyset$.

4.5.11. Entities navigation names

Definition 4.9 (Simple steps from entities):

For entities, names of lexicals, names of inner entities, names of inner relations, names of neighbouring entities and names of neighbouring relations are used by UCL in the same notation (*navigation simple steps*); all these names must be distinct.

- $EnNames_1(e) \stackrel{\text{def}}{=} En.lexNames(e) \stackrel{\text{def}}{=} \bigcup_{l \in E.lex(e)}\{L.name(l)\}$ is the set of lexical names.
- $EnNames_2(e) \stackrel{\text{def}}{=} En.inEnNames(e) \stackrel{\text{def}}{=} \bigcup_{f \in E.inEn(e)}\{E.name(f)\}$ is the set of inner entities names.
- $EnNames_3(e) \stackrel{\text{def}}{=} En.inRelNames(e) \stackrel{\text{def}}{=} \bigcup_{r \in E.inRel}\{R.name(r)\}$ is the set of inner relations names.
- $EnNames_4(e) \stackrel{\text{def}}{=} En.neEnNames(e) \stackrel{\text{def}}{=} \bigcup_{f \in E.neEn}\{E.name(f)\}$ is the set of neighbouring entities names.
- $EnNames_5(e) \stackrel{\text{def}}{=} En.neRelNames(e) \stackrel{\text{def}}{=} \bigcup_{f \in E.neRel}\{R.name(f)\}$ is the set of neighbouring relations names.

Collection of sets $\{EnNames_i(e)\}$ must be pairwise disjoint. All names of these elements must be distinct.

- $EnNames_i(e) \cap EnNames_j(e) = \emptyset$.

Definition 4.10 (Accessible relation steps from entities): Function $EnAccessNames(e)$ gets for an entity the set of the names of all relation ends which are accessible from the entity. The function $E.accEnds(e)$ gets all accessible relation ends for an entity. The names are used in UCL for navigation expressions *connected relations step* and *connected relations stop*.

- $EnAccessNames(e) \stackrel{\text{def}}{=} \bigcup_{a \in E.accEnds(e)}\{Re.name(a)\}$

4.5.12. Full entity descriptor

A child entity implicitly inherits all lexicals, inner entities, inner relations, neighbouring entities, neighbouring relations and all connected relations of its parent entity. Like in the previous subchapters, we define the full structure of entities and the sets of names of *simple steps* and *relation steps* from an entity together with its inherited properties.

Definition 4.11 (Full structure of entities):

$Entities^* =$
$(E, E.name, E.lex^*, E.inEn^*, E.inRel^*,$
$E.conRel^*, E.accEnds^*, E.neEn^*, E.neRel^*, \prec_{E.gen})$, where:

- $E.lex^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in parents(e)} E.lex(f)$ assigns the set of the lexicals; including all inherited lexicals.
- $E.inEn^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in parents(e)} E.inEn(f)$ assigns the set of the inner entities; including all inherited inner entities.
- $E.inRel^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in parents(e)} E.inRel(f)$ assigns the set of the inner relations; including all inherited inner relations.
- $E.conRel^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in parents(e)} E.conRel(f)$ assigns the set of the relations which are connected to the entity; including all inherited connected relations.
- $E.accEnds^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in parents(e)} E.accEnds(f)$ assigns to each entity a set of relations ends which are accessible (by a navigation) from the entity; including all inherited connected relations.
- $E.neEn^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in parents(e)} E.neEn(f)$ assigns the set of the neighbouring entities; including all inherited neighbouring entities.
- $E.neRel^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in parents(e)} E.neEn(f)$ assigns the set of the neighbouring relations; including all inherited neighbouring relations.

Given the definition of the full structure of entities, we define the following restrictions for the entities. The names of all elements which are accessible by a UCL navigation expression from an entity must by distinct; including the inherited elements:

Lexicals with the same name can be defined only in one entity.

- $(\forall e \in E)\big(\forall l, m \in E.lex^*(e)\big)\colon \big(L.name(l) = L.name(m)\big) \Rightarrow (l = m)$

Inner entities with the same name can be defined only in one entity.

- $(\forall e \in E)\big(\forall f, g \in E.inEn^*(e)\big)\colon \big(E.name(f) = E.name(g)\big) \Rightarrow (f = g)$

Inner relations with the same name can be defined only in one entity.

- $(\forall e \in E)\big(\forall r,s \in E.inRel^*(e)\big)\colon \big(R.name(r) = R.name(s)\big) \Rightarrow (r = s)$

Accessible relation ends with the same name can be defined only for one entity.

- $(\forall e \in E)\big(\forall a,b \in E.accEnds^*(e)\big)\colon \big(Re.name(a) = Re.name(b)\big) \Rightarrow (a = b)$

Neighbouring entities with the same name can be defined only in one entity.

- $(\forall e \in E)\big(\forall f,g \in E.neEn^*(e)\big)\colon \big(E.name(f) = E.name(g)\big) \Rightarrow (f = g)$

Neighbouring relations with the same name can be defined only in one entity.

- $(\forall e \in E)\big(\forall r,s \in E.neRel^*(e)\big)\colon \big(R.name(r) = R.name(s)\big) \Rightarrow (r = s)$

Definition 4.12 (Full simple steps from entities):

Names for all kinds of *simple step navigation* from an entity must be distinct including all inherited names.

- $EnNames_1^*(e) \stackrel{\text{def}}{=} En.lexNames^*(e) \stackrel{\text{def}}{=} \bigcup_{l \in E.lex^*(e)}\{L.name(l)\}$ is the full set of the lexical names.
- $EnNames_2^*(e) \stackrel{\text{def}}{=} En.inEnNames^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in E.inEn^*(e)}\{E.name(f)\}$ is the full set of the inner entities names.
- $EnNames_3^*(e) \stackrel{\text{def}}{=} En.inRelNames^*(e) \stackrel{\text{def}}{=} \bigcup_{r \in E.inRel^*}\{R.name(r)\}$ is the full set of the inner relations names.
- $EnNames_4^*(e) \stackrel{\text{def}}{=} En.neEnNames^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in E.neEn^*}\{E.name(f)\}$ is the full set of the neighbouring entities names.
- $EnNames_5^*(e) \stackrel{\text{def}}{=} En.neRelNames^*(e) \stackrel{\text{def}}{=} \bigcup_{f \in E.neRel^*}\{R.name(f)\}$ is the full set of the neighbouring relations names.

Collection of sets $\{EnNames_i^*(e)\}$ must be pairwise disjoint. The names of all inherited inner lexicals, all inherited inner entities, all inherited inner relations, all inherited neighbouring entities and all inherited neighbouring relations must be distinct. These names are used for UCL navigation expressions. It is possible to navigate the inherited element, too.

- $(\forall e \in E)(\forall i,j \in [5])\colon EnNames_i^*(e) \cap EnNames_j^*(e) = \emptyset.$

Definition 4.13 (Full accessible relation steps from entities): Names of all relation ends which are accessible from the entity including all inherited relation ends.

- $EnAccessNames^*(e) \stackrel{\text{def}}{=} \bigcup_{a \in E.accEnds^*(e)}\{Re.name(a)\}$

4.5.13. Formal concept of UCL Data meta-model

We combine together all previous defined components to formally define the full concept of UCL Data meta-model.

Definition 4.14 (Concept of UCL Data meta-model):
The general concept of UCL Data meta-model is a structure:

$\mathcal{M} = (Entities^*, Relations, RelEnds, Lexicals)$, where:

- $Entities^*$ is the full structure of *entities*.
- $Relations$ is the structure of *relations*.
- $RelEnds$ is the structure of *relation ends*.
- $Lexicals$ is the structure of *lexicals*.

4.6. Sample meta-model and UCL Data meta-model

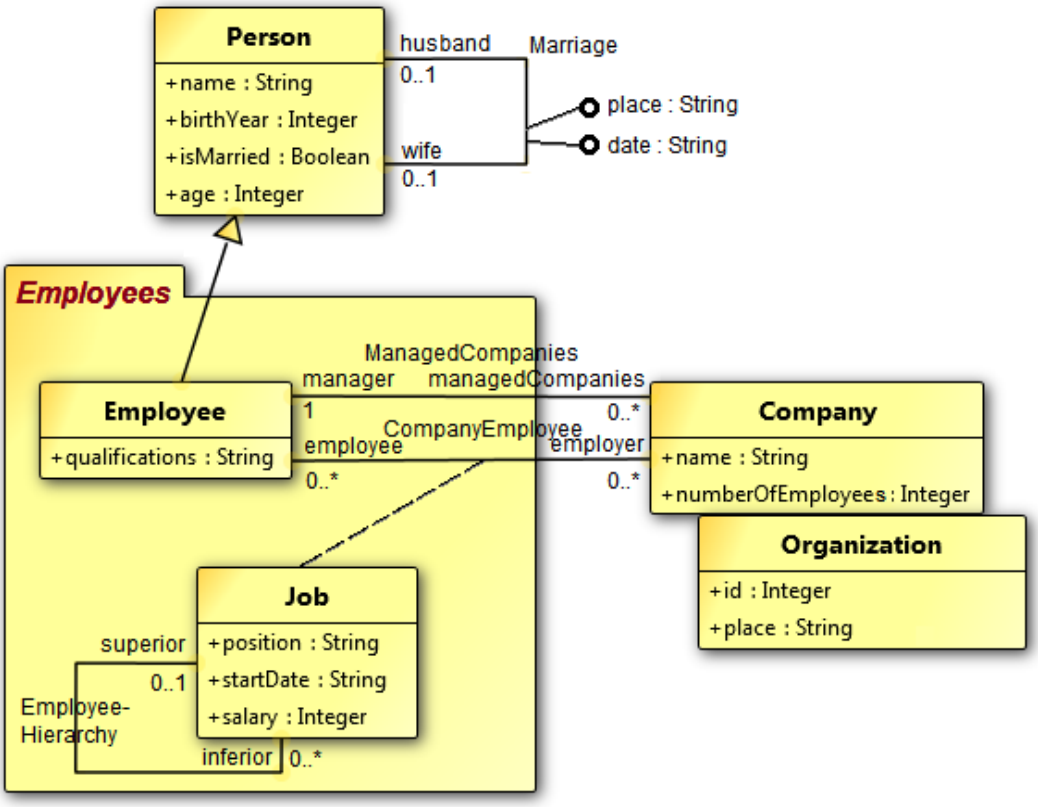


Figure 4.7: Model of Sample meta-model in view of UCL Data meta-model

Figure 4.7 shows a model (a diagram) of *Sample meta-model*. This *Sample meta-model* was invented only for the purpose of demonstrate *UCL Data meta-model*. The model in the figure is an instance model of *Sample meta-model*.

Sample meta-model defines four kinds of elements:

- *SampleClass* (e.g. *Person* in the model in the figure);
- *SamplePackage* (e.g. Employees);
- *SampleAttribute* (e.g. *name* of SampleClass *Person*);
- *SampleAssociations* (e.g. *Marriage*).

A SampleClass and a SampleAssociation can contain SampleAttributes. A SampleAssociation contains SampleAssociationEnds connected to SampleClasses. The generalization between SampleClasses is defined (e.g. *Employee* inherits *Person*). A SamplePackage can contain inner SampleClasses and SamplePackages. Two SampleClasses can be in the neighbour relationships (e.g. *Company* and *Organization* are neighbours). A SampleClass can be connected to a SampleAssociation in role of a SampleAssociationClass (e.g. *Job* is connected to *CompanyEmployee*); this SampleClass represents special information about the SampleAssociation.

Elements and relationships between elements of *Sample meta-model* must be mapped to instances of elements of *UCL Data meta-model*. And elements in a model of *Sample meta-model* must be mapped to elements of a model of UCL Data meta-model.

4.6.1. Mapping of Sample meta-model to UCL Data meta-model

The elements in UCL meta-model are:

- Elements SampleClass and SamplePackage are instances of UCL entity.
- SampleAssociation is an instance of UCL relation (with corresponding UCL relation ends).
- SampleAttribute is an instance of UCL lexical.

The relationships between elements in UCL meta-model are:

- A SampleAttribute of a SampleClass is an inner UCL lexical of UCL entity; and a SampleAttribute of a SampleAssociation is an inner UCL lexical of a UCL relation.
- A SampleClass in a SamplePackage is an inner UCL entity of a UCL entity.
- A neighbourhood between SampleClasses (e.g. *Company* and *Organization*) is a neighbourhood between UCL entities.

- A connection of a SampleClass to a SampleAssociation (e.g. *Job* with *CompanyEmployee*) is modeled to a neighbourhood relationship between a UCL entity and a UCL relation.
- A generalization between two SampleClasses (*Employee* is child of *Person*) is modeled to a generalization between two UCL entities.

4.6.2. Mapping of a model of Sample meta-model to UCL Data meta-model

Without a formal definition of *Sample data meta-model* we note the model in the figure Figure 2.9 in view of UCL Data meta-model.

The model in the figure Figure 2.9 in view of UCL Data meta-model is a structure: $\mathcal{M}_{Sample\ model} = (Entities^*_{SM}, Relations_{SM}, RelEnds_{SM}, Lexicals_{SM})$, according Definition 4.14;

where:

- the set of entities is $E = \{e_{Pe}, \mathrm{e}_{Ems}, e_{Em}, e_{Jo}, e_{Co}, e_{Or}\}$;
- the set of relations is $R = \{r_{Ma}, r_{MaCo}, r_{CoEm}, r_{EmHi}\}$;
- the set of lexical is $L = \{l_{PeN}, l_{PeBY}, l_{PeIM}, l_{PeA}, l_{EmQ}, l_{JoP}, l_{JoSD}, l_{JoS,} l_{CoN}, l_{CoNE}, l_{OrI}, l_{OrP}, l_{MaP}, l_{MaD}\}$;
- the set of relation ends is $Re = \{re_{MaH}, re_{MaW}, re_{MaCoE}, re_{MaCoC}, re_{CoEmE}, re_{CoEmC}, r_{eEmHiS}, re_{EmHiI}\}$;

and where:

- the names of the entities are $E.name$: $\begin{cases} e_{Pe} \to Person \\ \mathrm{e}_{Ems} \to Employees \\ \dots \end{cases}$;
- the lexicals of the entities are $E.lex$: $\begin{cases} e_{Pe} \to \{l_{PeN}, l_{PeBY}, l_{PeIM}, l_{PeA}\} \\ \mathrm{e}_{Ems} \to \emptyset \\ \dots \end{cases}$, e.g. the entity *Person* has lexicals *name*, *birthYear*, *isMarried* and *age*;
- the inner entities are $E.inEn$: $\begin{cases} \mathrm{e}_{Ems} \to \{e_{Em},\ e_{Jo}\} \\ \text{else} \to \emptyset \end{cases}$, the entity *Employees* has the inner entities *Employee* and *Job*;
- the inner relations are $E.inRel = \emptyset$, entities in the diagrams have no inner relations;
- the connected relations to entities are $E.conRel$: $\begin{cases} e_{Pe} \to \{r_{Ma}\} \\ e_{Or} \to \emptyset \\ \dots \end{cases}$, e.g. the entity *Person* is connected to *Marriage*, the entity *Company* has no connected relations;

- the accessible relation ends from entities are $E.accEnds: \begin{cases} e_{Pe} \rightarrow \{re_{MaH}, re_{MaW}\} \\ e_{Or} \rightarrow \emptyset \\ \dots \end{cases}$, e.g. from the entity *Person* the relation ends *husband* and *wife* of the relation *Marriage* are accessible;
- the neighbouring entities of the entities are $E.neEn: \begin{cases} e_{Co} \rightarrow \{e_{Or}\} \\ e_{Or} \rightarrow \{e_{Co}\} \\ \text{else} \rightarrow \emptyset \end{cases}$, the entities *Company* and *Organizations* are in a neighbour relationship;
- the neighbouring relations of the entities are $E.neRel: \begin{cases} e_{Jo} \rightarrow \{r_{CoEm}\} \\ \text{else} \rightarrow \emptyset \end{cases}$, the relation *CompanyEmployee* is the neighbour of the entity *Job*;
- the generalization hierarchy on entities is the relation $\prec_{E.gen} = \{(e_{Em}, e_{Pe})\}$, *Employee* is the child entity of *Person*;
- the names of the relations are $R.name: \begin{cases} e_{Pe} \rightarrow Person \\ \mathrm{e}_{Ems} \rightarrow Employees \\ \dots \end{cases}$;
- the lexicals of the relations are $R.lex: \begin{cases} r_{Ma} \rightarrow \{l_{MaP}, l_{MaD}\} \\ \text{else} \rightarrow \emptyset \end{cases}$, the relation *Marriage* has the lexicals *place* and *date*, rest of the relations does not have any lexical;
- the neighbouring entities of the relations are $R.neEn: \begin{cases} r_{CoEm} \rightarrow \{e_{Jo}\} \\ \text{else} \rightarrow \emptyset \end{cases}$, the entity *Job* is the neighbour of the relation *CompanyEmployee*;
- the relation ends of the relations are $R.ends: \begin{cases} r_{Ma} \rightarrow \{re_{MaH}, re_{MaW}\} \\ r_{CoEm} \rightarrow \{re_{CoEmE}, re_{CoEmE}\} \\ \dots \end{cases}$, e.g. the relation *Marriage* has the relation ends *husband* and *wife*;
- the role names of the relations ends are $Re.name: \begin{cases} re_{MaH} \rightarrow husband \\ \dots \end{cases}$;
- the (outer) relations of the relation ends are $Re.relation: \begin{cases} re_{MaH} \rightarrow r_{Ma} \\ \dots \end{cases}$;
- the connected entities of the relation ends are $Re.entity: \begin{cases} re_{MaH} \rightarrow e_{Pe} \\ \dots \end{cases}$, e.g the relation end re_{MaH} husband is connected to the entity *Person*;
- the cardinalities of the relations ends are $Re.card: \begin{cases} re_{MaH} \rightarrow (0,1) \\ \dots \end{cases}$;
- the names of the lexicals are $L.name: \begin{cases} l_{MaP} \rightarrow place \\ \dots \end{cases}$;
- the (basic) types of the lexicals are $L.name: \begin{cases} l_{MaP} \rightarrow String \\ \dots \end{cases}$.

5. UCL description

5.1. Introduction and comparison with OCL

In this chapter, the *Universal constraint language* (UCL) is introduced informally. The formal definition of UCL (according its meta-model) is in the next chapter (Chapter 6: Meta-model of UCL constraints).

UCL is a formal language to express integrity constraints over different data meta-models. Expressions of UCL are based on UCL Data meta-model. UCL Data meta-model is a general meta-model. It was created to enable the mapping of various data meta-models to UCL Data meta-model. Using this mapping, it is possible to express UCL expressions over different meta-models. We can express constraints in UCL over meta-models which we can map to UCL Data meta-model.

Expressions in UCL typically express simple or complex invariants. A UCL expression can be also used as a query without any data side effect over data which is independent of the data model.

UCL is based on *Object constraint language* (OCL). It is designated on UML object-oriented and state models. UCL is designed over UCL Data meta-model that is a general data meta-model.

Similarly to OCL, UCL is based on terms and predicates of the first-order predicate logic. Its syntax is similar to programming languages. UCL is a typed language; each expression has a type defined statically before the interpretation. The language has the predefined set of primitive types (*Integer*, *Boolean*, *String* and *Real*). Then each *UCL entity* and each *UCL relation* in UCL model represents special type in UCL. Also collection types from other type can be created. All constraints in UCL are *invariant expressions*. An invariant is a constraint expression which expresses a condition which must be satisfied all the time for all instances of the defined context. *Pre-conditions*, *post-conditions* and *guards* from OCL are not supported.

Not all language constructions of OCL are supported in UCL; e.g. collection literals, tuples, the iterate operation, constraints over operations and enumerations. And conversely UCL defines more kinds of navigational expressions; because it is based on more general data meta-model (UCL Data meta-model) than OCL (UML class and state diagrams).

OCL defines only 3 kinds of standard navigation expressions:

- from a class to its attributes;
- from a class to another class through an association;
- from a class to an association class which is connected to an association.

UCL defines 10 kinds of navigation expressions over UCL Data meta-model.

UCL defines more possibilities by expressing the context of declaring constraint invariants. In OCL, we can define the context of constraint only by UML classes and packages or states. In UCL, we can define the context of expressions over each root entity, over each root relation and we can use the navigation to inner entities and inner relations as it is proposed in UCL Data meta-model.

5.1.1. Sample model for example UCL constraints

Examples of UCL expressions in this chapter are expressed in the context of the model in Figure 4.7.

5.1.2. Lexical rules

UCL expressions consist of Unicode [58] characters. String literals and comments can contain any valid Unicode character. Identifier names of elements and types in the model can contain only standard Latin alphabet letters (A-Z) in lower and upper case, digits and underscore (_). An identifier cannot start with a digit or an underscore.

There are two styles of comments in UCL. A *single line style comment* which starts with two successive dashes (`--`) and ends at the end of the line. And a *multiple line* style opened with `/*` and closed with `*/`.

```
expressions… --this is a comment
expressions… /* this is a
comment */ expressions…
```

Figure 5.1: Styles of comments in UCL

A keyword is a reserved identifier which cannot be used in UCL expression as a name of a model element. Keyword in UCL are: `and`, `asSet`, `boolean`, `collect`, `context`, `def`, `else`, `endif`, `exists`, `false`, `forAll`, `if`, `implies`, `in`, `integer`, `inv`, `isEmpty`, `let`, `max`, `min`, `mod`, `not`, `notEmpty`, `or`, `real`, `reject`, `select`, `self`, `size`, `string`, `sum`, `then`, `true` and `xor`. The keywords are case-insensitive.

5.1.3. Precedence of operators rules

The precedence order of operators in UCL, from the operators with the highest precedence is in the list:

- parenthesis `()`
- navigation operators dot "`.`", step arrow "`->`", stop arrow "`:>`"
- unary "`not`", "`+`" and "`-`"
- multiplicative "`*`" and "`/`"
- additive "`+`" and "`-`"
- relational "`<`", "`>`", "`<=`" and "`>=`"
- equality "`=`" and "`!=`" / "`<>`"
- "`and`"
- "`or`"
- "`xor`"
- "`implies`" / "`=>`"
- "`if`" ... "`then`" ... "`else`"
- "`let`" ... "`in`" ...

5.1.4. Constraint expression example

Figure 5.2 demonstrates two constraint expressions in UCL. The keyword `context` defines the context entity or relation of the constraints. Here the context is the entity *Person*. The keyword `inv` specifies that we define an invariant constraint (however it is the only one possibility in UCL). In the code, there are three constraint invariants. After the keyword `inv`, there is an optional name of the invariant (e.g. *yearBefore2012*). The third invariant (in the fourth line) does not have any invariant name. After the colon, there are the invariant expressions.

```
Context Person
inv yearBefore2012: self.birthYear < 2012
inv ageNotNegative: self.age >= 0
inv: self.name <> "no name"
```

Figure 5.2: UCL expressions example

5.2. Relation to UCL Data meta-model

5.2.1. Context definition

Each UCL constraint is written in the specified context of a UCL entity or a UCL relation of the source model. This model is an instance of UCL Data meta-model. The context must be defined at the beginning of each block of UCL constraints.

```
Context Person
inv: …

Context Marriage
inv: …
```

Figure 5.3: Context of UCL constraints

Figure 5.3 demonstrates the definition of the context of UCL expressions. There are constraints in the context of the entity *Person* and in the context of the relation Marriage. Defining of the context is in UCL more complex than in OCL. In this way we can define context for root entities and for root relations. The entity *Person* is a root entity and the relation *Marriage* is a root relation. They are not an inner element in another UCL entity. To specify context for non root elements, we must notice the full path name to the element from the root entity through all inner entities transitive to the context entity or relation. Particular inner entities and relations in this path are syntactically separated by a double colon. The first is the root entity. Figure 5.4 shows definitions of complex context.

```
Context Employees::Job …
Context Employees::EmployeeHierarchy …
Context RootEntity1::Entity2::Entity3::TargetEntity …
```

Figure 5.4: Complex contexts of UCL constraints

5.2.2. Keyword "self"

Each constraint in UCL is defined in a context of an entity or of a relation. The keyword `self` in UCL expressions refers to instances of the context entity or relation. The demonstrational invariant *constraint1* does not refer to instances of the context entity because it does not contain the `self` expression. The second invariant *constraint2* refers to instances of the entity *Person*.

```
Context Person
inv constraint1: 0 < 1
inv constraint2: self.age >= 0
```

5.2.3. Invariants

To define integrity constraints, UCL uses invariant expressions. An invariant is an expression of the *Boolean* type. It must be evaluated to true for all instances of the context element at any time.

5.3. Types, values and operations

5.3.1. Basic types and operations

As in OCL, like in UCL there are 4 predefined basic types *Boolean* (`true` / `false`), *Integer*, *Real* and *String* (`"text"`). It is not possible to use other basic type. There are defined standard operations on the basic types:

- *Integer*: =, != / <>, <, >, <=, >=, +, –, *, /, unary +, unary –
- *Real*: =, != / <>, <, >, <=, >=, +, –, *, /, unary +, unary –
- *Boolean*: `xor`, `or`, `and`, =, != / <>, `not`
- *String*: =, != / <>, <, >, <=, >=

There are defined standard operations on UCL entities and UCL relations:

- =, != / <>

5.3.2. Types from the model

Each entity and relation from the model induces a special type in UCL expressions which are attached to UCL Data meta-model. E.g. type of expression `self` is the context UCL entity or UCL relation.

5.3.3. Collections

A UCL navigation expression over a UCL relation can result to a sequence of entities or relations. UCL contains also sequence types *Sequence* and *Set*. And the abstract base type *Collection*. A sequence is a collection of elements that can contain duplicates, a set cannot contain duplicates. Collection types are generic. They hold information about type of their elements. They can contain only elements of this type. It is possible to define collection of collection.

Examples of collection types are: `Sequence (Integer)`, `Set (Person)`, `Sequence (Sequence (Person))`.

5.3.4. Variables "def" and "let" definitions

Sometimes a part of an expression repeats in the invariant. It is useful to define this part of the expression as a constant or a variable. The `let` expression defines the variable of any kind of expression type. This variable can be used in the constraint.

But the `let` expression defines the variable only in the context of itself; we cannot use the variable outside the expression. E.g. we cannot use the variable *doubleAge* outside the `let` expression as in the example:

```
Context Person
inv:
  (let doubleAge = 2 * self.age in
    doubleAge >=0 and doubleAge < 400)
  or doubleAge mod 2 = 0 /* can not use doubleAge here */
```

To define a variable in the whole context block of invariants, we can use a `def` definition. The definition must be attached to the context of model element. Its syntax is similar to a `let` expression.

```
Context Person
def doubleAge = 2 * self.age
inv: doubleAge >=0
inv: doubleAge < 400
```

The names of variables `let` and `def` in a scope cannot conflict.

5.3.5. Type conformance

Each expression has a statically defined type and each valid expression has to satisfy the type conformance rules. Operations on the basic types and navigation operations in UCL expressions can be used only over expressions of the allowed types.

The types are organized in a tree type hierarchy which defines the conformance between different types. The type *one* conforms to the type two if an instance of the type *one* can be substituted by an instance of the type *two*. The relation *type conformance* is reflexive and transitive; the next rules exist:

- *Integer* conforms to *Real*;
- the entity type *T1* conforms to the entity type *T2* if the entity *T1* is a child entity of the entity *T2*;

- *Collection (T1)* / *Sequence (T1)* / *Set (T1)* conforms to *Collection (T2)* / *Sequence (T2)* / *Set (T2)* if *T1* conforms to *T2*
- *Sequence (T)* conforms to *Collection (T)*
- *Set (T)* conforms to *Collection (T)*

5.4. Expressions

In UCL, there are several kinds of expressions:

- literals (a value of a basic type: *Integer*, *Real*, *Boolean* or *String*);
- expression `self` (a variable referring to an instance of the context element);
- variables (a variable referring to a `let` or to a `def` definition or to a collection expression);
- operations on the basic types;
- collection expressions (e.g. `forAll`);
- collection operations (e.g. `size`);
- navigation expressions over the input model (a model of UCL Data meta-model).

5.4.1. Simple steps navigation expressions from an entity

Using the keyword `self`, an expression can refer to an entity from the input model. Simple step navigation from an entity is the technique how to access to other model element that is connected with the entity. It is possible to connect these elements:

- lexicals of the source entity;
- inner entities of the source entity;
- inner relations of the source entity;
- neighbouring entities of the source entity;
- neighbouring relations of the source entity.

According the formal definition of UCL Data meta-model, names of all these connected elements must be distinct. We can access these elements from an entity by the expression of the syntax: a dot operator (`.`) followed by the name of the connected element. For example:

```
-- in the context of Company:
  self.name /* a) lexical of Company */
-- in the context of Employees:
  self.Employee /* b) inner entity of Employees */
  self.EmployeeHierarchy /* c) inner relation of Employees */
-- in the context of Company:
  self.Organization /* d) neighbouring entity of Company */
-- in the context of Job:
  self.CompanyEmployee /* e) neighbouring relation of Job */
```

Figure 5.5: Simple steps navigation expressions from an entitiy

Value of this expression is the value of the target lexical (in the case *a*) or an expression referring to the target model element (in the other cases). The type of the expression is the type of the target lexical (in the case *a*) or the type of the target model element (in the other cases). Using these navigation expressions we can express calculations over the model. For example domain constraint of lexicals (e.g. “The age of persons is always greater than zero.”) or we can define complex constraints based on the relationship between elements which are connected by these kinds of navigation.

