UNIVERSITY OF MANNHEIM
School of Business Informatics and Mathematics

dACL: The Deep Constraint and Action Language for Static and Dynamic Semantic Definition in Melanee

Master Thesis

submitted: December 2016

by: Arne Lange
arlange@mail.uni-mannheim.de

Student ID Number: 1456883
Supervisor: Ralph Gerbig

University of Mannheim
Chair of Software Engineering
D – 68159 Mannheim
Phone: +49 621-181-3912, Fax: +49 621-181-3909
Internet: `http://swt.informatik.uni-mannheim.de`

Abstract

The Unified Modeling Language (UML) is the de facto standard for modeling software. But due to some limitations of UML a new modeling paradigm was born, called Multi-Level Modeling or Deep Modeling, allowing the user to model across multiple ontological levels. The software engineering group at the University of Mannheim has developed a multi-level modeling tool which is called "Melanee". To attract more users to this modeling paradigm, Melanee needs the same extensions as UML, such as a constraint language, an action language or a transformation language in order to help its users create precise and useful models.

There has been some effort to integrate a deep modeling constraint language into Melanee, but the usability of this dialect turned out to be limited. This thesis aims at raising the usability of the deep modeling constraint language dialect by customizing the Object Constraint Language (OCL) standard language definition, which is the constraint language extension for UML. Many of the semantic navigation definitions are based on the work of Kantner[33].

Contents

List of Figures

List of Tables

List of Abbreviations

Alf	Action Language for fUML
ANTLR	ANother Tool for Language Recognition
EMOF	Essential MOF
EMP	Eclipse Modeling Project
EOL	Epsilon Object Language
GLL	Generalized LL
GLR	Generalized LR
LML	Level-agnostic Modeling Language
MDA	Model-driven Architecture
MDD	Model-Driven Development
MOF	Meta-Object Facility
OCA	Orthogonal Classification Architecture
OCL	Object Contraint Language
OMG	Object Management Group
PLM	Pan-level Model
UML	Unified Modeling Language
WCRL	Weight Constraint Rule Language

1. Introduction

Since the rise of object-oriented software developing methods there has been an effort to unify the object-oriented developing methods. The goal of that unification effort was to create a general-purpose modeling language based on standardization of the Object Management Group (OMG) and resulted in the adoption and standardization of the UML in November 1997.[41] In the year 2000, the OMG accepted new proposals for UML2. The UML2 was necessary due to the limited capabilities of the first version of UML. The new version did not change the basic concepts of UML – in fact they remained mostly the same – but it gave UML a clearer design for users and toolmakers.[41] The UML core meta model was unified with modeling parts of the Meta-Object Facility (MOF) which "[permitted] UML models to be handled by generic MOF tools and repositories"[41]. From this Model-Driven Development (MDD) emerged a development paradigm in the object-oriented software community. The goal or "motivation for MDD is to improve productivity"[10]. As an alternative approach to model software, the chair of software engineering of the University of Mannheim developed a Level-agnostic Modeling Language (LML). This language allows the software modeler to model multiple ontological levels.[23]

The tool that implements the LML is called *Melanee*[37, 23] and aims for the same tool support as UML, therefore one of the main reasons to create a useful constraint language dialect for deep modeling. To some extent this support is already present in the current development of Melanee. There exists an implementation of the OCL in Melanee as of this date, but formulated constraints are not saved in any form. This is the first and most obvious feature that has to be implemented in either a totally new version of a deep modeling constraint language or in an extension of the original code written by Kantner[33]. His approach was to utilize the *Eclipse OCL*[19] implementation in terms of evaluating the OCL statements. Kantner's[33] main contribution was the navigation specification in a deep model. This navigation will be presented in chapter 3.1.

This thesis, however, will aim towards a new implementation of a deep modeling OCL dialect. The goals are to seperate this OCL dialect from the *EclipseOCL* implementation and to save the constraint statements on the respective models. The deep modeling constraint language is realized with a tool called "ANother Tool for Language Recognition (ANTLR)" which generates a parser and a lexer from a grammar which defines and checks the syntactic structure of expressions written in this dialect. The foundation of this particular technology will be introduced in chapter 2. Chapter 3 will introduce the modifications made to the OCL standard to fit the deep modeling properties; also the navigation semantics from Kantner will be presented. Chapter 4 will show how the meta-model of the Pan-Level Model (PLM) is extended with the constraint meta-model and how constraints are saved in that structure. The next chapter explains how and when defined constraints are evaluated and how the application, created in this thesis, deals with certain borderline cases in the implementation. Chapter 6 will display the tooling of ANTLR and how the generated software artifacts help to interpret OCL expressions. Chapter 7 will describe what parts of the constraint language are not yet supported and what features are still necessary to increase the usability of the deep OCL dialect. Chapter 8 will introduce other implementations of OCL in an deep modeling environment and will help to contextualize this thesis. The final chapter will sum up the work done so far and will give a final assessment of this thesis.

2. Foundations

This chapter will introduce the main concepts of Deep Modeling, OCL, formal languages and the Eclipse Modeling Project (EMP). With regard to the Deep Modeling topic. The modeling environment Melanee will be used to explain the concept in detail.

2.1. Formal Languages

In order to create a new dialect of OCL which is aware of the deep modeling environment, it is vital to understand how languages are constructed in regard to computer science. Programming languages have to be understood by the user and on the other hand from the computer which has to follow the written instructions and act accordingly. Almost all of the work in the field of formal languages is based on the work of Chomsky[14], who laid the theoretical linguistic foundation for language and grammar classification and later parsing theory.

Computers are only able to understand machine code. Any other higher-level language in which programmers write their programs in, such as JAVA[15], has to be translated into machine code, which the computer can then process. This translation is called *compiling*. Figure 2.1 shows the input and possible outputs of a compiler. If the compiler does not produce a target program as the result of

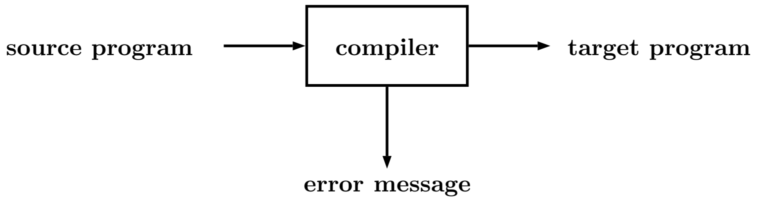


Figure 2.1.: A compiler

the translation of the source program and instead is performing the operations that are implied by the source program, then the translation process is called interpreting. The compiler itself is then called an interpreter.[1]

The whole compiling process is shown in figure 2.2. First the input is divided after each symbol and serves as an input stream for the lexical analyzer. The lexical analyzer takes each symbol and tries to classify each symbol or a set of symbols into categories. These categories are also called tokens. This then serves as an input for the syntax-directed translation process, which is, in this case, a parser. A parser takes these categories and orders them hierarchically into a data structure according to the defined rules of the language. The intermediate representation can vary, but in most cases it is a tree-like structure.[43]

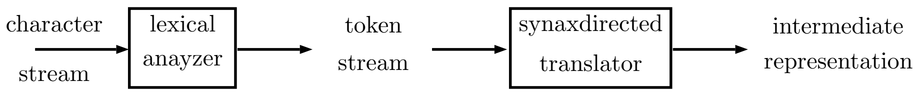


Figure 2.2.: The processing steps of a compiler

According to Aho and Ullman[2], there are some properties that have to be specified in order to create a higher-level programming language:

1. The set of symbols which can be used in valid programs
2. The set of valid programs
3. The *meaning* of each valid program

The first definition is relatively easy to achieve. For most modern high-level programming languages the set of symbols is a mixture of the English alphabet and arithmetic operation symbols. It is much more difficult to define the set of valid programs.[2] When specifying a program language, grammatical rules can be used to reduce the size of the set of valid programs. It may, however, evaluate statements as valid like listing 2.1 shows.[2] This kind of statement will either result in an endless loop or, if the meaning of that valid program is not defined, in some kind of error due to the consequences of the third definition.

Listing 2.1: *FORTRAN* statement that might be considered valid

```
L GOTO L
```

The third and final property says that a programming language has to determine the semantics of a valid program. With regard to third property listing 2.1 has to be either rejected because the meaning is ambiguous or the language accepts this expression and the execution context will be in a state of an infinite loop. Either way, the language specification has to be able to determine the semantics of a valid expression. This definition is the most difficult to achieve and will be part of the next chapters.[2]

The DeepOCL dialect translation has to fulfill these definitions, and every definition can be divided into different phases of translating a statement.

The words λ, 01110,01,00010,0,1 are words over the alphabet $\Sigma = \{0, 1\}$. The set of all words over an alphabet is denoted by Σ^* and by Σ^+ for all non-empty words. These two sets are infinite for any Σ, and they are the free monoids and free semigroup generated by Σ.[42] If a language L is in fact infinite it is not possible to enumerate all possible combinations of words in order to translate the language into executable code. Thus another representation for the translation must be sought. This specification of a language has to be of finite size, but the language specified is not required to be finite.[2]

There are two well-known methods to achieve this requirement. The first method is to define a generative system, which is called a grammar. The second method is being presented with a finite string, which is part of the input, and answers "yes" if the string is part of the language and "no" if the string is not part of the language. This procedure or algorithm is called recognizer.[2] The following sections will explain the first method in detail – because a grammar was used to generates the deep OCL dialect.

2.1.1. Theory

To fully understand the process of language engineering one has to understand how an alphabet, words, languages and their grammars are defined with regard to computer science languages or programming languages. This section defines all the parts that are needed to create a programming language.

Definition 2.1 (Alphabet)**.** An *alphabet* is a finite nonempty set. The elements of an alphabet Σ are called *letters* or *symbols*[42]

Definition 2.2 (Word)**.** Let $\Sigma = \{a, b, c, ...\}$ be a set of symbols, or *alphabet*, then a word w over Σ is a string where each character from w is from Σ. $w = a_0, a_1, a_2, ..., a_n$ where $a_i \epsilon \Sigma$. A special word is the empty word ε.

Definition 2.3 (Length of a word)**.** The length of a word is displayed by $|w|$ where $w = a_1a_2a_3...a_n$,$|w| = n$ and $|\varepsilon| = 0$.

The concatenation of two words is defined in the following.

Definition 2.4 (Concatenation)**.** Let $w_1 = a_0a_1a_2...a_n$ and $w_2 = b_0b_1b_2...b_n$ two different words where $a_ib_i\epsilon\Sigma$, then

$$w_1 \circ w_2 = a_0a_1a_2...a_nb_0b_1b_2...b_n$$

is the concatenation of w_1 and w_2.

Definition 2.5 (Language)**.** Let Σ^* be a set of words, then a language L is a subset of Σ^*

$$L \subseteq \Sigma^*$$

Definition 2.6 (Grammar)**.** A Grammar is a tuple $G = (N, \Sigma, P, S)$ where

(1) N and Σ are alphabets, $N \cap \Sigma = \emptyset$

(2) $S \in N$. The elements of N are called *nonterminals* and those of Σ *terminals.*

(3) S is called the *start symbol.*

(4) P is a finite subset of

$$(N \cup \Sigma)^*N(N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

Definition 2.7 (Regular grammar)**.** A grammar is called *regular* if each production is of the form $u\xi v \rightarrow uyv$, where ξ is in $N - \Sigma$, u and v are in $(N - \Sigma)^*$, and y is in $N^* - \{\varepsilon\}$. The language generated by a regular grammar is called a regular language.

A regular grammar or regular language is also called context-sensitive.[25] That term refers to the fact of using a rule of the from $u\xi v = uyv$ where ξ was rewritten to y. The equivalent automaton that accepts regular grammars is called *finite automaton.*[30]

Definition 2.8 (Context-free grammar)**.** A grammar is called *context-free* $G = (N, \Sigma, P, S)$ in which each production in P is of the form $\xi \rightarrow n$, where ξ is in $N - \Sigma$ and n is in N^*. $L \subseteq \Sigma^*$ is said to be a *context-free language* if and only if L is generated by some context-free grammar.

The term context-free is referring to the fact that rewriting the result of the productions is done independently of the context which each variable appears in.[25] Every context-free grammar is also context-sensitive, but the converse is not true.[25]

Definition 2.9 (Pushdown automaton)**.** A finite deterministic automaton can be defined as a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, where

- Q is a set of states
- Σ is an alphabet called *input alphabet*
- Γ is an alphabet called the *stack alphabet*
- q_0 in Q is the initial state
- Z_0 in Γ is a particular stack symbol called the *start symbol*
- $F \subseteq Q$ is the set of final states
- δ is a mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$

Like regular grammars the set of context-free grammars also have an equivalent automaton which is called *pushdown automaton.* This automaton accepts all context-free grammar definitions. A pushdown automaton is deterministic if the following restrictions hold[30]

1. whenever $\delta(q, a, X)$ is nonempty for some a in Σ, then $\delta(q, \epsilon, X)$ is empty
2. for each q in Q, a in $\Sigma \cup \{\epsilon\}$ and X in Γ, $\delta(q, a, X)$ contains at most one element.

The first restriction prevents the pushdown automaton to choose either the next input or making an ϵ-move. The second restriction prevents a choice on the same input.[30]

To sum up these definitions, first the alphabet of a language was defined, which is a set of arbitrary symbols out of which words can be build. Also the definitions provided a way to measure the length of a word and how two or more words can be concatenated into a new word. Further grammars were defined and and how they relate to languages. A subset of grammars are context-free grammars that generate context-free languages and are accepted by a pushdown automatons. "Context-free grammar are a generalization of regular grammars in that no restrictions are placed on the right hand sides of rules."[43]

By definition every context-free language is generated by a context-free grammar.[36]

An element (α, β) in P will be written $\langle\alpha\rangle \rightarrow \langle\beta\rangle$ and is called a production, where α is the left-hand side and β the right-hand side of the production. In other words, a grammar is context-free if the finite set of rules or productions has only non-terminals on the left-hand side of a production and an arbitrary number of combinations of terminals and non-terminals on the righ-hand side of a production. The left hand side of a production is also called syntactic categories, and every category itself represents a language.[30] The equivalent automaton for context-free grammars is the pushdown automaton. The deterministic version of this device accepts only a subset of all context-free grammars. This subset includes the syntax of most programming languages.[30]

2.1.2. Lexical Analysis

In a compiler or interpreter the first step of processing an expression is the linear analysis or lexical analysis. This step aims to categorize each token or a group of tokens. Assume the input of the translation process is the following statement:

Listing 2.2: Input for the lexical analysis

```
position := initial + rate * 60
```

The process of the lexical analysis would group the expression into the following token categories:

1. The identifier `position`
2. The assignment symbol `:=`
3. The identifier `initial`
4. The plus sign
5. The identifier `rate`
6. The multiplication sign
7. The number 60

The blanks between each token are usually eliminated during the lexical analysis.[1] If the result of the parsing process is fact a parse tree then every token or classification is a leaf in the hierarchical parse tree. The following sections will show

how these rules can organized in such data structures. It will also show that every leaf of the parse tree consist of a token.

2.1.3. Syntax and Semantics

Any language has some grammatical structure which consists of a set of rules on how words form a sentence or how words in a sentence relate to each other. For a natural language, e.g. English, every word of a sentence can be labeled or classified into syntactic categories. The sentence

The pig is in the pen.

can represented as a labeled tree, as presented in in figure 2.3. This representation

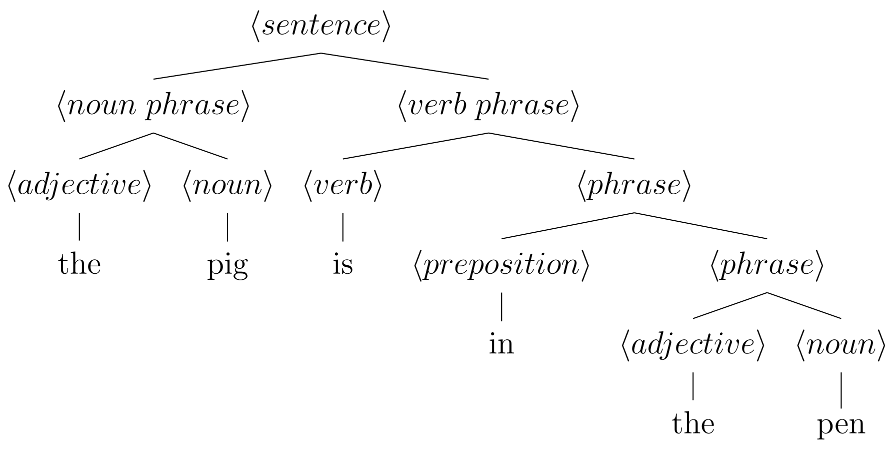


Figure 2.3.: Tree structure for a sentence in the English language[2]

helps understand the overall structure of the English language and how sentences are composed properly. The same principle can be applied for a programming language or any other language for that matter. For example, the arithmetic expression

$$a + b * c \tag{2.1}$$

can be also represented in a labeled tree, just like a sentence from a natural language. The labels may differ, but the tree helps to understand the relation of symbols in that particular language.[2] In figure 2.4, the labeled syntax tree is shown for an arithmetic expression. This example shows a calculation order in which the times calculation is computed before adding the left part of the tree to the result. So in fact this syntax tree ensures the correct calculation of an arithmetic expression. That is because the right term, the times calculation, has to be resolved before the result is added to *a*.

