University of Twente

Master Thesis

OCL constraints in GROOVE

Patrick Thijssen

supervised by
prof. dr. ir. A. Rensink
dr. L. Ferreira Pires

June 19, 2020

Abstract

Graphical modeling has been part of the Computer Science world for quite some time now. The standard graphical modeling language nowadays is the Unified Modeling Language (UML). Before 1997 the UML class diagram had one big shortcoming, since there was no way to create constraints over the values of attributes. That is why UML adopted the Object Constraint Language (OCL): OCL enables developers to create textual constraints over the values of attributes.

At the University of Twente, we use GROOVE as a graph-based object oriented verification tool. GROOVE makes use of a type graph that can be compared to a UML class diagram and so it has the same shortcoming as the original UML class diagrams had before 1997. In this work, we have used OCL to represent constraints for GROOVE models. So we have defined transformation rules from OCL to nested graph constraints, which are then represented in GROOVE logic such that we are able to define constraints in GROOVE based on OCL syntax. We have validated our implementation by rebuilding two case studies in both, our implementation and Eclipse OCL. Both systems accept and reject the same models for a given OCL constraint, and so we showed that our implementation is in line with the implementation of Eclipse OCL.

Contents

Chapter 1

Introduction

Graphical modeling has been part of the Computer Science world for quite some time now. Peter Chen was one of the first who investigated ways to create a graphical model of a software system. In 1976, Chen released his paper about the Entity-Relationship language in [1]. Several other graphical modeling languages have been derived from the Entity-Relationship language, one of these languages is the Unified Modeling Language (UML) [2]. Nowadays UML is the standard object-oriented modeling language.

A graphical modeling language, like UML, enables a software engineer to create a visual model of a software system. This makes it easier for engineers to have an overview of the system. One kind of model especially, the class diagram, gives the engineer an overview of which classes exist and the relations between these classes. The Unified Modeling Language contains many different kind of diagrams, but in this report we will focus on class diagrams.

Before 1997, it was not possible to add constraints to class diagrams. To overcome this issue, the Object Constraint Language (OCL) [3] was adopted and included into the UML standard. OCL is a formal textual language that has been created for defining constraints. Due to the limited expressiveness of graphical models, it is not possible to visualise a complex constraint inside a class diagram. So besides the visual UML class diagram, developers also define a set of textual OCL constraints, which valid instances of the class diagram have to satisfy.

A class diagram on its own can be seen as a graph, in which each node represents a class and the edges represent associations between classes. An example of a class diagram can be found in Figure 1.1. In this example, the Employee class is a node, and the association between Employee and Project is an edge. If you take an instance of this class diagram, in which all the attributes are initialized and therefore have values, then it is possible to check if the corresponding OCL constraints are satisfied and therefore if the instance of the class diagram is valid. An example of such an OCL constraint is: `context Employee inv positiveSalary: self.salary >= 0`, which encodes that the salary of all employees has to be positive.

The previous paragraph states that it is possible to encode the state of a system in graphs. Then it is possible to encode transitions between states of a system with graph transformations. This is exactly the line of thought that has been used to develop GROOVE [5]. GROOVE stands for GRaph-based Object-Oriented VErification and is a tool that uses graphs and graph transformations as a basis for model verification. GROOVE applies a defined graph transformation to a start graph from where the complete state space of a system is generated and can be verified.

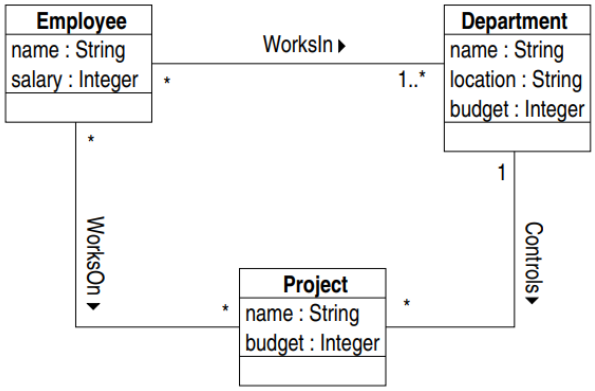


Figure 1.1: An example of a UML class diagram [4].