5.4.2. Simple steps navigation expressions from a relation

In the similar way as by the entities, we can also access model elements which are connected with a relation. It is possible to connect these elements:

- lexicals of the source relation;
- neighbouring entities of the source relation;
- relation ends of the source relation which are connected to an entity.

Names of these connected elements and names of these relation ends must be distinct according the formal definition of UCL Data meta-model. Examples of expressions:

```
-- in the context of Marriage:
  self.place /* a) lexical of Marriage */
-- in the context of CompanyEmployee:
  self.Job /* b) neighbouring entity of CompanyEmployee */
  self.employer
      /* c) relation end of the relation CompanyEmployee;
            it is connected to the entity Company;
            the type of expression is the entity Company */
```

Figure 5.6: Simple steps navigation expressions from a relation

The value of these expressions is the value of the target lexical (in the case *a*) or an expression referring to the neighbouring entity (in the case *a*) or an expression referring to the connected entity (in the case *c*). The type of the expression is the type of the target lexical (in the case *a*) or the type of the target entity (in the cases *b* and *c*).

Concrete examples of expressions:

```
Context Employees
inv: self.Job.position = "baker"
       implies self.Job.CompanyEmployee.employer.name <> "IBM"

Context Employees::EmployeeHierarchy
inv: self.inferior.salary <= self.superior.salary

Context Company
inv: self.name = "ABC" => self.Organization.place <> "USA"

Context Marriage
inv: self.place = "Prague" => self.husband.age >= 18
```

5.4.3. Navigation expressions through relations

Starting from an entity, we can create a navigation expression to other entity through a connected relation. This navigation can be created using the name of an opposite relation end of a connected relation.

For example, in context of the entity *Employee* we can navigate using expression `self->employer` to the entity Company. Or from the entity *Company*, we navigate to the entity *Employee* using `self->manager`. A value of such navigation expressions is the target entity or a sequence of the target entities; according the cardinality of the used opposite relation end to the navigation. If the cardinality is *(0, 1)* or *(1, 1)* then the value is an instance of the target entity; else it is a sequence of instances of the target entity. The type of the first exemplar expression is *Sequence (Company)*, the type of the second expression is *Employee*.

A navigation expression from an entity to other entity through a connected relation expressed with the operator `->` is called a *connected relation step expression*. The second type of a navigation expression through connected relation is *connected relation stop expression*. It is expressed with the operator `:>`. The value of this expression is not the opposite entity but the relation which connects the entities.

Concrete examples of expressions:

```
Context Company
inv: self->manager -- expression referrs to entity Employee
inv: self:>manager -- referrs to relation CompanyEmployee

Context Employee
inv: self->employer -- referrs to Sequence (Company)
inv: self:>employer -- referrs to Sequence (CompanyEmployee)

Context Company
inv: self->manager.qualifications = "manager"

Context Person
inv: (self.isMarried and self:>husband.place = "StateX")
       implies self.age >= 21
```

5.4.4. Generalization

In UCL Data meta-model, the generalization hierarchy on entities is defined. A child entity inherits all lexicals, all inner entities, all neighbour entities and relations and all connected relations of the parent entity.

In the example, the entity *Employee* is a child entity of the entity *Person*. The entity *Employee* inherits all these inner and connected elements of the entity *Person*. It is possible to define these expressions in the context of the entity *Employee*:

```
Context Employees::Employee
inv: self.age >= 0
inv: self->husband.age >= 18

Context Company
inv: self->manager.age >= 25
```

5.4.5. Collection expressions

The result of a navigation expression through a relation can be a collection. The language defines collection operation and collection expressions over collections. There are expressions to query items and expressions to express condition about (all or some) items of the collection. The collection operations and collection expressions are prefixed with the colon operator (`:`).

The expressions `select` and `reject` select a subset of the source collection using a condition. The expression `select` takes the items that satisfy the condition; `reject` takes the items that do not satisfy the condition. The result of the expression is a collection of the same type of item as the input collection.

Syntax of the expressions looks:

```
collection:select(variable | condition)
collection:reject(variable | condition)
```

A variable is used as the iterator over items of the input collection. The type of the variable is the type of the collection items. The condition is a *boolean* expression; it is used to specify which elements select (or not selects) to the result collection. The expression `select` gets the collection of the items for which the condition is evaluated to *true*; `reject the` items for which the consition is evaluated to *false*.

The concrete examples of expressions: the first expression selects *Employees* with age over 50; type of the expression is *Sequence (Employee)*. The second selects items of relation *CompanyEmployee* which are connected to the entity *Job* which *salary* is not under 100; the type of the expression is *Sequence (CompanyEmployee)*.

```
Context Company
inv: self->employee:select(empl| empl.age > 50)…
inv: self:>employee:select(comEmpl| comEmpl.Job.salary < 100)…
```

The expressions `select` and `reject` result in sub-collection of the original collection of the original type of items.

The operation `collect` gets the collection of results of the inside expression (*collect-expression*). The syntax of the expression looks:

```
collection:collect(variable | collect-expression)
```

The concrete examples of `collect` expressions: The first expression (*inv1*) selects the collection of salaries; its type is *Sequence (Integer)*. The second (*inv2*) gets the collection of salaries above 100; its type is *Sequence (Integer)*. The result of the thirst expression (*inv3*) is equivalent to the second.

```
Context Company
inv inv1:
  self:>employee:collect(comEmpl | comEmpl.Job.salary)…

inv inv2:
  self:>employee:select(comEmpl | comEmpl.Job.salary > 100)
  :collect(comEmpl | comEmpl.Job.salary)…
```

```
inv inv3:
  self:>employee:collect(comEmpl | comEmpl.Job.salary)
  :select(salaryValue | salaryValue > 100)…
```

To express a constraint if all items of a collection satisfy a specified condition or if there is at least one variation of items which satisfies the inside condition, there are expressions `forAll` and `exists`. The syntax of the expressions is:

```
collection:forAll(variables-list | condition)
collection:exists(variables-list | condition)
```

The type of the expression is *boolean*. The result values indicates if all items in collection satisfy the condition (`forAll`) or if there is at least one variation of items which satisfies the condition (`exists`). The type of the condition must be *boolean*.

```
Context Company
inv inv1:
  self.numberOfEmployees > 100
    implies self:>employee:forAll(ce | ce.Job.salary > 500)

inv inv2:
  self->employee:exists(emp | emp.qualifications = "manager")

inv inv3:
  self:>employee:collect(ce | ce.Job)
  :forAll(job1, job2 |
      job1.position = job2.position
        => job1.salary = job2.salary)
```

The concrete examples of constraints: The first (*inv1*) express that in a company over 100 employees must be all salaries over 500. The second (*inv2*) express that in a company must be at least one employee with the value of qualification *manager*. The third (*inv3*) tells that in a company must have all employees in the same position the same salary.

5.4.6. Collection operations

There are 7 predefined kinds of the operations on collections. They are accessed by the colon operator (`:`) and the name of the operation.

The operation `size()` gets the count of items in a collection; its return type is *Integer*. The operations `isEmpty()` and `notEmpty()` get if a collection contain (or do not contain) any item; their return type is *Boolean*. The operation `sum()` gets

the sum of all items in collection of *Integer* or *Real* values. The operations `min()` and `max()` get the smallest (or the highest) number in a collection of *Integer* or *Real* values; the return type of these operations is a *Integer* or *Real* number. The operation `sum()` gets the sum of all items in a collection of *Integer* or *Real* values; its return type is *Integer* or *Real*. The operation `asSet()` converts type *Sequence (T)* to the type *Set (T)*; it removes duplicity elements from the input collection.

The examples of collection operations:

```
Context Company
inv inv1:
  self.numberOfEmployees = self->employee:size()

inv inv2:
  self:>employee:collect(comE | comE.Job.salary):min() > 0

inv inv3:
  self:>employee:collect(comE | comE.Job.salary):max()
    <= self:>employee:collect(comE | comE.Job.salary):sum()

inv inv4:
  self:>employee:collect(ce | ce.Job.position)
  :asSet():size () > 2

inv inv5:
  self:>employee
  :select(comE | comE.Job:>superior:isEmpty())
  :notEmpty()
```

5.5. UCL syntax

In Appendix A (CD contents), there is the syntax of UCL constraints. Is defines the syntax of UCL expressions in *Extended Backus–Naur Form* (*EBNF*) [43]. EBNF is a metasyntax notation for expressing the grammar of formal languages.

There are also *syntax diagrams* (or *railroad diagrams*). It is a graphical alternative to EBNF for representing a context-free grammar. They are easier to read for people. A grammar of a language is in syntax diagrams expressed by a set of diagrams. Each diagram represents rules for a nonterminal. Each diagram has an entry and an end point. All possible paths from the entry to the end through other terminals and nonterminals describe production rules of the nonterminal. Terminals are depictured by round boxes, nonterminals by square boxes.

Syntax diagrams in the appendix were generated from EBNF by the application *Railroad Diagram Generator* [44].

5.6. Confrontation of OCL and UCL

The aim of this thesis is not to introduce a constraint language with many various constructions to express integrity constraints and with a very strong expressive power. Or aim is to introduce a simple language with base constructions and with the expressive power of the first-order predicate logic. Its expressive power must be suitable to express complex integrity constraints. But the contribution of the language is to use it for different meta-models, to enable mapping between different meta-models. This mapping must be suitable for the derivation of UCL constraints between different meta-models.

5.6.1. Unsupported constructions in UCL

Not all language constructions of OCL are supported in UCL. UCL does not support constrains for operations. It defines only constraints over data. Therefore pre- and post- conditions over operations and conditions of results of operations are not supported, too. Constraints for initial and derived values are not included. It is possible only to express invariants in UCL; invariants are conditions which must be satisfy at all the time for all instances of the context elements. Enumeration types and enumerations literals are not supported in UCL; they can be expressed using string literals. OCL defines 4 kinds of generic collections: a *set*, a *sequence*, a *bag* and an *ordered set*. Then there is the base collection type *collection*. UCL defines only a *sequence* (a collection of items) and a *set* (a distinct collection of items). UCL do not support collection literals and tuples. The standard library of functions in OCL defines some re-typing and casting functions; they are not included in UCL. These functions are: *oclIsTypeOf*, *oclIsKindOf*, *oclInState*, *oclIsNew* and *oclAsType*. UCL do not use the iterate collection operation and messages.

5.6.2. Added constructions to UCL

UCL defines more kinds of navigation expression over the input model (instance of UCL Data meta-model) than OCL. It defines more kinds of navigation because we need to use or for different meta-models, we need ma these meta-models and derive UCL expression between these mapped models.

OCL defines only 3 kinds of standard navigation expressions:

- from a class to its attributes;
- from a class through an association to another class;
- from a class through an association to an connected association class. But this kind of navigation is very specific only for UML class diagrams.

UCL defines 10 kinds of navigation expressions over models of UCL Data meta-model. We derive these navigations to different mapped meta-models.

5.6.3. Navigation to an association class in OCL and in UCL

OCL defines the navigation to class attributes, the navigation through association and the navigation to association classes. But this navigation is in OCL very specific and it is usable only for the UML meta-model.

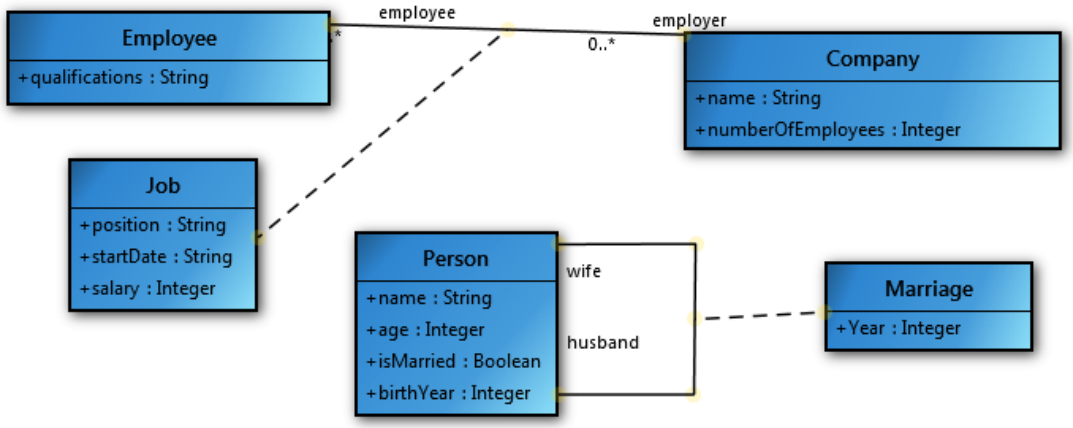


Figure 5.7: Association classes in UML

Figure 5.7 shows a UML model with association classes. To specify navigation to a navigation classes from a class, UCL use a dot notation. From a class we use the name of a connected association class; the result if this expression is the collection of instances of the target association class. E.g. from the class *Employee* we can navigate to the association class *Job* which is connected to the association between *Employee* and *Company*. The result of the first invariant (*inv1*) is the collection of instances of the association class *Job*. The second invariant defines the constraint that the *salary* value must be greater than 0.

```
Context Employee   /* in OCL */
inv inv1: self.job…   /* "job" with the small "j" */
inv inv2: self.job->forAll(job | jobs.salary > 0)
```

We must notice that the association or the association end is not in the OCL expression. However, it is used in the navigation path over the model.

The navigation in OCL to an association class which is connected to a recursive association is even more complex. When we want to navigate from the class *Person* to the association class *Marriage*, we must also express the used association end of the association (*husband* or *wife*). It is important because the path from the class *Person* to the association class must be unambiguous.

OCL defines the special syntax for the navigation to recursive association classes:

```
Context Person
inv1: self.marriage[husband]…
inv2: self.marriage[husband].Year <= 2011
```

The used association end in the navigation is expressed in square brackets.

The both kinds of the navigations to association classes are not suitable for UCL. In UCL, we must express constraints over different mapped meta-models. In different mapped models, association classes are mapped to other types of elements; e.g. in relational databases they are represented by tables which in UML models correspondent to the connected association.

The next problem is that in non-recursive association class in OCL we do not use the name of used association end. This is a problem because we must map this association end to the different mapped model. Otherwise the mapping will be not enough defined to derive UCL expressions to the mapped model.

UCL use the next syntax for the navigation to association classes. When we navigate from the class *Employee*, we must first navigate to the connected association using relation stop expression:

```
Context Employee
inv: self:>employer…
```

This expression ends in the association. It returns a sequence of instances of the association. To continue, we must access to this collection by a collection operation:

```
Context Employee
inv: self:>employer:forAll(association |… )
```

And from the association, we can navigate to the association class *Job* using the simple step expression. The association class is a neighbour UCL entity of the UCL relation. (But it depends on the definition of UCL meta-model for UML meta-model). We can express:

```
Context Employee
inv: self:>employer:forAll(association | association.Job… )
inv: self:>employer
       :forAll(association | association.Job.salary > 0)
```

When we use this syntax for the navigation to association classes, we can is map to different mapped meta-models. And we can derive UCL expression over UML mote-model to UCL expressions over different models.

5.6.4. Conclusion

Our aim is to create a framework to express constraints in one created language. It will be possible to derive constraints over one meta-model to a different mapped meta-model. Then we want to derive constraints from this language to different constraint languages.

In this chapter we have introduced this language. It is called UCL and it is based on OCL. Not all constructions of OCL are included in UCL. But UCL has the expressive power of the of the first-order predicate logic. OCL is based for the UML meta-model. It defines only navigation expressions over this meta-model. It defines only the navigations to attributes, through associations and to association classes. For example the navigation to association classes is very specific for UML. The navigations of OCL are not sufficient for our purposes. We need to define more possibilities for navigation expression in UCL because we need derive all kinds of navigations to different mapped meta-models. UCL is based on the general UCL Data meta-models. UCL defined more kinds for navigation expressions according all kinds of relationship between elements in UCL Data meta-model.

In the next chapter, we define the meta-model of UCL constraints.

6. Meta-model of UCL constraints

In the previous chapters, we have introduced UCL. It is important to define a meta-model of the whole UCL. We have yet defined UCL Data meta-model. It is a general data meta-model. UCL navigation expressions are based on this meta-model. In this chapter, we propose the meta-model of the whole UCL. It is the meta-model of UCL types and expressions. According this meta-model, it will be possible to represent all legal UCL expressions.

Then according it, we can derive UCL constraints to other constraint language or to UCL constraints over other data meta-model.

6.1. Structure of UCL meta-model

We present *UCL meta-model*. It is the meta-model of UCL in UML class diagrams. The model is separated into two parts:

- *Types*
- *Expressions*

The part (or the package in the terminology of UML) *Expressions* depends on the package *Types*. And the package *Types* depends on UCL Data meta-model; elements in model of UCL Data meta-model induce types in package *Types*.

A model of a concrete constraint language over a specific data meta-model (i.e. OCL over UML or Schematron over XML) should be separated into parts: *Types*, *Expressions* and *Values*. But for aims of this thesis, we are not going to describe and implement an interpretation of UCL expressions. We just describe and define the semantics of UCL expression informally. The individual parts of UCL meta-model represent different aspect. We get UML diagram of classes of these parts, we explain their classes and relationships between them. And we present some examples (UCL expression) as instances of this meta-model.

6.2. Types

A UCL expression has a type which is statically defined. Before the package *Expressions* we have to define the package *Types* as the part of UCL meta-model.

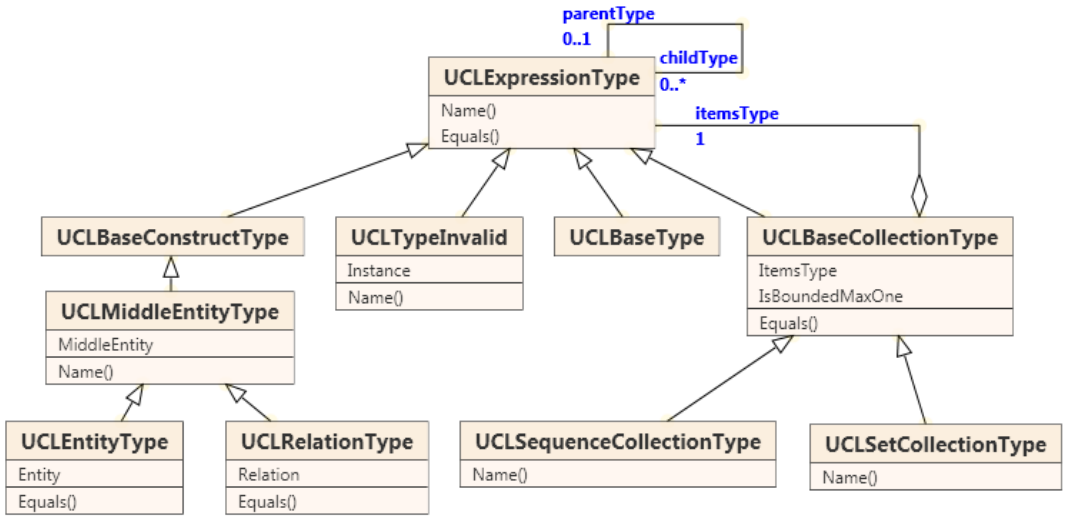


Figure 6.1: Meta-model of UCL types I

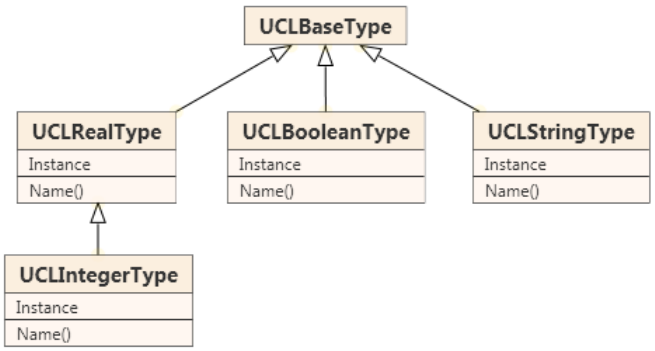


Figure 6.2: Meta-model of UCL types II

Figure 6.1 and Figure 6.2 show classes of the package *Types* of UCL meta-model. Instances of these classes represent the types of UCL expressions and UCL variables. The base abstract class for a type is *UCLExpressionType*; it models all available UCL types. Special class *UCLTypeInvalid* represents the type of expressions that do not satisfy the type conformance rules; the class is singleton, there is exactly one instance of class which represents this special type.

The class *UCLBaseType* and its child classes in Figure 6.2 represent the basic types *Real*, *Integer*, *Boolean* and *String*. These four classes are singletons; their instances represent the individual basic types. The class *UCLIntegerType* for the type *Integer* is the child class of the class for the type *Real*. An *Integer* expression conforms to a *Real* expression.

The class *UCLEntityType* is the type which is induced by an entity (*UCL entity*) from UCL Data meta-model. An instance of this class holds a reference to the UCL entity. An instance of *UCLEntityType* represents a type of UCL

expression derived from the referenced entity. E.g. if there is an entity *Car* in the model then instance of *UCLEntityType* that refers to entity *Car* represents the type of UCL expressions which represent the entity *Car*. Two instances of *UCLEntityType* over the same entity can exist coexistent; two instances of this class are equals if they refer to the same entity.

Analogue the class *UCLRelationType* represents the type induced by a relation (*UCL relation*) from the UCL Data meta-model. The abstract classes *UCLBaseConstructType* and *UCLMiddleEntityType* are parent classes of *UCLEntityType* and *UCLRelationType*.

UCLBaseCollectionType is the base class for collection types a sequence (*UCLSequenceCollectionType*) and a set (*UCLSetCollectionType*). The collection types hold the type of their items. E.g. *Set (Integer)* is represented by an instance of *UCLSetCollectionType* which refers to the (singleton) instance of *UCLIntegerType*. The nesting depth of collection types is not restricted; structured types like *Sequence (Sequence (Car))* are allowed.

6.3. Expressions

In this subchapter, we define meta-model of UCL expressions; including meta-model of definitions UCL contexts and UCL variables. The abstract class which defines an UCL expression is the class *UCLBaseExpression*. Instances of all its child classes represent all possible expressions.

6.3.1. Contexts

A block of UCL expressions consists of contexts blocks. These blocks are represented by instances of the class *UCLContextDefinition*; in the Figure 6.3.

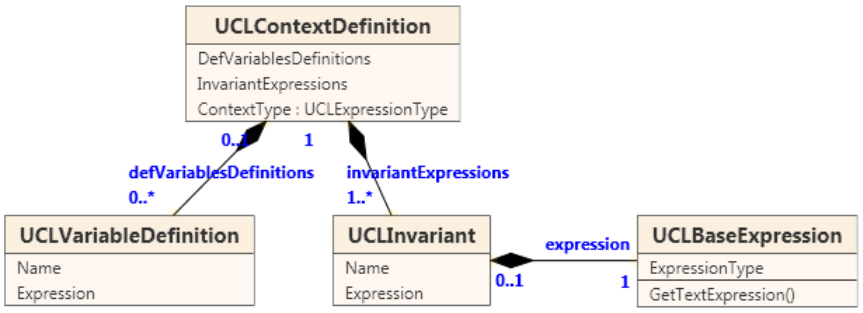


Figure 6.3: Meta-model and integration of contexts of UCL expressions

Each context refers to a UCL entity or to a UCL relation from the model. The attribute *ContextType* holds the type of this model element. A context can contain some (no or more) "def" variable definitions (instances of *UCLVariableDefinition*) and it contains at least one invariant (instances of *UCLInvariant*). A variable definition holds a unique variable name and an expression defining a value of the variable. An invariant has an optional name and it holds an UCL expression of the *Boolean* type; this expression defines the invariant condition.

6.3.2. Kinds of expressions

The abstract class *UCLBaseExpression* represents all possible expressions. Figure 6.4 shows the classes (some these classes are abstract) inherited from *UCLBaseExpression* which represent all individual possible kinds of expressions:

- *UCLLetsExpression* (a "let" variable)
- *UCLLiteralBaseExpression* (a literal value)
- *UCLInvalidExpression* (an incorrect expression; it does not satisfy the type conformance rules or it uses navigation over the input model in a wrong way)
- *UCLBaseOperationExpression* (an operation over the basic types)
- *UCLBaseCollectionOperationExpression* (and operation over collections)
- *UCLBaseCollectionExpression* (an expression over collections)
- *UCLBaseNavigationExpression* (a navigation referring the input model)

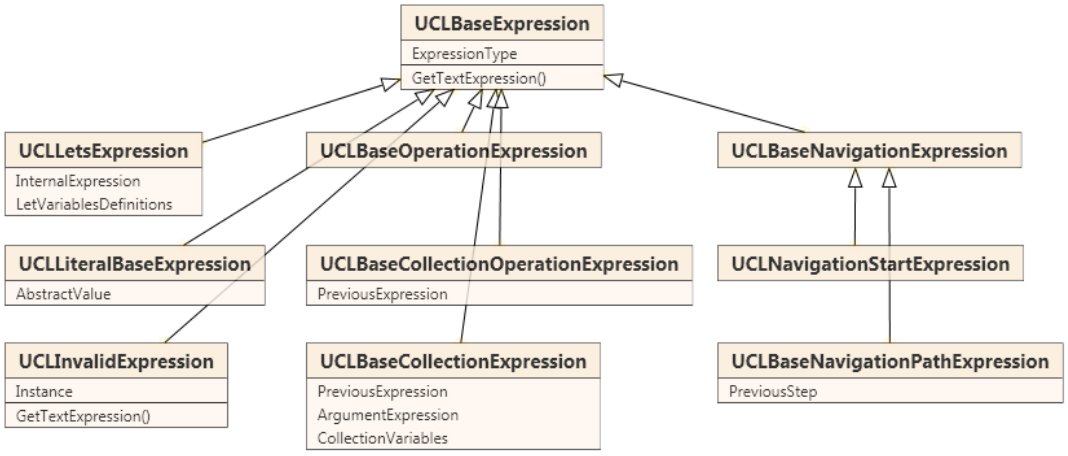


Figure 6.4: Meta-model of all kinds of UCL expressions

In next subchapters, we define the individual kinds of expressions according the classes in UCL meta-model.

6.3.3. Variables

Figure 6.5 shows the meta-model of variables definitions in UCL expressions. The class *UCLCollectionVariable* represents a variable which is defined in a collection expression; e.g. variable *varName* in expression `collection:select(varName | varName.age >= 18)`. It contains the variable name and the type of the previous (collection) expression.

The class *UCLVariableDefinition* represents "def" and "let" definitions of variables. It contains the variable name and holds the expression that defines the variable value (in the connected instance of *UCLBaseExpression*).

The abstract class *UCLVariable* is the parent class of these two classes. An expression which refers to a variable (*UCLVariableExpression*) holds a reference to instance of *UCLVariable*. This expression need not to know if the variable is a collection variable or if it is a "def" or a "let" variable definition.

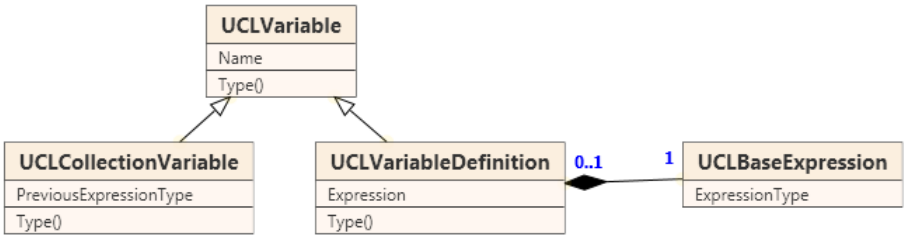


Figure 6.5: Meta-model of UCL variables definitions

Figure 6.6 illustrates the meta-model of expressions which contain "let" variable definition. Such an expression (*UCLBaseExpression*) is wrapped in an instance of *UCLLetsExpression*. This class holds definitions of "let" variables (instances of *UCLVariableDefinition*) and the internal expression (*internalExpression*).

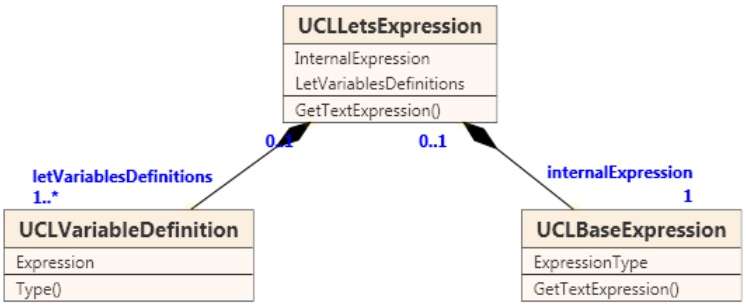


Figure 6.6: Meta-model of UCL expressions with inner "let" variables definitions

Figure 6.7 demonstrates an example of a "let" variable definition. It defines in the invariant *invariantName* two "let" variables `doubleAge` and `tripleAge`. These variables are visible only in the scope of the inner expression (*`expression…`*).