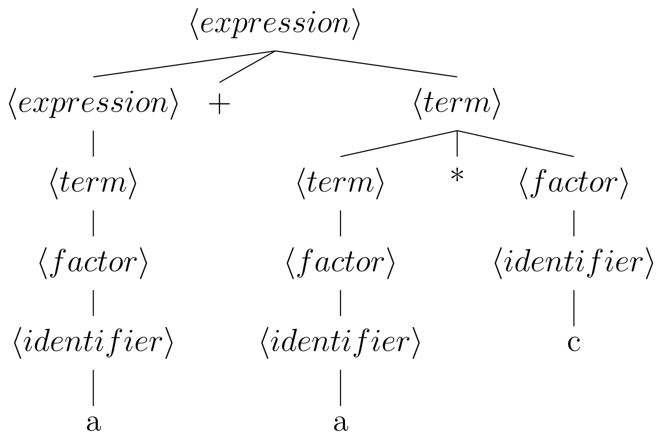


Figure 2.4.: Tree structure of an arithmetic expression[2]

The two examples above show the process of parsing or syntax analysis. We have seen that some rules or production have precedence over others and how these rule can the represented in a hierarchical structure like a parse tree.

The example that was shown earlier in listing 2.2 can also be represented in a parse tree, which is shown in figure 2.5. This example also shows that some

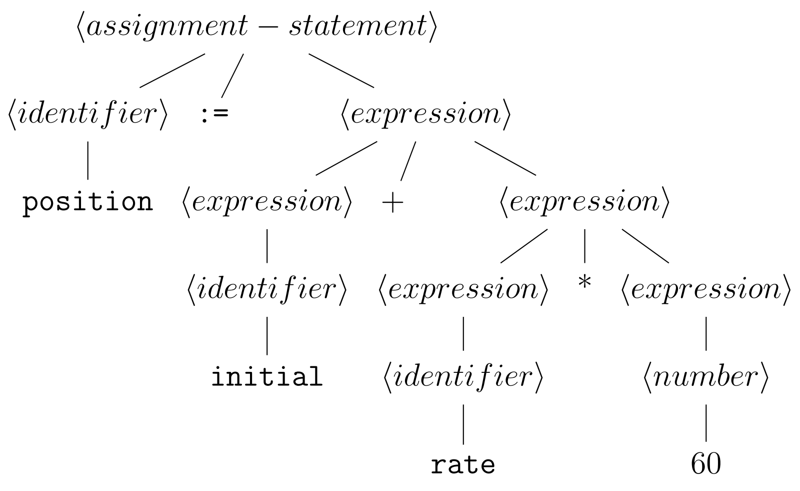


Figure 2.5.: Parse trees for `position := initial + rate * 60`

logical or arithmetic expressions have to be resolved before other expressions. The phrase `rate * 60` is a logical unit because arithmetic rules say that multiplication

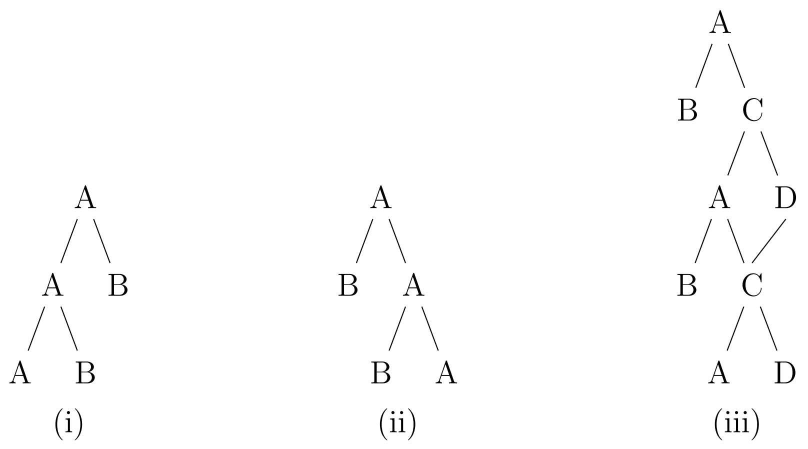


Figure 2.6.: Example parse tree for non-, left- and right-recursion[13]

is performed before addition. The rules that create that kind of hierarchical structure might look as follows:

1. Any `identifier` is an `expression`
2. Any `number` is an `expression`
3. If $\texttt{expression}_1$ and $\texttt{expression}_2$ are `expressions`, then so are

$$\texttt{expression}_1 + \texttt{expression}_2$$
$$\texttt{expression}_1 * \texttt{expression}_2$$
$$(\texttt{expression}_1)$$

The first two rules are non-recursive rules whereas the third rule defines a set of two expressions with different operators applied to the two expressions.[1] Thus, defined by the first rule, `initial` and `rate` are `identifiers`. The `number` 60 is a an expression, which is defined by the second rule. The third rule matches first `rate * 60`, which is an expression, and then to `initial + rate * 60` which is also an expression itself.

Figure 2.6 shows three parse trees that are non-recursive(i), left-recursive(ii) and right-recursive(iii). "We say that φ *dominates* $\psi(\varphi \rightarrow \psi)$ if there is a derivation $\sigma_1, ..., \sigma_n$ such that $\sigma_1 = \varphi$ and $\sigma_n = \psi$ (i.e., if ψ is a step of a σ-derivation)."[13] With that definition by Chomsky it can be shown that A in (i) is nonrecursive for non-null φ, ψ, because $A \Rightarrow \varphi A \psi$. A in (ii) is left-recursive if there is a non-null φ such that $A \Rightarrow A\varphi$. And A in (iii) is right-recursive if there is a non-null φ such that $A \Rightarrow \varphi A$.[13]

In the past it was believed that top-down parser cannot process grammars with left-recursion rules. Furthermore it was believed that the runtime complexity was at least exponential. Recent research showed that top-down parsing algorithms can accommodate grammars with left-recursion production in polynomial time.[22] How ANTLR, the tool used in this thesis, approaches this problem is discussed in chapter 2.1.5.

The separation of the two topics of lexical analysis and of syntactical analysis is to some extent arbitrary[1], because it can vary how the parser comes to the result of the intermediate representation of the expression. Most authors in the field of parsing, like Aho, Ullman[2] and Sippu[43], have decided to divide the parsing process into these two topics. In order show the difference between grouping tokens into lexical categories and reordering the whole expression into a hierarchical system, like a parse tree, the separation helps give a better understanding of the tasks involved in the parsing process.

2.1.4. Parsing Strategies

The parser obtains a stream of tokens from the lexical analyzer and checks if the whole stream of tokens can be generated by the defined grammar for the source language. The position in the compiling process is shown in figure 2.7. There are

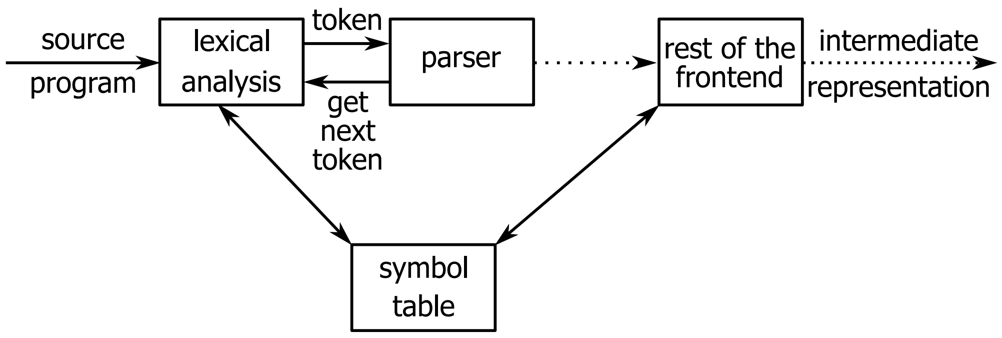


Figure 2.7.: The position of a parser in the compiling process

two types of algorithms that can be used for the task of syntactic analysis: top-down and bottom-up.[32] The top-down parser method builds parse trees from the top (root) to the bottom (leaves). The bottom-up parser method begins with the bottom (leaves) and works its way up to the top of the tree (root). The input for both parser methods is the same, it is scanned from left to right and one symbol at a time is processed. These parsing strategies only accept subclasses of grammars, such as LL and LR grammars.[1] The class of context-free grammars that can be parsed deterministically in a top-down fashion is called *LL(k)*, and the class of context-free grammar that can be parsed deterministically in a bottom-up fashion is called *LR(k)*.[40]

Due to the fact that the generated parser by ANTLR is a top-down parser[39], this thesis will concentrate on this parsing strategy and leave the further definitions for bottom-up parser strategies aside. With the definition of a context-free grammar (cf. definition 2.8) in mind a grammar with LL(k) properties can be defined as follows.

Definition 2.10 (LL(k) grammar)**.** A grammar $G = (N, \Sigma, P, S)$ is said to be a LL(k) grammar for some positive integer k if and only if given

1. a word w in N^* such that $|w| \leq k$
2. a non-terminal A in Σ
3. a word w_1 in Σ^*

There is at most one production p in P such that for some w_2 and w_3 in N

4. $S \Rightarrow w_1 A w_3$
5. $A \Rightarrow w_2$
6. $(w_2 w_3)/k = w$

With regards to the parsing process a "LL(k) grammar is a context-free grammar, such that for any words in its language, each production in its derivation can be identified with certainty by inspecting the word from its beginning (left end) to the *k-th* symbol beyond the beginning of the production. Thus when a non-terminal is to be expanded during a top-down parse, the portion of the input string which has been processed so far plus the next k input symbols determine which production must be used for the non-terminal."[40]

Any context-free language that is generated by a *LL(k)* grammar can be recognized by a deterministic push-down automaton.[35] Every *LL(k)* grammar is also an *LR(k)* grammar.[34]

2.1.5. ANTLR's parsing technology

In general, context-free grammar parsing can either be done in a top-down (*LL-style*) or in a bottom-up (*LR-style*) fashion. In a time where resources in a computer were scarce, programmers had to write their grammars to fit the deterministic parser generators. In modern days, programmers are able to use nondeterministic parsing strategies due to the fact that the resources, available to a computer, have grown.[39] These strategies are called Generalized LR (GLR) and Generalized LL (GLL), and they are able to handle nondeterministic ambiguous grammars. When a grammar is in fact ambiguous, they return multiple parse trees (forests) – because these strategies were intentionally designed for natural languages.[39]

ANTLR is using a top-down strategy called *ALL(*)*.[39] This strategy can also handle nondeterministic and ambiguous grammars, but does not return multiple parse tree, like GLL or GLR. The LL parsing style suspends at each production until the prediction mechanism has chosen the correct production to expand the tree and is resuming the parsing process. The ALL(*) strategy parses the whole expression dynamically. At each decision point in the grammar multiple subparsers are launched. For every possible decision at this point in the grammar one parser is created and tries to match the input. If the path a subparser has been taking fails to match the input it dies off and is not considered a valid production anymore.

Listing 2.3 shows a grammar that has rules that are left-recursive. The rule `expr` is in that form. Because of that form that kind of rule is unacceptable for ANTLR3. ANTLR 4 can handle these rules by automatically rewriting the grammar into a non-left-recursive and unambiguous grammar. The second reason this grammar cannot be accepted by ANTLR 3 is that the `stat` rule has alternative productions that have a common prefix, i.e. `expr`. This rule is undecidable for ANTLR 3 and for LL(*) style grammars.[39]

Listing 2.3: An example of a left-recursive grammar from Parr et al.[39]

```
grammar Ex;
// action defines ExParser member: enum_is_keyword
@members {boolean enum_is_keyword = true;}
stat: expr '=' expr ';' // production 1
    | expr ';' // production 2
```

```
;
expr: expr '*' expr
        | expr '+' expr
        | expr '(' expr ')' // f(x)
        | id
;
id  : ID | {!enum_is_keyword}? 'enum' ;
ID  : [A-Za-z]+ ; // match id with upper, lowercase
WS  : [ \t\r\n]+ -> skip ; // ignore whitespace
```

Before ANTLR4 generates the parser it will rewrite all direct left-recursions rules.

In theory the *ALL(*)* parsing strategy has a runtime complexity of $O(n^4)$ and the GLL parsing strategy has a runtime complexity of $O(n^3)$. Nevertheless when using this strategy for expressions in common languages, Parr et al. showed that the ALL(*) parsers exhibit a linear behavior and complete the parsing process faster than implementations of the GLL parsing strategies.[39]

2.2. Model-Driven Development

"The underlying motivation for MDD is to improve productivity [...]".[10] This sentence sums up what MDD is all about. MDD intends to automate as many routine programming tasks as possible and ease the workload of software developers. It should be clear that the main advantage of the MDD method is visual modeling. That means that everybody who has an understanding of the problem domain is able to model software. The most challenging difficulty in creating a complex software artifact is bridging the gap between the problem domain and the implementation domain.[21] MDD is just a developing method and needs a proper infrastructure which is called Model-Driven Architecture (MDA). Figure 2.8 shows the four-layer meta-model hierarchy of the UML specification. This infrastructure contains four model levels and "each (except the top) [is] characterized as an instance of the level above"[10]. The bottom level is call "M0" and contains the data object the software is meant to manipulate. The level above is called "M1" and holds the user model, which is actually the model of the user data instance. The next level is called "M2", and it incorporates the UML. This level contains the model of the user model and is called meta-model, because it is a model of a model. The *Attribute* class is the meta-class of the attribute in

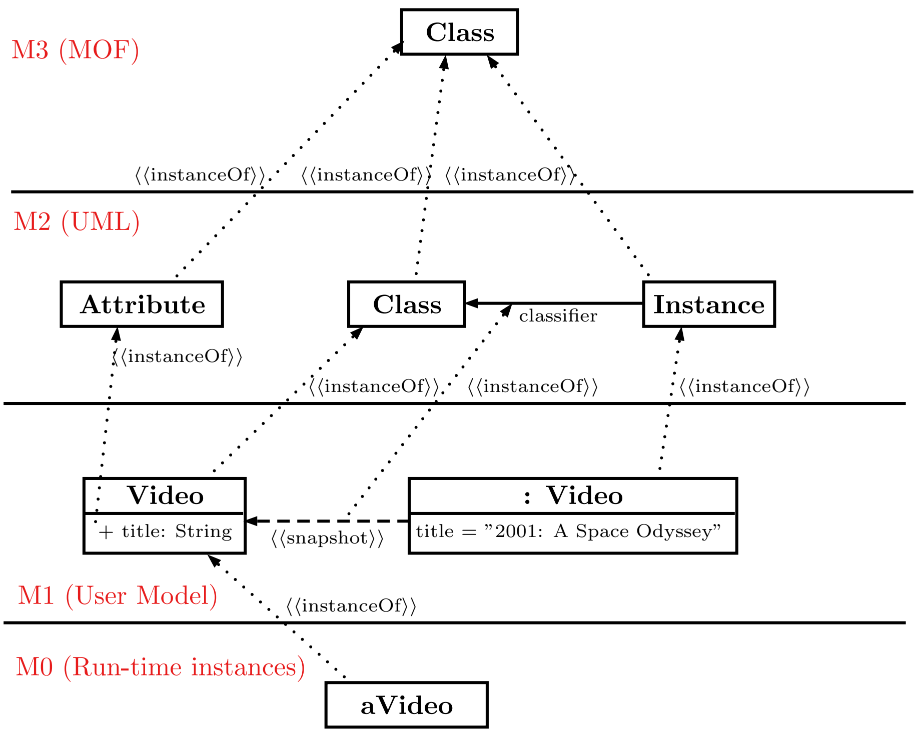


Figure 2.8.: An example of the four-layer meta-model hierarchy[28]

the *Video* class. The same can be said for all the other classification relationships from level M1 to M2, i.e. to their respective meta-classes. The last level is called "M3" and it contains the MOF. The MOF is the model specification for level M2 and all model on M2 are instances of the *Class* model that resides in the MOF.[10] "This venerable four-layer architecture has the advantage of easily accommodating new modeling standards (for example, the Common Warehouse Metamodel) as MOF instances at the M2 level."[10] This four-layer architecture supports MDD and can be characterized as the framework for MDD.[21]

2.3. Multi-Level Modeling

As several authors have pointed out, the classic hierarchical four-layer infrastructure is subject to limitations with regard to meta modeling.[8, 16] The biggest limitation is the existence of only two meta-levels and one kind of instantiation relationship specification.[16] In the meta modeling process it sometime proves useful to use more than two meta-levels, because two levels might able to cover the

”linguistic case, where an object is an instance of exactly one class”[16]. Nonetheless it is not enough to capture any ontological instantiation relationships within the problem domain.[16] If a problem would naturally span over more than two levels, because the problem domain is of such nature, the modeler would be forced to fit the solution of that problem into just two levels. This makes models of such fashion overly complex and crowded.[16]

The concept of *multi-level* or *deep modeling* is one approach to overcome the limitations of the four-layer modeling infrastructure. This thesis will use the term *deep modeling* when referring to this concept. All solutions for this dilemma have one idea in common, which is “[increasing] the flexibility of the metamodeling architecture by allowing an arbitrary number of meta-levels.”[16] Another principle the concept of deep modeling is based on is the principle of dual instantiation.[9, 16] Figure 2.9 displays the principle of dual instantiation. On

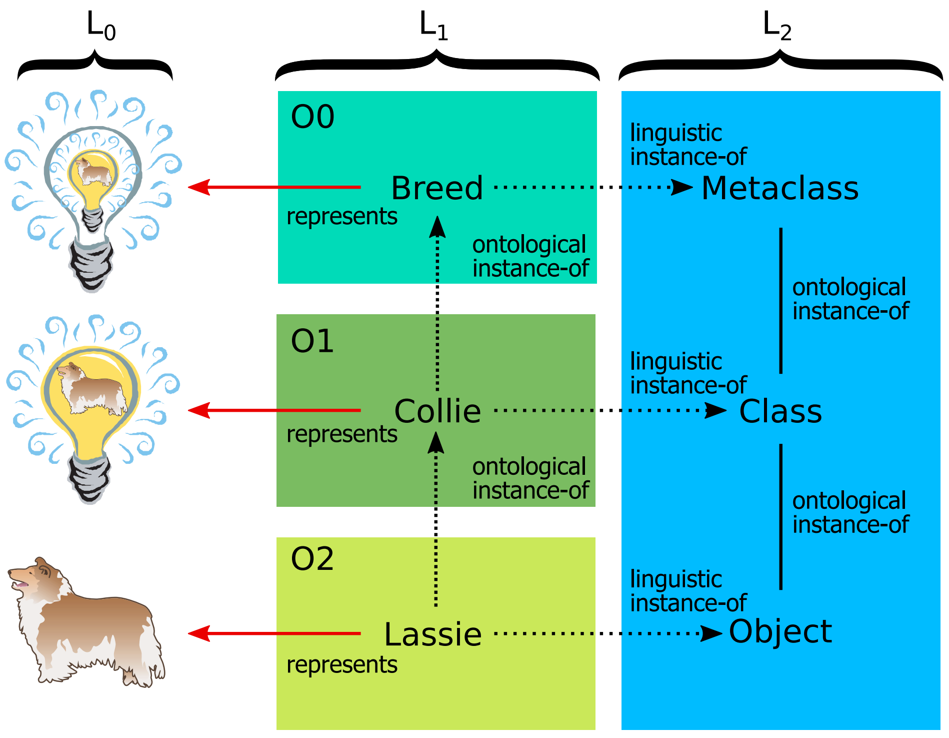


Figure 2.9.: The ontological metamodeling view[10]

the right-hand side in the blue box the linguistic meta-model is depicted. On the top resides the *Metaclass* entity, which is the ontological type of the *Class* entity. The *Object* entity is an ontological instance of the *Class* entity. This whole level is called *L0*. The *L1* level displays the logical dimension of the problem domain and can have an arbitrary amount of ontological levels. In this example, the model has three ontological levels, which are called *O0*, *O1* and *O2*. The first ontological level contains the *Breed* entity, which is a linguistic instance of *Metaclass*. The second level accommodates the *Collie* entity, which is classified by two types. Its ontological type is the *Breed* entity and the linguistic type is the *Class* entity. The third level consists of the *Lassie* entity which is also typed by two different entities. The ontological type of *Lassie* is the *Collie*, and the linguistic type is *Object*. Every element on the *L1* represents elements in the real world. *Breed* represent the idea of the idea of *Lassie* or the idea of a classification for dogs, which is called breed. The *Collie* element represents the concrete type of breed. *Lassie* represents the concrete type of *Collie*.