1.1 Problem Statement

Currently GROOVE only uses type graphs to enforce constraints throughout the state space. In a type graph, developers can define types and their attributes (and the types of these attributes). Such a type graph can be compared with a UML class diagram, and has therefore the same shortcomings concerning constraints. To overcome these shortcomings, UML has been extended with OCL; we expect that we can combine GROOVE and OCL.

In [6], Radke et al. transformed OCL constraints to nested graph constraints, similarly we aim to adopt OCL in GROOVE to represent constraints in GROOVE models.

1.2 Illustration

This section gives an illustration how we have implemented the transformation of OCL to nested graph constraints in GROOVE.

First of all we start with a type graph in GROOVE, in which we define the types, its attributes and the associations. For this example we use the type graph defined in Figure 1.2. This type graph defines four graph types: Employee, Project, Department and the common supertype Object. Each of the classes has its own attributes and allowed edges, the so-called associations. For example, class Employee has two attributes (salary and name), and two allowed outgoing edges (department and project).

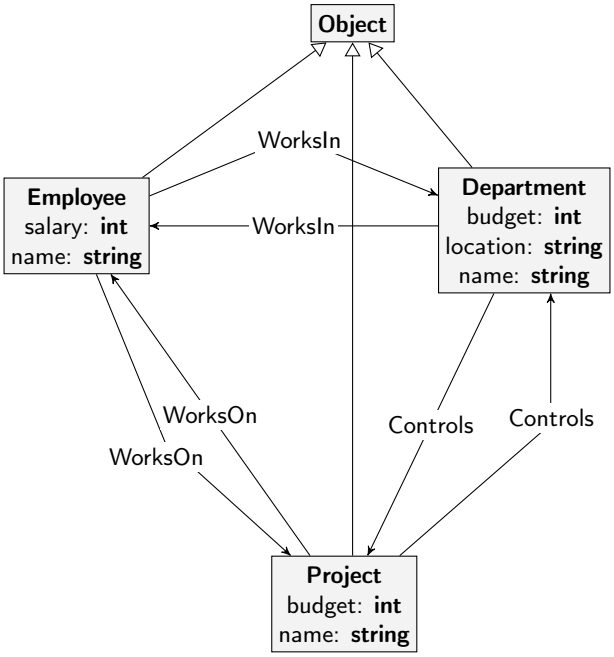


Figure 1.2: GROOVE type graph corresponding to the UML class diagram in Figure 2.3.

Now we can define an constraint. For example, we want to make sure that the salary of all employees is positive, because a negative salary makes no sense. This can be defined in the following OCL constraint: `context Employee inv positiveSalary: self.salary >= 0`. This OCL constraint is defined with the input window that is shown in Figure 1.3.

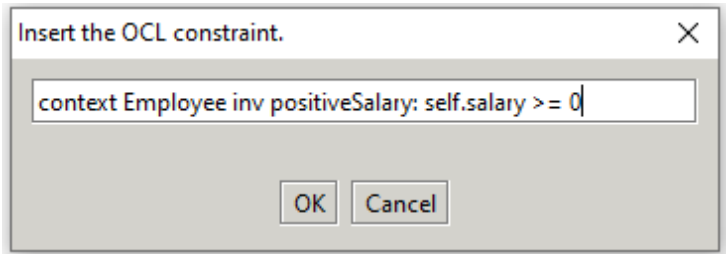


Figure 1.3: The way to create a new OCL constraint in GROOVE.

After creating the OCL constraint, this constraint is transformed to a nested graph constraint such that GROOVE is able to check it. The resulting graph is given in Figure 1.4. The layout of the graph has to be created by hand and is not generated.