The object diagram in the figure shows the object representation of the expression according UCL meta-model. There is one instance of *UCLLetsExpression* which represents the whole expression. It holds one instance of *UCLBaseExpression* (which represents the inner expression *`expression...`*) and two instances of *UCLVariableDefinition*; they represent variables `doubleAge` and `tripleAge`. These two objects also hold instances of *UCLBaseExpression* representing expression which define the values of the variables.

```
Context Person
inv invariantName:
  let doubleAge = 2 * self.age
  let tripleAge = 3 * self.age in
    expression…
```

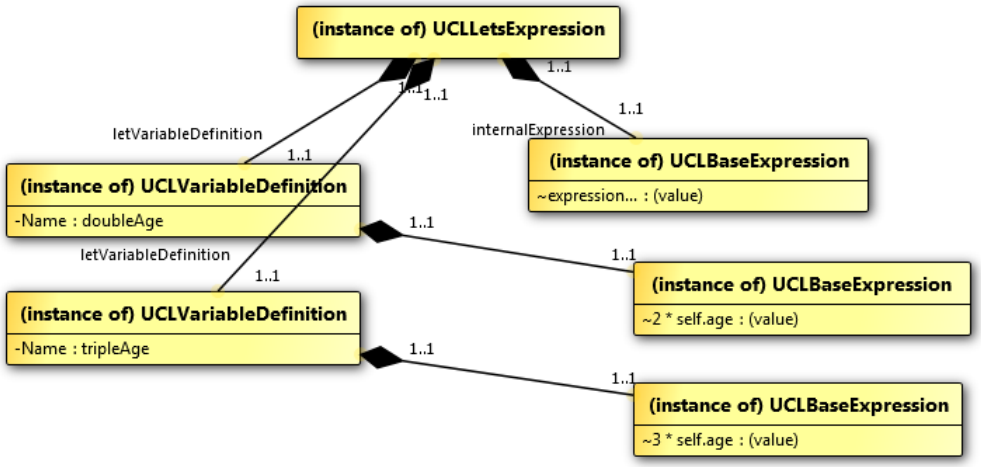


Figure 6.7: Example of "let" variables definitions with the Object diagram

6.3.4. Literals

The abstract class *UCLLiteralBaseExpression* inherits from the class *UCLBaseExpression*. It represents a constant lexical value in a UCL expression. Figure 6.8 shows the four non-abstract classes which represent separate literal values of the individual basic types *Real*, *Integer* (which conforms to *Real* literal), *String* and *Boolean*.

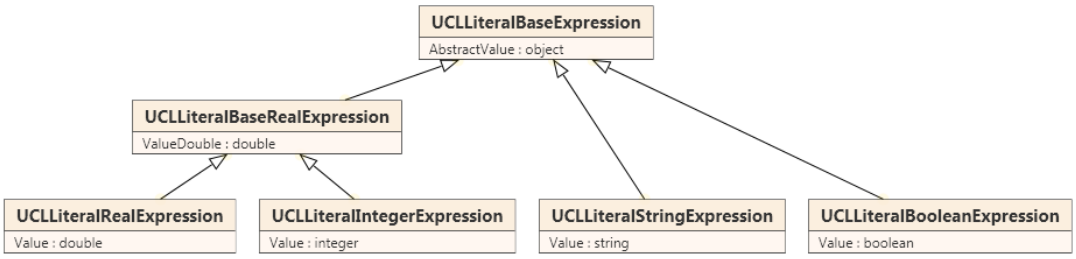


Figure 6.8: Meta-model of lexical values in UCL

6.3.5. Operations

The class *UCLBaseOperationExpression* inherits *UCLBaseExpression*. It defines the large group of all possible operations over expressions of the basic types and expressions `if-then-else`. We decompose operations into the three groups (and diagrams) according the count of their operands.

Figure 6.9 shows the model of unary operations expressions. All operations are represented by the abstract class *UCLUnaryBaseOperationExpression*. It holds an expression which represents the inner operand. There are child classes for the separate unary expressions: *arithmetic unary plus* and *unary minus*, *logical unary not* and *parenthesis* as expression wrapping the expression in the paranthesis.

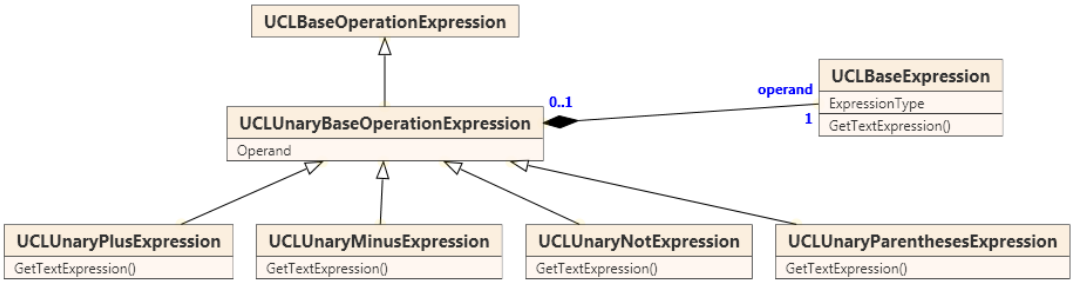


Figure 6.9: Meta-model of UCL unary operations

Figure 6.10 represents the meta-model of all binary operations. The abstract class for all binary operations is *UCLBinaryBaseOperationExpression*. It holds two expressions (*UCLBaseExpression*) which represent the left and the right operand of the operation. Then, there are the four non-abstract classes for all possible operations. Operations are divided into the classes according the precedence and type properties of the operations; classes hold kind of the concrete operation in a value of an attribute:

- multiplicative operations (`*`, `/`) over numerical types
- additive operations (`+`, `-`) over numerical types

- relational operations (=, !=, <, >, <=, >=)
- logical operations (`xor`, `or`, `and`, =, `implies`)

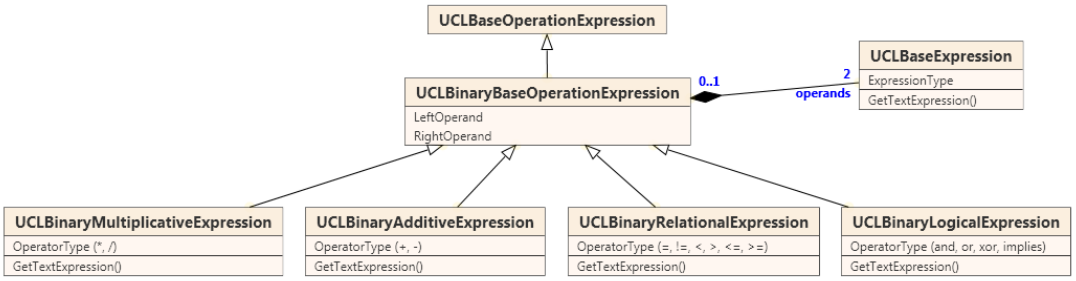


Figure 6.10: Meta-model of UCL binary operations

Figure 6.11 shows the meta-model of expressions `if-then-else`. The class *UCLIfExpression* holds three instances of *UCLBaseExpression*. The first is a condition; it must be of the *Boolean* type. The second represents the *true* (*then*) branch and the third expression represents the *false* (*else*) branch. The type of the third expression must be equal or must conform to the type of the second branch.

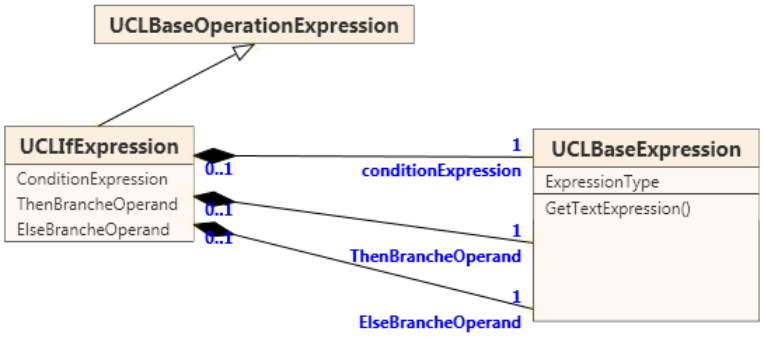


Figure 6.11: Meta-model of UCL if-then-else expressions

6.3.6. Navigation expressions

Figure 6.12 shows the meta-model of starting navigation expressions. An expression `self` is an instance of *UCLSelfEntityExpression* or *UCLSelfRelationExpression*. It refers to the context UCL entity or UCL relation. These two classes are connected through an association to the instance of *UCLEntity* or *UCLRelation* from UCL Data meta-model.

A navigation expression can start also by a reference to a variable. This variable must defined as a "def" or a "let" definition or it is a variable defined in a collection expression. Such an expression is an instance of *UCLVariableExpression*. This class is in an association with the class *UCLVariable*; which defines the variable.

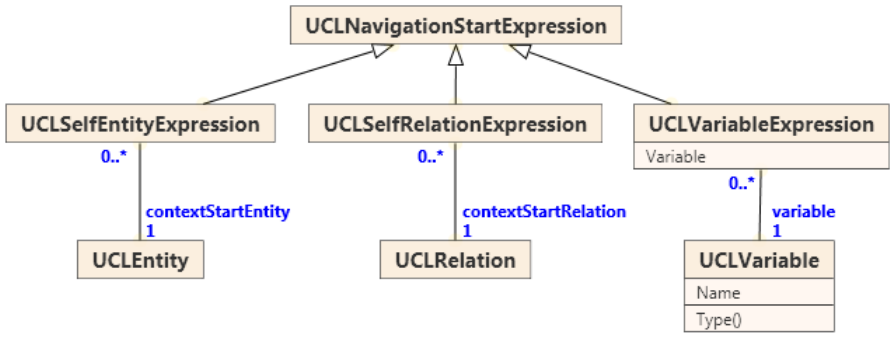


Figure 6.12: Meta-model of UCL navigation expressions – navigation start

Figure 6.13 shows the meta-model of navigation expressions *relation step* (expressed by ->) and *relation stop* (by :>). Both kinds of expressions are navigated from an entity through a connected relation according the name of an opposite relation end. The both classes *UCLBaseNavigationStepExpression* and *UCLBaseNavigationStopExpression* inherit *UCLBaseNavigatonPathExpression* in which they hold the source expression of the navigation (*previousStep*). The both classes contain reference to the source entity (*sourceEntity*) and to the connected relation (*navigatedRelation* / *targetRelation*). Class for the relation step contains also a reference to the target entity (*targetEntity*).

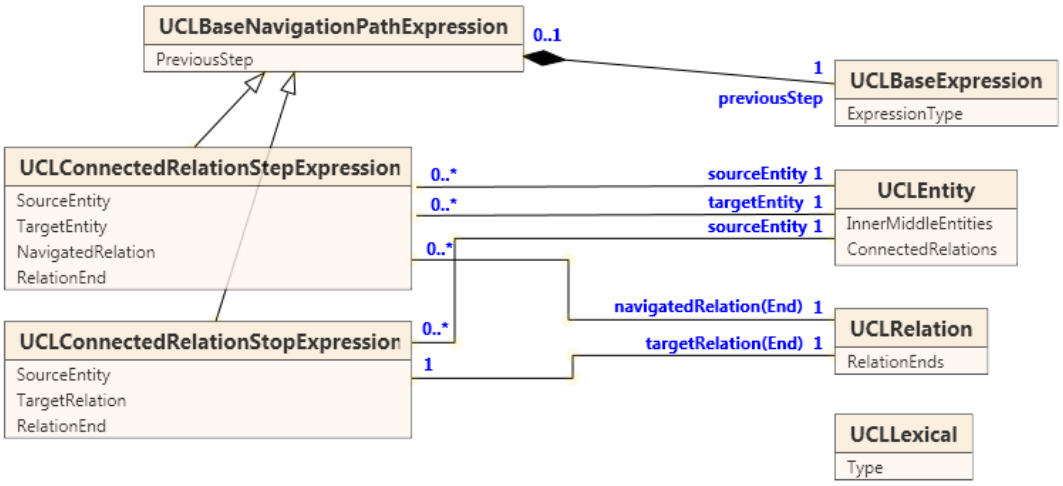


Figure 6.13: Meta-model of UCL navigation expressions – relation step and stop

Figure 6.14 shows the meta-model of *simple steps* navigation expressions. There are 8 non-abstract classes which represent the individual kinds of simple steps expressions. They all inherit from *UCLBaseNavigatonPathExpression* in which they hold the source expression of the navigation (*previousStep*). Then they inherit *UCLSimpleStepExpression*; the generic class that is templated according the types of the source and of the target model element (UCL entity, UCL relation or

UCL lexical). These individual 8 classes hold references to the source model element (5 classes have *sourceEntity*, 3 classes have *sourceRelation*) and to the target model element (4 classes have *targetEntity*, 2 classes have *targetRelation* and 2 classes have *targetLexical*). E.g. the class *UCLStepEntityLexical* represents the simple step navigation from an entity to its inner lexical; it holds references to the source entity and to the target lexical.

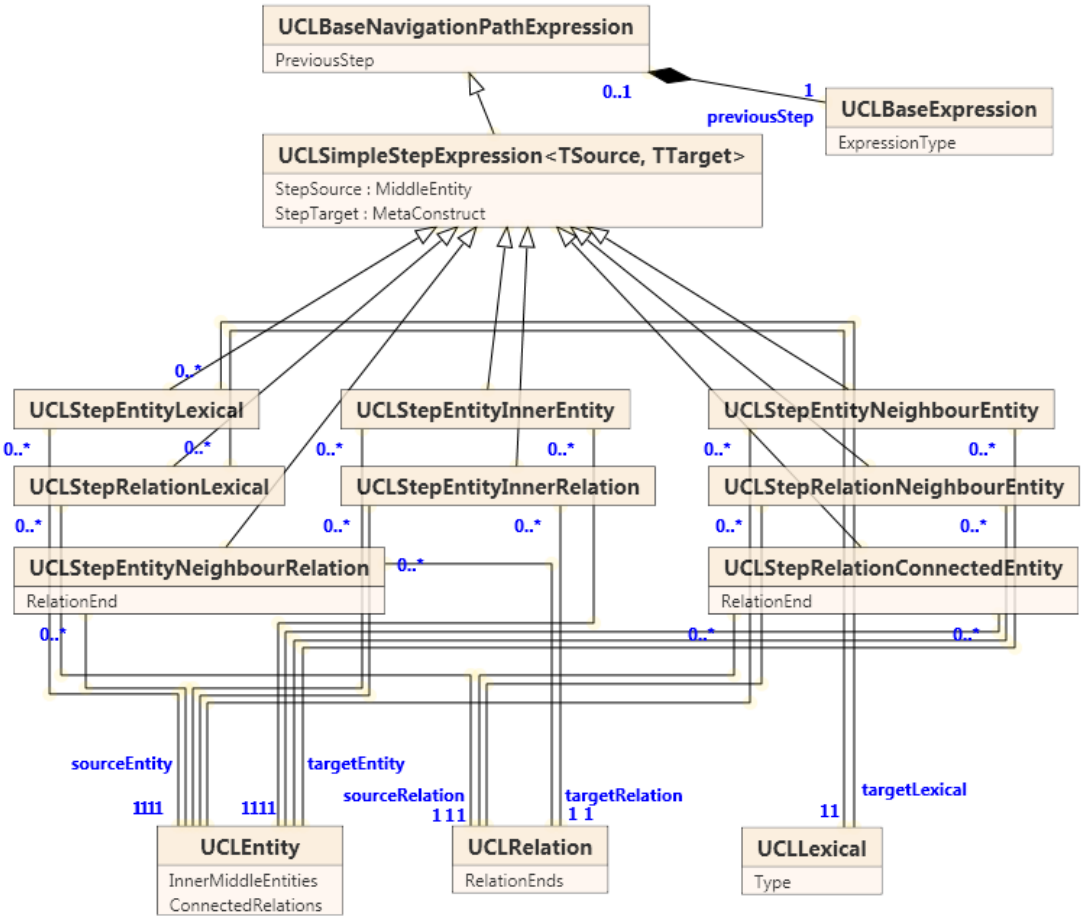


Figure 6.14: Meta-model of UCL navigation expressions – simple steps

6.3.7. Collection operations

Figure 6.15 shows the meta-model of collection operations. There are 7 nonabstract classes which represent the individual kinds of collection operations: `size()`, `sum()`, `max()`, `min()`, `isEmpty()`, `notEmpty()` and `asSet()`. All these classes inherit from the abstract class *UCLBaseCollectionOperation*. It holds the source expression of the collection operation (*previousStep*). These classes do not contain any other information.

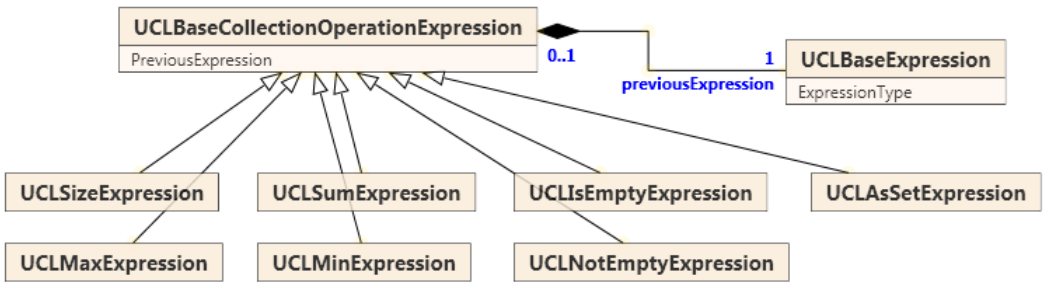


Figure 6.15: Meta-model of UCL collection operations

6.3.8. Collection expressions

Collection expressions differ from other kinds of expressions. They contain the *source expression* like other navigation expressions. But they contain also the *argument (or condition) expression* and definitions of context *variables* over items of the source collection.

In Figure 6.16, there is the meta-model of collection expressions. The individual 5 non-abstract classes represent collection expressions `forAll`, `exists`, `select`, `reject` and `collect`. They inherit from the abstract class *UCLBaseCollectionOperationExpression*. This class keeps the *source expression* of the collection expression (*previousExpression*). Then it contains the *argument expression* of the expression's condition (*argumentExpression*). And it keeps the definitions of all expression's variables (*collectionVariables*). Expressions `forAll` and `exists` can contain more variables definitions, the rest of the collection expressions can contain exactly one variable definition. The 5 classes for collection expression do not store any other information.

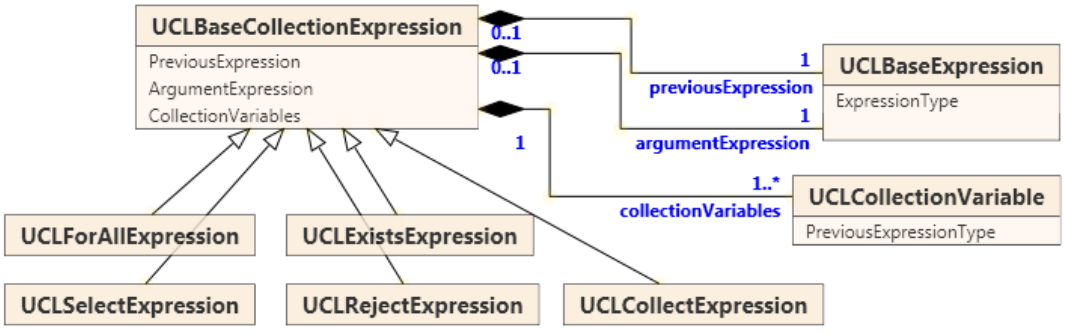


Figure 6.16: Meta-model of UCL collection expressions

6.3.9. Complex example

In the Figure 6.17, there is an example of a complex constraint in UCL. The constraint is used in the context of the example of the UCL Data meta-model in

Figure 4.7. The expression defines, that for all instances of the entity *Company* the next invariant must be fulfilled. The collection (sequence) of all its employees above 30 years must contain at least one person.

```
Context Company
inv: self->employee:select(p | p.age > 30):notEmpty()
```

Figure 6.17: Sample UCL constraint for the demonstration of UCL meta-model

We apply the presented UCL meta-model to the expression in Figure 6.17. We create an instance of the UCL meta-model for this expression. The part of the expression `self->employee` gets the collection of all employees of the source *Company*. Its result is an expression of the type *Sequence (Employee)*. This expression is used as the *source expression* of the *collection expression* `select`. The `select` expression defines a variable `p` of the type *Employee* (type of items of the source collection) which is bound to each item of the collection and for each item the condition of the *argument expression* (`p.age > 30`) is evaluated. The `select` expression creates a sequence with items of the source collection which satisfy the condition. The whole `select` expression (including its source expression, variable and the condition) is the *source expression* of the *collection operation* `notEmpty()` which gets if the result of the source expression's collection is not empty.

Figure 6.18 shows an Object diagram of the UCL constraint in Figure 6.17 as an instance of UCL meta-model. The model of the expression is created by a syntax tree according UCL meta-model; the root element of the tree is the collection operation `notEmpty()`. Operations of the expression which are evaluated at the end are placed in the tree as the parent elements of the operation which are evaluated at the beginning. The element for `notEmpty()` has one child element; its *source expression* – the `select` expression. The `select` expression has tree child branches: the first is the *source expression* (`self->employee`), the second is the definition of the variable (`p`) over the collection. And the third is the *argument expression* of the condition (`p.age > 30`).

The source expression (`self->employee`) is an expression *relation step* (instance of *UCLConnectedRelationStepExpression*). It has a child branch the *source expression* (`self`) and it is connected by associations with the source entity (*Company*), with the relation (*CompanyEmployee*) and with the target entity (*Employee*). Its source expression is the expression `self`; it refers to the context

entity *Company* and it is associated with the instance of the entity (*UCLEntity*) of UCL Data meta-model.

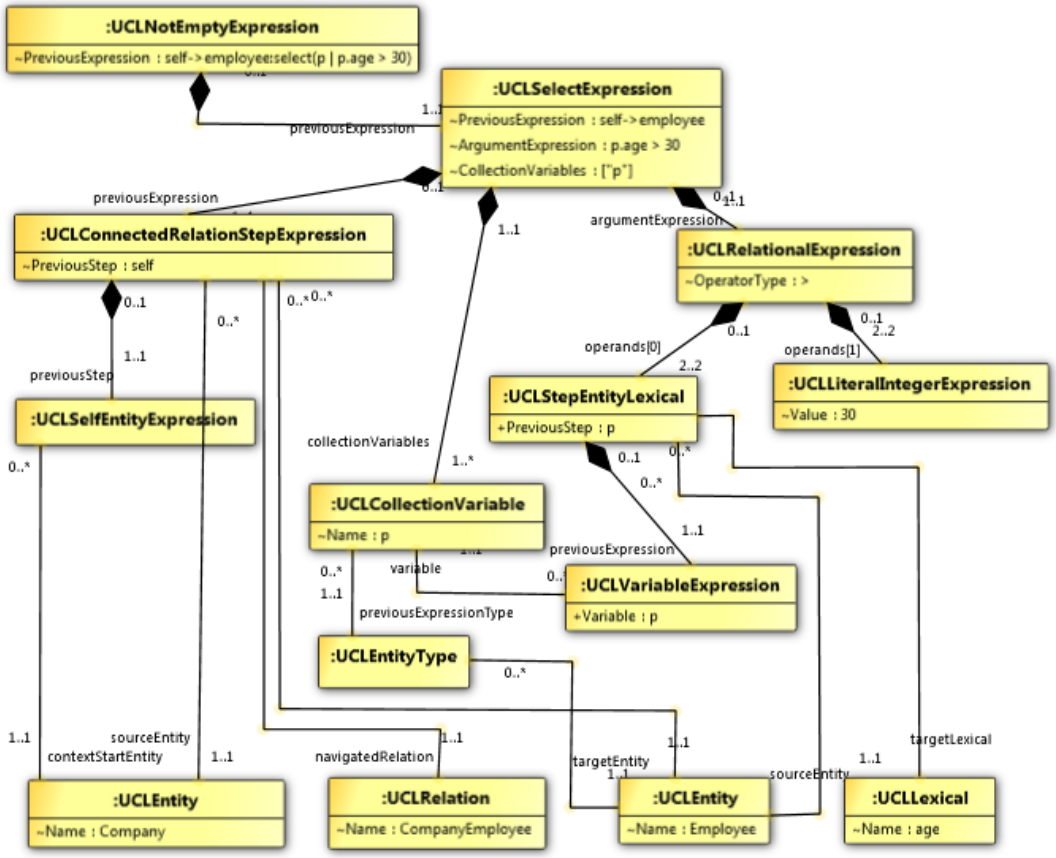


Figure 6.18: Object diagram of the sample UCL constraint

The definition of the variable `p` of the *select* expression is represented by an instance of *UCLCollectionVariable*. It holds the name of the variable and it is associated through an instance of *UCLEntityType* which represents the type of the entity *Employee* to the entity *Employee* (instance of *UCLEntity*). The instance of *UCLCollectionVariable* is also in an association with the using of the variable in the condition of the *select* expression.

The argument expression of the *select* expression (`p.age > 30`) is represented by the relational expression (*UCLRelationalExpression*). The binary relational operation `>` has two operands – two child branches. The first is the navigation expression (`p.age`) and the second is the literal integer value `30` (*UCLLiteralIntegerExpression*). The navigation expression (`p.age`) represented by *UCLStepEntityLexical* is the simple step from the entity *Employee* represented by the variable `p` (*UCLVariableExpression*) to its lexical *age* (*UCLLexical*).

6.4. Conclusion of UCL meta-model

In this chapter, we have defined *UCL meta-model*. It is the model of UCL types, constraints and expressions. The meta-model was defined as a set of class diagrams which were is separated into packages *Types* and *Expressions*. It contains classes which represent all possible types of expressions and all possible UCL expressions.

We have illustrated the usage of the meta-model in the example of a complex expression. The concrete model of a UCL expression (an instance of UCL meta-model) is represented by an abstract syntax tree according UCL meta-model. The last operation of an expression is the root of the syntax tree. Operations which are evaluated at the end are the parent elements of operations which are evaluated at the beginning.

The presented meta-model expresses a different, more detailed view to UCL expressions. It defines the abstract syntax of expressions. The presented UCL meta-model is a suitable form (which represents UCL constraints) for a computer representation and for purposes like the adaptation of UCL expressions when the input model evolves or the derivation of UCL constraints to other constraint languages.

7. Using UCL for UML Class diagrams

In the previous chapters, we have formally defined UCL Data meta-model, in semi-formal way in English text we have defined language UCL for constraints and we have formally defined syntax and model of UCL expressions. Now we have established all the necessary facilities to demonstration and formal definition how to use constraints in UCL for different data models. In this chapter, we apply UCL constraints to UML Class diagrams; and in the next chapter to XML.

Using UCL for UML Class diagrams is trivial task because UCL is based on language OCL for UML Class diagrams. Difference is that UCL is based on more general data model; so for example navigation to *Association classes* is bit different.

First we must formally define model of schemas in UML Class diagrams and we map this model to UCL Data meta-model. Afterwards we semi-formally apply model of UCL to UML Class diagrams; we analyse all possible navigations through elements in UML.

7.1. Notation of the model

Definition 7.1 (Model of UML Class diagram):

UML Class diagram is a structure $\mathcal{M}_{UML} = (Classes_{UML}, AssociationClasses_{UML}, Associations_{UML}, AssociationsEnds_{UML}, Attributes_{UML}, \prec_{\mathrm{UML}})$.

Structure of *classes* is $Classes_{UML} = (C_{UML}, C_{UML}.name, C_{UML}.att, C_{UML}.conAs, C_{UML}.accEnds)$; structure of *associations* is $Associations_{UML} = (A_{UML}, A_{UML}.name, A_{UML}.asClass, A_{UML}.ends)$; structure of *association ends* is $AssociationsEnds_{UML} = (Ae_{UML}, Ae_{UML}.association, Ae_{UML}.name, Ae_{UML}.card, Ae_{UML}.class)$; structure of *attributes* is $Attributes_{UML} = (At_{UML}, At_{UML}.name, At_{UML}.type)$;

where:

- C_{UML} is a finite set of all classes;
- $AssociationClasses_{UML}$ is a set of association classes, it is a subset of classes $AssociationClasses_{UML} \subseteq C_{UML}$;
- A_{UML} is a finite set of all associations;
- Ae_{UML} is a finite set of all association ends;
- At_{UML} is a finite set of all attributes;
- $\prec_{\mathrm{UML}}$ is generalization hierarchy on classes, it is a partial order over C_{UML};

then:

- $C_{UML}.name : C_{UML} \longrightarrow \mathcal{L}$ assigns to each class a name;
- $C_{UML}.att : C_{UML} \longrightarrow \mathcal{P}(Attributes_{UML})$ assigns to each class a set of its attributes;
- $C_{UML}.conAs: C_{UML} \longrightarrow \mathcal{P}(A_{UML})$ assigns to each class a set of associations which are connected to the class;
- $C_{UML}.accEnds : C_{UML} \longrightarrow \mathcal{P}(Ae_{UML})$ assigns to each class a set of association ends which are accessible (by a navigation) from the class;
- $A_{UML}.name : A_{UML} \longrightarrow \mathcal{L}$ assigns to each association a name;
- $A_{UML}.asClass : A_{UML} \longrightarrow (AssociationClasses_{UML} \cup \emptyset)$ assigns to each association one or no association class which is connected to the association;
- $A_{UML}.ends : A_{UML} \longrightarrow \mathcal{P}(Ae_{UML})$ assigns to each association a set of its association ends;
- $Ae_{UML}.name : Ae_{UML} \longrightarrow \mathcal{L}$ assigns to each association end a role name;
- $Ae_{UML}.association : Ae_{UML} \longrightarrow A_{UML}$ assigns to each association end its association;
- $Ae_{UML}.class : Ae_{UML} \longrightarrow C_{UML}$ assigns to each association end a connected class;
- $Ae_{UML}.card : Ae_{UML} \longrightarrow \mathcal{C}$ assigns to each association end the cardinality;
- $At_{UML}.name : At_{UML} \longrightarrow \mathcal{L}$ assigns to each attribute a name;
- $At_{UML}.type : At_{UML} \longrightarrow \mathcal{L}$ assigns to each attribute a type (name of the type).