2.4. The Level-Agnostic Modeling Language

In order to create a tool to support deep modeling, Atkinson, Kennel and Groß[7] developed the LML. As stated by Atkinson et. al[7], the goals of the LML are to resemble UML. Undeterred by the weaknesses of UML, it is the de facto standard for modeling software in a graphical fashion and "the LML was designed to adhere to the concrete syntax and modeling conventions of the UML [to the greatest extent possible]."[7] The second goal for the LML was to be level-agnostic and to support the Orthogonal Classification Architecture (OCA) which builds the framework for the deep modeling paradigm. The third goal of the LML is to support as many mainstream modeling paradigms as possible. The last goal is to provide support for a reasoning service, which UML lacks.

Figure 2.9 is an example to demonstrate the structure of the OCA. It consists of two classification dimensions, the ontological and linguistic dimension, which are orthogonal to each other. The *instanceOf* classification relationship can cross level boundaries.[6] The LML resides at level *L1* in the aforementioned figure and every element or entity of that level is an instance of one of the entities shown in level *L0*, without regard to the ontological classification. The *L0* level contains the PLM which is the linguistic meta-model for the LML. *L2* is designed to contain elements of the real world.

From an ontological standpoint, the element in level *O1* contains an object that is an instance of *Bred* which is simultaneously a type for the *Lassie* entity. Due to the fact that entities in deep modeling can both be an object, i.e., an instance of something, and a type for other objects in the lower levels, theses entities are called *Clabject*, a combination of *Class* and *Object.*[6]

2.5. Object Constraint Language

This chapter introduces the Object Constraint Language which is closely linked to this thesis. The OCL is a vital part of the implementation of the application presented in this thesis and it is important to grasp the basic ideas of what OCL is.

Kleppe and Warmer[44] characterize OCL as follows:

- OCL is a modeling language which is an add-on to UML
- OCL is a query and a constraint language
- OCL is a declarative language

The consequences of these characteristics are that OCL can never be used as a stand-alone modeling language. It has to be applied to a model of some kind. In the context of this thesis, OCL statements are part of a deep model. Every OCL expression is based on a class specification in a model.[44] A constraint, with regard to Kleppe and Warmer[44], is a restriction on one or more values of a model or part of a model. OCL is used to add information and preciseness to a model, which UML diagrams may not always be able to express. Every constraint defined on that model has to be evaluated to true, i.e., if the defined constraint is not a *def* or *body* constraint.

Since UML2, the viewpoint of the OMG has been changing towards a combined query and constraint language. With OCL it is also possible to query and reference models. Kanter[33] implemented a dialect which used the semantics of OCL but only queried models instead of saving different types of constraints to a model. Due to the nature of UML and the fact that these models cannot be executed, OCL is reckoned a declarative language. It describes *what* has to be done and not *how.*[44]. The modeler is able to express procedures of the model if the model is eventually translated, implemented and then executed. The modeler who writes OCL statements in a UML model just expresses rules in the modeling realm and declares these rules on an abstract level. Kleppe and Warmer[44] also state that using OCL has no side effects on the UML model. Again, due to the fact that UML models do not possess the ability to be executed out of box, this statement is true. How these properties can be transferred is subject to the chapter 6.

2.5.1. The OCL Meta-Model

OCL is a typed language. Every expression is either typed explicitly or the type is derived statically.[27] Figure 2.10 shows the OCL types specification. All the instances of these meta-model classes are types themselves and are not instances of the domain they represent.[27]

The OCL specification divides the whole functionality up into three packages. The first package is called BasicOCL and is exposing a very limited amount of functionality of OCL. The second package is called EssientialOCL and is exposing the minimum required amount of functionality to work with EMOF. It is also built using the BasicOCL package, and there is no structural difference between those two. These combined packages are basically a subset of OCL.[27] The third package is called CompleteOCL and represents the OCL standard library.

Figure 2.10 shows the type meta-model for the OCL 2.4 specification. The types in the white boxes are imported from the Essential MOF (EMOF) package, and the yellow boxes represent the types that are specifically defined for EssientialOCL.[27] The abstract syntax definition of OCL is itself divided into several packages. The *Types* and the *Expression* package are described in the following.

The *Types* packages describes the type system of OCL. It shows the predefined OCL types and "which types are deduced from the UML models."[27] In OCL there are four different collection types defined. These are *OrderedSet*, *Set*, *Bag* and *Sequence*. Every collection type in OCL conforms to the MOF Collection type as shown in table 2.1.

Figure 2.11 shows the core of the OCL expression package. This package describes the structure of OCL expressions. Every *OclExpression* has a type, hence every element that inherits from *OclExpression* has a type. Every expression can be statically determined by the application through analyzing the expression in the given context.[27] The *OclExpression* is the abstract superclass for all other

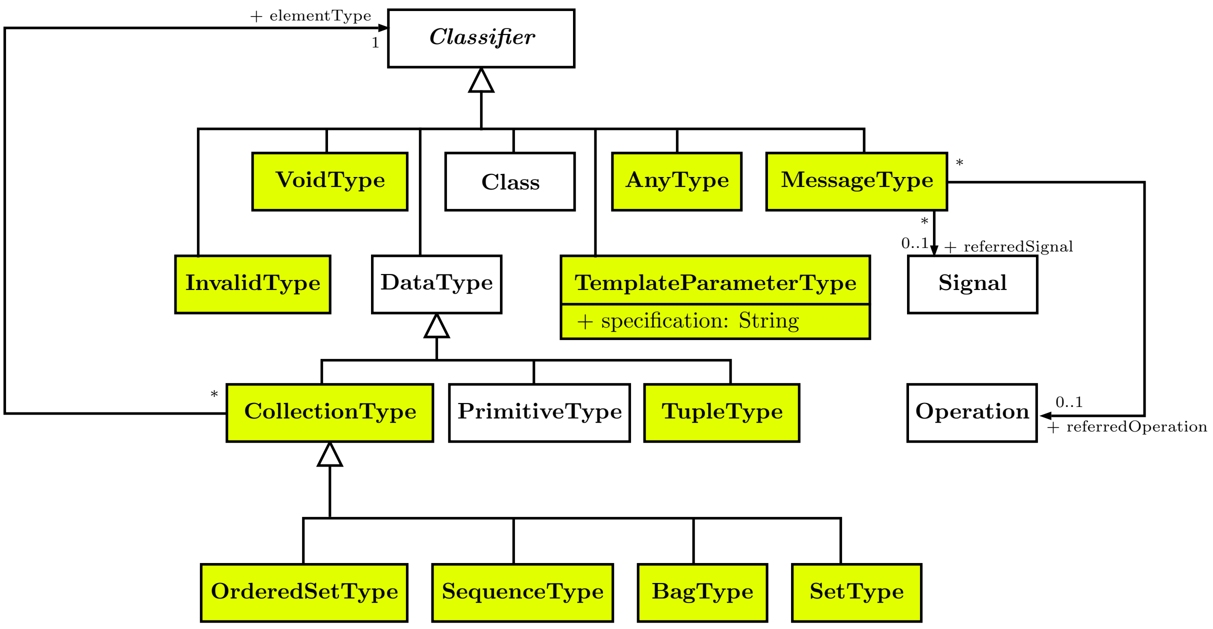


Figure 2.10.: Abstract Syntax Kernel Metamodel for OCL Types as specified in[27]

Type	**Conforms to**	**Condition**
Set(T1)	Collection(T2)	if T1 conforms to T2
Bag(T1)	Collection(T2)	if T1 conforms to T2
OrderedSet(T1)	Collection(T2)	if T1 conforms to T2
Sequence(T1)	Collection(T2)	if T1 conforms to T2

Table 2.1.: Collection type conformance to the MOF; adapted from the OCL 2.4 specification[27]

expression types in the meta-model. Every evaluation of an expression results in a value. Expressions with a boolean result value can be used as constraints, and expressions with any type can be used to formulate queries or to initialize and derive attributes.[27]

2.5.2. Collection and Loop Operations

There are many operations that can be applied on collections. This part of the thesis shows those operations and gives an introduction on their semantics. An *X* implies that the operation at hand is defined for that particular collection type.

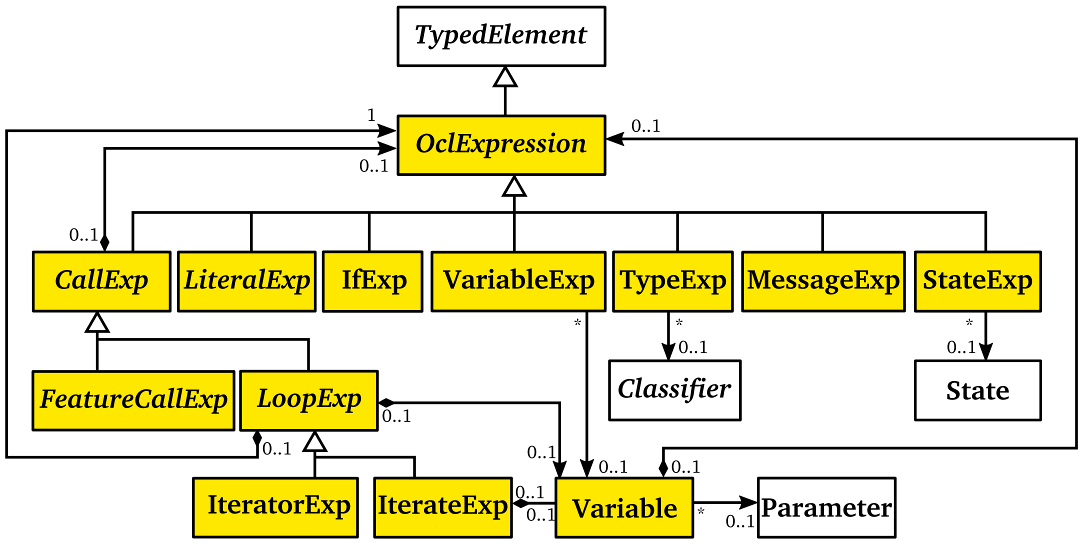


Figure 2.11.: The basic structure of the abstract syntax kernel meta-model for expressions as specified in[27]

Operation	Set	OrderedSet	Bag	Sequence
=	X	X	X	X
<>	X	X	X	X
-	X	X	-	-
append(object)	-	X	-	X
asBag()	X	X	X	X
asOrderedSet()	X	X	X	X
asSequence()	X	X	X	X
asSet()	X	X	X	X
at(index)	-	X	-	X
excluding(object)	X	X	X	X
including(object)	X	X	X	X
first()	-	X	-	X
flatten()	X	X	X	X
indexOf(object)	-	X	-	X
insertAt(index, object)	-	X	-	X
intersection(collection)	X	-	X	-
last()	-	X	-	X
prepend(object)	-	X	-	X
subOrderedSet(lower, upper)	-	X	-	-

subSequence(lower, upper)	-	-	-	X
symmetricDifference(collection)	X	-	-	-
union(collection)	X	X	X	X

Table 2.2.: Collection Operations and their semantics

As shown in table 2.2, there are some operations that are defined for a certain type of collection or for a set of collection types. There are, however, some operations that are defined for any collection type. These operations are referenced in table 2.3.

Operation	Description
count(object)	How often does the object occur in the collection
excludes(object)	True if the object occurs not an element in the collection
excludesAll(collection)	True if no element of the passed collection occurs in the actual collection
includes(object)	True if the object occurs in the collection
includesAll(collection)	True if every element of the passed collection occurs in the actual collection
isEmpty()	True if the collection does not contain any element
notEmpty()	True if the collection contains at least one element
size()	Amount of elements in the collection
sum()	The sum of all elements of the collection. Each element has to be from a type that supports addition (Real, Integer)

Table 2.3.: Collection Operations for any collection type

Loop operations in OCL are different from all the operations shown in 2.2 and 2.3. Such operations iterate over an entire collection and compare each element or the value of an attribute of each element with the given expression, and either return a new collection, which is a subset of the source collection, or return a boolean value. These operations are displayed in table 2.4.

Operation	Description
any(expr)	returns a random element of the source collection if for any element the expression holds
collect(expr)	returns a new collection of objects, which are usually values of attributes of each element
collectNested(expr)	returns a collection of collections which yield the evaluation of the expression for each element
exists(expr)	returns true if there exists at least one element in the source collection for which the expression holds
forAll(expr)	returns true if the expression holds for every element in the source collection
isUnique(expr)	returns true if the evaluated value of the expression is different for each element in the source collection
iterate(...)	iterated over every element in the source collection
one(expr)	returns true if there is exactly one element in the source collection for which the expression holds
reject(expr)	returns a subset of the source collection without any element for which the expression holds
select(expr)	returns a subset of the source collection with any element for which the expression holds
sortedBy(expr)	returns a new collection where all elements of the source collection are sorted by the expression

Table 2.4.: Loop Operations and their semantics

Every loop operation in OCL can be reduced or rewritten as an *iterate* operation. An example of an *iterate* operation can be seen in listing 2.4. The *element* variable is the iterator variable, i.e., it changes with every iteration. The *result* variable is the variable which will accumulate the results of each iteration. The body of this operation can be found after the "|" symbol, which is the expression that will produce the result of each iteration.

Listing 2.4: Iterate operation OCL expression

```
collection -> iterate( element: Type1; result: Type2 = <expression>
    | <expression-with-element-plus-result>)
```

This is example shows the abstract syntax definition of an *iterator* operation. Listing 2.5 shows the concrete application of an *iterate* statement.

Listing 2.5: Concrete example of the iterate operation

```
Set{1,2,3} -> iterate( i: Integer; sum: Integer = 0 | sum + i)
```

This statement iterates over the set with three elements, which are from the type *Integer*, and adds them up to a sum. The result of this operation is six. The listing 2.5 can actually serve as the *sum()* implementation from table 2.3.

3. DeepOCL Revisited

This chapter discusses the modifications made to the OCL grammar specification in order to build a valid grammar that creates a lexer-parser pair which accepts expressions made in the DeepOCL dialect. The complete grammar, written in ANTLR4, can be found in appendix A.

The DeepOCL dialect is applied on deep models, hence the semantic definitions of OCL have to be modified to fit the deep modeling context. But not only the semantic definitions have to modified, but several syntactic specification have to be altered to support the full functionality of a deep modeling environment.

3.1. OCL Modifications for Deep Models

As stated in chapter 2.3, the deep modeling environment provides a linguistic and an ontological dimension. The consequence for a deep constraint language is that constraints are definable for both dimension. For example, a user can define a constraint that expresses the need for at least three levels in the deep model. The default context for the constraint is the type of the context, i.e., if the context is the *Clabject* meta class, the context lies in the linguistic dimension. If on the other hand the context is an instance of the *Clabject* meta class, or any other meta class for that matter, the context lies in the ontological dimension.

If, for some reason, a switch of the context dimension is needed, the user can wrap the navigation expression in '#' symbols. In the previous implementation of DeepOCL, as provided by Kanter[33], the proposed way of switching the dimension context is displayed in listing 3.1.

Listing 3.1: Linguistic Navigation as proposed by Kantner

```
context SportsCar
self._l_.getAllAttributes()->any(name="acceleration").value
```

This rather complicated way of escaping the keyword *l* with an underscore on either side is due the fact that the implementation from Kantner was based on the EclipseOCL implementation. This dependency and the fact that the grammar is not changeable without knowing the exact ramifications of said change prohibited Kantner from defining his own syntax definitions for a DeepOCL dialect. This syntax construct was the only way to add the feature of linguistic navigation.

Although this syntax definition seems simple enough to prevent ambiguities, it leaves some questions unanswered. Is it possible to combine two linguistic navigations? If yes, how? And if we switch to the linguistic dimension by invoking _*l*_, is it possible to switch back to the ontological dimension in the same expression?