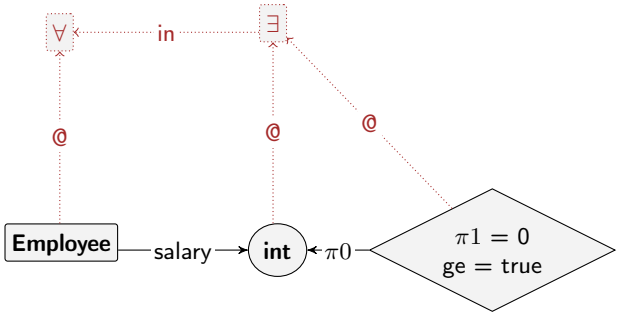


Figure 1.4: The positiveSalary invariant in GROOVE.

1.3 Research Goals

The main goal of our research is the following: evaluate the use of OCL in GROOVE to represent constraints in GROOVE models, such that constraints are enforced throughout the state space of a software system.

This should make GROOVE a better verifier because we will be able to ensure that some behaviour never occurs in the complete state space. Because OCL is way too expressive we will focus on the OCL fragment that has the same expressive power as first-order logic, this fragment is called OCL_{FO}. To achieve this we have split up our goal into the following steps:

1. **Define transformation rules from OCL_{FO} to nested graph constraints**
   We have defined transformation rules from OCL to the language used in GROOVE, graph based semantics. This includes an overview of transformation rules for each OCL operation.
2. **Represent graph constraints in GROOVE**
   We have defined how to represent nested graph constraints in GROOVE, where a nested graph constraint can consist of multiple graphs, consists a graph in GROOVE always of a single graph. So we have defined how to represent the multiple graphs of a nested graph constraint in a single graph in GROOVE.
3. **Implement the transformation rules in GROOVE**
   We have implemented the defined transformation rules in GROOVE, such that we could use OCL constraints in GROOVE.

1.4 Validation

One of the important aspects research is validation. How are we going to demonstrate that what we implement accepts the intended valid models? To validate our implementation, the models including the OCL constraints of two case studies have been rebuilt in both Eclipse OCL and in our implementation of GROOVE. In this way, we checked if our implementation accepts and rejects the same models as the existing Eclipse OCL, and validated if our implementation enforces the constraints correctly. If the implementation works as desired, there should be no difference between the accepted and rejected models of Eclipse OCL and our implementation. The following two case studies have been used for validation:

1. USE case study [4], Richters et al. have defined seven basic OCL constraints. Because it are basic constraints this could be a good starting point for the research.
2. DBLP case study [7], Planas et al. have created a well elaborated case study with 26 unique OCL constraints ranging from basic constraints to more complex constraints.

Besides these two case studies we have compared the results of some additional OCL constraints. In this way we have validated the supported OCL operations that were not part of one of the case studies.

In this thesis we have not looked into the real performance of our implementation. The only thing we can say about it is that the constraints are transformed almost immediately. After the transformation we have the same performance as the common graph constraints in GROOVE.

1.5 Report Outline

This document is further structured as follows. Chapter 2 gives the necessary background information to understand the transformation rules from OCL to nested graph constraints. Chapter 3 discusses related work regarding transformations of OCL. Chapter 4 defines the OCL fragment and transformation rules from OCL to nested graph constraints with corresponding equivalence rules. Chapter 5 explains how a nested graph constraint is represented in GROOVE. Chapter 6 discusses the actual implementation. Chapter 7 validates our GROOVE implementation on the basis of two selected case studies. Chapter 8 concludes our research.

Chapter 2

Background

In Section 2.1 propositional logic is explained and in Section 2.2 first order logic, an extension of propositional logic, is explained. In Section 2.3 the Unified Modeling Language is introduced and in Section 2.4 extended with the Object Constraint Language. In Section 2.5 the concept of decidability is explained. In Section 2.6 an introduction to graph theory is given. In Section 2.7 the graph transformation tool GROOVE is introduced and lastly in Section 2.8 the concept of parsing is explained.