7.2. Mapping to UCL Data meta-model

In this chapter, we define a mapping of elements of UML Class diagrams to UCL Data meta-model; we formally define how to create a schema in UCL Data meta-model from an existing schema in UML Class diagrams.

Definition 7.2 (Mapping model of UML Class diagram to UCL Data meta-model):

Let be $\mathcal{M}_{UML}$ a schema in UML Class diagrams (Definition 7.1), then mapping of this schema to UCL Data meta-model (Definition 4.14) is schema $\mathcal{M} = (Entities^*, Relations, RelEnds, Lexicals)$, where:

- $E \coloneqq C_{UML}$, set of UML classes is directly mapped to UCL entities;
- $R \coloneqq A_{UML}$, set of UML associations is directly mapped to UCL relations;

- $L \coloneqq At_{UML}$, set of UML class attributes is directly mapped to UCL lexicals;
- $Re \coloneqq Ae_{UML}$, set of UML association ends is directly mapped to UCL relation ends;

where:

- $E.name \coloneqq C_{UML}.name$, name of an UML class is mapped to name of the mapped UCL entity;
- $E.lex \coloneqq C_{UML}.att$, attributes of an UML class are mapped to inner lexicals of the mapped UCL entity;
- $(\forall e \in E)E.inEn(e) \coloneqq \emptyset$ and $(\forall e \in E)E.inRel(e) \coloneqq \emptyset$, there is no relationship between UML elements which is mapped to relationship inner entities or inner relations of an UCL entity, UCL entity has no inner entities or inner relations. All UCL entities and UCL relations are *root entities* and *root relations*;
- $E.conRel \coloneqq C_{UML}.conAs$, connected associations to an UML class are mapped to connected relations to the mapped UCL entity;
- $E.accEnds \coloneqq C_{UML}.accEnds$, accessible association ends from an UML class are mapped to accessible relation ends from the mapped UCL entity;
- $(\forall e \in E)E.neEn(e) \coloneqq \emptyset$, there is no relationship between UML elements which is mapped to relationship neighbouring between UCL entities, UCL entity has no neighbouring entities;
- $E.neRel(e) \coloneqq \begin{cases} e \in AssociationClasses_{UML} \coloneqq \{r \in R | A_{UML}.asClass(r) = e\} \\ else \coloneqq \emptyset \end{cases}$, relationship between an UML association and a connected UML association class is mapped to UCL relation and UCL entity which are in neighbourhood relationship, function $E.neRel(e)$ assigns to a class that is not an association class an empty set and to an association class it assigns set of UCL relations which are connected to this association class (condition $A_{UML}.asClass(r) = e$);
- $\prec_{E.gen} \coloneqq \prec_{\text{UML}}$; $(\forall e_1, e_2 \in E)(e_1 \prec_{E.gen} e_2) \Leftrightarrow (e_1 \prec_{\text{UML}} e_2)$, generalization hierarchy on UML classes is mapped to generalization on the mapped UCL entities;
- $R.name \coloneqq A_{UML}.name$, name of an UML association is mapped to name of the mapped UCL relation;
- $(\forall r \in R)R.lex(r) \coloneqq \emptyset$, UML association cannot contain attributes so UCL relation does not contain any UCL lexicals;
- $R.neEn \coloneqq A_{UML}.asClass$, relationship between an UML association class and a connected UML association is mapped to UCL entity and UCL relation which are in neighbouring relationship, if an UML association is not connected to any UML association class then its mapped UCL relation does not have any neighbouring entity;

- $R.ends := A_{UML}.ends$, association ends of an UML association are mapped to relation ends of the mapped UCL relation;
- $Re.name := Ae_{UML}.name$, role name of an UML association end is mapped to role name of the mapped UCL relation end;
- $Re.relation := Ae_{UML}.association$, UML association of an UML association end is mapped to an UCL relation of the mapped UCL relation end;
- $Re.entity := Ae_{UML}.class$, UML class connected through an UML association end is mapped to an UCL entity which is connected through the mapped UCL relation end;
- $Re.card := Ae_{UML}.card$, cardinality of an UML association end is mapped to cardinality of the mapped UCL relation end;
- $L.name := At_{UML}.name$, name of an UML attribute is mapped to name of the mapped UCL lexical;
- $L.type(l) := \begin{cases} At_{UML}.type(l) = "Integer" := Integer \\ At_{UML}.type(l) = "Real" := Real \\ At_{UML}.type(l) = "Boolean" := Boolean \\ else := String \end{cases}$, type of an UML attribute (of string type) is mapped to the type of the mapped UCL lexical (item of the set of predefined basic type according definition Definition 4**.3**) according this function. Type with name *"Integer"* is mapped to the basic type Integer, etc. and all other types are mapped to the basic type *String*.

We have formally defined how to create a schema in UCL Data meta-model but it is also important to check all restrictions of UCL Data meta-model of distinct names which are presented in chapters 4.5.7 – 4.5.12. Otherwise the created schema in UCL Data meta-model will be not valid, it will contain duplicate navigation names (e.g. the same name for an *association end* and for connected *association class*) and created UCL expressions can be ambiguous.

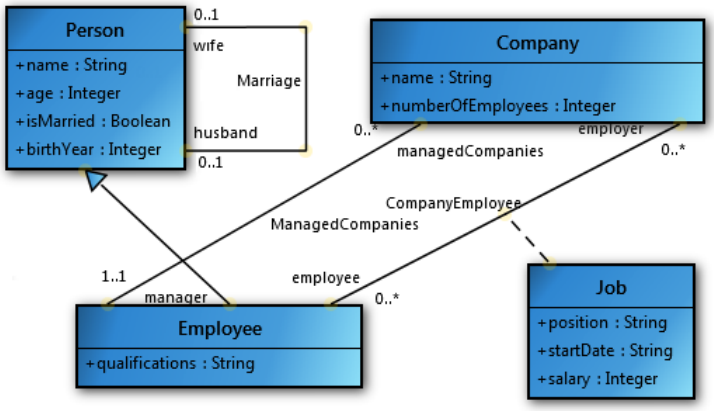


Figure 7.1: Sample UML Class diagram

In the Figure 7.1: Sample UML Class diagram, there is example of UML Class diagram. It contains 4 classes with attributes and 3 associations. Class *Job* is also an association class connected to the association *CompanyEmployee*; association *Marriage* is a recursive association; class *Employee* is the child class of class *Person*, it inherits its attributes and connected associations.

When we map this schema to UCL Data meta-model according Definition 7.2, we create a schema with 4 UCL entities with UCL lexicals and 3 UCL relations. Mapped entity *Employee* is the child entity of entity *Person*; entity *Job* is in neighbourhood relationship with relation *CompanyEmployee*. All these entities and relations are *root entities* and *root relations*. Types of UML attributes are mapped to UCL basic types.

7.3. UCL constraints over UML Class diagrams

7.3.1. Context of constraints

UCL constraints over UML can be expressed in context of all classes and associations. All UML classes are *UCL root entities* and all associations are *UCL root relations*. We can define contexts of constraints:

```
Context Person
inv: self…

Context Job
inv: self…

Context CompanyEmployee
inv: self…
```

7.3.2. Expressions

In UCL constraints over UML are allowed definitions of "def" and "let" variables and using of their values in expressions. In UCL expressions are allowed literal expressions of all basic types, expression `if-then-else`, all standard unary and binary operations over basic types, all collection operations (e.g. `:size()`) and all collection expressions (e.g. `:forAll(…)`).

7.3.3. Navigation

Not all UCL navigation expressions are supported, if some kind of relationship in UCL Data meta-model mapped from UML Class diagram is not

defined then kind of navigation expressions which use this relationship is not used. Navigation expression can start using expression `self` in an entity or in a relation; or it can start with a reference to a variable.

7.3.4. Simple steps navigation

UCL defines 5 kinds of *simple step* navigation expressions (operator dot and followed by a name of a connected element) from an entity; in this way it is possible to navigate UML attribute of UML Class:

```
Context Person
inv: self.age…

Context Person
inv: self.age > 0
inv: self.isMarried implies self.age >= 18
```

From an UML association class it is possible to navigate the connected UML association through the name of the UML association:

```
Context Job
inv: self.CompanyEmployee…

Context Job
inv: self.position = "baker"
     implies self.CompanyEmployee.employer.name <> "IBM"
```

Other kinds of *simple steps* from an entity are not used. It is not possible to navigate inner entities and inner relations of an entity because an entity cannot contain inner entities and inner relations. It is not possible to navigate a neighbouring entity of an entity because this relationship is not defined in UML mapped to UCL.

UCL defines 3 kinds of *simple step* navigation from a relation. From an UML association we can navigate a connected UML association class using the name of the association class:

```
Context CompanyEmployee
inv: self.Job…

Context CompanyEmployee
inv: self.Job.salary > 100
```

It is possible to navigate to a connected class through a role name of an association end:

```
Context Marriage
inv: self.wife…

Context Marriage
inv: self.wife.age >= 18
```

It is not possible to navigate UCL lexicals from an UCL relation because UML association cannot have any UML attributes.

7.3.5. Relation step through an association

In context of a *class*, we can navigate to a connected *class* through an *association* using operator `->` and name of an opposite *association end* of the *association*. The example expression starts in class *Employee*, goes through association *CompanyEmployee* and ends in class *Company*. The type of expression is *Sequence (Company)*:

```
Context Employee
inv: self->employer…

Context Employee
inv: self->employer:forAll(comp | comp.numberOfEmployees > 0)
```

7.3.6. Relation stop to an association

In context of a *class*, we can navigate to a connected *association* using operator `:>` and name of an opposite *association end* of the *association*. The example expression starts in class *Employee* and ends in association *CompanyEmployee*. The type of expression is *Sequence (CompanyEmployee)*:

```
Context Employee
inv: self:>employer…

Context Employee
inv: self:>employer:collect(ce | ce.Job.salary):sum() > 1000
```

7.3.7. Generalization

The generalization on UML classes in mapped to the generalization on UCL entities. In context of an UML class, we can navigate connected elements of parent classes. We can navigate attributes, connected associations and in case of an association class also the connected association.

It is possible to express:

```
Context Employee
inv: self.age >= 0
inv: self->husband.age >= 18

Context Company
inv: self->employee:forAll(em | em.age >= 15)
```

8. Using UCL for XML schemas

In the previous chapter, we have applied UCL for model of UML Class diagrams. It was not difficult task because UCL is based on OCL which is constraint language for UML; the difference is that UCL is based on more general UCL Data meta-model and it has more kinds of navigation expressions over the data model.

In this chapter, we apply UCL to XML data; we apply UCL over schemas of XML documents in PSM layer of XSEM model. In a similar way as in the previous chapter, we define XSEM PSM schema and we map it to UCL Data meta-model. Then we apply model of UCL to XSEM and we analyse all possible navigation expressions over XSEM schemas.

8.1. Notation of the model

Definition 8.1 (Model of XSEM PSM schema):

XSEM PSM schema is a structure:

$\mathcal{M}_{XSEM} = (Classes_{XSEM}, Attributes_{XSEM}, Associations_{XSEM}, AsEnds_{XSEM}, ContModels_{XSEM})$.

Structure of classes is $Classes_{XSEM} = (C_{XSEM}, C_{XSEM}.name, C_{XSEM}.att, C_{UML}.conAs, C_{UML}.accEnds)$, structure of attributes is $Attributes_{XSEM} = (At_{XSEM}, At_{XSEM}.name, At_{XSEM}.dataType, At_{XSEM}.xForm)$, structure of associations is $Associations_{XSEM} = (A_{XSEM}, A_{XSEM}.parent, A_{XSEM}.child, A_{XSEM}.name, A_{XSEM}.card)$, structure of association ends is $AsEnds_{XSEM} = (Ae_{XSEM}, Ae_{XSEM}.association, Ae_{XSEM}.part)$, structure of content models is $ContModels_{XSEM} = (CM_{XSEM}, CM_{XSEM}.type)$;

where:

- C_{XSEM} is a finite set of classes;
- At_{XSEM} is a finite set of attributes;
- A_{XSEM} is a finite set of associations;
- Ae_{XSEM} is a finite set of association ends;
- CM_{XSEM} is a finite set of content models;

then:

- $C_{XSEM}.name : C_{XSEM} \longrightarrow \mathcal{L}$ assigns to each class a name;

- $C_{XSEM}.att : C_{XSEM} \longrightarrow \mathcal{P}(At_{XSEM})$ assigns to each class a set of its attributes;
- $C_{XSEM}.conAs: C_{XSEM} \longrightarrow \mathcal{P}(A_{XSEM})$ assigns to each class a set of all parent and child associations which are connected to the class;
- $C_{XSEM}.accEnds : C_{XSEM} \longrightarrow \mathcal{P}(Ae_{XSEM})$ assigns to each class a set of association ends which are accessible from the class;
- $At_{XSEM}.name : At_{XSEM} \longrightarrow \mathcal{L}$ assigns to each attribute a name;
- $At_{XSEM}.dataType : At_{XSEM} \longrightarrow \mathcal{L}$ assigns to each attribute a data type (name of the data type).
- $At_{XSEM}.xForm : At_{XSEM} \longrightarrow \{element, attribute\}$ assigns to each attribute an XML form (a simple element or an attribute) of the XSEM attribute;
- $A_{XSEM}.name : A_{XSEM} \longrightarrow \mathcal{L}$ assigns to each association a name;
- $A_{XSEM}.parent : A_{XSEM} \longrightarrow Ae_{XSEM}$ assigns to each association an association end which is connected to association's parent class or content model;
- $A_{XSEM}.child : A_{XSEM} \longrightarrow Ae_{XSEM}$ assigns to each association an association end which is connected to association's child class or content model;
- $A_{XSEM}.card : A_{XSEM} \longrightarrow \mathcal{C}$ assigns to each association cardinality;
- $Ae_{XSEM}.association : Ae_{XSEM} \longrightarrow A_{XSEM}$ assigns to each association end its association;
- $Ae_{XSEM}.part : Ae_{XSEM} \longrightarrow C_{XSEM} \cup CM_{XSEM}$ assigns to each association end a connected participant element (a class or a content model);
- $CM_{XSEM}.type : CM_{XSEM} \longrightarrow \{sequence, choice, set\}$ assigns to each content model its type (a sequence, a choice or a set).

8.2. Mapping to UCL Data meta-model

Now, we define a mapping of elements of an existing XSEM PSM schema to a schema in UCL Data meta-model.

Definition 8.2 (Mapping model of XSEM PSM schema to UCL Data meta-model):

Let be $\mathcal{M}_{XSEM}$ a XSEM PSM schema (Definition 8.1), then mapping of this schema to UCL Data meta-model (Definition 4.14) is schema $\mathcal{M} = (Entities^*, Relations, RelEnds, Lexicals)$;

where:

- $E \coloneqq C_{XSEM} \cup CM_{XSEM}$, classes and content models are directly mapped to UCL entities;
- $R \coloneqq A_{XSEM}$, associations are directly mapped to UCL relations;
- $Re \coloneqq Ae_{XSEM,}$ association ends are directly mapped to UCL relations ends;
- $L \coloneqq At_{XSEM}$, attributes are directly mapped to UCL lexicals;

where:

- $E.name(e) \coloneqq \begin{cases} e \in C_{XSEM} & \coloneqq C_{XSEM}.name(e) \\ e \in CM_{XSEM} & \coloneqq A_{XSEM}.name\big(parentAssoc(e)\big) \end{cases}$;

  In the first case ($e \in C_{XSEM}$), name of a class is mapped to name of the mapped UCL entity (*e*).

  In the second case ($e \in CM_{XSEM}$), a content model does not have any name, name for the mapped UCL entity is taken from the name of the content model's parent association. Function $parentAssoc : C_{XSEM} \cup CM_{XSEM} \rightarrow A_{XSEM}$ assigns to a class and a content model the parent association:

  $parentAssoc(e) \coloneqq (a \in A_{XSEM})\big(Ae_{XSEM}.part\big(A_{XSEM}.child(a)\big) = e\big)$;

  $E.lex(e) \coloneqq \begin{cases} e \in C_{XSEM} & \coloneqq C_{XSEM}.att(e) \\ e \in CM_{XSEM} & \coloneqq \emptyset \end{cases}$; In the first case, attributes of a XSEM class are mapped to inner lexicals of the mapped UCL entity; in the second case, a content model and the mapped UCL entity does not have attributes;

  $E.inEn(e) \coloneqq \begin{cases} e \in C_{XSEM} & \coloneqq childEntities(e) \cup contModelChilds(e) \\ e \in CM_{XSEM} & \coloneqq \emptyset \end{cases}$; In the second case, the entity mapped from a content model does not have inner entities. In the first case, inner entities of an entity are child entities connected through an association ($childEntities(e)$); and child entities of a child content model of the entity *e* ($contModelChilds(e)$).

  Where function $childEntities: C_{XSEM} \rightarrow C_{XSEM}$ assigns to an entity set of entities (*f*) for which exist an association which connects entities *e* and *f*:

  $childEntities(e) \coloneqq \{f \in C_{XSEM} \mid isAssoc(e, f)\}$;

  Predicate $isAssoc : C_{XSEM} \times C_{XSEM} \rightarrow \{true, false\}$ gets for a pair of entities (*e* and *f*) if there exist an association (*a*) whose parent ends is connected to the entity *e* and child end to the entity *f*:

  $isAssoc(e, f) \coloneqq (\exists a \in A_{XSEM}) \begin{pmatrix} \big(Ae_{XSEM}.part\big(A_{XSEM}.parent(a)\big) = e\big) \\ \wedge \big(Ae_{XSEM}.part\big(A_{XSEM}.child(a)\big) = f\big) \end{pmatrix}$;

  Function $contModelChilds : C_{XSEM} \rightarrow C_{XSEM}$ assigns to a XSEM class (*e*) set of its grandchildren (*g*) which are child of content model (*f*):

  $contModelChilds(e) :=$

$\left\{g \in C_{XSEM} \middle| \left((\exists f \in CM_{XSEM}) \left(isAssoc(e,f) \wedge isAssoc(f,g)\right)\right)\right\}$.

- $(\forall e \in E)E.inRel(e) := \emptyset$, UCL entities have no inner relations. All UCL relations which are mapped from XSEM associations are *root relations* with an empty name;
- $E.conRel := C_{XSEM}.conAs$, connected associations to a class are mapped to connected relations to the mapped UCL entity;
- $E.accEnds := C_{XSEM}.accEnds$, accessible association ends from a class are mapped to accessible relation ends from the mapped UCL entity;
- $(\forall e \in E)E.neEn(e) := \emptyset$ and $(\forall e \in E)E.neRel(e) := \emptyset$, there is no relationship between XSEM elements which is mapped to relationship neighbouring between two UCL entities or between an UCL entity and UCL relation;
- $\prec_{E.gen} := \emptyset$, generalization hierarchy is not defined;
- $R.name := \lambda$, all UCL relations (they all are *root relations*) has an empty name;
- $(\forall r \in R)R.lex(r) := \emptyset$, so UCL relations do not contain any UCL lexicals because XSEM association cannot contain attributes;
- $(\forall r \in R)R.neEn(r) := \emptyset$, there is no relationship between XSEM elements which is mapped to relationship neighbouring between an UCL entity and UCL relation;
- $R.ends(r) := \{A_{XSEM}.parent(r), A_{XSEM}.child(r)\}$, the parent and the child association ends of an association are mapped to relation ends of the mapped UCL relation;
- $Re.name(x) := \begin{cases} parentEnd(x) := "parent" \\ else \qquad := A_{XSEM}.name\left(Ae_{XSEM}.association(x)\right) \end{cases}$;

  where $parentEnd : Ae_{XSEM} \rightarrow \{true, false\}$ gets if the association end is connected to the parent class or content model; it is a predicate, a set of association ends whose are connected to the parent class or content model:

  $parentEnd := \left\{x \in Ae_{XSEM} \middle| A_{XSEM}.parent\left(Ae_{XSEM}.association(x)\right) = x\right\}$;

  XSEM association ends do not have a name. Function $Re.name$ assigns to an association end to the parent element name *"parent"* and to an association end to the child element name of the association (expression $A_{XSEM}.name$);
- $Re.relation := Ae_{XSEM}.association$, an association of an association end is mapped to an UCL relation of the mapped UCL relation end;

- $Re.entity \coloneqq Ae_{XSEM}.part$, participant (class or content model) connected through an association end is mapped to an UCL entity which is connected through the mapped UCL relation end;
- $Re.card(x) \coloneqq \begin{cases} parentEnd(x) \coloneqq (1,1) \\ else \quad \coloneqq A_{XSEM}.card\left(Ae_{XSEM}.association(x)\right) \end{cases}$;

  XSEM association ends do not have cardinality. Function $Re.card$ assigns to an association end to the parent element cardinality $(1,1)$ because an XML element can be placed in exactly one parent element. And to an association end to the child element it assigns cardinality of the association (expression $A_{XSEM}.card$);
- $L.name \coloneqq At_{XSEM}.name$, name of an attribute is mapped to name of the mapped UCL lexical;
- $L.type(l) \coloneqq \begin{cases} At_{XSEM}.dataType(l) = "xs:integer" \coloneqq Integer \\ At_{XSEM}.dataType(l) = "xs:double" \coloneqq Real \\ At_{XSEM}.dataType = "xs:boolean" \coloneqq Boolean \\ else \quad \coloneqq String \end{cases}$, data type of an XSEM attribute (of string type) is mapped to the type of the mapped UCL lexical (item of the set of predefined basic type according definition Definition 4**.3**) according this function. Type with name *"xs:integer"* is mapped to the basic type Integer, etc. and all other types are mapped to the basic type *String*.

Schema in UCL Data meta-model which is created according Definition 8.2 is valid only if all restriction defined in chapters 4.5.7 – 4.5.12 are satisfied. Otherwise there can be duplicate navigation names in the model and ambiguous UCL navigation expressions can be created.

Figure 8.1 shows a sample XSEM PSM diagram. There are two XML trees with root classes *Company* and *Person*. Classes contain attributes, attributes with XML form *simple element* are prefixed by a character "-", attributes of XML form attribute are prefixed by "@". Class *Company* contains as child a content model a *sequence*, class *Person* contains a *choice*. Associations have a name and cardinality.

Figure 8.2 shows mapping a sample XSEM PSM diagram to UCL Data meta-model according Definition 8.2. Root classes *Company* and *Person* are mapped to two root entities *Company* and *Person*. For simplicity, the UCL lexicals are not marked. Company connected is through the association *manager* connected to its child class *Manager*; in UCL entity *Person* has the inner entity *Manager* and they are connected through the relation with relation end names *parent* and *manager*. Child classes *Employee* and *Job* of content model sequence are inner entities of entity

Company. Class *Person* contains class *Cars* which contains class *Car*; this relationship is to UCL mapped to entities *Person* which contains inner entity *Cars* which contains inner entity *Car*; entities are connected by UCL relations too.

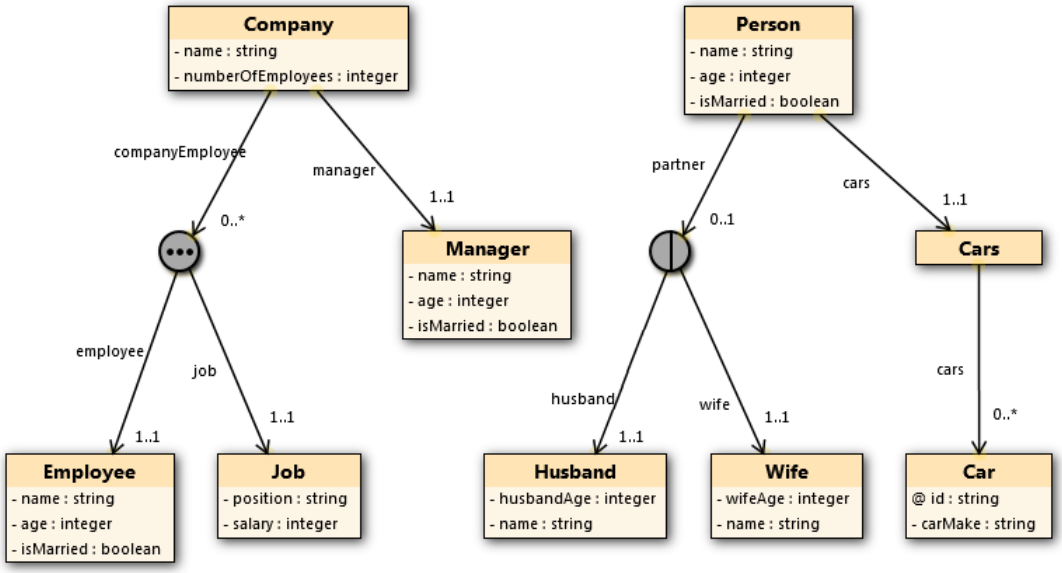


Figure 8.1: Sample XSEM PSM diagram

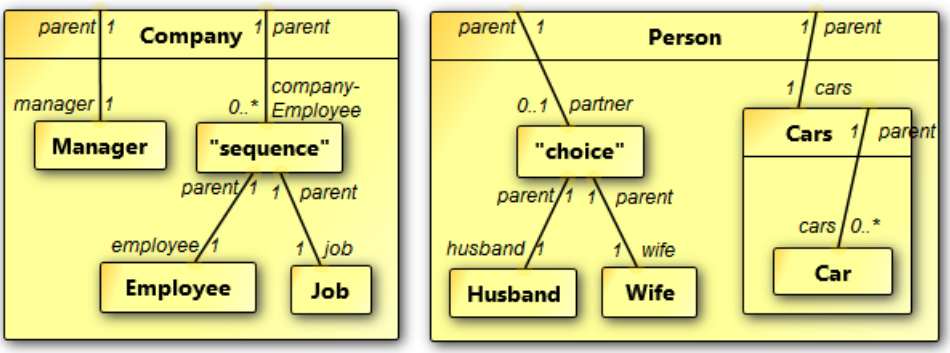


Figure 8.2: Mapping of XSEM PSM diagram to UCL Data meta-model

8.3. UCL constraints over XSEM PSM schemas

8.3.1. Context of constraints

UCL constraints over XSEM can be expressed in context of all classes and content models; they cannot be expressed over associations. To specify context for element which is not a root entity, we must specify the full path to the element from its root entity through all inner entities:

```
Context Company
inv: self…
```

```
Context Company::Job
inv: self…

Context Company::companyEmployee
inv: self…

Context Person::Cars::Car
inv: self…
```

8.3.2. Expressions

UCL constraints over XSEM support definitions of "def" and "let" variables and references to their values in UCL expressions. Literal expressions of all basic types, expression `if-then-else`, all standard unary and binary operations over basic types, all collection operations (e.g. `:size()`) and all collection expressions (e.g. `:forAll(…)`) are allowed in standard syntax and semantics.

The generalization on entities is in the mapping from XSEM schemas not defined.

8.3.3. Navigation

Not all kinds of UCL navigation expressions are supported because not all kinds of relationship in UCL Data meta-model mapped from XSEM are defined. Relationships between entities and between an entity and a relation are not defined. Navigation expression can start in an entity, in a content model and in a variable.

8.3.4. Simple steps navigation

UCL defines 5 kinds of *simple step* navigation expressions (operator dot followed by a name of a connected element) from an entity. It is possible to navigate attribute of a class:

```
Context Person
inv: self.age…

Context Person
inv: self.age > 0
```

It is possible to navigate inner class and inner content model of a class:

```
Context Company
inv: self.Manager…
inv: self.Job…
inv: self.companyEmployee…
```

```
Context Company
inv: self.Job.salary > 0
```

It is not possible to navigate inner relations of an entity; entities do not have inner relations, all relations are root relations without a name. It is not possible to navigate a neighbouring entity and a neighbouring relation because this relationship is not defined in the mapping from XSEM to UCL Data meta-model.

UCL defines 3 kinds of *simple step* navigations from relations. From a relation we can navigate to the connected entity (class or content model) using a name of a relation end name. To navigate the child element we use the name of the association and to navigate the parent element we use name "parent" (according definition of the mapping XSEM to UCL Data meta-model):

```
(expression ending in association "manager").manager...
(expression ending in association "manager").parent...
```

```
Context Company
def variable = self:>manager
inv: variable.manager.age >= 18
inv: variable.parent.numberOfEmployees >= 1
```

It is not possible to navigate lexicals of a relation because an association cannot have attributes.

8.3.5. Relation step through an association

In context of a *class* or a *content model*, we can navigate to a connected *class* or *content model* through an *association* using operator `->` and name of an opposite *association end* of the *association*. *Relation end name* to the child element is *name* of the *association* and *relation end name* to the parent element is "parent".