Examining listing 3.1 we find that two conceivable outcomes are possible. The first one is that after one navigation in the linguistic context the context switches back to the "normal" ontological context. The second one is that once the linguistic navigation is triggered, the dimension stays the same until the end of the OCL statement.

To resolve these possible ambiguities the following will propose a new syntax definition for navigating the linguistic dimension in the deep model.

Listing 3.2: A new proposal for linguistic navigation in a deep model

```
context SportsCar
self.#getAllAttributes()# -> any(a|a.#getName()# = "acceleration").#
    getValue()#
```

This thesis proposes another way of switching to the linguistic dimension of a deep-model element. The '#' symbols enclose the linguistic navigation statement. As shown in listing 3.2, the user knows exactly where the linguistic navigation begins and where it ends. When the collection iteration operation `any` is querying for the name of each attribute, another switch into the linguistic dimension is necessary. The listing 3.1 seems to stay in the linguistic dimension for the whole OCL expression, because in front of `name` there is no explicit operator that would perform the dimension switch.

The DeepOCL dialect displayed in this thesis performs just one dimension switch at the time and immediately switches back to the ontological dimension after the linguistic navigation has been resolved. That means if the user wants navigate in the linguistic context twice each expression has to be enclosed by '#'. Listing

3.3 shows an example of such expressions. After the linguistic navigation ends the user can navigate and perform collection operations again on the ontological level. This syntax definition of a linguistic navigation is precise and leaves no room for ambiguities.

Listing 3.3: A new proposal of linguistic navigation in a deep-model

```
context Manager
inv salary: self.#getDirectSupertype()#.#getSubtypes()# -> reject(
    self) -> forAll(m|m.salary < self.salary)
```

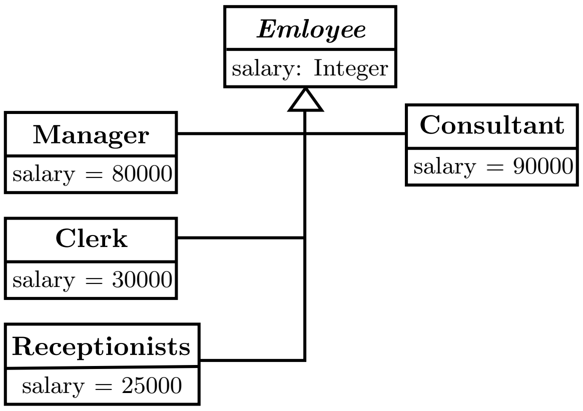


Figure 3.1.: An example for linguistic navigations

Figure 3.1 displays the model which the OCL expression from listing 3.3 is applied on. The expression will not hold until the *Manager* is the highest-paid employee. The context is *Manager* and the expression navigates to the direct super type of itself, which is the abstract class *Employee* from which *Manager* inherits. After the first navigation, the linguistic dimension context is left and the user can decide to either continue navigating in the ontological dimension or add another linguistic navigation.

The second navigation of listing 3.3 is also a linguistic one and yields a collection of all sub-types of *Employee*, which include *Manager*, *Clerk*, *Receptionist* and *Consultant*. This collection is the input for the `reject` operation with the parameter `self`, which means that every element of the collection is collected and returned in a new collection except the *Manager* class. Then this collection is the input for the `forAll` collection operation, which returns *true* if the *Manager* is the highest-paid employee and false if any other *Employee* has the same or a higher salary value. If in fact the `forAll` operation returns false, then the invariant constraint evaluates to false and the model is, with regard to the defined constraint in listing 3.3, not valid.

3.2. Unambiguous Multi-Level Navigation

The ontological navigation is also different from the OCL standard. In the normal OCL implementation there is no need to navigate on and to different levels, because in the UML context the user models only in one level. Kantner[33] also proposed a way to navigate on different ontological levels. At the moment it is only possible to navigate to higher levels. If the context resides at the third level, the navigation allows only to reach levels one and two. Listing 3.4 shows how this type of navigation was realized in the previous DeepOCL implementation.

Listing 3.4: Ontological navigation as proposed by Kantner

```
context SUV
inv wheels: self._CarType_.wheels
```

Again the navigation target is decorated with underscores. Here there is no misunderstanding where the ontological navigation begins and when it ends. Once the navigation reached the model in the upper levels the user uses the navigation from the upper levels, in this example the *wheels* navigation. The result of this kind of navigation is a set that contains every entity which can be classified by *Wheel* but is connected to *SUV* by the *wheels* connection.

Listing 3.5: A new proposal for ontological navigation

```
context SUV
inv SUVWheels: self.$CarType$.wheels
```

The new implementation uses '$' symbols to enclose the ontological navigation.

3.3. Deep (Re-)Classification Operations

To check the classification relationship of models OCL, provides two operations, `oclIsKindOf(type)` and `oclIsTypeOf(type)`, and to retype or cast a model to another type, OCL provides the `oclAsType(type)` operation. The operations can be invoked on a source object and is then checked against the type of the passed argument. The `oclIsTypeOf(type)` operation evaluates to true only if the source object's type is identical to the argument. The `oclIsKindOf(type)` operation evaluates to true if either the source object's type is identical to the argument's type or identical to any of the subtypes of the argument.

To illustrate the semantics of these operations in a normal not-multi-level modeling context, figure 3.2 shows a model in which two classes, *Earning* and *Burning*, inherit from the *Transaction* class.[44] Listing 3.6 shows valid invariant con-

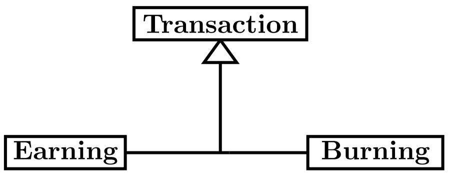


Figure 3.2.: An example model for the `oclIsKindOf()` and `oclIsTypeOf()` operations

straints. It be seen that the rules, that are described above, are holding. The first invariant constraint evaluates to true because the *Burning* class inherits from the *Transaction* class, i.e., *Burning* is identical to one of the subtypes of *Transaction*. The second invariant constraint evaluates to false because *Burning* is not identical to *Transaction*.

Listing 3.6: oclIsKindOf and oclIsTypeOf invariant constraints

```
context Burning
inv: self.oclIsKindOf(Transaction) = true
inv: self.oclIsTypeOf(Transaction) = false
inv: self.oclIsTypeOf(Burning) = true
inv: self.oclIsKindOf(Burning) = true
inv: self.oclIsTypeOf(Earning) = false
inv: self.oclIsKindOf(Earning) = false
```

Now the DeepOCL dialect has to evaluate OCL constraints in the deep modeling environment and the dual facets of *Clabjects*, which are described in chapter 2.3, prevent the proper use of those operations. Kantner tried to fit the operations into the deep modeling environment and proposed a semantic definition that allows these operations to work in a two-level window of the whole model. Figure 3.3 shows a two-level window of a model which illustrates the usage of the `oclIsKindOf` and `oclIsTypeOf` operation. The red dotted arrows represent an evaluation to false and the green arrows indicate an evaluation to true. This

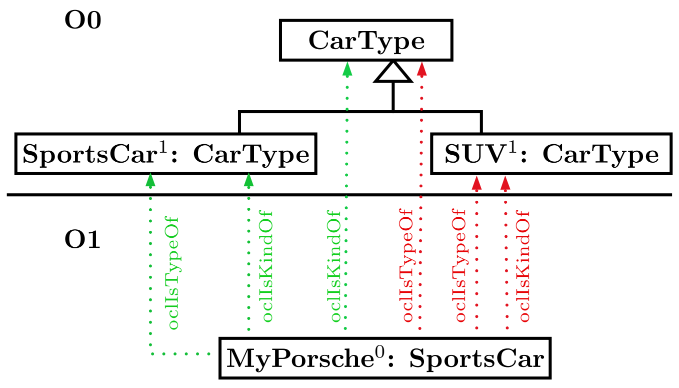
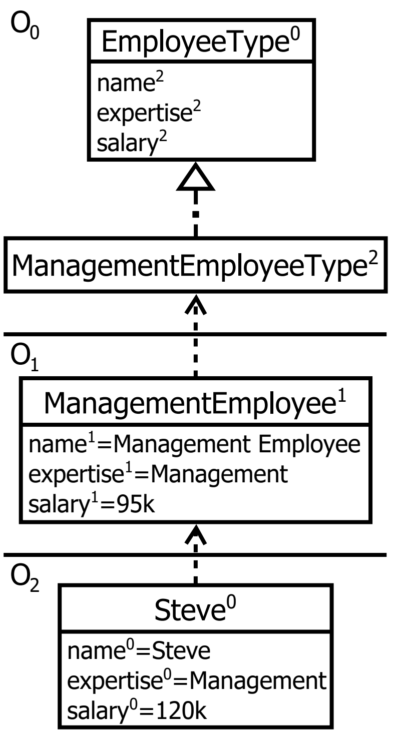


Figure 3.3.: An example model for the `oclIsKindOf()` and `oclIsTypeOf()` operations in a deep modeling environment as proposed by Kantner[33]

thesis proposes a new way to handle both operations. Because the multi-level modeling paradigm does not have the classical distinction between `oclIsKindOf` and `oclIsTypeOf`, these operations have to be either redefined with regard to their semantic definition or they have to be substituted with other operations which provide a similar functionality. There are many linguistic operations that can be used to query the relationship from one *Clabject* to another. The approach of this thesis is to limit the scope of the arguments to the elements of the linguistic dimension. Thus the arguments have to be of type *Clabject*, *DeepModel*, *Feature* or their respective subtypes.

To substitute the missing functionality, the following operations and their semantic definitions are available which are displayed in the table on the right-hand side of figure 3.4. The aforementioned table is supplemented by an example model (a) to illustrate the semantic definitions (b) of the depicted operations.[24] In a deep modeling environment the classification relationship of *Clabjects* can be to some extent translated to the `oclIsKindOf` operation. The operation that corresponds to the `oclIsKindOf` operation is the `isDeepInstanceOf` operation that iterates over the whole classification hierarchy and checks if the source object's type is identical with one the iteration elements. The `instanceOf` operation just checks the direct type, i.e., *Steve* is of type *ManagementEmployee*.

To recap this section, `oclIsKindOf` and `oclIsTypeOf` are only allowed to accept arguments from the linguistic meta types of the meta-model, i.e., *Clabject*, *Deep-*



(a)

context Steve isInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	false
EmployeeType	false

context Steve isDeepInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	true
EmployeeType	true

context Steve isDirectInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	false
EmployeeType	false

context Steve isDeepDirectInstanceOf(...)	
ManagementEmployee	true
ManagementEmployeeType	true
EmployeeType	false

context Steve isIndirectInstanceOf(...)	
ManagementEmployee	false
ManagementEmployeeType	false
EmployeeType	false

context Steve isDeepIndirectInstanceOf(...)	
ManagementEmployee	false
ManagementEmployeeType	false
EmployeeType	true

(b)

Figure 3.4.: Classification checking methods on the example of *Steve*.

Model, *Level* and *Feature* and their subtypes. Kantner redefined the semantics of both OCL operations. This thesis also changed the semantic definitions of both operations, but instead of attempting to fit them into the multi-level modeling paradigm they are limited in their functionality. To substitute these operations the framework provides a precise set of operations for comparing types in a deep modeling environment.

The `oclAsType(type)` operation is also only valid for types from the linguistic dimension. When using this operation in the ontological dimension, it means that source element has to be cast to an element of a classification relationship from an upper level. For this scenario the level cast operation was defined and is indicated by '$' symbols.

The OCL specification defines a `allInstances()` operation that is executable on every element of a model and returns a set of all instances of the element and all its subtypes. In order to apply this operation to a deep-modeling environment the scope of this operation might not be sufficient. In a deep model the classification of instances can be stretched over multiple levels. The operation that returns all deep instances of a deep model is called *allDeepInstances()* and returns the whole classification tree of the element, that can extend over multiple levels. It is also possible to execute the `allInstances()` operation but it returns only the direct ontological instances of the element.

3.4. Deep Constraints

When navigating ontologically, the user of the DeepOCL dialect has to be aware that when the definition and the execution context of the constraint is not the same, then the lower bound cardinality of the connection ends function as a value of instances for the respective *Clabject*. Figure 3.5 shows the model which the following OCL expressions are applied on.

Listing 3.7: ontological navigation example

```
context JoesCar
inv wheels: self.$Car$.wheel -> includesAll(Bag{frontLeft,frontRight
    ,rearLeft,rearRight})
```

Listing 3.7 shows the normal ontological navigation which elevates the execution context of the constraint to the level where *Car* is located. Due to the fact that all cardinalities of the connections from *JoesCar* to each wheel on the level *O2* have the value *1*, the *wheel* navigation yields all the instances of *Wheel* which are connected to *JoesCar* at level *O2*. The *includesAll* collection operation checks if all instances of *Wheel* that are connected to the *Car* instance are actually the result of said navigation.

Anther example is shown in listing 3.8 and shows the same principle of navigation but serves to clarify the consequences of the ontological navigation in combination with cardinality definitions on connections in the model.

Listing 3.8: Ontological navigation with a higher cardinality higher than one

```
context Dragster
inv test: self.$Car$.wheel -> includesAll(Bag{SmallWheel, SmallWheel
    , BroadWheel, BroadWheel})
```

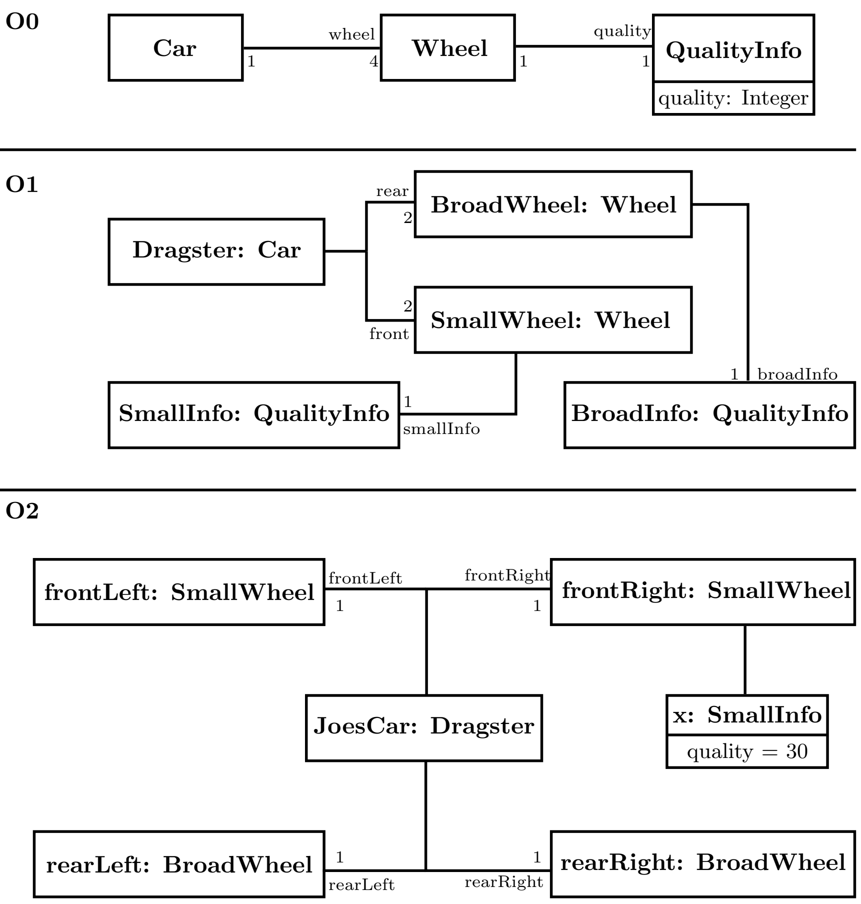


Figure 3.5.: An example for ontological navigation where the definition and execution context are not the same

Instead of returning a *Bag* that just contains two elements, one *SmallWheel* and one *BroadWheel*, the navigation returns a *Bag* with four items, which is displayed in the *includesAll* collection operation of listing 3.8. Due to the fact that the lower bound of the cardinality is two for the connection from *Dragster* to *SmallWheel* and to *BroadWheel*, the cardinality of said connections behaves like an indicator of how many instances are concealed in it. The model is correct if every instance of *Dragster* has exactly two *SmallWheel* instances and two instances of*BroadWheel* connected to it.

If navigating in a ontological level above the context-level, the rule is that the navigation returns a *Bag* with as many elements in it as high as the lower bound of the cardinality of the navigation.

If navigating at the same level as the definition context of the constraint, without using the ontological cast navigation, and if the lower bound of the cardinality of that navigation is *1*, then one atomic element is returned.

Listing 3.9: Ontological navigation with cardinality equal to one

```
context JoesCar
self.frontLeft -- frontLeft
```

Listing 3.9 serves to display the result of the navigation from *JoesCar* to *frontLeft*. The cardinality of the displayed navigation in listing 3.9 takes the value *1*, and as the result of this navigation the *frontLeft* entity is returned without being enclosed in a collection.

4. Constraint Definition Meta-Model

To support constraints in a deep modeling workbench they either have to be part of the meta-model definition or they have to be stored and loaded in another format beside the models. This thesis opted to use the first option. By extending the PLM meta-model with constraints, storing them is done automatically by the application if they are put properly in the containing list of their respective element. This chapter will show how the constraints are supported in Melanee and how the application will save constraints in a partly change agnostic fashion.

4.1. Constraint Meta-Model

The first and most obvious change to Kantner's[33] implementation is the persistence of constraints in the model. In his implementation constraints were not saved in any form.

Figure 4.1 shows the extension point in the PLM meta-model to support constraints. The abstract class *AbstractConstraint* has a name attribute as every constraint has to have a name for the purpose of identification. The class is connected to the abstract class *Element* in a composition relationship, i.e., every element of type *AbstractConstraints* and its subtypes are contained in that relationship. The *Element* class is the most generic class in this meta-model and all relevant classes, like *Clabject*, *DeepModel*, *Level*, *Attribute* and *Method*, inherit ultimately from that class. That means every instance of any of those classes, as displayed in the diagram, can contain an arbitrary number constraints.

In figure 4.2 the constraint meta-model is displayed in detail. This is the meta-model that extends the PLM meta-model with constraint definition. A constraint contains zero or one *Level* class, where the `startLevel` and `endLevel` are defined as integer values. Some constraint types do not need a level specification, like definition, body, pre and post constraints, because a method is only executed once

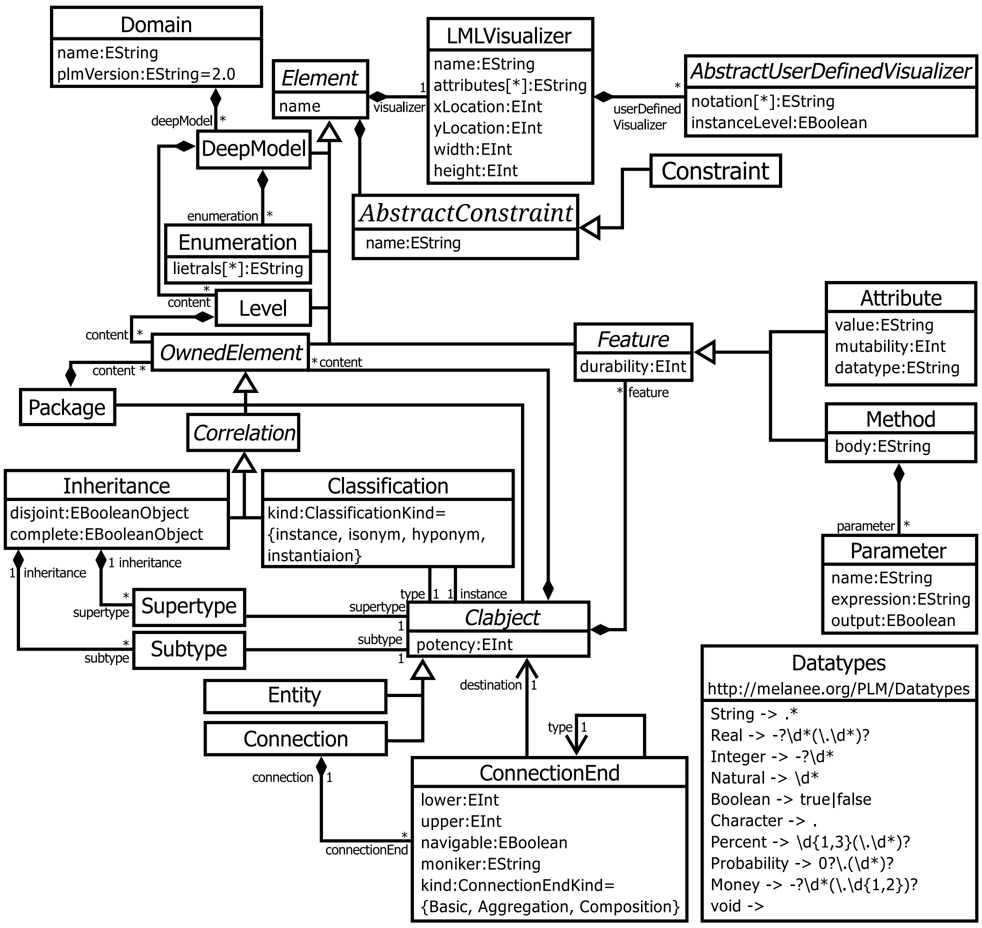


Figure 4.1.: PLM meta model with constraints

in the current navigation context. If the constraint definition is not equal to the current navigation context, the search algorithm searches for next most concrete constraint definition for that method. Every other constraint has to specify a level range for which they are to be evaluated. A constraint also contains an arbitrary amount of expressions, which is split up into *Text* classes and *Pointer* classes. The *Expression* class is abstract, i.e., the class cannot be instantiated, only the subclasses, *Text* and *Pointer* are instantiatable. The classes that inherit from *Constraint* reflect the fact that there are seven different constraint types in the OCL specification. The user can define a severity and a message with every constraint. If a constraint evaluates to false or is not valid, this information will be displayed to the user. The *Severity* enumeration is the type for the `severity` attribute, and the `message` attribute is from type *EString*.

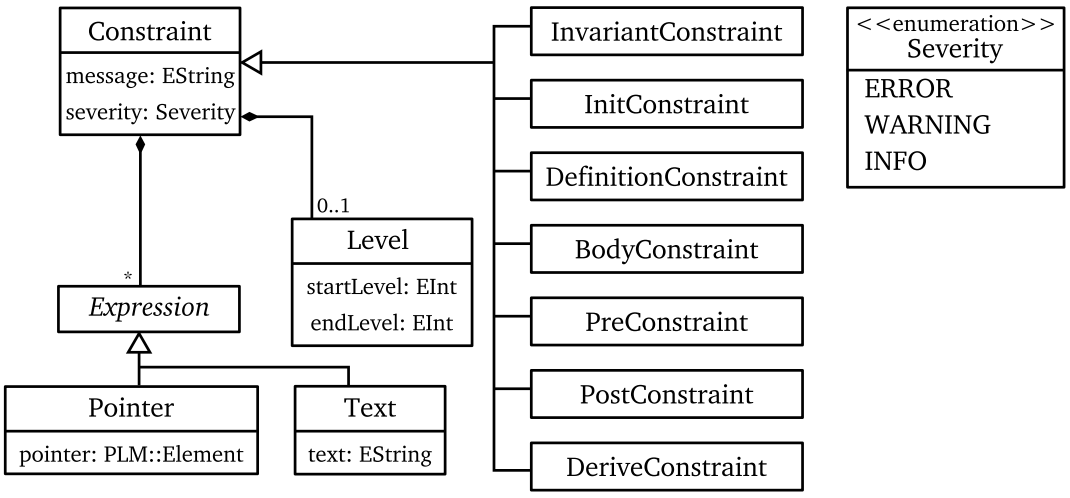


Figure 4.2.: The constraint meta model

4.2. Saving Constraints

Figure 4.3 shows the result of the algorithm that is responsible for saving a constraint. On the top right side the OCL expression that is to be saved is shown. Under that expression the *Person* model is displayed, which is the context the constraint is defined on. This model contains an *Attribute* with the name “age”. This attribute is referenced in the OCL expression after the `self` expression. The algorithm splits up the expression string after every dot (“.”) because it indicates an object navigation. The result of that split operation is an array of substrings of that OCL expression. Then the algorithm iterates over every substring and checks for every substring if it is contained in the defined connections, attributes, methods or is a reference to another *Clabject* that is contained in the deep model. If in fact the algorithm finds a substring that can be matched to any of these aforementioned types, the algorithm creates an instance of the *Pointer* class. The only attribute of this class is called `pointer` and is of type *PLM::Element* which allows this class to save the reference to the found element in that attribute. If the algorithm cannot identify the current substring as a reference pointer it creates a *Text* class and saves the substring as a normal string. The advantage of this saving algorithm is that the user, to stick with this example, can rename the `age` attribute without having to rename the reference in the OCL expression as well. After renaming the attribute the reference is correctly resolved by the `pointer` attribute, which has an operation `getName()` available to retrieve the current

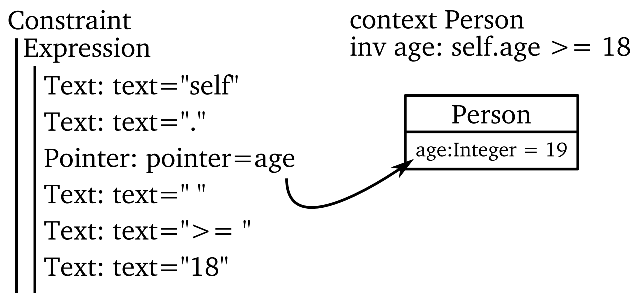


Figure 4.3.: The saving algorithm result with reference pointers to identifiable elements

name of the aforementioned types. This saves the user time when refactoring. It also ensures a higher probability of the OCL expression being correct, because if it is identified as a *Pointer* it ensures that this element is in navigable. Once identified as a pointer reference it reacts to every name change.

5. Executing DeepOCL

This chapter explains what the data flow looks like when an OCL expression is evaluated. Besides it explains when a constraint evaluation is triggered for the particular constraint types.

5.1. Data Flow

In order for any OCL expression to be evaluated, the expression has to be parsed and then interpreted. Figure 5.1 shows a high-level overview of the data flow in the part of the application that is responsible for parsing and interpreting these expressions. The process begins by retrieving the constraint and resolving

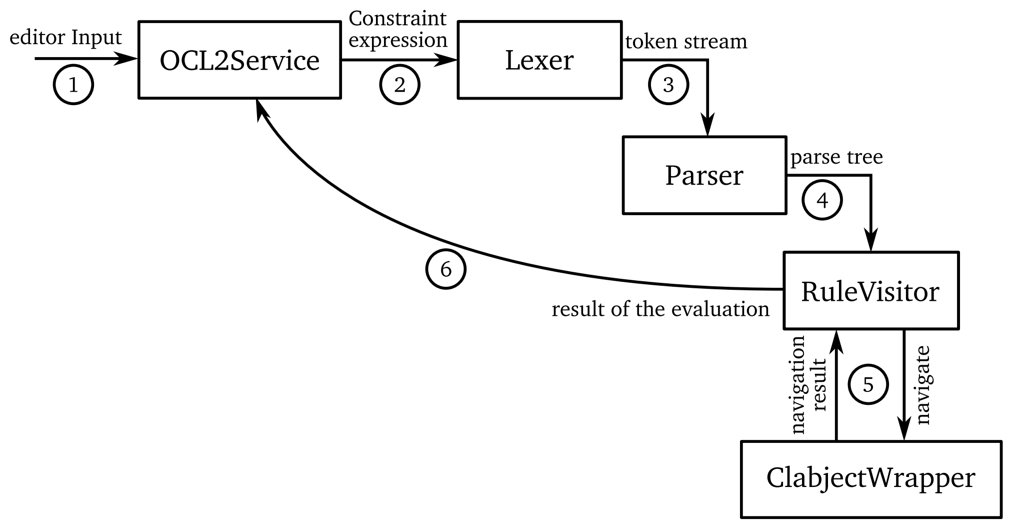


Figure 5.1.: An illustration of the data flow in the DeepOCL evaluation application

the expression from the constraint. The arrow labeled “1” indicates this first step, and the constraint string serves as an input for the *OCL2Service*, which implements the *IConstraintLanguageService* interface. When interpreting invariant constraints, the method takes the whole deep model as a parameter and iterates over the whole content of each level to find constraints from type *invariant*. The step, indicated by the arrow labeled “2”, prepares the statement for parsing. The application retrieves the text from the constraint and passes it on to a new instance of the *DeepOCL2Lexer*. The lexer creates a token stream which is the desired input format for a new instance of the *DeepOCL2Parser*. This parser creates a hierarchical data structure that can be transformed into the typical parse tree for this dialect. The fourth step is to instantiate the *DeepOCL2RuleVisitor*. As a parameter this class takes the element that is to be evaluated with the constraint for an input. With the method *visit* the interpretation part of the process starts and as an argument this method takes the parse tree as an argument. It then visits every rule from top to bottom and with help of the *DeepOCL2ClabjectWrapper* the application performs navigation, query and comparing operations, which is indicated by the number *5*. The last step in this process is that the *RuleVisitor* returns the result of the constraint expression that was applied to the current element. When an invariant constraint was evaluated to false or something went wrong during the evaluation process the *OCL2Service* creates resource marker to call attention to the user that the model is not valid in terms of the defined constraints.

5.2. Implicit/Explicit Causes of Triggering Constraint Evaluation

There are seven different constraint types. These types are invariant, derive, init, body, definition, pre and post constraints. Each type has a different scope in regard to the evaluation context. The following shows how and when each type has to be evaluated.

Invariant Constraint An invariant can only be defined for *Clabjects* and not for attributes or methods. It has to be a valid Boolean expression which has to be evaluated to true. In the DeepOCL dialect invariant constraint must also hold for all instances that exist in the level range specification. This type of constraint is not evaluated every time the model changes. The user has to trigger the evaluation, by pressing a button, of every invariant constraint that is defined in the deep model. If an invariant constraint evaluates to false the respective *Clabject* is marked and the user will notice the violated constraint. It is also possible to define constraints for the *Deep Model* and the containing *Levels*.

It is possible to define an arbitrary amount of invariant constraints on one *Clabject*.

Init Constraint The init constraint can only be defined on *Attributes* of a *Clabject*. When a new instance of a *Clabject* on which an init constraint is defined, is created in the deep model the DeepOCL application will be noticed by the underlying framwork to evaluate the defined constraint and initialize the attribute with the result of the computation. The execution is implicitly triggered by the user who is creating a new instance of an entity on which a *init* constraint is defined.

It is only possible to define one init constraint on one *Attribute*.

Pre/Post Constraint Both pre- and post-constraints are only applicable to methods that are defined on a *Clabject*. The pre-type constraint checks the state of the model before the method is executed. It has to be evaluated to true before the method can be executed. If this type of constraint is evaluated to false, the execution of said method is canceled.

The post-constraint type checks the state of the model after the method execution and has to be evaluated to true for a successful method execution. If it evaluates to false, the changes the execution has caused have to be rolled back. This feature has not been implemented yet but a implementation proposal will be discussed in chapter 7.

Both constraint types are also implicitly executed by an OCL expression which calls a defined method on a *Clabject* for which either a pre- or post-constraint exists.

For these constraint types it is possible to define an arbitrary amount on one *Method* entity.

Derive Constraint The derive type constraint is only applicable to *Attributes*. One *Attribute* can contain only one derive constraint. The evaluation of a derive constraints is triggered if the model changes in any way. Even totally unrelated changes with regard to the computation result of this constraint type will trigger an evaluation. When the application receives the notice that the model changes in some way the whole deep-model is searches for this constraint type. This obvious performance issues will be discussed in chapter 7.

The user triggers the evaluation implicitly by changing something in the deep-model.

Definition Constraint The definition type constraint defines a whole operation on a *Clabject*. Compared to the body constraint type, it also defines the signature of the method, i.e., how many and what kind of parameters are passed on and what the return type of the operation is. This constraint type is evaluated when the *Method* is called by any other OCL expression evaluation.

There must be only one definition of a *Method*, but the definition constraint type is defined on the *Clabject* itself. The user can define multiple operations on a *Clabject*. Hence an arbitrary amount of definition constraints can be defined.

Body Constraint The last constraint type defines the body of a *Method*. Like the previous constraint type this type is also evaluated when the operation is called by another part of an OCL expression.

For one *Method* only one such constraint ought to be used on it.

All but one DeepOCL constraints are implicitly evaluated,the only exception is the invariant constraint. When a new *Clabject* is instantiated the derive and init constraints are triggered to compute their respective results. Pre-, post-, body- and definition constraints are trigger when other OCL expression accessing a *Method* on which those constraint types are defined. This functionality had to be integrated into the *Melanee* application. The aforementioned only exception, the invariant constraint type, can only be triggered by the user. When we consider that the only plug-in that can trigger any operation execution in a Deep-Model is the DeepOCL dialect then the invariant constraint triggers every other constraint execution, with the exception of the init and derive constraint. The user is only able to explicitly trigger the execution of invariant constraints which can contain a reference to a body or definition constraint that is then triggered and evaluated.

5.3. Nested Collection Operations

In OCL it is possible to write nested collection operation expressions, like the expression shown in listing 5.1. This expression iterates over a collection and then performs a navigation, which results in another collection, which is then iterated over. In other words, this statement is an iteration over another iteration.

Every collection iteration operation creates a new instance of the *DeepOCLClabjectWrapper* is created for every element of the collection that is iterated over. This class needs an argument for the constructor which defines the current context, which is the current iteration element.

In the first *select* expression, the algorithm creates a new instance for every *customer* entity in the model. In every new instance, the navigation operation to every connected *transaction* is performed. Afterwards the second *select* expression creates new instances for every *Transaction* entity in order to perform the comparing operation.

Listing 5.1: Nested collection operation

```
context Company
inv VIPCustomer: self.customer -> select(c|c.transaction -> select(
    value > 100)) -> size()>2
```

5.4. Constraint Search Algorithm

The algorithm that finds the constraints that are actually applicable to a model element is based on the aspect aware visualizer search algorithm by Gerbig[24]. Listing 5.2 shows the core functionality of the algorithm.

Listing 5.2: Constraint Search Algorithm

```
private List<AbstractConstraint> search(Clabject c, Attribute a,
    List<Class> constraintTypes) {
        List<AbstractConstraint> constraints = new ArrayList<>();
        LinkedList<Clabject> superTypesToSearch = new LinkedList<
            Clabject>();
        LinkedList<Clabject> typesToSearch = new LinkedList<Clabject
            >();
        typesToSearch.add(c);
        Clabject currentClabject = null;
        Clabject currentType = null;
        // Go through the type hierarchy
        while ((currentType = typesToSearch.poll()) != null) {
                // We need to have the type of the current type at
                    the beginning
                typesToSearch.addAll(currentType.getDirectTypes());
                superTypesToSearch = new LinkedList<Clabject>(
                    currentType.getDirectSupertypes());
                superTypesToSearch.add(0, currentType);
                // Go through the inheritance hierarchy
                while ((currentClabject = superTypesToSearch.poll())
                     != null) {
                        if (a == null) {
                                constraints.addAll(
                                    getConstraintFromElement(
                                    currentClabject, c,
                                    constraintTypes));
                        } else {
                                constraints.addAll(
                                    getConstraintFromElement(
                                    currentClabject, a,
                                    constraintTypes));
                        }
                        superTypesToSearch.addAll(currentClabject.
                            getDirectSupertypes());
                        typesToSearch.addAll(currentClabject.
                            getDirectTypes());
                }
        }
        return constraints;
}
```

The *search* method has three parameters defined. The first parameter is the *Clabject* for which the constraints are searched for. The second parameter is an *Attribute*, which can be `null` if the constraint is not contained in an *Attribute*. The third parameter is a list with types of constraints that have to be found for the current attribute or entity. If the application evaluates invariant constraints, the list consists of one element that is `InvarianConstraintImpl.class`. The algorithm then retrieves the classification hierarchy and iterates over this hierarchy to find the next concrete applicable constraint. The algorithm adds all valid and applicable constraints to the list and returns the list at the end. If no constraints are found the empty list is returned. This algorithm provides a clear structure to find relevant constraints in the classification hierarchy and and adds to the maintainability of the source code in the rest of the DeepOCL application.