The expression starts in class *Person::Cars*, goes through association *cars* to class *Car*. The type of expression is *Sequence (Person::Car ::Car)*:

```
Context Person::Cars
inv: self->cars…
```

```
Context Employee
inv: self->employer:forAll(comp | comp.numberOfEmployees > 0)
```

The second example starts in class *Person::Cars::Car* and goes through navigation cars (using the relation ends name *"parent"*) to class *Person::Cars*.

```
Context Person::Cars::Car
inv: self->parent…

Context Person::Cars::Car
inv: self->parent->parent.age >= 18

Context Person
inv: self.age < 18
     implies self->cars->cars:forAll(c | c.carMake = "moped")
```

8.3.6. Relation stop to an association

Although the defined mapping XSEM PSM schemas to UCL does not support definitions of constraints in the context of associations, but it is possible to use *relations stop* navigation expressions which navigate to associations.

In context of a *class* or *content model* we can navigate to a connected *association* using operator `:>`. The example expression starts in class *Company* and ends in association *manager*. The type of expression is *Sequence (manager)*:

```
Context Company
inv: self:>manager…

Context Company
inv: self:>manager.manager.age >= 20
```

9. Mapping and deriving constraints

In chapter 6, we presented UCL meta-model (model of UCL types and constraints). As we mentioned, it is a suitable form to formal representation of UCL constraints and to deriving UCL constraints to a different model or to a different constraint language. A model (according UCL meta-model) of a UCL expression represents its abstract syntax tree. An operation of an expression which is evaluated at the end is the root element of the syntax tree of the expression.

In this chapter, we define deriving of UCL expressions for XML to Schematron constraints. We formally define the mapping between different models of different meta-models. And according this mapping between different models we define deriving of UCL constraints over one model to UCL constraints over a different mapped model.

First we define a collection of UCL constraints over a schema.

Definition 9.1 (UCL constraints over a schema):

Let be $\mathcal{M} = (Entities^*, Relations, RelEnds, Lexicals)$ a schema of UCL Data meta-model.

Then a *set of UCL expressions* over the schema $\mathcal{M}$ is a set $Expressions_{\mathcal{M}}$ which contains instances of the class $UCLBaseExpression$ (Figure 6.4) with navigation expressions over the schema $\mathcal{M}$.

A *set of variables definitions* over the schema $\mathcal{M}$ is the set $VarDefinitions_{\mathcal{M}}$ which contains instances of classes $UCLCollectionVariable$ and $UCLVariableDefinition$ (Figure 6.5). The attribute $Expression$ of $UCLVariableDefinition$ defines the variable's value; the value of this attribute must be member of the set $Expressions_{\mathcal{M}}$, the set of expressions over schema $\mathcal{M}$.

A *collection of UCL context definitions* over the schema $\mathcal{M}$ is the a $UclContexts_{\mathcal{M}}$ which contains all instances of the class $UCLContextDefinition$ (Figure 6.3). All variables definitions must be member of $VarDefinitions_{\mathcal{M}}$, all invariant expressions must be members of $Expressions_{\mathcal{M}}$. And context type of the context must be $UCLEntityType$ over the set of entities in schema $\mathcal{M}$ of $UCLRelationType$ in the schema $\mathcal{M}$ (Figure 6.1).

9.1. Deriving UCL for XML to Schematron

In this subchapter, we define the algorithm *UclToSchematron* which derives UCL constraints for XML to constraint rules in Schematron. We define the recursive function *ExpressionToSchematron* which derives all kinds of UCL expressions (all kinds of UCL expressions according UCL meta-model) to Schematron rules. The function *TypeToSchematron* derives context type to XPath navigation to the type.

9.1.1. Context of constraints

Definition 9.2 (Deriving UCL to Schematron):

The function *UclToSchematron* generates from a set of UCL constraints on XML $UclContexts_{\mathcal{M}}$ the constraint rules in Schematron.

```
function
UclToSchematron(contextDefs : list of UCLContextDefinition)
{
  print <?xml version="1.0" encoding="utf-8"?>
  print <schema xmlns="http://www.ascc.net/xml/schematron">
  print <pattern name="Schematron rules">

  //a) all blocks of context definitions
  foreach (contextDef in contextDefs)
  {
    typeName := TypeToSchematron(contextDef.ContextType)
    print <rule context="typeName">

    //b) all "def" variable definitions
    foreach (variable in contextDef.DefVariablesDefinitions)
    {
      value := ExpressionToSchematron(variable.Expression)
      print <let name="variable.Name" value="value" />
    }

    //c) all context's invariants
    foreach (invariant in contextDef.InvariantExpressions)
    {
      name := invariant.Name
      expr := ExpressionToSchematron(invariant.Expression)
      print <assert test="expr">name</assert>
    }

    print </rule>
  }
```

```
  print </pattern>
  print </schema>
}
```

The function first prints the XML header and Schematron root elements *schema* and *pattern*. In the first loop *(a)* it parses all blocks of UCL context definitions.

For each block it creates the XML element <*rule*> with its invariants. In element <*rule*>, there is the attribute *context* with XPath navigation to the context entity. This XPath navigation is derived by function *TypeToSchematron*.

The second loop *(b)* derives all "def" variables definition in the context block to Schematron *let* variables. Schematron can define a variable in the scope of an element *rule*. A variable definition is represented by an element `<let name="name" value="value" />`. To reference the variable we must prefix the variable name with `$`, e.g. `$variableName`. The variable name in Schematron is assigned from variable name in UCL. Expression of the variable's value is derived to Schematron by function *ExpressionToSchematron*.

The third loop *(c)* derives all invariants in the context block to Schematron rules. Invariants are in Schematron represented by an element `<assert test="expression">name</assert>`. The invariant's name is the UCL invariant's name. Attribute *expression* is the constraint expression which must be satisfied for each instance of the context element type. It is derived from UCL by function *TypeToSchematron*.

9.1.2. Types

The function *TypeToSchematron* converts UCT entity type over XML to XPath navigation expression. An XSEM class or a content model is represented by an inner entity of the parent XSEM class or content model. XPath expression representing the type begins with `/` and the name of the root UCL entity. It follows by `/` and the name of the inner entity. And it ends by the name of the searched UCL entity which represents converted XSEM class or content model. But a content model does not represent real XML element, so name of a content model is not added to the created XPath expression.

Definition 9.3 (Type name of UCL entity in XPath):

```
function TypeToSchematron(uclEntity : UCLEntity)
{
  //search parent entity (if uclEntity is inner entity of e)
  parentEntities := {e | uclEntity ∈ E.inEn(e)}

  //uclEntity is a root entity
  if (parentEntities == empty set)
    return /E.name(uclEntity)

  parentEntity := e ∈ parentEntities
  parentEntityName := TypeToSchematron(parentEntity)

  if (uclEntity is XSEMContentModel)
    return parentEntityName

  return parentEntityName/E.name(uclEntity)
}
```

E.g. for UCL entities mapped from XSEM diagram in Figure 8.1, function assigns: for root class *Company* expression */Company*, for class *Car* expression */Person/Cars/Car* and for content model *partner* (in class *Person*) expression */Person*. Content models does not represent real XML element, only a content of their parent element.

9.1.3. Expressions

In this subchapter we define the function *ExpressionToSchematron*. It derives all kinds of UCL expressions (expressions according UCL model in chapter 6) to XPath expressions for Schematron rules.

Definition 9.4 (Deriving UCL expressions to XPath expressions):

The function *ExpressionToSchematron* assign to an UCL expression *uclExpression* from the set $Expressions_{\mathcal{M}}$ an XPath expression. According kind of UCL expression it assigns:

- If *uclExpression* is a literal value (*UCLLiteral-Expression*); the function assigns directly the literal value of *uclExpression*.

  An XPath expression can contain literal values of basic types *real*, *integer*, *string* and *boolean*.

- If *uclExpression* is a unary operator (*UCLUnaryBaseOperation-Expression*): `+ expression`, `- expression`, `not (expression)`, `(expression)`;

the function gets an XPath expression which is the same as the UCL expression. The function calls recursive the function *ExpressionToSchematron* to the operand `op`.

XPath defines unary operators `+`, `-`, `not` and parentheses.

- If *uclExpression* is a multiplicative operation (*UCLBinaryMultiplicative-Expression*) `op1 * op2` or `op1 / op2`; the function gets XPath expression `op1 * op2` or `op1 div op2`. The function calls recursive function *ExpressionToSchematron* to the operands `op1` and `op2`.
- If *uclExpression* is an additive operation (*UCLBinaryAdditive-Expression*) `op1 + op2` or `op1 - op2`; the function gets XPath expression `op1 + op2` or `op1 - op2`. The function calls recursive the function *ExpressionToSchematron* to the operands `op1` and `op2`.
- If *uclExpression* is a relational operation (*UCLBinaryRelational-Expression*) `op1 = op2`, `op1 != op2`, `op1 < op2`, `op1 > op2`, `op1 <= op2`, or `op1 >= op2`; the function gets XPath expression `op1 = op2`, `op1 != op2`, `op1 < op2`, `op1 > op2`, `op1 <= op2`, or `op1 >= op2`. The function calls recursive the function *ExpressionToSchematron* to the operands `op1` and `op2`.

  Characters "<" and ends with ">" must be replaced by XML entities "<" and ">".
- If *uclExpression* is a logical operation (*UCLBinaryLogical-Expression*) `op1 or op2`, `op1 and op2`, `op1 implies op2` or `op1 xor op2`; the function gets XPath expression `op1 or op2`, `op1 and op2`, `not(op1) or op2`,
  (`op1 and not(op2)) or (not(op1) and (op2))`. The function calls recursive the function *ExpressionToSchematron* to the operands `op1` and `op2`.

  XPath does not define operations *implies* and *xor*. They are replaced by equivalent expressions.
- If *uclExpression* is an *if-then-else* expression (*UCLIf-Expression*); the function does not get any result. XPath does not define *if-then-else* expression.

- If *uclExpression* is a "let" variable definition (*UCLets-Expression*); the function does not get any result. Schematron cannot define a variable in a scope of an XPath expression. It is possible to use "def" variable definitions.
- If *uclExpression* is *self* expression (*UCLSelfEntity-Expression*); the function gets XPath expression *dot* ("`.`") which refers to the current node.

  An expression *self* can refer only an UCL entity because context cannot be expressed only over an UCL relation.
- If *uclExpression* is a reference to a "def" variable definition (*UCLVariable-Expression* which refers *UCLVariableDefinition*); the function gets the variable's name prefixed by `$`, e.g. `$variable`.
- If *uclExpression* is a reference to a variable in a collection expression; (*UCLVariable-Expression* which refers *UCLCollectionVariable*); the function gets XPath expression *dot* ("`.`") which refers to the current node in context of a collection expression. XPath current node refers to the context of condition expression inside the collection expression.
- If *uclExpression* is a *relation step* expression *`expression`*`->R` from entity *A* through relation end *R* to entity *B* (*UCLConnectedRelationStep-Expression*), *`expression`* is the source expression;

  In the case of navigation from class *A* to child class *B*; the function gets expression *`expression`*`/B`. The expression navigates to the child node;

  In the case of navigation from class *A* to parent class *B*; the function gets expression *`expression`*`/..`. The expression navigates to the parent node;

  In the case of navigation from content model *A* to parent entity *B*; the function gets only the source expression *`expression`*. A content model does not represent real XML node;

  In the case of navigation from class *A* to child content model *B*; the function gets only the source expression *`expression`*. A content model does not represent real XML node;

  The function calls recursive the function *ExpressionToSchematron* to the source expression *`expression`*.

  E.g. constraint in UCL and in Schematron:

```
Context Company
inv: self->manager.age >= 20
```

```
<rule context="/Company">
     <assert test="./Manager/age &gt;= 20"></assert>
</rule>

Context Person::Cars::Car
inv: self->parent->parent.age >= 18

<rule context="/Person/Cars/Car">
     <assert test="./../../age &gt;= 18"></assert>
</rule>
```

- If *uclExpression* is a *simple step* expression `expression.B` from XSEM class *A* to its inner XSEM attribute *B* (*UCLStepEntityLexical*), `expression` is the source expression:

  In the case if *XML form* of attribute *B* is *XML simple element*; the function gets expression `expression/B`. The expression navigates to the child XML node;

  In the case if *XML form* of attribute *B* is *XML attribute*; the function gets expression `expression/@B`. The expression navigates to the XML attribute;

  The function calls recursive the function *ExpressionToSchematron* to the source expression `expression`.

```
Context Company
inv: self.numberOfEmployees >= 0

<rule context="/Company">
     <assert test="./numberOfEmployees &gt;= 0"></assert>
</rule>

Context Person::Cars::Car
inv: self.id != "0"

<rule context="/Person/Cars/Car">
     <assert test="./@id != '0'"></assert>
</rule>
```

- If *uclExpression* is a *simple step* expression `expression.B` from XSEM class *A* to its inner XSEM class *B* (*UCLStepEntityInnerEntity*), `expression` is the source expression; the function gets expression `expression/B`. The expression navigates to the child XML node;

  The function calls recursive the function *ExpressionToSchematron* to the source expression `expression`.

```
Context Company
inv: self.Job.salary > 0

<rule context="/Company">
     <assert test="./Job/salary &gt; 0"></assert>
</rule>
```

- Other kinds of *simple step* expressions are not supported to by UCL over XML or it is not possible to derive them to an XPath navigation expression.

- If *uclExpression* is a collection operation:

  In the case of operation *size* `collection:size()` (*UCLSize-Expression*); the function gets an XPath aggregate the function `count (collection)`;

  In the case of operation *sum* `collection:sum()` (*UCLSum-Expression*); the function gets an XPath aggregate the function `sum (collection)`;

  In the case of operation *isEmpty* `collection:isEmpty()` (*UCLIsEmpty-Expression*); the function gets an XPath sequence the function `empty (collection)`;

  In the case of operation *notEmpty* `collection:notEmpty()` (*UCLNotEmpty-Expression*); the function gets an XPath sequence the function `exists (collection)`;

  In the case of operation *max* `collection:max()` (*UCLMax-Expression*); the function gets an XPath aggregate the function `max (collection)`;

  In the case of operation *min* `collection:min()` (*UCLMin-Expression*); the function gets an XPath aggregate the function `min (collection)`;

  In the case of operation *asSet* `collection:asSet()` (*UCLAsSet-Expression*); the function gets an XPath sequence the function `distinct-values (collection)`;

  The function calls recursive the function *ExpressionToSchematron* to the source expression `collection` which returns a collection;

```
Context Company
inv: self.numberOfEmployees
     = self->companyEmployee:collect(ce | ce->employee):size()

<rule context="/Company">
     <assert test="./numberOfEmployees=count(./Employee)">
     </assert>
</rule>
```

- If *uclExpression* is a collection expression:

In the case of expression *select* (*UCLSelect-Expression*) `collection:select(variable | condition)`; the function gets an XPath expression with a predicate `collection[condition]`. This predicate selects from the collection items that satisfy the condition.

In the case of expression *reject* (*UCLReject-Expression*) `collection:reject(variable | condition)`; the function gets an XPath expression with a predicate `collection[not(condition)]`. This predicate selects from the collection items that do not satisfy the condition.

In the case of expression *collect* (*UCLCollect-Expression*) `collection:collect(variable | navigation)`; the function gets an XPath navigation expression `collection/navigation`. This expression navigates according the expression *navigation*.

In the case of expression *exists* (*UCLExists-Expression*) `collection:exists(variable | condition)`; the function gets an XPath sequence function with a predicate `exists(collection[condition])`. This predicate finds if there is at least one item that satisfies the condition.

In the case of expression *forAll* (*UCLForAll-Expression*) `collection:forAll (variable | condition)`; the function gets a composite XPath expression:

`count(collection[condition]) = count(collection).`

XPath does not have expressive power to find if all items in a collection satisfy a condition. This complex example get if size of a collection is equal to size of a collection with items that satisfy the condition.

The function calls recursive the function *ExpressionToSchematron* to the source expression `collection` and also to the condition expression `condition`.

Collection expressions derived to Schematron do not use collection variables from UCL. Collection variables refer to the current node, they are derived to a dot (".").

```
Context Person
inv:
  self.age < 18
  implies self->cars->cars:forAll(c | c.carMake = "moped")
```

```
<rule context="/Person">
      <assert test=
            "not(./age &lt; 18)
                  or
            (count(./Cars/Car[./carMake = 'moped'])
            = count(./Cars/Car))">
      </assert>
</rule>
```

In this subchapter, we have defined algorithm *UclToSchematron* and the functions *TypeToSchematron* and *ExpressionToSchematron* which automatically derive UCL constraints over XSEM PSM schema to Schematron rules for XML documents.

9.2. Mapping between data models

Today's complex software systems consist of more individual software components. Individual components are responsible for different parts of the software system. The whole software system used to be modeled by *PIM* (platform-independent-model). It is typically modeled using of UML Class diagrams. Particular software components are modeled using various data meta-models. To represent data of an application in a relational database we use an ER diagram. To represent structure of XML documents or SOAP messages send by an application we can use a XSEM diagram.

Elements in one model are semantically related and interconnected with elements in other model. For example UML Class in PIM diagram models a runtime object. This object is in a relational database stored as a column of a table; or in XML document it is represented by an element with attributes. A table in an ER diagram and an XSEM class are interconnected with a related UML class in PIM diagram.

In this subchapter, we define a directed mapping between elements of two different schemas of UCL Data meta-model. Such mapping must be created manually by a software architect or automatically by deriving a model to a different model.

Definition 9.5 (Directed mapping between two different models of UCL Data meta-model):

Let A is a schema in data model $Model_A$ and B is a schema in data model $Model_B$.

$\mathcal{M}_A$ is UCL Data meta-model schema (Definition 4.14) which has been created by mapping of elements from A to UCL Data meta-model. And $\mathcal{M}_B$ is UCL Data meta-model schema which has been created by mapping of elements from B to UCL Data meta-model.

$\mathcal{M}_A = (Entities^*_A,\ Relations_A,\ RelEnds_A,\ Lexicals_A)$;

$\mathcal{M}_B = (Entities^*_B,\ Relations_B,\ RelEnds_B,\ Lexicals_B)$;

Mapping from UCL Data meta-model $\mathcal{M}_A$ to UCL Data meta-model $\mathcal{M}_B$ is a function:

$Map_{A\to B} : E_A \cup R_A \cup L_A \cup Re_A \to E_B \cup R_B \cup L_B \cup Re_B$; which

- is an injective function, it does not maps distinct elements of its domain to the same element,

  $(\forall a, b \in E_A \cup R_A \cup L_A \cup Re_A)\ Map_{A\to B}(a) \neq Map_{A\to B}(b)$;

- maps entities from E_A to entities in E_B,

  $(\forall e \in E_A)\ Map_{A\to B}(e) \in E_B$;

- maps relations from R_A to relations in R_B;

  $(\forall r \in R_A)\ Map_{A\to B}(r) \in R_B$;

- maps relation ends from Re_A to relation ends in R_B;

  $(\forall x \in Re_A)\ Map_{A\to B}(x) \in Re_B$;

- maps lexicals from L_A to lexicals in L_B;

  $(\forall l \in L_A)\ Map_{A\to B}(l) \in L_A$.

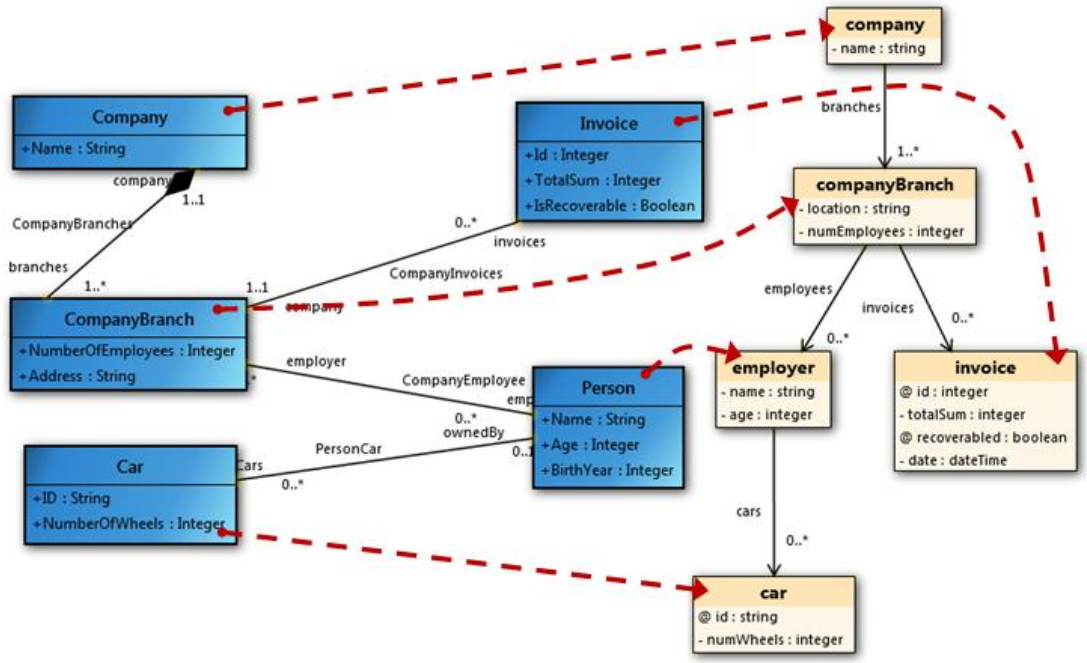


Figure 9.1: Mapping between UCL entities

Figure 9.1 illustrates by arrows the mapping from a UCL Data meta-model of UML class diagram to a UCL Data meta-model of XSEM PSM schema. In the figure, there is only mapping of UCL entities. The mapping function defines mapping of UCL relations, lexicals and relation ends, too.

9.3. Deriving UCL to different mapped model

By modeling of complex software systems we must often express a constraint over several models. E.g. we must define a constraint for PIM model in UML diagram and also the same constraint for databases in ER diagram. In UCL it is possible to express constraints over various models. We can express an UCL constraint expression over one model and another UCL constraint expression over different model. Although these expressions can semantically represent the same constraint, textual notation of these expressions may be different. Names of elements and relationships between them can be in individual UCL Data meta-model schemas different.

E.g. the same constraint in UCL over different models:

```
Context Person
inv: self.Age < 18 implies self->hasCars:isEmpty()

Context company::companyBranch::employer
inv: self.age < 18 implies self->cars:isEmpty()
```

Consider a situation by modeling a complex software system. We have two models and we have created schemas of UCL Data meta-model of these schemas. And we have created mapping of elements from the first UCL Data meta-models to elements of the second UCL Data meta-models. We have expressed integrity constraints in UCL over the first model.

Then it is possible to derive UCL constraints for the second model according the mapping between models. UCL constraints for the second model can be created only if relationships between elements of the first models are similar to relationships between elements of the second model. For example a mapping between UCL entities must correspondent with mapping of their inner UCL lexicals. Next definition formalizes the derivation of UCL constraints to a mapped data model.

Definition 9.6 (Derivation of UCL constraint to a mapped UCL Data meta-model):

Let $\mathcal{M}_A = (Entities^*_A, Relations_A, RelEnds_A, Lexicals_A)$ is an UCL Data meta-model schema over model A. And $\mathcal{M}_B = (Entities^*_B, Relations_B, RelEnds_B, Lexicals_B)$ is a UCL Data meta-model schema over model B.

Let $Map_{A\to B} : E_A \cup R_A \cup L_A \cup Re_A \to E_B \cup R_B \cup L_B \cup Re_B$ is a mapping function from UCL Data meta-model $\mathcal{M}_A$ to UCL Data meta-model $\mathcal{M}_B$ (Definition 9.5).

Let we have expressed UCL constraints over model A. They consist of a sequence of context blocks that is a subset of $UclContexts_{\mathcal{M}_A}$. Constraints contain definition of variables $VarDefinitions_{\mathcal{M}_A}$ and they are compound of expressions from the set $Expressions_{\mathcal{M}_A}$ (Definition 9.1).

Then function:

$DeriveUCL_{A\to B} : UclContexts_{\mathcal{M}_A} \cup VarDefinitions_{\mathcal{M}_A} \cup Expressions_{\mathcal{M}_A} \to UclContexts_{\mathcal{M}_B} \cup VarDefinitions_{\mathcal{M}_B} \cup Expressions_{\mathcal{M}_B}$ derives UCL constraint over model *A* to UCL constraint over model *B* if the mapping is correct defined.

The recursive function $DeriveUCL_{A\to B}$ assigns to a source UCL constraint *source* an UCL constraint according the kind of the source constraint:

- If *source* is a *context block* (*UCLContext-Definition*); function creates a *context block* where:

  Context type is $Map_{A\to B}(source.ContextType)$, if the context UCL entity or UCL relation is not mapped to model *B*, context block is not created;

The set of "def" variable definitions is the set of derived variables definitions $\{DeriveUCL_{A\to B}(var)|var \in source.DefVariablesDefinitions\}$, if the variable value's derivation is defined;

The set of invariants is derived to set of invariants. The name of created invariant is the source invariant name. Value of the invariant expression if the derived value of the source invariant expression. If the derived value is not defined then the derived invariant is not defined, too. $DeriveUCL_{A\to B}(invariant.expression)$;

- If *source* is a "let" variable definition (*UCLLets-Expression*); function gets a "let" variable definition with the same name and derived variable's value $DeriveUCL_{A\to B}(source.expression)$, if it is defined;
- If *source* is a literal value of a basic type (*UCLLiteralBase-Expression*); function get directly the literal value *source*;
- If *source* is an if-the-else operation (*UCLIf-Expression*)

  `if` *condition* `then expression1 else expression2 endif`;

  function gets:

  `if` $DeriveUCL_{A\to B}(condition)$

  `then` $DeriveUCL_{A\to B}(expression1)$

  `else` $DeriveUCL_{A\to B}(expression2)$ `endif`;

  where the inside expressions are derived. If one of the derived expressions is not defined then the whole derived expression is not defined, too;
- If *source* is a unary operation (*UCLUnaryBaseOperation-Expression*) $operator\ expression$; function gets the unary operation where the operand expression is derived $operator\ DeriveUCL_{A\to B}(expression)$; if the derived operand expression is not derived then the whole expression is not define, too;
- If *source* is a binary operation (*UCLBinaryBaseOperation-Expression*) $expression1\ operator\ expression2$; function gets the binary operation where the operand expressions are derived:

  $DeriveUCL_{A\to B}(expression1)\ operator\ DeriveUCL_{A\to B}(expression2)$; If one of the derived expressions is not defined then the whole derived expression is not defined, too;
- If *source* is a self expression which refers the content UCL entity or UCLRelation (*UCLSelfEntity-Expression*, *UCLSelfRelation-Expression*); function gets a self expression which refers to the mapped entity or relation

$Map_{A\to B}(source.context)$. If the mapping function is for the context element not defined then the whole expression is not defined, too, but then all context block is not defined;

- If *source* is a reference to a variable (*UCLVariable-Expression*); function gets the name of the variable;
- If *source* is a simple step expression from relation *R* through relation end *A* to entity *E* (*UCLStepRelationConnectedEntity*) `prevStep.A`;

  If the derivation of the previous expression is not defined then the whole expression is not defined.

  Let *derivPrevStep* $\coloneqq DeriveUCL_{A\to B}(prevStep)$ is the derivation of the previous expression. It must refer to relation $Map_{A\to B}(R)$.

  Let *mapA* $\coloneqq Map_{A\to B}(A)$ is the mapping of relation end *A*. If *mapA* does not belong to mapped relation $Map_{A\to B}(R)$ or it is not connected to mapped entity $Map_{A\to B}(E)$ then the whole expression is not defined.

  Else function gets derivation of the expression `derivPrevStep.A'`. Where *A'* is name of the mapped relation end $Re.name(mapA)$.
- If *source* is another kind of simple step expression from element (UCL entity or UCL relation) *A* to element *B* (*UCLSimpleStep-Expression*) `prevStep.B`;

  If the derivation of the previous expression is not defined then the whole expression is not defined.

  Let *derivPrevStep* $\coloneqq DeriveUCL_{A\to B}(prevStep)$ is the derivation of the previous expression. It must refer to element $Map_{A\to B}(A)$.

  Let *mapB* $\coloneqq Map_{A\to B}(B)$ is the mapping of the target element.

  If the mapping *mapB* is not defined or if navigation from $Map_{A\to B}(A)$ to *mapB* does not exist then the whole expression is not defined.

  Else function gets a simple step derivation of the expression `derivPrevStep.B'`. Where *B'* is name of the mapped target element (UCL entity or UCL relation) $Re.name(mapB)$.
- If *source* is a relation step (or stop) expression from entity *A* through relation *R* with relation end *X* to entity *B* (*UCLConnectedRelationStep-Expression* or *UCLConnectedRelationStop-Expression*)

  `prevStep->X` or `prevStep:>X`;

If the derivation of the previous expression is not defined then the whole expression is not defined.

Let *derivPrevStep* $\coloneqq DeriveUCL_{A\to B}(prevStep)$ is the derivation of the previous expression. It must refer to entity $Map_{A\to B}(A)$.

Let *mapR* $\coloneqq Map_{A\to B}(R)$ is the mapping of relation *R*; *mapX* $\coloneqq Map_{A\to B}(X)$ is the mapping of relation end *X*; and *mapB* $\coloneqq Map_{A\to B}(R)$ is the mapping of entity *B*.

If there *mapR* does not contain any relation end connected to the source entity $Map_{A\to B}(A)$ or if *mapX* is not the relation end of relation *mapR* or *mapX* is not connected to entity *mapB* then the whole expression is not defined.

Else function gets derivation of the expression `derivPrevStep->X'` or `derivPrevStep:>X`. Where *X'* is name of the mapped relation end $Re.name(mapX)$.