6. Implementation

This chapter presents the way to implement an OCL dialect for a Deep Model infrastructure. The first part deals with the grammar of this dialect and how it differs from the OCL specification. The second part handles the implementation of the *DeepOCLRuleVisitor* class, which is the core of the application and responsible for the correct interpretation of any OCL statement. The next part will introduce the new Meta Model for *Constraints*, which is now a part of the PLM Meta Model for *Melanee*. Then the navigation semantics and examples of interpreted OCL statements will be explained in detail. As will be how and what proposals from Atkinson, Kühne and Gerbig[4] are implemented.

6.1. ANTLR

This section aims to introduce the implementation of the grammar and the rule visitor. In chapter 2.1 the theoretical foundation was laid for the principles used in the implementation of those parts. "ANTLR is an exceptionally powerful and flexible tool for parsing formal languages."[38] This tooling is used to define the DeepOCL dialect grammar and to generate the pair of lexer and parser.

6.1.1. Grammar

In order to create a new dialect that can untie the dependency between a Deep-OCL implementation and EclipseOCL implementation, it is necessary to define a grammar that can generate a DeepOCL language (cf. chapter 2.1). Most parts of the grammar are adopted from the OCL specification.[19] Some parts, like the linguistic or ontological navigation specification, differ. This chapter presents the main differences to the EclipseOCL specification, which is the de facto implementation of the OCL standard.

In order to give a better idea on how the grammar processes OCL statements, a few example will be displayed in the following. The subsequent step, the interpretation of OCL statements, will be presented in section 6.1.2

Listing 6.1: Simple OCL expression

```
context Customer
inv ofAge: age >= 18
```

Listing 6.1 can represented in a labeled parse tree as shown in figure 6.1. The

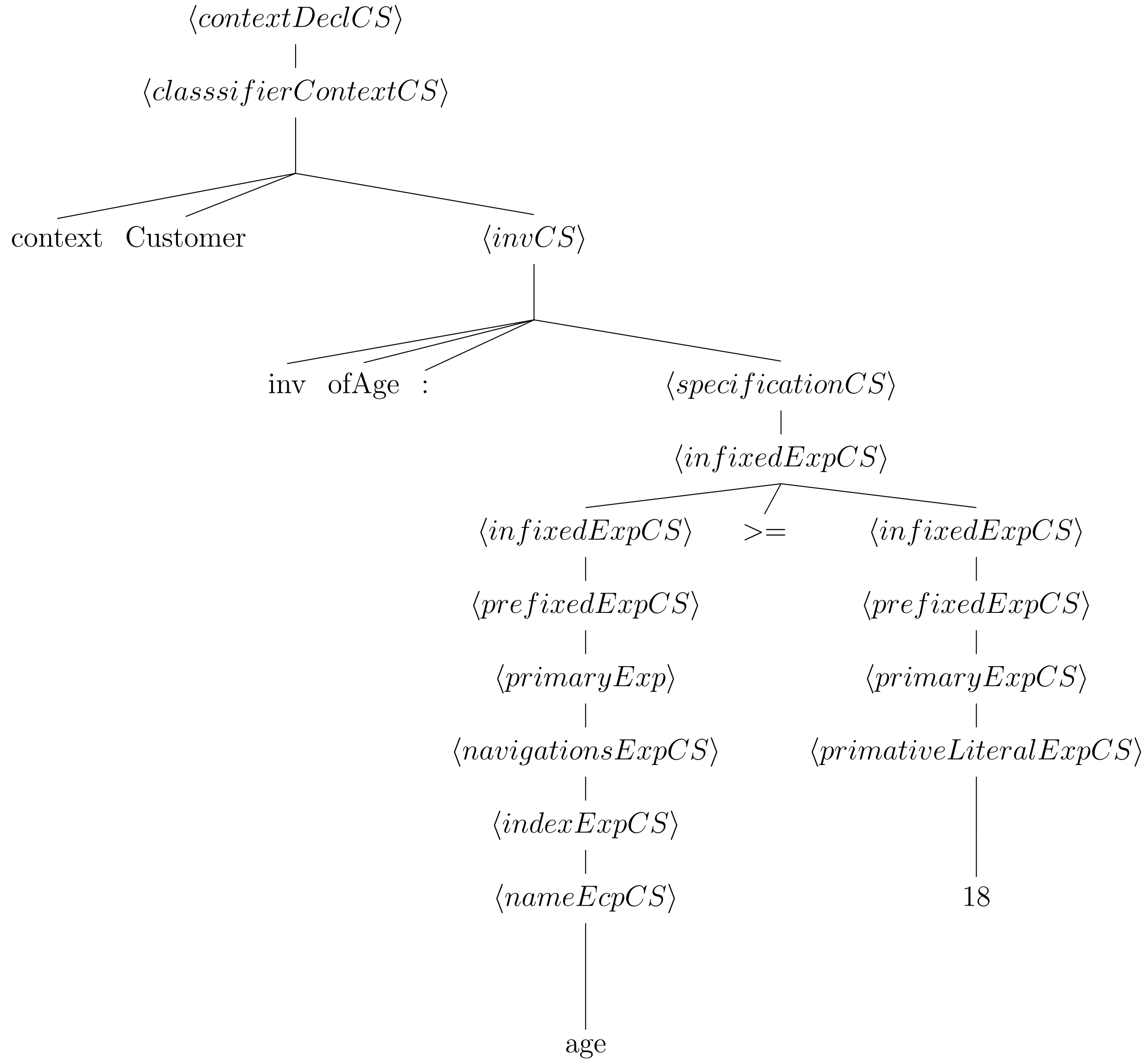


Figure 6.1.: first example of a parsed OCL statement

process of parsing the OCL expression with the help of the generated code that was derived from the grammar specification prepared the expression for interpretation. Note that the comparing operator >= has a left-hand and a right-hand side. First the navigation on the left side has to be resolved. The result of that navigation is the value of the *age* attribute. It is then compared to the value *18*. The whole expression holds if the all instances of *Customer* are over the age of 18.

The next example showcases how the grammar handles logical expression, because some logical operators have precedence over other operators.

Listing 6.2: Logical expression showcase

```
context Person
inv logic: a and b implies c and d
```

Listing 6.2 shows how logical operators can be combined in an OCL expression. In this particular statement the *and* operator has precedence over the *implies* operator. The figure 6.2 shows how the grammar reorders this OCL expression hierarchically after the defined rules. If one had to evaluate this statement only reading it from left to right the result would be false. The *implies* operator would compare the result of the first *and* expression and the value of *c*. Then this result and the value of *c* would be the input for *implies* operator witch would then pass the result to the second *and* operator.

This precedence order for these operators conforms to the OCL 2.4 specifications.[27]

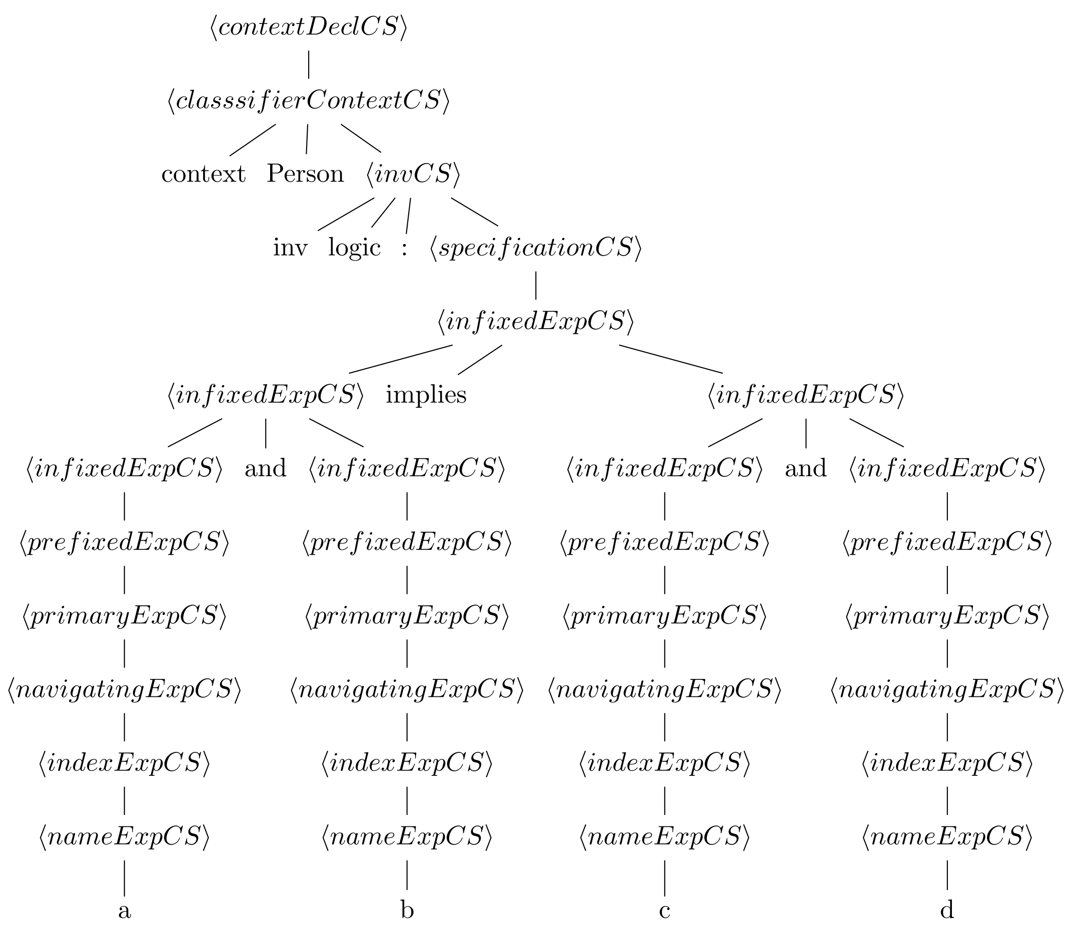


Figure 6.2.: Example of a parse tree with a combined logical expression

Listing 6.3: The infixedExpCS rule as defined in the ANTLR grammar

```
infixedExpCS
 :
        prefixedExpCS
        | iteratorBarExpCS
        | left = infixedExpCS op =
        (
                '/'
                | '*'
        ) right = infixedExpCS
        | left = infixedExpCS op =
        (
                '+'
                | '-'
        ) right = infixedExpCS
        | left = infixedExpCS op =
        (
                '<='
                | '>='
                | '<>'
                | '<'
                | '>'
                | '='
        ) right = infixedExpCS
        | left = infixedExpCS op = '^' right = infixedExpCS
        | left = infixedExpCS op =
        (
                'and'
                | 'or'
                | 'xor'
        ) right = infixedExpCS
        | left = infixedExpCS op = 'implies' right = infixedExpCS
  ;
```

Listing 6.3 shows how the logical operation rules have to be arranged to give precedence to the *and* operator over the *implies*. The operation that has precedence over other rules has to be defined at the top of the grammar rule. That means that rules that precedence over other rules are located relatively below in the parse tree to the rules they have precedence over. The ANTLR tooling reacts implicitly to the order of the rule definition. The structure of each possible matching of this rule is the same. There is always a left- and right-hand side of the production and the operator. The left- and right-hand side of the production have to take the form that they can also match the *infixedExCS* rule, which can be another logical expression, a navigation in the model or a primitive literal.

These few examples show that the grammar is a very important step in the parsing process. First it identifies the necessary tokens in the lexical analysis and then the grammar orders the expressions according to the defined productions or rules. This reordering prepares DeepOCL expressions for interpretation.

6.1.2. Rule Visitor

As stated in section 6.1.1, the OCL expressions are first parsed with the help of the generated lexer and parser. The expressions are then prepared for interpretation by splitting up the statement and matching these parts against rules that are defined in the grammar. Then these parts can be represented in a labeled tree, as it was shown in section 6.1.1. In the case of the second example, which is displayed in figure 6.2, the tree visitor, which functions as the core element of the interpretation process, visits every rule that can be found in the labeled tree.

The rule visitor visits every rule that can be identified in the OCL expression. Every instance of the rule visitor crates an instance of a *DeepOCLClabjectWrapper*. Every operation is then delegated to this class.

6.2. Interpreting OCL Expressions

This section will give a better understanding on the semantics of the DeepOCL dialect. The primary focus is both the navigation semantics of said dialect and the execution mechanisms of the different constraint types.

6.2.1. Linguistic Context Operations

The syntax of linguistic navigations was discussed in section 3.1, and two semantic definitions were clarified with the help of the *Manager* example. This section will give a complete overview of the capabilities and semantics of linguistic navigations.

For the meta model entities *Clabject*, *DeepModel*, *Level* and *Connection* there are methods defined that can be invoked by using the linguistic context navigation. But not only methods can be used to navigate in the linguistic context; references that contain other meta-model instances can be navigated, too. These references point to containments of other instances of *EObject* that are defined in the PLM meta-model of Melanee.

For the *DeepModel* meta model the following methods and references are defined:

- `getContent()` – returns all containing elements; in this case all levels that are defined within the *DeepModel* instance
- `enumeration` – returns all the defined enumerations
- `getLevelAtIndex(int level)` – returns the level that is identified by the parameter

- `getPrimitiveDatatypes()` – returns all primitive data types
- `getAllDatatypes()` – returns all primitive data types and enumerations

The reference navigation can be identified by the missing parentheses at the end. For the *Level* meta model the following operations and reference navigations are defined:

- `getContent()` – returns all containing elements of the *Level* instance.
- `getAllInheritances()` – returns all the generalizations that are present at the level
- `getClabjects()` –returns all elements that are of the type *Clabject* of the *Level*
- `getEntities` – returns all entities which are a subset of all *Clabjects* of the *Level*
- `getConnections()` – returns all *Connections* that are present at this *Level*
- `getClassifications()` – returns all classifications if the instance is present at this *Level*
- `getDeepModel` – returns the *DeepModel* that contains this *Level*
- `isRootLevel()` – returns true if the *Level* is the topmost level in the *DeepModel*, else false
- `isLeafLevel()` – returns true if the *Level* is the bottom level in the *DeepModel*, else false

Here the `content` navigation as well is referring to references of the meta-model.

The next list will display a selection of *Connection* operations and reference navigations, which are supposed to be useful for writing statements in the DeepOCL dialect efficiently.

- `getDomain()` – returns all destinations of the navigable connection ends of this *Connection*
- `getNotDomain()` –returns all *Clabjects* that participate in this *Connection* but are not navigable
- `getHumanReadableName()` – returns a human readable name of this *Connection*

- `getParticipants()` – returns all participants, i.e. destinations of the connection ends, of this *Connection*
- `getMoniker()` – returns the moniker for this *Connection*
- `getMonikerForParticipant(Clabject)` – returns the moniker of this *Connection* for the parameter *Clabject* if it is reachable through this *Connection*
- `getOrder()` – returns the number of connection ends in the *Connection*
- `getParticipantForMoniker(String)` – returns the *Clabject* reachable through the *Connection* via the parameter *moniker*
- `getAllConnectionEnd` – returns the connection ends that the connection inherits from its supertypes

The operation `getAllConnectionEnd()` could also be replaced by the `connectionEnd` reference navigation.

Even though the *Clabject* meta-model contains many methods and references that can be invoked by any OCL statement, the following will only display a few operations and references which have a higher chance of being used when writing DeepOCL expressions.

- `getPotency()` – returns the potency of the *Clabject*
- `getContent()` – returns all the elements that are contained by the *Clabject*
- `getAllFeatures()` – returns all attached *Feature* entities, which could be from type *Attribute* or *Method*
- `getTypes()` – returns a collection of all *Clabjects* that are of the type of the source *Clabject*
- `getInstances` – returns all the *Clabjects* that are an instance of the source *Clabject* based on classification elements.
- `getAllAttributes()` – returns all *Attributes* of the source *Clabject*
- `getAllMethods()` – returns all the methods that are contained by the source *Clabject*
- `getDefinedNavigations()` – returns all defined navigation of the source *Clabject*
- `getDirectTypes()` – returns the direct types of the source *Clabject*

- `getDefinedInstances()` – returns the instances and their subtypes of the *Clabject* only
- `getSubtypes()` – returns all entities that inherit from the source *Clabject*
- `getSupertypes()` – returns the *Clabjects* this *Clabject* inherits properties from
- `getConnections()` – returns all connections from the source *Clabject*
- `getLevelIndex()` – returns the level index the source *Clabject* is located on
- `detDeepModel()` – returns the *DeepModel* the *Clabject* is contained in
- `isTypeOf(Clabject)` – returns true if the *Clabject* is in the classification tree of the *Clabject* that was passed in the parameter

These linguistic navigations are a vital part of navigating the deep model and keeping the model valid with respect to the constraints. Combined with the ontological navigation it shows the capabilities of the DeepOCL dialect.

6.2.2. Ontological Context Operations

Due to the nature of deep modeling, *Methods* can be executed without being translated into a executable source program, with the help of the DeepOCL dialect. If another OCL expression is invoking an operation, which is defined by either a *body* or *definition* constraint, the execution of the current OCL expression is paused and the constraint which defines the method is evaluated.

If in addition to a body or definition constraint pre- and/or post-constraints are defined, the evaluation of the method is extended by these constraints. If the pre-constraint is evaluated to `false`, the method is not invoked and the entity that contains the OCL statement the invocation originated from is marked with an error marker by the application. If the post-constraint is evaluated to `false`, the method has been already executed. At this point in time the application is not able to restore the model to the state before the execution if the operation is able to change something in the model. But the entity the OCL statement originated from is marked faulty to indicate that something went wrong in the evaluation of the original statement. A conceivable implementation of a mechanism to rollback the ramifications of the operation execution will be discussed in chapter 7.4.

6.3. Extending the Functionality of the DeepOCL Dialect

Currently it is not possible to extend the DeepOCL dialect dynamically, i.e., writing DeepOCL expressions that extend the functionality scope of the application. Chapter 7.6 discusses that functionality in detail. To add a function to the application first the class `DeepOCLRuleVisitor` has to gain a new section in the `visitNavigatingExpCS` function. If the new operation is executable on a source collection by the `"->"` identifier, the new section has to be placed inside the if block that checks if the Boolean value of `collectionOperation` is true. If not, the new function has to be placed just outside of that block. Then the new function name has to be added to the list of functions the `DeepOCLClabjectWrapper` class is maintaining in the `loadOperations` method. Then the else-if-block of the `invoke` method has to be extended with that name, too. From there the newly implemented method is ultimately called.

Listing 6.4 shows an example of the `last` method definition. First the algorithm checks for the name of the operation. The second if block checks whether the `tempCollection`, which is an attribute that is used in nested collection expressions, is not null nor empty. If this if block evaluates to true the last element is selected and returned. In the case the `tempCollection` is null or empty, the collection that is currently on top of the navigation stack, which is controlled by the `DeepOCLClabjectWrapper`, is retrieved and the last element of that collection is returned for a result. Before the return statement returns the actual result, the algorithm places the result on top of the `navigationStack`, because this element could be used in a another navigation.

Listing 6.4: The example function `last` to show the structure of a method definition in the DeepOCLRuleVisitor

```
// last
else if (ctx.opName.getText().equals("last")) {
        if (this.tempCollection != null && this.tempCollection.size
           () > 0) {
                Element e = (Element) tempCollection.toArray()[
                   tempCollection.size() - 1];
                tempCollection = null;
                this.wrapper.getNavigationStack().add(new Tuple<
                   String, Collection<Element>>("last", Arrays.
                   asList(e)));
                return e;
        } else {
                int size = this.wrapper.getNavigationStack().peek().
                    getSecond().size();
                Element e = (Element) this.wrapper.
                    getNavigationStack().peek().getSecond().toArray()
                    [size - 1];
                this.wrapper.getNavigationStack().add(new Tuple<
                    String, Collection<Element>>("last", Arrays.
                    asList(e)));
                return e;
        }
}
```

In some cases it is not necessary to add the returned result to the navigation stack, like operations that return a Boolean or an integer type value. The result of those operations cannot be navigated on and are compared immediately afterwards in the `DeepOCLRuleVisitor` class.

7. Future Work

This chapter discusses the possible directions the new DeepOCL can take in the future and what features need to be added to the constraint language to reach the highest conformity to the OCL specifications as possible. This is important, because the user who is already familiar with OCL expects a dialect of that same language to behave in a very similar way. To use the DeepOCL dialect the users should not learn a new language in terms of the functionality scope and syntax, if they are already familiar with OCL.

First the missing action language for the deep modeling context is discussed, and which already defined action languages can serve as a blueprint for an action language in deep modeling. Then a series of topics will discuss how the DeepOCL dialect can advance to be more user friendly and facilitate the full potential of a constraint language in the deep modeling context. Above all, sections 7.4 and 7.6 deserve a mentioning beforehand, because they have the potential to elevate the level of functionality dramatically.

7.1. Action Language

A deep modeling action language could be based on the syntax of the Epsilon Object Language (EOL) or the Action Language for fUML (Alf).[20, 26]

Although there is no exact specification of a deep modeling action language yet, this thesis showed a way to create a language extensions inside a deep modeling environment. Any other language that needs to the added to the scope of functionality of a deep modeling environment, like Melanee, can be implemented in the same way. First the grammar has to be developed, then the ANTLR tooling creates the parser and lexer pair and finally the semantic definitions of said language are implemented.

An action language has to support CRUD operations in the deep modeling context, which are to create, to read, to update and to delete entities, attributes, methods, deep models, levels or clabjects. The language also has to be able to call methods that are defined on clabjects and evaluate the results. It also has to have to support loop expressions, like *for* or *while* loops, and variable assignments. OCL is only able to create variables in a *let* expression context, which is rather inconvenient compared to the other programming languages. A better example to to declare variables and assign values to them can be found in languages like Python[18] or Javascript[31].

A lot of the semantic definitions of the action language can be reused from the DeepOCL implementation. All the collection operation implementations, that were worked on in the context of this thesis, have the same semantics. Also the navigation specification, i.e., how to navigate in a deep model with multiple levels, that originated from Kantner[33] are also basically the same.

EOL and ALF are partly based on OCL and combine the features of an imperative language, like Javascript, and OCL collection query expressions. Hence, these two language are candidates to serve as the syntactical basis for any action language that will be implemented in a deep modeling environment. With regards to the semantics of the OCL part of these languages the new action language can borrow the semantic definitions of this work as much as possible.

7.2. Level Specific Constraints

For now it is only possible to specify constraints for a level range, i.e., for the first level below the context *Clabject*, from where the constraint originated, and to the last level. It is not possible to skip a level in that range. Instead of defining two parameters, which are the bounds of the level range the constraint will be evaluated for, the user should be able to choose whether he passes a list of levels or defines a level range. This functionality has to be added to the editor which then delegates the evaluation of defined constraints. The default mechanism will always be to define a level range. This is a more advanced feature for a more experienced user of deep modeling and OCL.

7.3. Constraints on Elements of the Linguistic Dimension

Right now the default and only dimension to define constraints in, is the ontological dimension. To define constraints on the elements of the PLM, additional functionality in the source code is needed. It is on the other hand possible to define a constraint on a specific instance of a *Level* or a *DeepModel*. The user can define constraints that would be similar to those the user would define in the linguistic dimension. If a constraint would be defined on the *Clabject* meta-model, the constraint would apply to all *Clabjects* in this model and not only a specific instance of an element. Due to the fact, that the most common structure of a deep model has only one instance of the *DeepModel* meta-class the constraint would look essentially the same. Nevertheless it is a necessary feature that has to be implemented in the next steps to utilize the full potential of a deep constraint language.

7.4. Rollback mechanism

OCL is per definition a language that is side effect free, with regard to the model on which constraint expressions are applied on. Nonetheless when constraints are combined with statements from an action language or when the whole model is translated into executable code the constraints are applied on a system that changes. It is even possible to execute a deep model.[5]

If a model is extended with both the DeepOCL and the possibility to execute this model, then there is a need to control constraints in a more restrictive way. Let us assume a *Clabject* is extended with a definition constraint, which defines a *Method*. Further assume that this *Method* needs a pre- and post-constraint to function properly. To enforce the pre-constraint is rather straight forward. If the pre-constraint does not hold, the *Method* is not executed at all. Dealing with a failing post-constraint is more difficult for obvious reasons. If the post-constraint fails, the *Method* has been already executed and that is why in the future the DeepOCL dialect needs a rollback mechanism to reverse the effect of said *Method* and restore the old state of the system. Hence *Melanee* needs to enforce the ACID[29] principles on the model in order to keep the model consistent with regard to the defined constraints.

A practicable way to deal with that problem would be to register every change this *Method* execution makes. If now a post-condition fails after the execution the system can be restored to a previous state. This logging data would only be stored temporary and is only needed until the post-condition is checked.

The Eclipse Modeling Framework, which is used to create Melanee, has already the capabilities to serve as a transaction system with which the rollback mechanism can be realized.[11] It is called the EMF.Edit[11] framework and resides inside the EMF project. This framework is already used throughout the Melanee workbench. In this way the model can be stored in its old state and if the post-condition does not hold, the old model can be restored without any further calculation. If any further research is necessary in this matter I propose to examine the applicability of integrating said framework into the method execution of deep-models.

7.5. Unambiguous OCL

There is at least one instance where the OCL grammar is ambiguous. The listing 7.1 shows that if a *Let* variable is created the "=" operator is used to assign a value to the variable. The listing 7.2 is selecting every customer that has a transaction volume that equals "100" at least once. These two listings show that the same operator is used for two different operations. The first operation used the operator to assign a value to a variable and the second operation used the operator to compare two values with each other. This an ambiguity that can easily be fixed due to the fact that the grammar of the DeepOCL dialect is part of this thesis and a contribution to Melanee.

Listing 7.1: Assignment OCL statement

```
context A
        inv correctDate: let correctDate:Boolean = false in
        if self.notValid then correctDate = false else correctDate =
            true endif
```

Listing 7.2: Comparing OCL statement

```
context Company
        inv VIPCustomer: customer -> select(c|c.transaction ->
            select(volume = 100))->size()=1
```

The only time the “=” operator is assigning a value to an attribute is the let expression. If an action language is derived from the DeepOCL grammar this ambiguity is much harder to control, because the context for assigning values is not just the let expression anymore. The obvious proposal is to change the operator for comparing to “==”. So that there is special terminal, like in JAVA, for assigning a value to a variable and one for comparing values.

7.6. Dynamically Extending DeepOCL Functionality

For now there is only one way to extend the functionality scope of the DeepOCL dialect. One has to implement the new functionality in the `DeepOCLRuleVisitor` class and the `ClabjectWrapper` class which are located in the service plug-in of the application. For the purpose of rapid prototyping it would be much more convenient to just add a new functionality in the DeepOCL dialect. The dimension context for adding functionality to the dialect can be both the linguistic and ontological dimension.

Assume that a new functionality for all *Clabjects* is needed and the programmer can express this new function in the DeepOCL dialect. The description of this process is only referring to operation definitions like *body* or *def* constraints. These are the only constraints for which a need of such rapid prototyping could be identified. All other constraints do not have the properties of defining functional instructions and therefore are not included in the process description. The process for adding those constraints or methods could look like the following.

Every constraint that adds new constraints or methods for any element in the deep model has to be bootstrapped and stored either inside the model or outside in an extra file. If the constraint is of type *def* the method name has to be added to the list of all valid functions. If the constraint is of type *body* the name of the method does not have to be added to the list, this name should already be there. Assume further the newly defined functionality is called, let us say the constraint is of type *def*, then the expression has to be parsed by grammar and on the exact spot where the method is called the method the syntax tree has to be substituted with the parsed definition of the method. Then the parse tree, with the substituted bit that adds the new functionality, is parsed again and interpreted.

ANTLR does not support substituting nodes or subtrees with other nodes or subtrees. This topic needs further investigation into the inner works of ANTLR. Specifically into the syntax tree generation or extending the functionality of the `ParseTree` class of ANTLR, so that the support for those operations are included.

7.7. Editor

In order to improve the user experience the editor has to support code completion proposals and syntax highlighting. Due to the time constraints of this thesis all features that could elevate the user experience and help the user to produce a correct and valid OCL expression were prioritized very low. The syntax highlighting mechanism is already implemented in the DeepOCL implementation of Kanter[33]. The code completion proposals can be derived from the parser class. A parser instance that was generated from the ANTRL tooling contains a method by the name of `getExpectedTokens()`.[38] When parsing an unfinished and partly faulty OCL expression it is possible to react to parsing errors and retrieve the expected tokens from said method.

7.8. The Transitive Closure Operation

Since the version 2.3.1 of OCL, which was published in January 2012, a new iteration operation on collections was defined, the “closure” operation. With this operation it is possible to navigate over transitive relationships of models. The closure operation “supports returning results from the elements of a collection, the elements of the elements of a collection, the elements of the elements of the elements of a collection, and so forth.”[27] Consider the model displayed in figure

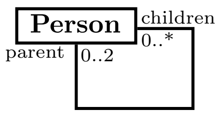


Figure 7.1.: The `Person` closure operation example

7.1, a person has zero or at maximum two parents and zero to an arbitrary amount of children. With an OCL expression the following listings shows how to query for a collection of all children and the second body constraint queries for all parents. The syntax definition is the same like the `reject` and `select` operations.

Listing 7.3: Closure operation on the person model

```
context Person::allDescendants():Set(Person)
body: self.parents -> closure(children)

context Person::allAnchestors ():Set(Person)
body: self -> OrderedSet() -> closure(parents)
```

As the `closure` operation differs from the regular implementation schema of the other collection iteration operation, the `closure` is not yet supported in the DeepOCL dialect. The problem is to find an efficient implementation for the semantic definition of this operation, because the algorithm has to determine weather or not the transitive relationship is a circular one or not. If the relationship is in fact a circular one, then this circle is returned as a `Set()`.

So in this regard the DeepOCl dialect does not conform to the OCL 2.4 specification, which is the newest available version of OCL.

7.9. Performance Optimization

The implementation of the DeepOCL dialect in this thesis is a research prototype. There are several conceivable bottlenecks where performance, when working with a model that contains a lot of entities with several constraint definitions, could be an issue with regard to the user experience.

The evaluation of all constraints happens in an implicit fashion, except invariant constraints. Consider the evaluation of init constraints, every time a new *Clabject* is created, the algorithm has to search for a possible init constraint in the type classification hierarchy over all levels the hierarchy spreads over. Because at the time the new *Clabject* is created the application has no knowledge of whether there is a defined init constraint in the classification hierarchy or not. The operation is triggered by an event that passes an *Attribute* from where the search for a valid constraint starts. With the implementation of the action language in Melanee, the user should be able to create many new entities in a bulk, which could need a lot of time to complete. If the new entities have attributes defined, it does not matter whether an init constraint is defined in the hierarchy or not, the algorithm starts the search every time a new *Attribute* is contained in the newly created *Clabject*.

Another conceivable performance bottleneck is the evaluation of derive constraints. The performance reducing impact is possibly higher than the evaluation of the init constraint. The evaluation of derive constraints is triggered every time anything changes within the model. Because derive constraints can have also an arbitrary complexity and the expression can navigate the whole model, they have to be evaluated as soon as a change in the model is detected.

The fact that the OCL expression are saved in a more or less parsed fashion could help to avoid the performance bottlenecks. Every reference to a *Attribute*, *Method* or *Clabject* is stored as such and not as plain text. The expression should register to these reference somehow and if something changes, the subscribers to the change event are notified. If the receiver of such notification is a derive constraint it can be reevaluated without searching for every derive constraint in the model and reevaluate all of them.

8. Related Work

This chapter gives an introduction into other realizations of OCL in the deep modeling context. As the master thesis of Kantner is the basic point of reference throughout this thesis, his contribution to this topic, deep constraint languages, will be briefly summed up. Other deep modeling tools that provide the possibility to define constraints on deep models will also be introduced in this chapter.

8.1. DeepOCL by Kantner

The most similar implementation to this work, is the master thesis of Kantner[33]. The basis for his implementation was the EclipseOCL[19] implementation of OCL. That is why his implementation is conforming, syntactically speaking, to OCL standard, with the exception of two changes in the syntax definition. The only syntactic changes from Kantner was the dimension switch operation, which was indicated by _l_ and the level cast operation in the upper hierarchy levels of the source object.

With regard to the semantic definitions of his dialect, he worked out a useful way on how to navigate in a deep model.

His work lacked the flexibility to change syntactic structures in the DeepOCL dialect due the restrictions the EclipseOCL enforced on the well-formedness of OCL expression.

8.2. metaDepth

MetaDepth, which was introduced by de Lara and Guerra[16], is a deep-modeling tool that supports textual modeling over an arbitrary amount of ontological levels and dual instantiation of *Clabjects*.[16] The tool utilizes EOL, which extends in part OCL, to define constraints. EOL is part constraint language and part action language. Chapter 7.1 indicated that EOL could be the basis for the deep action language dialect for Melanee.

In order to make EOL aware of the deep-modeling environment the authors adjusted the standard OCL in two aspects.[17]

The first change to the specification of OCL was to be able to assign a potency with the constraint definition. The potency shows the level where the constraint is evaluated on. If the potency is `1` then the constraint is evaluated one level below the level where the constraint is defined on. The second adjustment is that the constraints can use methods and attributes of the linguistic dimension. Properties of the linguistic dimensions are accessed like properties from the ontological dimension. If there is a name collision of attributes or methods from the linguistic and ontological dimension and the user wants to access the property from the linguistic dimension the prefix "ˆ" can be used to indicate the dimension switch.[17]

This thesis described a similar dimension switch into the linguistic dimension, but the user has to indicate that switch explicitly in contrast to the meta-depth method, which is implicitly accessing the properties of the linguistic dimension.

The idea to define potencies on the constraints to indicate the level the constraint is evaluated on, is to some extent consistent with the idea of potency in general in the deep-modeling context. Normally it would indicate a level span on which the constraint is valid and has to be evaluated on, because the potency defined on a *Clabject* indicate on how many level below the *Clabject* can the instantiated the las time. This thesis lets the user define a level span, that marks the range of the evaluation of the constraint.

8.3. Nivel

Nivel is another deep modeling framework which was created by Asikainen and Männistö[3]. Nivel uses the Weight Constraint Rule Language (WCRL), which is a general-purpose knowledge representation language, to create models and constraints.[3] The framework enables the user to define *cardinality constraints*, which are constraints that affect instances of associations holding values for the cardinality and potency.

Aiskainen and Männistö state that “Nivel defines no constraint language of its own” [3] and the construct of cardinality constraints is the only possibility to define constraints for the model or elements of the model. According to the authors, adopting WCRL for Nivel would cause a number of problems and is not desired by them. They claim that any user that is familiar with Nivel would assumed to be familiar with the WCRL and be able to write constraint in it.[3]

In this thesis Melanee was used to create models in a graphical fashion. Compared to UML, Melanee has also a similar look and feel with regard to the modeling. Hence, the user of Melanee is most probably familiar with OCL and other modeling tools that the Eclipse modeling project provides.