- If *source* is a collection expression (*UCLBaseCollection-Expression*) `prevStep:expression(variables | argument)`;

  Let *derivPrevStep* $\coloneqq DeriveUCL_{A\to B}(prevStep)$ is the derivation of the previous expression; and *derivArgument* $\coloneqq DeriveUCL_{A\to B}(argument)$ is the derivation of the argument expression. If expression *derivPrevStep* or *derivArgument* is not defined then the whole expression is not defined.

  Function gets derivation of the expression:

  `derivPrevStep:expression(variables | derivArgument)`;

- If *source* is a collection operation (*UCLBaseCollection-Expression*) `prevStep:operation()`;

  Let *derivPrevStep* $\coloneqq DeriveUCL_{A\to B}(prevStep)$ is the derivation of the previous expression. If the derivation of the previous expression is not defined then the whole expression is not defined.

  Function gets directly the derived previous expression with the collection expression `derivPrevStep:operation()`.

Function $DeriveUCL_{A\to B}$ derives UCL constraints over one model *A* to another model *B*; if there is mapping $Map_{A\to B}$ from UCL Data meta-model for *A* to UCL Data meta-model for *B* and it satisfies conditions in the definition of $DeriveUCL_{A\to B}$.

9.4. Conclusion

In this chapter, we have formally defined UCL constraints over a model of UCL Data meta-model. We have defined the directed mapping between two such models. This mapping must be created manually by software architects. And last we have defined how it is possible to derive UCL constraints over one model to UCL constraints over another model if the created mapping satisfies specific conditions.

A significant limitation is that this mapping must be a mapping one to one. To one element of the source model we can assign only one element in the target model. Otherwise the derivation algorithm is not defined.

We have also defined the derivation of UCL constraints over XML schemas to rules in Schematron. According all these definitions it is possible to convert UCL constraints over PIM schema to Schematron rules or other specific constraint languages for various meta-models.

For example in the context of UML model and mapped XSEM PSM model in Figure 9.1, we have UCL constraints over a UML diagram:

```
Context Car
inv: self.NumberOfWheels >= 2
inv: self.NumberOfWheels >= 4 implies self->ownedBy.Age >= 18

Context Person
inv: self.Age < 18 implies self->hasCars:isEmpty()

Context Company
inv:
self->branches:forAll(branch |
  branch->employee:select(empl | empl.Age > 30):size() > 10
)
```

Constraints can be derived to UCL constraints over XSEM:

```
Context company::companyBranch::employer::car
inv: self.numWheels >= 2
inv: self.numWheels >= 4 implies self->parent.age >= 18

Context company::companyBranch::employer
inv: self.age < 18 implies self->cars:isEmpty()
```

```
Context company
inv:
self->branches:exists(branch |
  branch->employees:select(empl | empl.age > 30):size() > 10
)
```

Constraints can be derived to Schematron rules:

```
<?xml version="1.0" encoding="utf-8"?>
<schema xmlns="http://www.ascc.net/xml/schematron">
  <pattern name="Constraints">

    <rule context="/company/companyBranch/employer/car">
      <assert test="./numWheels &gt;= 2"></assert>
      <assert test=
        "not(./numWheels &gt;= 4) or ./../age &gt;= 18" />
    </rule>

    <rule context="/company/companyBranch/employer">
      <assert test="not(./age &lt; 18) or empty(./car)" />
    </rule>

    <rule context="/company">
      <assert test=
        "exists(./companyBranch
          [count(./employer[./age &gt; 30]) &gt; 10]
        )"
      />
    </rule>
  </pattern>
</schema>
```

10. Implementation

The aim of this thesis is also to implement a software component with the support for UCL constraints. This software component must demonstrate the described applications of UCL to different data meta-models. We must create a universal and extensible framework. When the framework supports a data meta-model, it must be possible to add the support for mapping this of this data meta-model to its UCL Data meta-model. E.g. if the framework supports creating of UML Class diagrams then the it must be able to extend the framework to the support for creating of mapped UML-UCL models.

The framework must implement a parser of UCL expressions over UCL Data meta-model. It must be able to express UCL constraints over various data meta-models. The framework must also implement the presented derivation of UCL constraint over one model to UCL constraints to another mapped model.

And finally, the framework must support the derivation of UCL constraint to different specific constraint languages for different data meta-models. It must be able to add to the framework a support for a derivation to different other constraint languages.

10.1. DaemonX framework

DaemonX [45] is an existing software system for modeling. It was developed as a student software project in *Faculty of Mathematics and Physics, Charles University in Prague*. The whole DaemonX system was not developed as part of this thesis. Only the support for UCL is the part of this thesis.

DaemonX is a modeling framework. It is able to add the support for various data meta-models. DaemonX defines its own general meta-meta-model and it enables adding of plug-ins to the framework. According the documentation of the meta-meta-model, a programmer can create his own (data or process) meta-model. A programmer can develop a plug-in for this meta-model. And he can add it to the framework to enable creating models of the created meta-model in the DaemonX framework. The version DaemonX 1.0.1 contains plug-ins for meta-models PIM (as subset of UML class diagrams), relational model of databases, UML class and state diagrams, XSEM PSM model and BPMN. The demonstration of modeling in DaemonX is in Appendix C (User documentation).

Individual models of real complex software systems are related. DaemonX framework enables to create directly mapping between models. Such a mapping must be created manually by software architects. Or it is possible to derive a PSM diagram from a PIM diagram, in such a case the mapping between the PIM and the created PSM diagram is created automatically. An illustration of a mapping is in Appendix C (User documentation).

DaemonX supports an evolution process between mapped models. When a software architect modifies a PIM model, this change is propagated to all mapped models.

10.2. Constraints module, features

As a part of this thesis, we have implemented a software component in DaemonX framework. It must demonstrate the usage of UCL constraints over different data meta-models. DaemonX is an existing framework; it supports adding of data meta-models as plug-ins to the framework. DaemonX also implements creating of mapping between data models. We can use the existing implementation of this mapping for demonstration of deriving UCL constraints over one model to UCL constraints over the other mapped model (of different meta-model).

Using the implemented software component and UCL plug-ins, it is implemented and it is possible to:

- Create UCL Data meta-model for UML class diagrams meta-model (it is called UML-UCL Data meta-model);
- Create UCL Data meta-model for XSEM meta-model (it is called XSEM-UCL Data meta-model);
- It is possible to create automatically UCL Data meta-model for a UML Class diagrams model and UCL Data meta-model for an XSEM model (UML-UCL Data meta-model). This is implemented by individual UCL Data meta-models in the previous point;
- We can express UCL constraints over a diagram of the general UCL Data meta-model. So we can express UCL constraints over a UML class model and over an XSEM model;
- We can manually create mapping from elements of a UML model to elements of an XSEM model (This is provided by DaemonX framework);

- According this mapping, UCL constraints module deduces the mapping from elements of the UML-UCL model to elements of the XSEM-UCL model. And according this mapping we can automatically derive UCL constraints over a UML-UCL model to UCL constraints over an XSEM-UCL model;
- We have implemented also the derivation from UCL constraints over XSEM models to Schematron rules.

The user's guide with an example of the supported operations is in Appendix C (User documentation).

10.3. Architecture

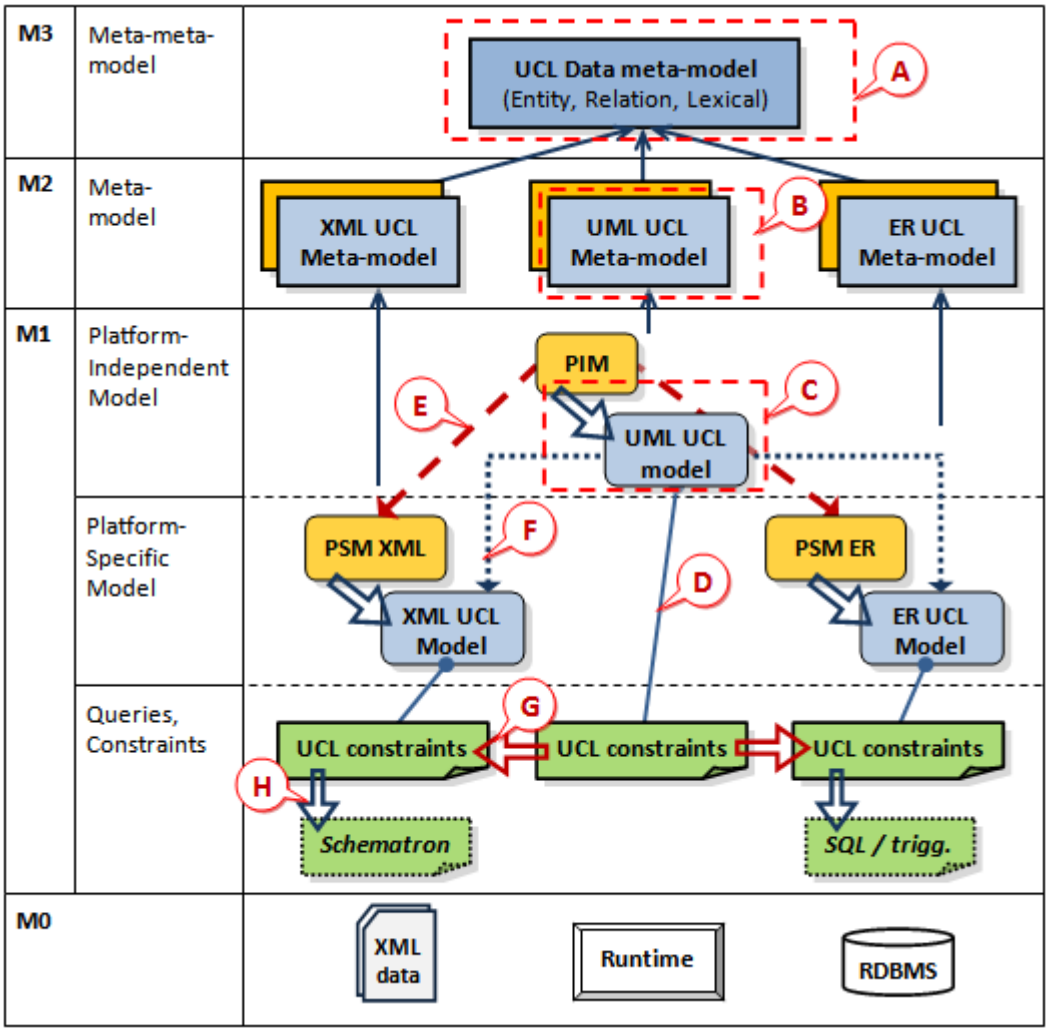


Figure 10.1: Architecture of the whole framework

Figure 10.1 shows the architecture of the whole framework DaemonX with the implemented component for UCL. The architecture is described in detail in Appendix D (Programmer's documentation).

Label *A* shows to the general UCL Data meta-model. It is a set of classes for *UCL entity*, *UCL relation* and *UCL lexical* and relationship between them.

To add the support of UCL for a data meta-model, we must create plug-in for it (label *B*). E.g. a plug-in to add the support of UCL for UML meta-model, we created UML-UCL meta-model. It defines mapping from elements from UML (class, attribute) to element in UCL Data meta-model (entity, relation, lexical) and relationships between them. The elements in UML UCL meta-model inherit element in UCL Data meta-model.

In the layer M1, concrete UCL meta-model plug-ins (e.g. UML UCL meta-model) (label *B*) creates UCL models from models (label *C*), e.g. a UML UCL model from a UML model.

Label *D* shows to the most complex and most comprehensive part of the implemented software component. It is the parser of textual UCL constraint expressions. It includes the lexical analyzer, the syntax parser and the semantic analyzer. It searches elements in UCL model in according navigation expressions; it checks type conformance rules of expression and operations. And it creates a syntactical tree of UCL expressions according UCL meta-model. We can express UCL constraints only in context of a UCL model (label *C*). This part was implemented in the software component of UCL as a part of this thesis.

Part *E* represents a direct mapping from one model to other model. This mapping must by created manually by a software architect – a user of DaemonX. The support of this mapping is implemented in DaemonX; its development was not a part of this thesis.

We have a mapping from one model to another model (created by a user, label *E*). We have mapped the first model to elements of its UCL model and analogue we have mapped the second model to elements of its UCL model. We can create the mapping from the first UCL model to the second UCL model (label *F*). This mapping is created automatically according the mentioned mappings.

According the mapping from one UCL model to another UCL model (label *F*) we can derive UCL constraints over the first model to UCL constraint over the second model (label *G*). This deriving is processed automatically. It is implemented in the developed component.

The implemented software component supports creating plug-ins to derivation of UCL constraints into other specific constraint languages (label *H*). E.g.

we have implemented plug-in which derives Schematron rules from UCL constraints over a XML model.

11. Conclusion

Today's software systems are typically composited of a complex system of several components. These individual components are responsible for different parts of the whole system; they communicate with each other. By modeling of these components, software architects use various meta-models for model individual components. E.g. UML class diagrams, ER models, XSEM. Models of these components are related and interconnected.

By modeling of software systems, we need integrity constraints. We need to express the consistency of modeled data. We often need to express conditions which must be satisfied by modeled data. To express these conditions a graphic annotation is not suitable. We must express them in textual constraint languages. When we model a complex system, we must use different constraint languages for different data meta-models; e.g. *OCL*, *SQL expression*, *Schematron*.

We often solve a task that one integrity constraint over one meta-model must be expressed also over other meta-model in a different constraint language. We must translate the same constraint from one language in one language to another language over a different related model. This is a nontrivial task which must be handled manually. The expressive power of individual constraint languages can be different, too.

The aim of this thesis was to automate this process. Our aim was to achieve a situation that it will be sufficient to express integrity constraints only over one meta-model. Typically in the meta-model UML class diagrams in the state that it is the platform-independent model. A software architect defines the interconnection between all meta-models which represent individual software component of the complex modeled system. When a software architect defines the mapping, it will be automatically possible to derive constraints from the PIM to other meta-models.

First, our aim was to define a new common language for integrity constraints in which it will be possible to express integrity constraints over different meta-models.

- In chapter 5 (UCL description), we introduced Universal Constrain Language (UCL). We have defined its syntax and its meta-model, in chapter 6 (Meta-model of UCL constraints). UCL is based on OCL. It does not support all language constructions of OCL. But it adds more possibilities for navigation expressions over the input model. For example, it uses different way for the

navigation to association classes as OCL; because OCL is based on the meta-model for UML.

The aim was that the created language must be suitable to express integrity constraints over different data meta-models. E.g. we want to express constraints over UML, over XML, over relational databases in this language.

- In UCL, it is possible to express constraints over different meta-models. UCL is a language which is based on the general data meta-model (UCL Data meta-model). Individual meta-models can be mapped to UCL Data meta-model; so we can express integrity constraints over these meta-models in UCL. In chapter 7 (Using UCL for UML Class diagrams), we have defined how to map meta-model of UML class diagrams to UCL Data meta-model. And we have defined how we can express constraints in UCL over UML models mapped to UCL Data meta-model. Then in chapter 8 (Using UCL for XML schemas), we have defined the mapping of XML models to UCL Data meta-model. In UCL, we can express constraints over an XML model.

Formulations of integrity constraints in UCL over different meta-models are possible through the general meta-model (UCL Data meta-model). UCL is based on this meta-model. The aim of this thesis was to analyse and formally define this general meta-model.

- In chapter 4 (Architecture, UCL Data meta-model), we have analyzed elements of the general meta-model. We have formally defined its elements and relationship between these elements. Then we defined a set of restrictions which must be satisfied for an instance of UCL Data meta-model. We defined which names of elements must be distinct because the names of elements are used in UCL navigation expression. Created UCL expressions cannot be unambiguous.

The aim was that it must be possible to translate constraint in UCL to other constraint languages.

- According meta-model of UCL, we have defined in chapter 9.1 (Deriving UCL for XML to Schematron) the derivation from UCL constraints over an XML model to constraint rules in language Schematron. But various constraint languages can have different expressive power. Derivation between different constraint languages must not be possible for all kinds of expression and for all pairs of constraint languages.

Some current modeling CASE tools support to define integrity constraints in OCL over created model. They can also check if instances of model satisfy the constraint. Some tools can also derive constraints in OCL to other constraint languages (to SQL expression over relational databases). In some tools, we can also express OCL constraints over other meta-models (over databases).

This thesis goes on in the process of the automation. Our aim was to support mapping between different meta-models. And we wanted to derive constraints in UCL between different related meta-models.

- In chapter 9.2 (Mapping between data models), we have formally defined the directed mapping from elements of one model (of one meta-model) to elements of other model (of different meta-model). This mapping must be created manually by a software architect. This mapping must map UCL entities to UCL entities, UCL relations to UCL relations, UCL lexicals to UCL lexicals and UCL relation ends to UCL relation ends. The next significant restriction is that this mapping must map one element to one element. To an element of the source model we can assign most one (one or no) element in the target model. And last we have defined how it is possible according the created mapping to derive UCL constraints over one model to UCL constraints over another model.
- The aim of this thesis is also to define deriving of UCL constraints over one data model to UCL constraints over another data model. If elements in separate models and parts of a complex software system are related and interconnected then it must be possible (according this interconnection of models) to derive constraints over one model to constraints over another related model.

The aim of this thesis was also to implement a framework which demonstrates the contribution of the theoretical work in this thesis. Our aim was to implement the application of UCL over different meta-models which are mapped to UCL Data meta-model. Then we wanted to demonstrate the derivation of UCL constraints between related meta-models; and the derivation from UCL constraints to other constraint languages.

- As we have described in chapter 10 (Implementation), we have implemented the support for UCL constraints into modeling framework *DaemonX*. In this framework data meta-models can be inserted as plug-ins. And then it support to create directed mapping from one model to other model. As a part

of this thesis, we have implemented the software module into this framework. This module implements the support for UCL. It is possible create a plug-in which map a concrete meta-model to UCL Data model. The implemented module contains a parser of UCL expressions. It is possible to express UCL constraints over different meta-models.

- Then we have implemented the derivation of UCL constraints over one meta-model to UCL constraints over a different meta-model. When a user of this framework creates a directed mapping from one model (of one meta-model) to other model (of a different meta-model), our component derives UCL constraints over the source model to UCL constraints over the target model. We have implemented the algorithm in chapter 9.3 (Deriving UCL to different mapped model).
- We have also implemented the support for the derivation from UCL constraints to other constraint languages. It is possible to implement a plug-in which derives UCL constraints over a model on a concrete meta-model to expressions in other constraint languages. We have implemented the plug-in which derives UCL constraints over a XML model to Schematron rules. We have implemented the algorithm in chapter 9.1 (Deriving UCL for XML to Schematron).

The contribution of this thesis is to reduce the work by designing of complex software systems. We have defined the common constraint language UCL. It can be used for integrity constraints over different meta-models which are mapped to the general UCL Data meta-model. Using UCL, software architects have not to express constraints manually for each individual meta-model in different constraint languages. They express constraints only in one language over one meta-model. And it is possible to derive these constraints automatically over other meta-models; and into other constraint languages.

Existing tools enable to use OCL for different meta-models. We created the tool which can use our constraint language over different meta-models. And then it can derive constraints to constraints over other meta-models and into other languages. We have implemented the tool in which user can define constraint over UML model in UCL; he can derive this constraint over XSEM model; and he can derive it to a Schematron rule.

Bibliography

[1] Object Management Group: *Unified Modeling Language Specification, Version 2.3. May 2010*. http://www.omg.org.

[2] Object Management Group: *Object Constraint Language OMG Available, Specification Version 2.2. February 2010*. http://www.omg.org.

[3] Nečaský, M.: Conceptual Modeling for XML, volume 99 of Dissertations in *Database and Information Systems Series. IOS Press/AKA Verlag, January 2009*.

[4] Codd, E. F.: *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks*. IBM Research Report, 1969.

[5] Bray, T. – Paoli, J. – Sperberg-McQueen, C. M. – Maler, E. – Yergeau, F.: *Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C, November 2008*. http://www.w3.org/TR/2008/REC-xml-20081126/.

[6] Thompson, H. S. – Beech, D. – Maloney, M. – Mendelsohn, N.: *XML Schema Part 1: Structures (Second Edition). W3C, October 2004*. http://www.w3.org/TR/xmlschema-1/.

[7] International Organization for Standardization.: *ISO/IEC 19757-3:2006 Information technology - Document Schema Definition Language (DSDL) - Part 3: Rule-based validation – Schematron*. http://www.schematron.com/

[8] W3C: *XML Path Language (XPath). Version 1.0, W3C Recommendation 16 November 1999*. http://www.w3.org/TR/xpath/.

[9] W3C: *XQuery 1.0: An XML Query Language, W3C Candidate Recommendation 8 June 2006*. http://www.w3.org/TR/xquery/.

[10] *List of Unified Modeling Language tools*. http://en.wikipedia.org/wiki/List_of_UML_tools.

[11] Richta K.: *Jazyk OCL a modelem řízený vývoj, accepted for publication in Moderní databáze 2010*, Nesuchyně, Komix, pp. 1-10, 2010.

[12] Richters, M. – Gogolla, M.: *OCL: Syntax, Semantics, and Tools. In: Object Modeling with the OCL, LNCS 2263, Springer Berlin/Heidelberg*, ISBN978-3-540-43169-5, pp. 447-450, 2002.

[13] Toval, A. – Requena, V. – Fernández, J. L.: *Emerging OCL tools. Software and System Modeling 2(4)*, 248-261 (2003).

[14] Hussmann, H. – Demuth, B. – Finger, F.: *Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International onference, York, UK, October 2000, Proceedings, volume 1939 of LNCS*, pages 278–293. Springer, 2000.

[15] *IBM OCL Parser*. http://www-01.ibm.com/software/awdtools/library/standards/ocl-download.html.

[16] BoldSoft: *ModelRun*. http://www.rationalrose.com/addins/non-free/modelrun.htm.

[17] Demuth, R.: *UML tools with OCL support*. http://st.inf.tu-dresden.de/ocl/.

[18] Octopus OCL Toolkit – *Octopus (OCL Tool for Precise UML Specifications)*. http://sourceforge.net/projects/octopus/.

[19] *Eclipse*. http://www.eclipse.org/.

[20] Miller, J. - Mukerji, J.: *MDA Guide Version 1.0.1.* Object Management Group, 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[21] Gogolla, M. - Büttner, F. - Richters, M.: *USE: A UML-Based Specification Environment for Validating UML and OCL*. Science of Computer Programming, 69:27-34, 2007.

[22] USE: A UML-based Specification Environment.
http://www.db.informatik.uni-bremen.de/projects/USE/.

[23] Sparx Systems: *Enterprise Architect*. http://www.sparxsystems.com.au.

[24] Sparx Systems: *OCL Conformance in Enterprise Architect.*
http://www.sparxsystems.com/enterprise_architect_user_guide/test_and_quality_control/ocl_conformance_element_relati.html.

[25] *Microsoft Visual Studio*. http://www.microsoft.com/visualstudio/en-us.

[26] *Dresden OCL*. http://www.dresden-ocl.org/index.php/DresdenOCL.

[27] Dresden OCL, Documentation.
http://www.emftext.org/index.php/DresdenOCL:Documentation.

[28] Wilke, C. – Thiele, M. – Freitag, B.: Dresden OCL, Manual for installation, use and development. http://141.76.65.213/dresdenocl_updatesite/manual.pdf.

[29] *NetBeans IDE*. http://netbeans.org/.

[30] *Eclipse Modeling Framework Project*. http://www.eclipse.org/modeling/emf/.

[31] *ECore meta-model: Package org.eclipse.emf.ecore*.

http://download.eclipse.org/modeling/emf/emf/javadoc/2.7.0/org/eclipse/emf/ecore/package-summary.html.

[32] Bräuer, M.: *Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment, Großer Beleg*, May 2007. http://dresden-ocl.sourceforge.net/gbbraeuer/index.html

[33] Dresden OCL Pivot model.
http://svn-st.inf.tu-dresden.de/svn/dresdenocl/trunk/ocl20forEclipse/eclipse/tudresden.ocl20.pivot.pivotmodel/src/tudresden/ocl20/pivot/pivotmodel/.

[34] Wilke C.: *Java Code Generation for Dresden OCL2 for Eclipse, Master thesis* Technische Universität Dresden.

[35] Brandt, R.: Ein OCL - Interpreter für das Dresden OCL2 Toolkit basierend auf dem Pivotmodell, Diploma Thesis, November 2007.
http://dresden-ocl.sourceforge.net/gbbraeuer/index.html.

[36] *Eclipse OCL*. http://www.eclipse.org/modeling/mdt/?project=ocl.

[37] Garcia, M.: *How to process OCL Abstract Syntax Trees*. Available under the EPL v1.0, June 28th, 2007. http://www.eclipse.org/articles/article.php?file=Article-HowToProcessOCLAbstractSyntaxTrees/index.html.

[38] Gamma, E. – Helm, R. – Johnson, R. – Vlissides, J.: *Design patterns, software engineering, object-oriented programming. Addison-Wesley, 1994*. ISBN 0-20163361-2. QA76.64.D47 1995.

[39] Akehurst, D. – Linington, P. – Patrascoiu, O.: *OCL 2.0: Implementing the Standard. Computer Laboratory*, University of Kent, November 2003. http://www.cs.kent.ac.uk/projects/ocl/index.html.

[40] Akehurst, D. - Linington, P. - Patrascoiu, O.: OCL 2.0 – Implementing the Standard for Multiple Metamodels. Electronic Notes in Theoretical Computer Science, (102):21–41, Elsevier, 2004.

[41] Demuth, B. - Hussmann, H.: *Using OCL Constraints for Relational Database Design. In: R. France, B. Rumpe (eds), <<UML>>'99 - The Unified Modeling Language, Proc. 2nd International Conference, Springer LNCS 1723, pp. 598-613, 1999*. http://www.inf.tu-dresden.de/ST2/ST/papers/uml99draft.pdf.

[42] Object Management Group. *MetaObject Facility*. 2.4 - Beta 2. ISO/IEC 19470:2005. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF.

[43] International Organization for Standardization. ISO/IEC 14977 : 1996(E): *Extended Backus–Naur Form*.

[44] Rademacher, G.: *Railroad Diagram Generator*. http://railroad.my28msec.com/.

[45] *DaemonX – a case tool framework for modeling and adaptibility of complex applications*. http://daemonx.codeplex.com.

[46] *Lexical analysis*. http://en.wikipedia.org/wiki/Lexical_analysis.

[47] *lex*. http://en.wikipedia.org/wiki/Lex_programming_tool.

[48] Imriska, S.: *C# LEX Manual*. http://www.seclab.tuwien.ac.at/projects/cuplex/lex.htm.

[49] Berk, E.: *JLex: A Lexical Analyzer Generator for Java. Version 1.2, May 5, 1997. Manual revision October 29, 1997*.
http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html.

[50] *Syntactic analysis*. http://en.wikipedia.org/wiki/Syntax_analysis.

[51] *Comparison of parser generators*. http://en.wikipedia.org/wiki/List_of_Parsers.

[52] *GPPG (Gardens Point Parser Generator)*. http://gppg.codeplex.com/.

[53] *jay*. http://www.cs.rit.edu/~ats/projects/lp/doc/jay/package-summary.html.

[54] Imriska, S.: *C# CUP Manual*. http://www.seclab.tuwien.ac.at/projects/cuplex/cup.htm.

[55] *Windows Presentation Foundation (WPF)*. http://msdn.microsoft.com/en-us/library/ms754130.aspx.

[56] *Extensible Application Markup Language (XAML)*. http://msdn.microsoft.com/en-us/library/ms747122.aspx.

[57] Structured Query Language (SQL). IBM. October 27, 2006. Retrieved 2007-06-10.
http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0004100.htm.

[58] The Unicode Consortium. *The Unicode Standard, Version 6.0.0. Mountain View, CA: The Unicode Consortium, 2011. ISBN 978-1-936213-01-6*.
http://www.unicode.org/versions/Unicode6.0.0/.

[59] Object Management Group: *XMI Specification. December 1, 2007.*
http://www.omg.org/technology/documents/formal/xmi.htm

List of Figures

Appendix A
CD contents

The attached CD contains:

- An electronic version of the text of this thesis (directory */thesis_text*);
- The whole implemented application with sample projects (directory */application_binary*);
- The source codes of the whole DaemonX framework with implemented component for purposes of this thesis (directory */application_sources*);
- The documentation of the whole DaemonX framework (directory/ *daemonX_documentation*).

The sample projects are in the directory */application_binary/DaemonX/Save*. There are files:

- *ucl_sample_data_model-examples.dx*
  Demonstration UCL expressions over the sample model in Figure 4.7;
- *ucl_sample_data_model-chapter-4.dx*
  UCL expressions which over the same sample model (Figure 4.7). There are all UCL expressions from UCL description;
- *ucl_for_uml.dx*
  The project with the UML Class diagram in Figure 7.1 with all UCL constraints over this model which are used in Using UCL for UML Class diagrams (Using UCL for UML Class diagrams);
- *ucl_for_xsem.dx*
  The project with the XSEM PSM diagram in Figure 8.1 Figure 7.1with all UCL constraints over this model which are used in Using UCL for XML schemas (Using UCL for XML schemas);
- *ucl_derivation.dx*
  The project with the UML Class diagram and the mapped XSEM PSM diagram in Figure 9.1 with UCL constraints over the UML diagrams. It is able to derive these constraints to constraints over the XSEM model; and to derive into Schematron rules according Mapping and deriving constraints (Mapping and deriving constraints).

Appendix B
Syntax of UCL

This appendix describes the syntax of UCL expressions in *Extended Backus-Naur Form* (*EBNF*) and railroad syntax diagrams.

Legend and conventions

- `'string'` terminal symbol
- `string` nonterminal symbol
- `::=` definition
- `symbol*` repetition (no or more occurrences) of symbol
- `symbol+` repetition (one or more occurrences) of symbol
- `symbol?` option (no or one occurrence) of symbol
- `one | two` choice from more symbols or rules
- `(* string *)` comment

Headers and common symbols

Block of UCL expressions consists of some blocks with context definitions. *Contexts* is the start nonterminal.

```
Contexts ::= ContextDefinition*
```

Contexts:

The context block contains `Context` keyword, definition of the context model element (*ContextPath*), some 'def' variables definitions (DefDefinition) and at least one invariant expression definition (Invariant).

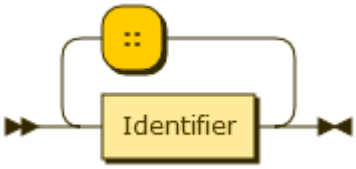
```
ContextDefinition ::=
   'Context' ContextPath DefDefinition* Invariant+
```

ContextDefinition:

The context of model element contains at least one name of root element and it can continue with names of inner entities or relations separated by the double colon.

```
ContextPath ::= Identifier ('::' Identifier)*
```

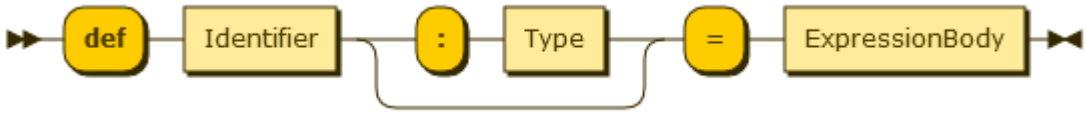
ContextPath:



A definition of 'def' variable consists from the keyword `def`, an identifier (name of the variable); optionally type of the variable (`Type`) can be presented; and from the expression (*ExpressionBody*) which defines a value of the variable. This expression cannot contain any 'let' variable definition. If the type (Type) is presented then it must be equal to type of the expression.

```
DefDefinition ::=
  'def' Identifier (':' Type)? '=' ExpressionBody
```

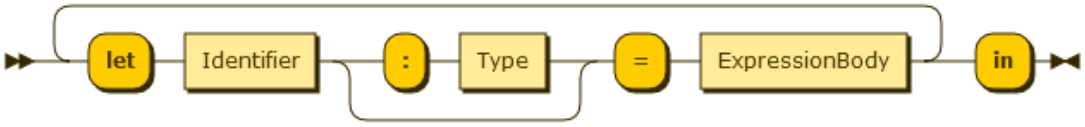
DefDefinition:



A definition of 'let' variables is similar. Definitions of variables are terminated by in to determine where the end of the last variable expression is and where the invariant expression starts. Definitions of 'def' variables are part of context headers; definitions of 'let' variables are part of particular expressions.

```
LetExpressions ::=
  ('let' Identifier (':' Type)? '=' ExpressionBody)+ 'in'
```

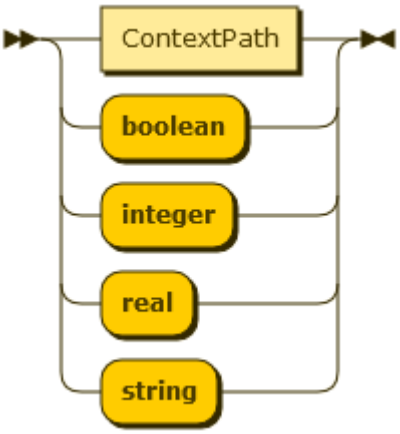
LetExpressions:



Type (of defined variable) can be type of model element (*ContextPath*) or one of basic types.

```
Type ::=
  ContextPath | 'boolean' | 'integer' | 'real' | 'string'
```

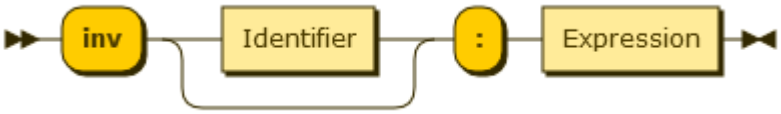
Type:



Definition of an invariant constraint consists from `inv`, optional name of the invariant (*Identifier*) and an UCL expression of boolean type.

```
Invariant ::= 'inv' Identifier? ':' Expression
```

Invariant:



Literals for all basic types:

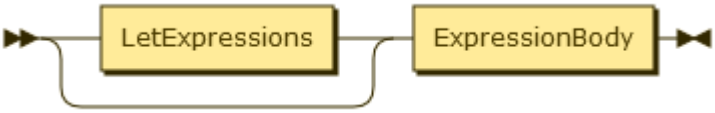
```
ExpressionLiteral ::=
  IntegerValue | RealValue | StringValue | BooleanValue
```

Operation expressions

Nonterminal symbol *Expression* represents an UCL expression. Optionally it can contain definitions of 'let' variables which can be used in the expression.

```
Expression ::= (LetExpressions)? ExpressionBody
```

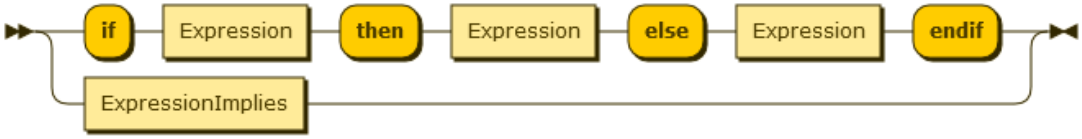
Expression:



Nonterminal *ExpressionBody* describes UCL expression without any definition of let variables. It has rules for implies-expression and for if-expression. Condition of if-expression must be of boolean type; the true and false branches must be of the same type; their type is also the result type.

```
ExpressionBody ::=
  ExpressionImplies |
  'if' Expression 'then' Expression 'else' Expression 'endif'
```

ExpressionBody:

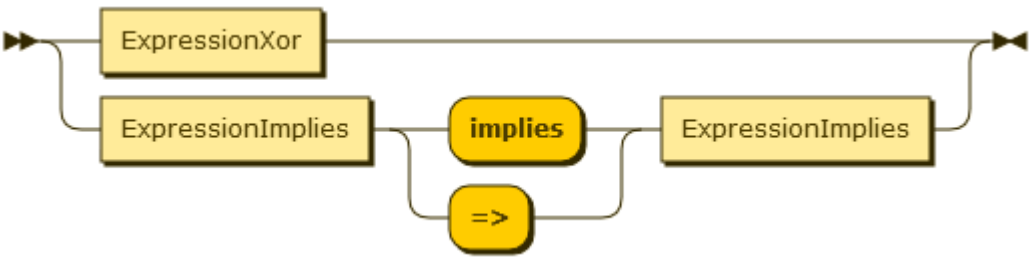


In *ExpressionBody* are nonterminals which refer to operations of individual operator according their operators precedence; from the lowest to the highest precedence.

Operands of expression *implies* must be of boolean type. It is allowed to use aggregated occurrence of operator; the first nonterminal of the second ruler is also ExpressionImplies.

```
ExpressionImplies ::=
  ExpressionXor |
  ExpressionImplies ('implies' | '=>') ExpressionImplies
```

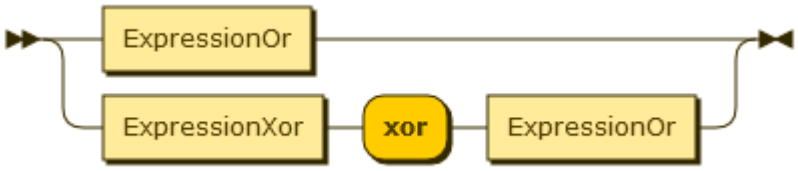
ExpressionImplies:



Operands of `xor` must be of boolean type; operator can be aggregated.

```
ExpressionXor ::=
  ExpressionOr |
  ExpressionXor 'xor' ExpressionOr
```

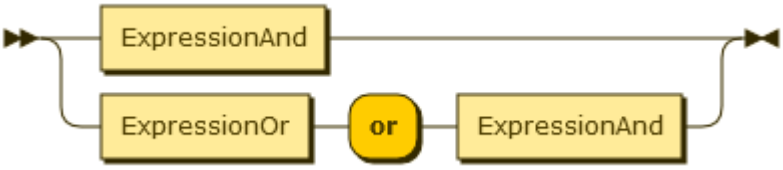
ExpressionXor:



Operands of `or` must be of boolean type; operator can be aggregated.

```
ExpressionOr ::=
  ExpressionAnd |
  ExpressionOr 'or' ExpressionAnd
```

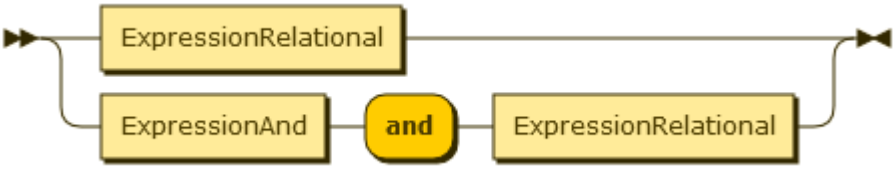
ExpressionOr:



Operands of `and` must be of boolean type; operator can be aggregated.

```
ExpressionAnd ::=
  ExpressionRelational |
  ExpressionAnd 'and' ExpressionRelational
```

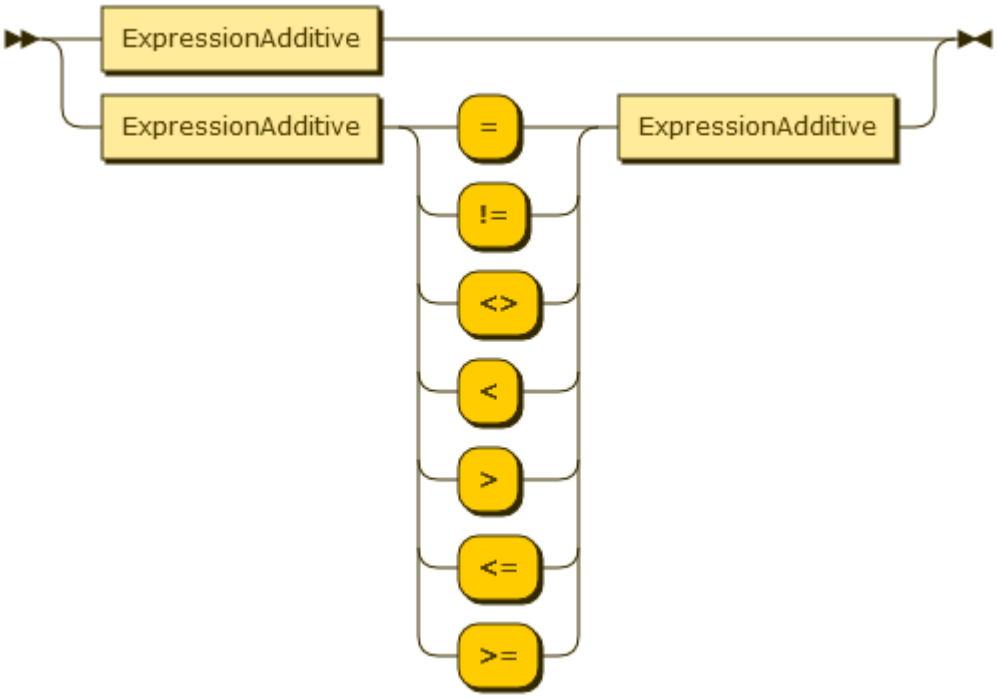
ExpressionAnd:



Comparing operators (*equals to* and not equals to) can be used with expressions of all types, relational operators can be used with basic types Integer, Real and String; operators cannot be aggregated

```
ExpressionRelational ::=
  ExpressionAdditive |
  ExpressionAdditive
    ('=' | '!=' | '<>' | '<' | '>' | '<=' | '>=')
    ExpressionAdditive
```

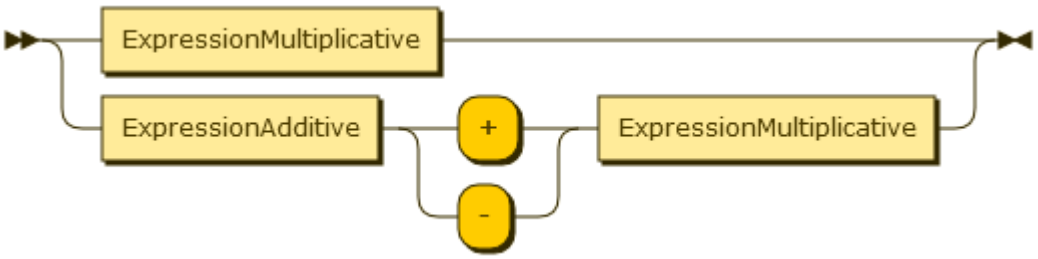
ExpressionRelational:



The operators addition and subtraction can be aggregated.

```
ExpressionAdditive ::=
  ExpressionMultiplicative |
  ExpressionAdditive ('+' | '-') ExpressionMultiplicative
```

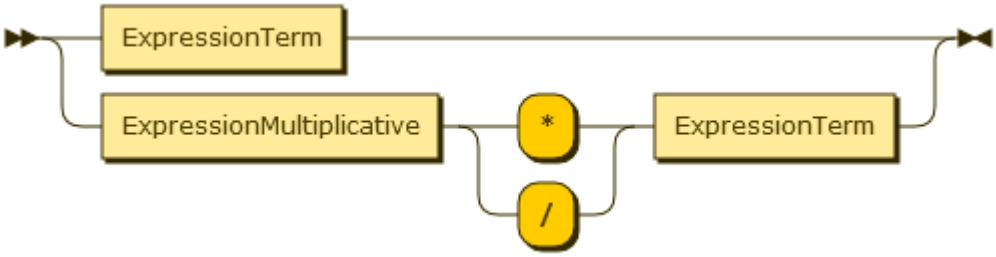
ExpressionAdditive:



The operators multiplying and dividing can be aggregated.

```
ExpressionMultiplicative ::=
  ExpressionTerm |
  ExpressionMultiplicative ('*' | '/') ExpressionTerm
```

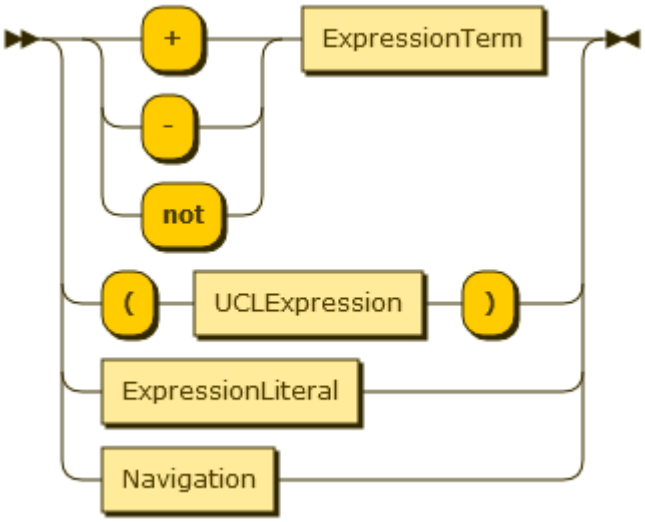
ExpressionMultiplicative:



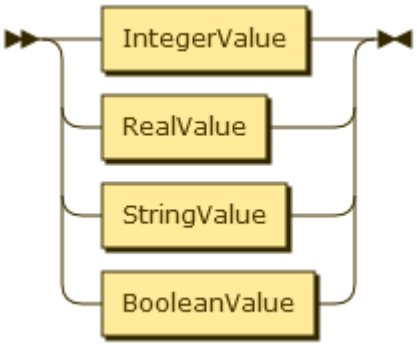
Unary operators + and – (for *Real* and Integer types), `not` (for Boolean) and parentheses has the highest priority.

```
ExpressionTerm ::=
  ('+' | '-' | 'not') ExpressionTerm |
  '(' UCLExpression ')' |
  ExpressionLiteral |
  Navigation
```

ExpressionTerm:



ExpressionLiteral:

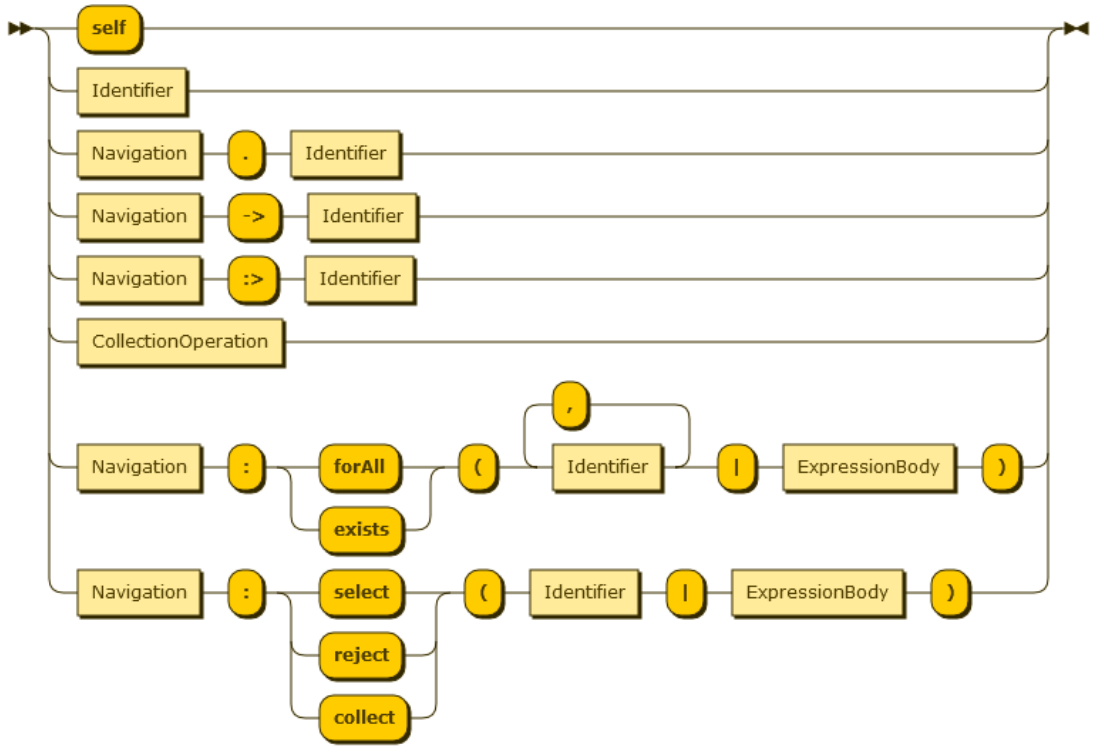


Navigation and collections

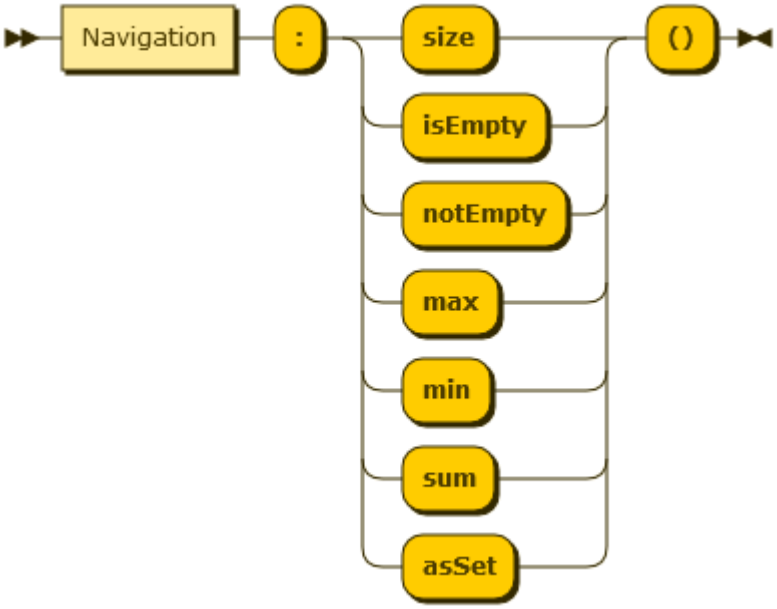
Navigation and collection expressions and operations have several forms. The first two rules define the starts of navigation over data model (`self` referring the context element and a variable referring the definition of variable). The rest nonterminals have *Navigation* as the prefix which defines the source of the navigation over data model. There is rule for the simple step navigation (from entities and relations) which is distinguished by a dot; for the relation step and the relation stop distinguished by `->` or `:>`. Collection operations and collection expressions are prefixed by a colon. Collection expressions contain identifier of expression's variable and the internal expression; in the case of expressions `forAll` and `exists` more variables can be defined.

```
Navigation ::=
  'self' |
  Identifier (* variable name *) |
  Navigation '.' Identifier (* simple step *) |
  Navigation '->' Identifier (* relation step *) |
  Navigation ':>' Identifier (* relation stop *) |
  CollectionOperation |
  CollectionExpression1 |
  CollectionExpression2

CollectionOperation ::=
  Navigation ':' ('size' | 'isEmpty' | 'notEmpty' |
    'max' | 'min' | 'sum' | 'asSet') '()'

CollectionExpression1 ::=
  Navigation ':'
    ('forAll' | 'exists')
   '(' Identifier (',' Identifier)* '|' ExpressionBody ')'

CollectionExpression2 ::=
  Navigation ':' ('select' | 'reject' | 'collect')
    '(' Identifier '|' ExpressionBody ')'
```

Navigation:



CollectionOperation:



Appendix C
User documentation

Launch and open a project in DaemonX

Before the executing of the system, copy the whole directory with the application (directory */application_binary/* in the CD) to a directory where the current user has permissions to read and write files.

To launch the application, execute file */application_binary/DaemonX.exe*. The application starts and loads all plug-ins from directory */application_binary/Plugins*. To load a project, press *Ctrl-O* or click to the *DX* icon in the top Ribbon panel and click to “Open project”. The sample projects are in the directory */application_binary/DaemonX/Save*.

Modeling and mapping

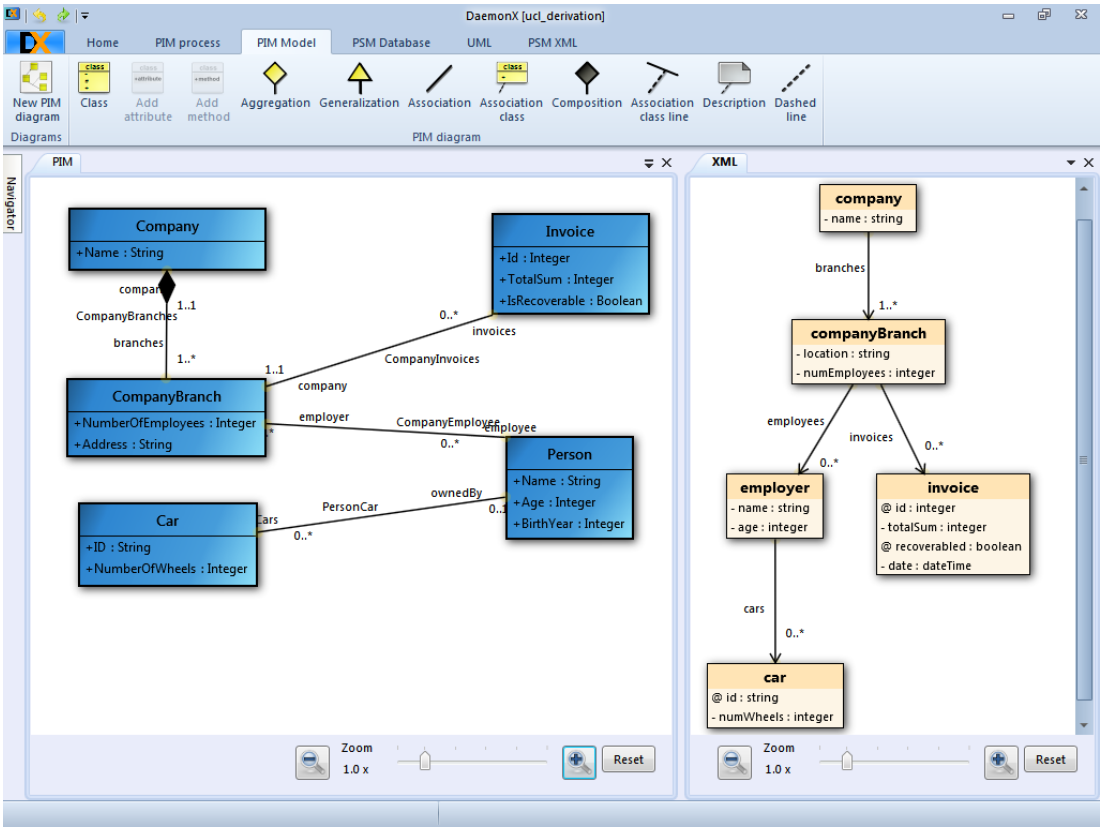


Figure C.1: Modeling in DaemonX, PIM and XSEM diagram

In the top Ribbon panel, there is the item “Home” and there is one item for each modeling plug-in. In the Ribbon panel items of meta-models, there are buttons for adding and editing diagrams and element of the meta-model. In the Ribbon panel

"Home", there are buttons to access individual DaemonX modules, e.g. "Evolution" (mapping between elements) and "UCL constraints".

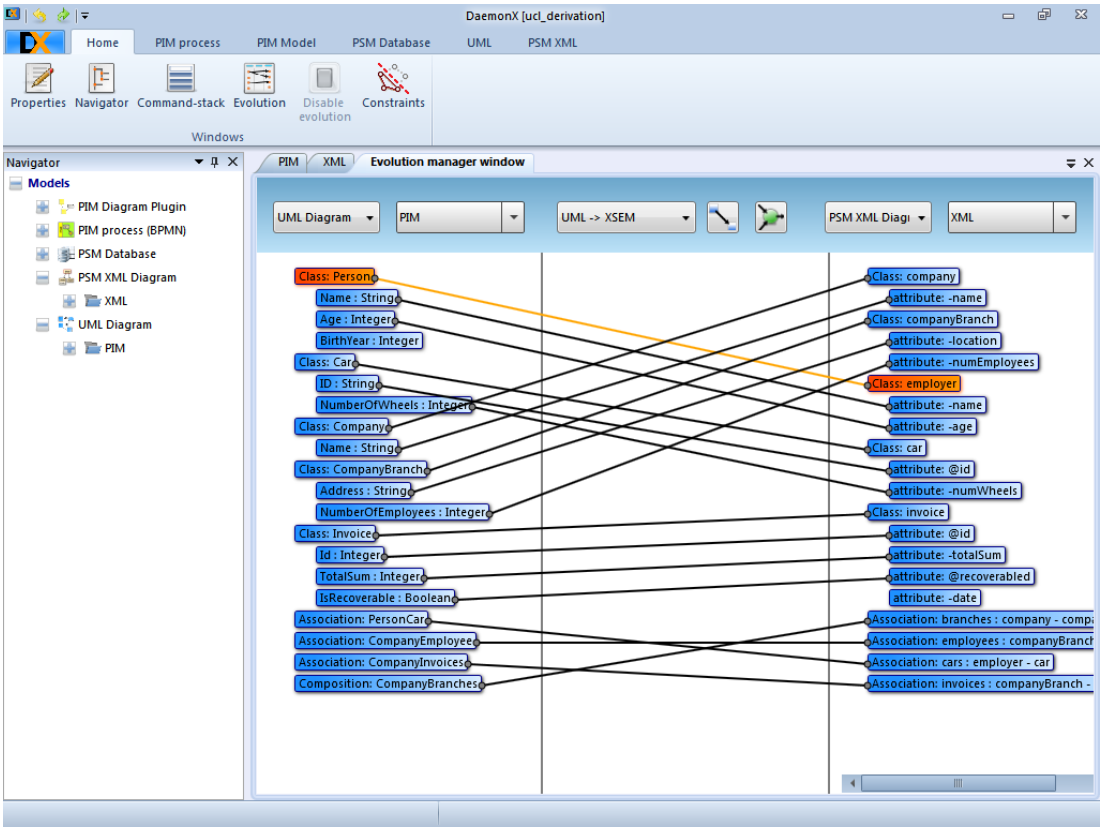


Figure C.2: Mapping from elements of UML model to elements of XSEM model

Figure C.2 shows the "Evolution window". Here a user can create a directed mapping from elements of one model to elements of another model. E.g. there is the mapping from the UML model to the XSEM model.

Constraints window

In the "Home" item in the Ribbon panel, there is the button "Constraints" to show windows the "UCL constraints" window. We can see this window in Figure C.3. In the combo box (label *M*), we can select a data meta-model for which exist at least one diagram in actual opened project. E.g. for the sample project in file "*ucl_derivation.dx"*, there are models of meta-models UML and XSEM. According the architecture of the framework (Figure 10.1), we select the modeling plug-in. When we select a model, we can write UCL constraints in text box (label *K*).

When we have selected a model, we can select in the combo box (label *N*) a UCL meta-model plug-in for the selected model. This plug-in creates the UCL model of the selected model; in the framework architecture (Figure 10.1) it is the

UCL model at label *C*. We implemented UCL meta-model plug-ins for UML and XSEM meta-models.