9. Conclusion

Out of the limitation of UML, the LML was conceived and in order to attract users to the new modeling paradigm, deep-modeling, a certain tool support is necessary. One vital tool to create exact and valid models is OCL. Kantner did the initial work of defining the navigation semantics in a deep modeling environment and implemented a console to query models in the deep model.

The implementation of the new deep OCL dialect of this thesis elevated the usability of said dialect in Melanee. I was able to evolve the dialect from a console to an integral part of modeling in Melanee. The user is able to save constraints inside the LML, which is possible due to the extension of the PLM, and evaluate different kinds of constraints. This new dialect is also united from the EclipseOCL implementation and can evolve further in any desired direction without having to adapt to changes in the EclipseOCL application interfaces. New developments can introduce new syntactical or semantic constructs to customize the deep constraint dialect to meet occurring requirements.

Storing constraints is done in a homogeneous format for models and their constraints, which avoids inconsistencies in the combination of constraints and models.[12] When using OCL in most tools, the constraints can be stored in various formats (txt, annotations in Ecore) and the "MOF and OCL parts are not always stored in the same format, both parts of the metamodel tend to be inconsistent."[12]

As a next step the software has to be extensively tested in order to ensure a certain level of functionality. Since this project is operated by an university, students should be able to work on further scientific research in this field. Tests help to give anybody an idea of the workings of the DeepOCL dialect. There are already a number of tests that cover some cases, but the software has to deal with real modeling cases and constraint definitions to make sure, that the semantic definitions of the DeepOCL dialect behave the way they are specified. All the statements displayed in this thesis are covered by test and deliver the correct result. In total there are 74 tests that range from rather simple tests, like parsing test to show that certain OCL expression can be parsed by the defined grammar, to very complex tests that navigate deep model over multiple levels and nested collection operation tests.

This work provides a very useful DeepOCL dialect that will hopefully further advance the popularity of deep-modeling. The dialect is very close to the OCL standard specification and users that are switching from UML to LML, i.e., from the 4-layer modeling architecture to OCA, will have no difficulties to adapt to the new dialect that is presented in this thesis.

Bibliography

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley series in computer science. Addison-Wesley, Reading, Mass. [u.a.], 38 edition, 2002.

[2] Alfred V Aho and Jeffrey D Ullman. *The theory of parsing, translation, and compiling. 1, Parsing*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[3] Timo Asikainen and Tomi Männistö. Nivel: a metamodelling language with a formal semantics. *Software & Systems Modeling*, 8(4):521–549, 2009.

[4] Colin Atkinson, Ralph Gerbig, and Thomas Kühne. Opportunities and challenges for deep constraint languages. In Achim D. Brucker, Marina Egea, Martin Gogolla, and Frèdèric Tuong, editors, *Proceedings of the 15th International Workshop on OCL and Textual Modeling*, volume 1512 of *OCL 2015*, pages 3–18. CEUR Workshop Proceedings, 2015.

[5] Colin Atkinson, Ralph Gerbig, and Noah Metzger. On the execution of deep models. In Tanja Mayerhofer, Philip Langer, Ed Seidewitz, and Jeff Gray, editors, *Proceedings of the 1st International Workshop on Executable Modeling*, volume 1560 of *EXE 2015*, pages 28–33. CEUR Workshop Proceedings, 2015.

[6] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Transactions on Software Engineering*, 35(6):742–755, 2009.

[7] Colin Atkinson, Bastian Kennel, and Björn Goß. *The Level-Agnostic Modeling Language*, pages 266–275. Springer Berlin Heidelberg, 2011.

[8] Colin Atkinson and Thomas Kühne. The essence of multilevel metamodeling. In *International Conference on the Unified Modeling Language*, pages 19–33. Springer, 2001.

[9] Colin Atkinson and Thomas Kühne. Rearchitecting the uml infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, October 2002.

[10] Colin Atkinson and Thomas Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, Sept 2003.

[11] Frank Budinsky. *Eclipse modeling framework : a developer's guide.* The eclipse series. Addison-Wesley, Boston, Mass. ; Munich [u.a.], 2004.

[12] Juan Cadavid, Benoit Baudry, and Benoit Combemale. Empirical evaluation of the conjunct use of MOF and OCL. In Michel Chaudron, Marcela Genero, Parastoo Mohagheghi, and Lars Pareto, editors, *Experiences and Empirical Studies in Software Modelling (EESSMod 2011)*, Wellington, New Zealand, October 2011. CEUR.

[13] Noam Chomsky. On the notion 'rule of grammar'. In *Proceedings of the Twelfth Symposium in Applied Mathematics*, volume 12, pages 6–24. American Mathematical Society, 1961.

[14] Noam Chomsky. *Syntactic structures.* Ianua linguarum / Series minor 4,8 4 (DE-576)015598004. Mouton, The Hague [u.a.], 8 edition, 1969.

[15] Oracle Corporation. Java, 2016.

[16] Juan de Lara and Esther Guerra. Deep meta-modelling with metadepth. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 1–20. Springer, 2010.

[17] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. Model-driven engineering with domain-specific meta-modelling languages. *Software & Systems Modeling*, 14(1):429–459, 2015.

[18] Python Software Foundation. Python, 2016.

[19] The Eclipse Foundation. Eclipse ocl project, 2016.

[20] The Epsilon Foundation. Epsilon object language, 2016.

[21] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Future of Software Engineering, 2007. FOSE '07*, pages 37–54, May 2007.

[22] Richard A. Frost and Rahmatullah Hafiz. A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time. *SIGPLAN Not.*, 41(5):46–54, May 2006.

[23] Ralph Gerbig. The level-agnostic modeling language: Language specification and tool implementation. 2011.

[24] Ralph Gerbig. *Deep, Seamless, Multi-format, Multi-notation Definition and Use of Domain-specific Languages.* PhD thesis, University of Mannheim, to be published 2017.

[25] Seymour Ginsburg. *The mathematical theory of context free languages.* McGraw-Hill, New York [u.a.], 1966.

[26] Object Management Group. ALF Action Language for Foundational UML, 2005.

[27] Object Management Group. Object management group ocl specification 2.4, 2016.

[28] Object Management Group. Object management group uml specification 2.5, 2016.

[29] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[30] John E Hopcroft and Jeffrey D Ullman. *Introduction to automata theory, languages, and computation.* Addison-Wesley series in computer science. Addison-Wesley, Reading, Mass. [u.a.], 1979.

[31] Ecma International. Ecmascript, 2016.

[32] Stanislaw Jarzabek and Tomasz Krawczyk. Ll-regular grammars. *Information Processing Letters*, 4(2):31–37, 1975.

[33] Dominik Kantner. Specification and implementation of a deep ocl dialect. Master's thesis, Department of Business Informatics and Mathematics Chair of Software Engineering - Prof. Dr. Colin Atkinson, University of Mannheim, 2014.

[34] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607 – 639, 1965.

[35] P. M. Lewis, II and R. E. Stearns. Syntax-directed transduction. *J. ACM*, 15(3):465–488, July 1968.

[36] Anton Nijholt. *Context-free grammars : covers, normal forms, and parsing.* Lecture notes in computer science 93 (DE-576)014492687. Springer, Berlin ; Heidelberg [u.a.], 1980.

[37] University of Mannheim Software Engineering Group. Melanee, 2016.

[38] Terence Parr. *The definitive ANTLR 4 reference.* The pragmatic programmers. Pragmatic Bookshelf, Dallas, Tex. [u.a.], 2012.

[39] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 579–598, New York, NY, USA, 2014. ACM.

[40] D. J. Rosenkrantz and R. E. Stearns. Properties of deterministic top down grammars. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, pages 165–180, New York, NY, USA, 1969. ACM.

[41] James Rumbaugh, Grady Booch, and Ivar Jacobson. *The unified modeling language reference manual.* Addison-Wesley, Boston, 2nd edition, 2004.

[42] Arto Salomaa. *Jewels of formal language theory.* Computer Science Pr., Rockville, Md., 1981.

[43] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing theory. 1, Languages and parsing.* European Association for Theoretical Computer Science EATCS monographs on theoretical computer science 15 (DE-576)009692398. Springer, Berlin ; Heidelberg, 1988.

[44] Jos Warmer and Anneke G Kleppe. *Object constraint language 2.0 : [die neuen Sprachkonstrukte der OCL 2.0; so werden Ihre Modelle MDA-tauglich ...].* Software-Entwicklung. mitp-Verl., Bonn, 1 edition, 2004.

Appendix

A. The ANTLR4 DeepOCL grammar

```
/*************************************************************************
 * Copyright (c) 2012, 2013 University of Mannheim: Chair for
   Software Engineering
 * All rights reserved. This program and the accompanying materials
 * are made available under the terms of the Eclipse Public License
   v1.0
 * which accompanies this distribution, and is available at
 * http://www.eclipse.org/legal/epl-v10.html
 *
 * Contributors:
 *    Ralph Gerbig - initial API and implementation and initial
   documentation
 *    Arne Lange - ocl2 implementation
 *************************************************************************/

grammar DeepOcl;

contextDeclCS
:
        (
                propertyContextDeclCS
                | classifierContextCS
                | operationContextCS
        )+
;

operationContextCS
:
        CONTEXT levelSpecificationCS?
        (
                ID ':'
        )?
        (
                ID '::'
                (
                        ID '::'
                )* ID
                | ID
        ) '('
        (
                parameterCS
                (
                        ',' parameterCS
                )*
        )? ')'
        (
                ':' typeExpCS
        )?
        (
                preCS
                | postCS
                | bodyCS
        )*
;

levelSpecificationCS
:
        '(' NumberLiteralExpCS
        (
                ','
                (
                        '_'
                        | NumberLiteralExpCS
                )
        )? ')'
;

CONTEXT
:
        'context'
;

bodyCS
:
        'body' ID? ':' specificationCS
;
```

```
postCS
:
        'post' ID? ':' specificationCS
;

preCS
:
        'pre' ID? ':' specificationCS
;

defCS
:
        'def' ID? ':' ID
        (
                (
                        '(' parameterCS?
                        (
                                ',' parameterCS
                        )* ')'
                )? ':' typeExpCS? '=' specificationCS
        )
;

typeExpCS
:
        typeNameExpCS
        | typeLiteralCS
;

typeLiteralCS
:
        primitiveTypeCS
        | collectionTypeCS
        | tupleTypeCS
;

tupleTypeCS
:
        'Tuple'
        (
                '(' tuplePartCS
                (
                        ',' tuplePartCS
                )* ')'
                | '<' tuplePartCS
                (
                        ',' tuplePartCS
                )* '>'
        )?
;

tuplePartCS
:
        ID ':' typeExpCS
;

collectionTypeCS
:
        collectionTypeIDentifier
        (
                '(' typeExpCS ')'
                | '<' typeExpCS '>'
        )?
;

collectionTypeIDentifier
:
        'Collection'
        | 'Bag'
        | 'OrderedSet'
        | 'Sequence'
        | 'Set'
;

primitiveTypeCS
:
        'Boolean'
        | 'Integer'
        | 'Real'
        | 'ID'
        | 'UnlimitedNatural'
        | 'OclAny'
        | 'OclInvalID'
        | 'OclVoID'
;

typeNameExpCS
:
        ID '::'
        (
                ID '::'
        )* ID
        | ID
;

specificationCS
:
        infixedExpCS*
;

expCS
:
        infixedExpCS
;

infixedExpCS
:
        prefixedExpCS # prefixedExp
        | iteratorBarExpCS # iteratorBar
        | left = infixedExpCS op = '^' right = infixedExpCS #
           Message
        | left = infixedExpCS op = 'implies' right = infixedExpCS #
           implies
        | left = infixedExpCS op =
        (
                'xor'
                | 'or'
                | 'and'
        ) right = infixedExpCS # andOrXor
        | left = infixedExpCS op =
        (
                '='
                | '<>'
                | '<='
                | '>='
                | '<'
                | '>'
        ) right = infixedExpCS # equalOperations
        | left = infixedExpCS op =
        (
                '+'
                | '-'
        ) right = infixedExpCS # plusMinus
        | left = infixedExpCS op =
        (
                '*'
                | '/'
        ) right = infixedExpCS # timesDivide
;

iteratorBarExpCS
:
        '|'
;

navigationOperatorCS
:
        '.' # dot
        | '->' # arrow
;

prefixedExpCS
:
        UnaryOperatorCS+ primaryExpCS
        | primaryExpCS
        (
                navigationOperatorCS primaryExpCS
        )*
        | primaryExpCS
;

UnaryOperatorCS
:
        '-'
        | 'not'
;
```

```
primaryExpCS
:
        letExpCS
        | ifExpCS
        | navigatingExpCS
        | selfExpCS
        | primitiveLiteralExpCS
        | tupleLiteralExpCS
        | collectionLiteralExpCS
        | typeLiteralExpCS
        | nestedExpCS
;

nestedExpCS
:
        '(' expCS+ ')'
;

ifExpCS
:
        'if' ifexp = expCS+ 'then' thenexp = expCS+ 'else' elseexp =
            expCS+ 'endif'
;

letExpCS
:
        'let' letVariableCS
        (
                ',' letVariableCS
        )* 'in' in = expCS+
;

letVariableCS
:
        name = ID ':' type = typeExpCS '=' exp = expCS+
;

typeLiteralExpCS
:
        typeLiteralCS
;
```

```
collectionLiteralExpCS
:
        collectionTypeCS '{'
        (
                collectionLiteralPartCS
                (
                        ',' collectionLiteralPartCS
                )*
        )? '}'
;

collectionLiteralPartCS
:
        expCS
        (
                '..' expCS
        )?
;

tupleLiteralExpCS
:
        'Tuple' '{' tupleLiteralPartCS
        (
                ',' tupleLiteralPartCS
        )* '}'
;

tupleLiteralPartCS
:
        ID
        (
                ':' typeExpCS
        )? '=' expCS
;

selfExpCS
:
        'self'
;

primitiveLiteralExpCS
:
        NumberLiteralExpCS # number
        | STRING # string
        | BooleanLiteralExpCS # boolean
        | InvalIDLiteralExpCS # invalid
        | NullLiteralExpCS # null
;

InvalIDLiteralExpCS
:
        'invalid '
;

NumberLiteralExpCS
:
        INT
        (
                '.' INT
        )?
        (
                (
                        'e'
                        | 'E'
                )
                (
                        '+'
                        | '-'
                )? INT
        )?
;

fragment
DIGIT
:
        [0-9]
;

INT
:
        DIGIT+
;

BooleanLiteralExpCS
```

```
:
        'true'
        | 'false'
;

NullLiteralExpCS
:
        'null'
;

navigatingExpCS
:
        opName = indexExpCS
        (
                '@' 'pre'
        )?
        (
                '(' '"'? onespace? arg = navigatingArgCS* commaArg =
                    navigatingCommaArgCS*
                barArg = navigatingBarAgrsCS* semiArg =
                   navigatingSemiAgrsCS* '"'? ')'
        )*
;

navigatingSemiAgrsCS
:
        ';' navigatingArgExpCS
        (
                ':' typeExpCS
        )?
        (
                '=' expCS+
        )?
;

navigatingCommaArgCS
:
        ',' navigatingArgExpCS
        (
                ':' typeExpCS
        )?
        (
                '=' expCS+
        )?
;

navigatingArgExpCS
:
        iteratorVariable = infixedExpCS iteratorBarExpCS nameExpCS
        navigationOperatorCS body = infixedExpCS*
        | infixedExpCS+
;

navigatingBarAgrsCS
:
        '|' navigatingArgExpCS
        (
                ':' typeExpCS
        )?
        (
                '=' expCS+
        )?
;

navigatingArgCS
:
        navigatingArgExpCS
        (
                ':' typeExpCS
        )?
        (
                '=' expCS+
        )?
;

indexExpCS
:
        nameExpCS
        (
                '[' expCS
                (
                        ',' expCS
                )* ']'
        )?
```

```
;

nameExpCS
:
        (
                (
                        ID '::'
                        (
                                ID '::'
                        )* ID
                )
                | ID
                | STRING
        ) # name
        | '$' clab = ID '$' # ontologicalName
        | '#' aspect = ID
        (
                '('
                (
                        NumberLiteralExpCS
                        | ID
                )?
                (
                        ','
                        (
                                NumberLiteralExpCS
                                | ID
                        )
                )* ')'
        )? '#' # linguisticalName
;

parameterCS
:
        (
                ID ':'
        )? typeExpCS
;

invCS
:
        'inv'
        (
                ID
                (
                        '(' specificationCS ')'
                )?
        )? ':' specificationCS
;

classifierContextCS
:
        CONTEXT levelSpecificationCS?
        (
                ID ':'
        )?
        (
                (
                        ID '::'
                        (
                                ID '::'
                        )* ID
                )
                | ID
        )
        (
                invCS
                | defCS
        )*
;

propertyContextDeclCS
:
        CONTEXT levelSpecificationCS?
        (
                (
                        ID '::'
                        (
                                ID '::'
                        )* ID
                )
                | ID
        ) ':' typeExpCS
        (
                (
                        initCS derCS?
                )?
                | derCS initCS?
        )
;

derCS
:
        'derive' ':' specificationCS
;

initCS
:
        'init' ':' specificationCS
;

ID
:
        [a-zA-Z] [a-zA-Z0-9]*
;

WS
:
        [ \t\n\r]+ -> skip
;

onespace
:
        ONESPACE
;

ONESPACE
:
        ' '

;

STRING
:
        '"'
        (
                ~[\r\n"]
```

```
                        |  '""'
            )*  '"'
;

COMMENT
:
            '--' .*? '\n' -> skip
;
```