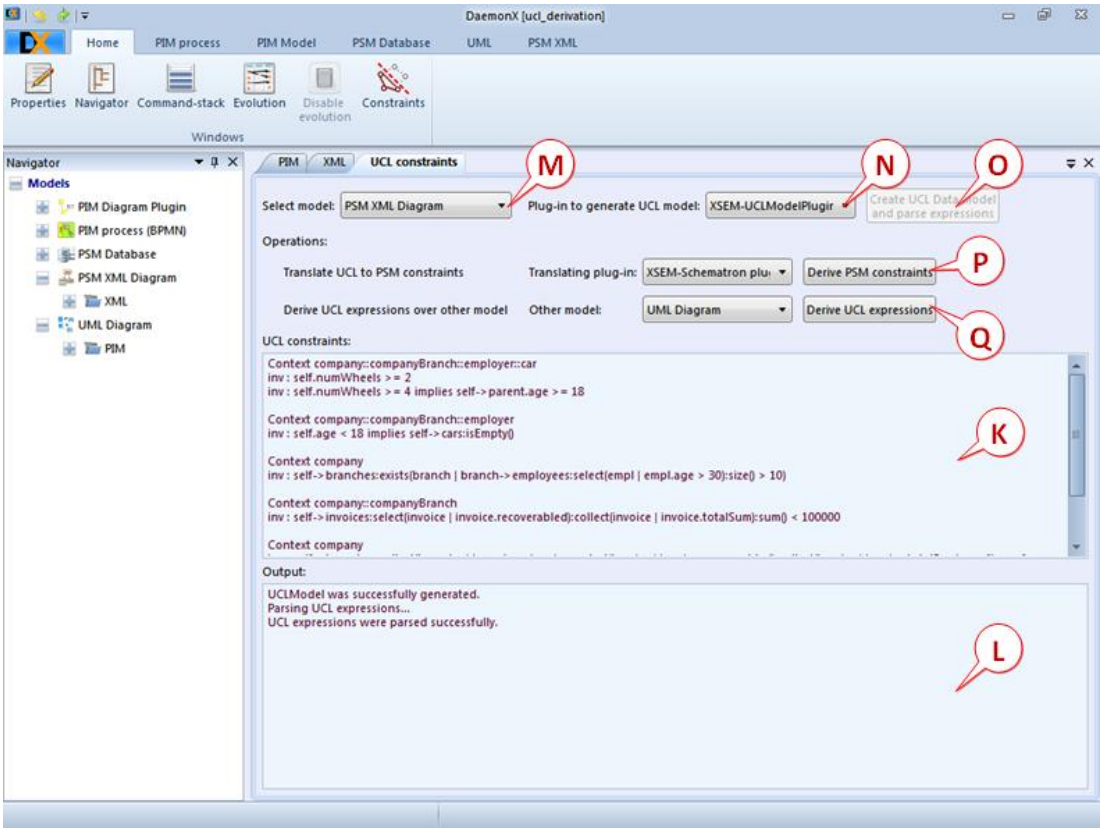


Figure C.3: UCL constraints over created models, derivation to another model and derivation to another constraint language

When we select a meta-model plug-in, we can click to button “Create UCL Data meta-model and parse expressions”. Then the UCL Data meta-model is created and UCL expressions in the text box (label *K*) are parsed and processed according the model. The parser of expressions is in the architecture (Figure 10.1) represented by label *D*. If any lexical, syntax or semantic (type or model) error in the UCL expressions is found then such an error is printed the “Output” text box (label *L*).

When the UCL constraints over the model are correctly parsed, we can derive the UCL constraints to a specific constraint language or we can derive UCL constraints over another mapped model.

When we want to derive the UCL constraints over one data model to UCL constraints over another model, we must in the combo box (label Q) select the target model. We must select the meta-model of the target model. If there is created a mapping from the source model to the target model (in the “Evolution window”) and the UCL Data meta-model of the target model was created then the framework

derives the UCL constraints over the target model. In the framework architecture, the derivation is label *G*.

E.g. we open the sample project "*ucl_derivation.dx"*. We open "Constraint window". We select the model XSEM and we create the XSEM-UCL Data meta-model by click to the button "Create UCL Data model and parse expressions". Then we select the model UML and we click to the same button again. The UML-UCL Data meta-model is created. The framework parses the UCL expressions in the text box. Then we must select the target model for deriving of the UCL constraints. We select the XSEM meta-model to derive the UCL constraints over UML to UCL constraints over XSEM. Then we click to the button "Derive UCL expressions". In the output text box the derived UCL constraints are printed. Or if a mapping between data model does not exist or it is not correct, error messages are printed.

In the combo box (label *P*) we can select a plug-in which translates the UCL constraints over the selected UCL meta-model to constraints in a specific constraint language. We have implemented the plug-in which translates UCL over XSEM to Schematron rules. When a user click to the button (label *Q*), Schematron rules are printed in the "Output" text box. Or if the derivation is not possible, error messages are printed.

Appendix D
Programmer's documentation

The source codes of the framework are in directory */application_sources*. The framework's documentation is in directory */documentation*.

Assemblies

The whole framework (Microsoft Visual Studio .NET solution) is divided into a few .NET assemblies (Visual Studio projects). Projects of the DaemonX framework are *AppLayer*, *Command*, *Controller*, *DaemonX*, *EvolutionManager*, *Model*, *ResourceLibrary*, *UndoRedo*, *Utilities* and *View*. These projects were not implemented as a part of this thesis. But we added functionality to some of these projects. All changes in the framework's projects are in pre-processor's directives `#if UCL_CA … #endif`.

- Into the project *DaemonX*, class *MainWindow*, we added the Ribbon panel button to open "Constraint window".
- Into the project *Applayer*, class *AppController*, we added the call to the initialization of the constraints module (`UCLCore.UCLManager.Init()`). Then there are calls to methods which save a load UCL expressions in "Constraint window" to a project file by loading and saving the whole project. Method `AppController.Load` calls method `UCLManager.Load` and method `AppController.Save` calls method `UCLManager.Save`.
- In the project *Applayer*, class *PluginManager*, we added functionality of loading plug-ins for the constraints module. The whole framework loads also plug-ins which are represented by interfaces *UCLModel.Plugins.IUCLModelPlugin* (a kind of plug-ins which represent UCL Data meta-model of a concrete data meta-model) and *UCLModel.Plugins.IUCLTranslatingPlugin* (a kind of plug-ins which derive UCL constraints to expression in another constraint language).

In the solution folder *Plugins.Modelling*, there are projects which represent modeling plug-ins – meta-models: *BPMNPlugin*, *ERPlugin*, *PIMPlugin*, *UMLPlugin* and *XSEMPlugin*. In the solution folder *Plugins.Evolution*, there are plug-in projects which represent a directed mapping from one meta-model to another meta-model.

They also implement an evolution propagation of changes from the source model to the target model.

All projects in the solution folder *Constraints* were implemented as a part of this thesis. The projects *UCLModel* and *UCLCore* represent the software component of the UCL support. Projects in the folder *Constraints.UCLModelPlugins* represent UCL Data meta-model plug-ins and projects in the folder *Constraints.UCLTranslatingPlugins* represent plug-ins which derive UCL constraints to other constraint languages. In the folder *UCLModelPlugins*, there are the plug-ins *Test_UCLModel* (plug-in which creates the sample model from Figure 4.7), *UML_UCLModelPlugin* (UCL Data meta-model plug-in for UML meta-model), and *XSEM_UCLModelPlugin* (UCL Data meta-model plug-in for XSEM meta-model). In the folder *UCLTranslatingPlugins*, there is the plug-in *XSEMSchematron_UCLTranslatingPlugin* which derives UCL constraint over XSEM to Schematron rules. The relevant difference between the projects *UCLModel* and *UCLCore* is that *UCLModel* defines mainly data structures and interfaces and *UCLCore* defines the functionality of the UCL expressions parser. Constraints plug-ins refer to the project *UCLModel* but they cannot refer to the project *UCLCore*.

Project UCLModel

Project *UCLModel* defines data structures and interfaces. They are used by parsing of UCL constraints. The project *UCLCore* and all constraints plug-ins refer this project.

Project defines: exceptions used in all UCL projects and plug-ins; class model for representing UCL expressions; read-only interfaces for plug-ins; class models for UCL Data meta-model; abstract factory which creates UCL expressions; classes for all lexical, syntax and semantic errors in parsed UCL expressions and interfaces for plug-ins.

Exception

Classes in the namespace *UCLModel.Exceptions* represent all exceptions which can be thrown by creating of UCL Data meta-model or derivation of UCL constraints. The base exception class is *UCLException*.

The exception *UCLInvalidDerivationToOtherModel* represents invalid derivation of UCL constraints to UCL constraint over another model.

The all remaining exceptions represent an incorrect creation of UCL Data meta-model. There are thrown if the created UCL Data meta-model does not satisfy the restrictions in Chapter 4.5 (UCL Data meta-model Concept definition). There are exceptions: *UCLDupplicateAccessibleRelationEndName* (there are relation ends of the same name which are accessible from a UCL entity); *UCLDupplicateRelationRelationEndName* (there are relation ends of the same name in a UCL relation); *UCLDupplicateSimpleStepName* (there are multiply possible simple step navigations from an entity or a relation); *UCLDupplicateTopMiddleEntityName* (there are more root entities or relations with the same name); *UCLGeneralizationCycle* (the generalization hierarchy over entities is not a partial order, there exist a cycle in the hierarchy) and *UCLGeneralizationEntityAlreadySet* (a generalization parent of an entity was already set).

UCL Data meta-model

Classes in the namespace *UCLModel.Model* represent elements of UCL Data meta-model which is presented in Chapter 4.5 (UCL Data meta-model Concept definition). The class model correspondents with the diagram in Figure 4.6. There are classes *UCLMetaConstruct* (the base class of UCL Data meta-model), *UCLMiddleEntity* (the base class for UCL entity and UCL relation), *UCLEntity*, *UCLRelation*, *UCLRelationEnd*, *UCLLexical* and *UCLModelHelper* (static class with methods to creating neighbourhood and generalization relations between elements).

All these classes are abstract. Plug-in for a concrete UCL meta-model must create child classes for these classes. E.g. UML-UCL meta-model must define class *UML-UCL-Class* which must inherit class *UCLEntity*.

UCL meta-model

The collection of 80 classes in the namespaces *UCLModel.Expressions*, *UCLModel.Types* and *UCLModel.Manager* represent UCL meta-model of UCL constraints which is presented in Chapter 6 (Meta-model of UCL constraints).

The classes in the namespace *UCLModel.Types* (14 classes) represent types of UCL expressions according diagrams in Figure 6.1 and Figure 6.2. The base class representing a UCL expression type is *UCLExpressionType*. There are classes representing the all basic types (they are singleton classes), collection types and types which refer UCL entities and UCL relations of UCL Data meta-model. Classes representing a type of UCL expressions override methods *Name* and *DeriveToOtherModel*. Method *Name* gets the name of the type, in case of the type

of UCL entity of UCL relation it gets the name including its outer entities. Method *DeriveToOtherModel* derives the type to the type over other mapped model; it is called by deriving UCL constraint to constraints over another model.

The classes in the namespace *UCLModel.Expressions* (61 classes) represent all possible kinds of UCL expressions. There are classes for the concrete literal values, operations over expression of the basic types, collection operations, collection expressions, navigation expressions and "let" variables definitions according Chapter 6.3 (Expressions). Classes representing a UCL expression override methods *GetTextExpression* and *DeriveToOtherModel*. Method *GetTextExpression* gets the textual representation of the expression. Method *DeriveToOtherModel* derives the expression to the type over other mapped model.

The classes in the namespace *UCLModel.Manager* (5 classes) represent definitions of variables, blocks of UCL expression and collections of these blocks. Classes *UCLVariable*, *UCLCollectionVariable* and *UCLVariableDefinition* represent definitions of "def", "let" and collection variables, according Chapter 6.3.3 (Variables) and Figure 6.5. Class *UCLInvariantExpression* represent invariant UCL expression of boolean type with a name; class *UCLContextDefinition* represent block of context definition of UCL constraint over a UCL entity or a UCL relation, according Chapter 6.3.1 (Contexts).

UCL model and UCL expressions

Class *UCLModelManager* represents UCL model (e.g. UML-UCL model) with instances of UCL entities, relation and lexicals and parsed UCL constraints. It is created by class *UCLModelManager*. It holds all UCL entities and al UCL relations which are root entities or relations. They are stored in attribute *topMiddleEntities*; they are mapped according their names. Attribute *contextsDefinitions* holds all instances of *UCLContextDefinition*; all blocks of context definitions with UCL invariants. Attribute *allVariablesList* holds all definition of UCL variables. Instance of this class is created by the creating of UCL model from a model. Root elements of UCL model are inserted into it.

By the parsing of UCL constraints, created context blocks are inserted into it. Attribute *variablesLayers* holds layers of variables and attribute *actualVariablesMap* holds all variables which valid are the current scope. By the parsing of a context block with "def" definitions of variables, new layer in *variablesLayers* is created and the variables are inserted to *actualVariablesMap*. Analogue, by the parsing of expression with "let" definitions of variables, new layer in *variablesLayers* is created

and the variables are inserted to *actualVariablesMap*. After the parsing of a context block or the parsing of a *let* expression, the top layer in *variablesLayers* is removed and the variables are removed from *actualVariablesMap*.

Class holds also descriptions with all founded errors in UCL expression in attribute *modelErrors*.

Class implements two proxy interfaces with specific methods which are accessible only from certain classes. Interface *IModelManagerProxy* is a proxy interface with methods which can read and add elements of UCL model. This proxy interfaces is used by plug-ins which create UCL model (e.g. *UML_UCLModelPlugin*). The second proxy interface is *IModelManagerTranslatingProxy*. It contains only one method to read all blocks of context definitions. It is used by plug-ins which derives UCL constraints to another constraint language.

Class *UCLDiagramManager* is responsible for the creating of UCL model. It is created by class *UCLManager* from project *UCLCore*. It holds an instance of *UCLModelManager*, UCL model plug-in, expression factory which creates instances of UCL meta-model and container with errors found in UCL constraints.

Expressions factory

Interface *IUCLExpressionFactory* is designed following the design pattern Abstract factory. It creates instances of classes of UCL meta-model. The parser of UCL constraints does not create concrete instances of class in UCL meta-model but it calls the *expressions factory* to create instances of concrete classes which inherit base class from UCL meta-model.

E.g. UCL meta-model contains class *UCLForAllExpression* which represents *forAll* expressions. But plug-in which derives UCL constraints to Schematron rules defines its own class for *forAll* expressions; because it must override methods of class *UCLForAllExpression*. This plug-in defines *expressions factory* which does not create instances of class *UCLForAllExpression* but instances of its own class.

Errors in UCL expressions

The collection of 32 classes in the namespace *UCLModel.Parser.Errors* represents all possible lexical, syntax and semantic errors in parsed UCL expressions which can occur by parsing of UCL constraints. The base class for the errors is *ExpressionError*.

The lexical errors are a bad format of a literal number, an overflow value of a literal number, an unexpected character, an unterminated string or comment. The syntax errors are unexpected token or unexpected end of UCL expressions. The semantic errors are: an unknown identifier (of a root entity or relation, of an inner entity or relation, of all possible navigation expressions), a bad type of an expression (of a source expression, of a collection argument expression), a collection expression / operation over a not collection type, an invariant of not boolean type, an aggregation operation (*sum*, *min*, *max*) over not numerical collection, a *set* operation over not a sequence type, a duplicate name of a variable, an illegal type of an operations over basic types.

Plug-ins interfaces

In the namespace *UCLModel.Plugins*, there are interfaces which represent kinds of plug-ins.

Interface *IUCLModelPlugin* represent plug-in with UCL meta-model. This plug-in creates UCL Data meta-model for a model. E.g. for UML Class diagrams meta-model, we can implement UCL-UML meta-model plug-in. This plug-in creates from a UML model UML-UCL model.

The interface contains properties *PluginInfo* with information of the plug-in and *ModelPluginInfo* about information the plug-in which represents the data meta-model. It contains method *GenerateModel* which generates UCL model of.

Interface *IUCLTranslatingPlugin* represents plug-in which derives UCL constraints over to constraints in other constraint language. E.g. plug-in *XSEMSchematron_UCLTranslatingPlugin* derives UCL constraints over XSEM to Schematron rules.

The interface contains property *PluginInfo* with information of the plug-in. It contains method *CanTranslateModel*, which gets if the plug-in can translate UCL constraints over it. (E.g. the plug-in for Schematron gets *true* value only for UCLModelPlugin over XSEM.) Method *GetExpressionsFactory* gets an expression factory which creates instances of UCL meta-model for UCL constraints. Method *TranslateExpressions* derives UCL constraints to the target constraint language.

UCL meta-model plug-ins

How to create a UCL meta-model plug-in (*UCLModelPlugin*)

To create a UCL meta-model plug-in of over an existing meta-model plug-in we must create a .NET project. Then we must:

- create a plug-in class – a class which implements interface *IUCLModelPlugin*;
- create UCL meta-models – we must create a class for each element in the source meta-model, these classes must inherit a class from UCL Data meta-model (*UCLEntity*, *UCLRelation* or *UCLLexical*).

UML_UCLModelPlugin

UML-UCL meta-model plug-in (*UML_UCLModelPlugin*) is UCL meta-model plug-in over UML meta-model (*UMLPlugin*).

Plug-in defines UML-UCL meta-model according the algorithm in Chapter 7.2 (Mapping to UCL Data meta-model). Class *UCL_UMLClass* inherits *UCLEntity*; it represents a UML class or a UML Association class. Class *UCL_UMLAttribute* inherits *UCLLexical*; it represents a UML attribute. Class *UCL_UMLAssociation* inherits *UCLRelation*; it represents a UML association, a composition or an aggregation.

The plug-in class is class *UML_UCLModelPlugin*. It implements interface *IUCLModelPlugin*. Class contains the default constructor. It overrides properties *PluginInfo* and *ModelPluginInfo* (the meta-model plug-in is *UMLPlugin*).

Class overrides method *GenerateModel* which generates the UML-UCL model from a UML model. UML classes and UML Association classes are mapped to instances of class *UCL_UMLClass*. UML attributes are mapped to instances of class *UCL_UMLAttribute* and they are inserted into mapped instances of class *UCL_UMLClass*. UML associations, compositions and aggregations are mapped to instances of class *UCL_UMLAssociation* and they are connected with mapped connected UML classes. The generalization between UML classes is mapped to the generalization between instances of class *UCL_UMLClass*. Connections between UML Association classes and connected associations are mapped to neighbourhood relationships between instances of class *UCL_UMLClass* and instances of class *UCL_UMLAssociation*.

XSEM_UCLModelPlugin

XSEM-UCL meta-model plug-in (*XSEM_UCLModelPlugin*) is UCL meta-model plug-in over XSEM meta-model (*XSEMPlugin*).

Plug-in defines XSEM-UCL meta-model according the algorithm in Chapter 8.2 (Mapping to UCL Data meta-model). Class *UCL_XSEMClass* inherits *UCLEntity*; it represents an XSEM class. Class *UCL_XSEMContentModel* inherits *UCLEntity*; it represents an XSEM content model. Class *UCL_XSEMBaseAssociationEnd* is the base class of *UCL_XSEMClass* and *UCL_XSEMContentModel*. Class *UCL_XSEMAttribute* inherits *UCLLexical*; it represents an XSEM attribute. Class *UCL_XSEMAssociation* inherits *UCLRelation*; it represents an XSEM association.

The plug-in class is class *XSEM_UCLModelPlugin*. It implements interface *IUCLModelPlugin*. Class contains the default constructor. It overrides properties *PluginInfo* and *ModelPluginInfo* (the meta-model plug-in is *XSEMPlugin*).

Class overrides method *GenerateModel* which generates the XSEM-UCL model from an XSEM model. XSEM classes are mapped to instances of class *UCL_XSEMClass*; and they are inserted into the mapped parent class (it such a class exists) XSEM content models are mapped to instances of class *UCL_XSEMContentModel*; and they are inserted into the mapped parent class (it such a class exists). XSEM attributes are mapped to instances of class *UCL_XSEMAttribute*; and they are inserted into mapped instances of class *UCL_XSEMClass*.

XSEM associations are mapped to instances of class *UCL_XSEMAssociation*; and they are connected with mapped connected XSEM classes or content models. The XSEM association name is used as the name for the relation end to the child class. Name “parent” is used as the name for the relation end to the parent class. The XSEM association cardinality is used as the cardinality for the relation end to the child class (or content model). Cardinality (1, 1) is used as the cardinality for the relation end to the parent class (or content model).

Translating plug-ins

A translating plug-in is plug-in which derives UCL constraints over to constraints in other constraint language.

E.g. plug-in *XSEMSchematron_UCLTranslatingPlugin* derives UCL constraints over XSEM to Schematron rules.

How to create a Translating plug-in (*UCLTranslatingPlugin*)

To create a Translating plug-in of over an existing meta-model plug-in we must create a .NET project. Then we must:

- create a plug-in class – a class which implements interface *IUCLTranslatingPlugin*;
- create special UCL meta-model – a set of classes which inherits class in the namespace *UCLModel.Expressions*. In the namespace, there is a class for each kind of UCL expressions. We must create a child class for each class in this namespace. And we must override method *GetTextExpression*. This method prints the textual representation of expressions. We must override this method with the implementation which derives UCL constraints to constraints in another constraint language.
- create an expression factory – a class which creates instances of classes from the created special UCL meta-model.

XSEMSchematron_UCLTranslatingPlugin

Plug-in *XSEMSchematron_UCLTranslatingPlugin* is a plug-in (*UCLTranslatingPlugin*) which derives UCL constraint over XSEM model to Schematron rules. The derivation is implemented according the algorithm in Chapter 9.1 (Deriving UCL for XML to Schematron).

In project's directory *Expressions*, there is a set of classes. These classes represent special type of UCL meta-model. Each class inherits a class from UCL meta-model (classes in namespace *UCLModel.Expressions*). E.g.: In UCL meta-model, there is class *UCLSelfEntityExpression* which represents a *self* UCL expression. Method *GetTextExpression* of this class returns expression "`self`". It is an UCL expression which refers to the context entity or relation. In this plug-in, there is class *SchematronSelfEntityExpression* which inherits *UCLSelfEntityExpression*. Method *GetTextExpression* of class *SchematronSelfEntityExpression* returns XPath expression "`.`". It is an expression which refers to the current XML node.

Some kinds of UCL expressions cannot be derived to Schematron rules. Classes which represent these kinds of expressions implement method *GetTextExpression* but they throw exception *UCLInvalidDerivation*.

Class *XSEMSchematronExpressionFactory* represent abstract factory. It inherits *IUCLExpressionFactory* (the interface for creating UCL expressions). It creates instances of classes in directory *Expressions* which represent UCL expressions.

The main plug-in class is class *XSEMSchematron_UCLTranslatingPlugin*. It implements interface *IUCLModelPlugin*. Class contains the default constructor. It overrides property *PluginInfo*. It overrides method *CanTranslateModel*, this plug-in derives UCL constraint over model plug-in *XSEM_UCLModelPlugin*. Method *GetExpressionsFactory* returns an instance of class *XSEMSchematronExpressionFactory*. Method *TranslateExpressions* prints Schematron document with constraint rules. It calls methods to derive UCL expressions over all UCL context blocks.

Project UCLCore

Project *UCLCore* consists of:

- lexical and syntax parser of UCL expression;
- “Constraint window”;
- manager class UCLManager.

Lexical analyzer

Lexical analyzer [46] is a component which converts a sequence of characters to a sequence of tokens. A token is an order of characters; e.g. an identifier, a keyword. Lexical analyzer is class *UCLCore.Parser.LexicalAnalyzer*. This class was generated according the rules specifying the lexical analyzer in file *UCLCore/Parser/LexRules.lex*. A lexical generator [46] is software which creates source code according rules specifying the lexical analyzer. The best-known and most used lexical generator is lex [47]. The popular version of lex is Flex. We used as a lexical generator C# lex [48]. It is C# version of lex. It is version of well documented software which creates .NET source code. C# lex was developed from JLex [49] what is a version of lex for Java.

Lexical analyzer class *LexicalAnalyzer* implements interface *UCLCore.Parser.YYParser.ILexer*. It contains methods to get next token identifier, its position in text and other information about the token. This interface is used by parsers. Values of tokens are integer constants in class *UCLCore.Parser.Token*. The type of a token is not the entire Information about the token. With a token,

advanced values are associated. E.g. for an integer literal, type of a token is *Token.INTEGER* and the associated value is its integer value. Complex token information are represented by enums in namespace *UCLCore.Parser.TokenValues*. E.g. it is kind of the basic type and type of operators. Static class *TokenToText* converts type of tokens to its textual representation for message expression errors.

Lexical analyzer contains method *SetErrorsContainer* to set the container to inserting of founded lexicals errors in input expressions.

Parser generator

A syntactic analysis [50] is the process of analyzing a structure of tokens. It determines if the structure satisfies the syntax of the input language. It also creates an abstract syntax tree in the hierarchical structure according the syntax of the input language. A parser generator is software which creates the source code of a syntactic analyzer from the grammar of an input language in BNF.

Lex is commonly used [47] with the yacc parser generator (Yet Another Compiler Compiler). It is an LALR parser (Look-Ahead LR parser) [50]. Well-known parsers which use YACC input grammar notation and generate C# code are [51] GPPG (Gardens Point Parser Generator) [52], jay [53] or C# cup [54]. We use jay parser generator because it is best-known and most used of these parser generators. File *ParserRules.jay* it the input of the parser generator. It contains the list of all tokens and the grammar of UCL in YACC-BNF notation. The created parser is represented by class *UCLCore.Parser.UCLParser*.

The parser calls the lexical analyzer using methods of interface *UCLCore.Parser.YYParser.ILexer*. The lexical analyzer and parser are created, interconnected and launched by:

```
System.IO.TextReader inputTextReader = …
UCLDiagramManager diagramManager;

var lexAnalyser = new LexicalAnalyzer(inputTextReader);
var uclParser = new UCLParser(diagramManager);
uclParser.ParseInput(lexAnalyser);
```

The parser class's constructor has as a parameter instance of *UCLDiagramManager*. To this class is inserts parsed expressions and founded syntax and semantic errors.

Semantic analysis

Class *UCLParser* contains many methods in file *Parser_partial.cs*. These methods respond to the semantic analysis. The methods are called by parsing of UCL expressions according rules in file *ParserRules.jay*.

Semantic analysis creates the abstract syntax trees from UCL expressions. It processes all kinds of UCL expression according UCL meta-models. It does not create instances of classes from UCL meta-model; it calls the expression factory to create instances of these classes. Semantic analysis does not check the type conformance rules by operations over the basic types. This control is provided by classes of UCL meta-model.

Semantic analysis processes navigation expressions. It searches root UCL entities or root UCL relations according it names. It searches accessible elements by *simple step* navigation, *relation step* and *relation stop* navigation.

Semantic analysis processes collection expressions and collection operations. In that case, it controls type rules.

Constraints window

Class *ConstraintsWindow* represent the “Constraint window”. The visualization is created by *Windows Presentation Foundation* (WPF) [55]. It is represented by an *Extensible Application Markup Language* (XAML) [56] document in file *ConstraintsWindow.xaml*. It is a singleton class. The instance is accessible by property *Instance*.

Combo box *modelComboBox* holds items with meta-models (modeling plug-ins) which are used in the current project. (There exists a diagram of this meta-model.) The items (meta-models) are filled in method *ChangeModelComboBoxData*. By change of the selected item, method *ModelComboBox_SelectionChanged* is called. This method calls method *ChangeSelectedModel*. It loads UCL constraints for model of the selected meta-model. Then it calls *FillUCLModelPlugins* to load UCL meta-model plug-ins; and *FillDerivationOtherModelPlugins* to get list of other meta-model for derivation UCL constraint to other model.

Combo box *modelPluginComboBox* contains UCL meta-model plug-ins for the selected meta-model. Its items are filled by method *FillUCLModelPlugins*. By the change of the selected item, it calls *ChangeSelectedUCLModelPlugin*. This method

calls *FillUCLTranslatingPlugins* to get list of translating plug-ins and enable button *generateModelBtn*.

Click to button *generateModelBtn*, method *UCLManager.GenerateUCLModel* is called. It generates UCL model of the current selected meta-model; and parses UCL expressions over the model.

Combo box *uclTranslatingPluginComboBox* contains translating plug-ins from UCL constraints over the selected meta-model to another constraint language. Its items are filled by method *FillUCLTranslatinglPlugins*. Combo box *derivationOtherModelComboBox* contains list of meta-models as the target models to derive UCL constraints over the current source model. Its items are filled by method *FillDerivationOtherModelPlugins*.

UCLManager

UCLManager is the main class of project *UCLCore*. It holds all UCL meta-model plug-ins and all translating plug-ins. It implements *DaemonXModuleInfo* to set information about the module “UCL constraints” of DaemonX framework.

Method *GenerateUCLModel* creates UCL model according the selected UCL meta-model plug-in. It sets the expression factory from the selected translation plug-in; or it set the default expression factory. It initializes *DiagramManager*, lexical analyzer and parser to parse UCL constraints.

Method *DeriveUclExpressionsToOthermodel* derives UCL constraint over the selected model to constraint over another model. It calls method *DeriveToOtherModel* to all context block of UCL constraints.

Method *TranslateToUCLToPsmExpressions* derives UCL constraints to constraints in another constraint language. It calls the translating plug-in.