2.1 Propositional Logic

Propositional logic is the simplest logic and provides a way to start reasoning about logic. A propositional formula consists of propositions that capture a global fact such as, *Snow is white*. Such a proposition can be either true or false, but never both. The assignment of true or false is called the truth value of the proposition. The relations between truth values of propositions can be found in Table 2.1 and a small example of how to use propositional logic can be found in Example 1. The symbol $\equiv$ is used to show that two things are equivalent.

$\neg S$	is true iff	S	is false		
$S_1 \wedge S_2$	is true iff	S_1	is true and	S_2	is true
$S_1 \vee S_2$	is true iff	S_1	is true or	S_2	is true
$S_1 \oplus S_2$	is true iff	$S_1 \vee S_2$	is true and	$S_1 \wedge S_2$	is false
$S_1 \implies S_2$	is true iff	S_1	is false or	S_2	is true
$S_1 \iff S_2$	is true iff	$S_1 \implies S_2$	is true and	$S_2 \implies S_1$	is true

Table 2.1: The relations between truth values of propositions.

Example 1. A propositional logic example where the relations of Table 2.1 are used.

$p =$ it rains
$q =$ it snows

$Q =$ It rains or it snows $= p \vee q$
$R =$ It does not rain $= \neg p$

By combining the propositions Q and R:
$R \wedge Q \equiv \neg p \wedge (p \vee q) \equiv (\neg p \wedge p) \vee (\neg p \wedge q) \equiv false \vee (\neg p \wedge q) \equiv \neg p \wedge q$

Therefore it snows

2.2 First Order Logic

Propositional logic is only able to reason about global facts and not about individuals. First Order Logic (FOL) extends propositional logic and is able to overcome this shortcoming by defining a syntax with constants, variables, predicates and quantifiers.

- **Constants** are values that will not change and so will not contain another value than defined. For example, the constant True will always evaluate to True.
- **Variables** are values that can have different values. For example, a boolean variable can evaluate to True or False.
- **Predicates** are functions, given one or more input arguments a predicate will evaluate to True or False. For example, *even(x)* takes as input a number x and will return True iff x is an even number.
- **Quantifiers**, there are two quantifiers:
  - Universal, a universal quantifier should be read as "for all" and is denoted as $\forall$.
  - Existential, an existential quantifier should be read as "exists" and is denoted as $\exists$.

Given a set A of constants, V of variables and P of predicates the FOL syntax is defined as:

$$
\begin{aligned}
&t ::= a(\in A) \mid x(\in V) \\
&\phi ::= p(t_1, ..., t_{n_p}) \text{ (atomic formula)} \\
&\phi ::= \phi \ op \ \phi \mid \neg\phi \\
&\phi ::= \forall_x \phi \mid \exists_x \phi \\
&op ::= \wedge \mid \vee \mid \oplus \mid \implies \mid \iff
\end{aligned}
$$

Example 2. A few FOL examples can be found below, given that the domain is natural numbers, **N**, and the available predicates on this domain are *even*, *odd* and *geq* (greater than or equal).

All numbers are even or odd:	$\forall_x(even(x) \vee odd(x))$
If a number is not even then it is odd:	$\forall_x(\neg even(x) \implies odd(x))$
There exists an odd number:	$\exists_x(odd(x))$
For every number, there exists a number that is greater than or equal to this number:	$\forall_x(\exists_y(geq(y, x))$

2.3 Unified Modeling Language

The Unified Modeling Language (UML) [2] is the standard object-oriented modeling language and provides a way to create a visualization of the design of a system. The different kinds of visualizations, called diagrams, can be found in Figure 2.1. In this thesis, we will focus on one kind of diagram, the class diagram, which is a diagram that shows the structure of a system in terms of its classes and their relations.

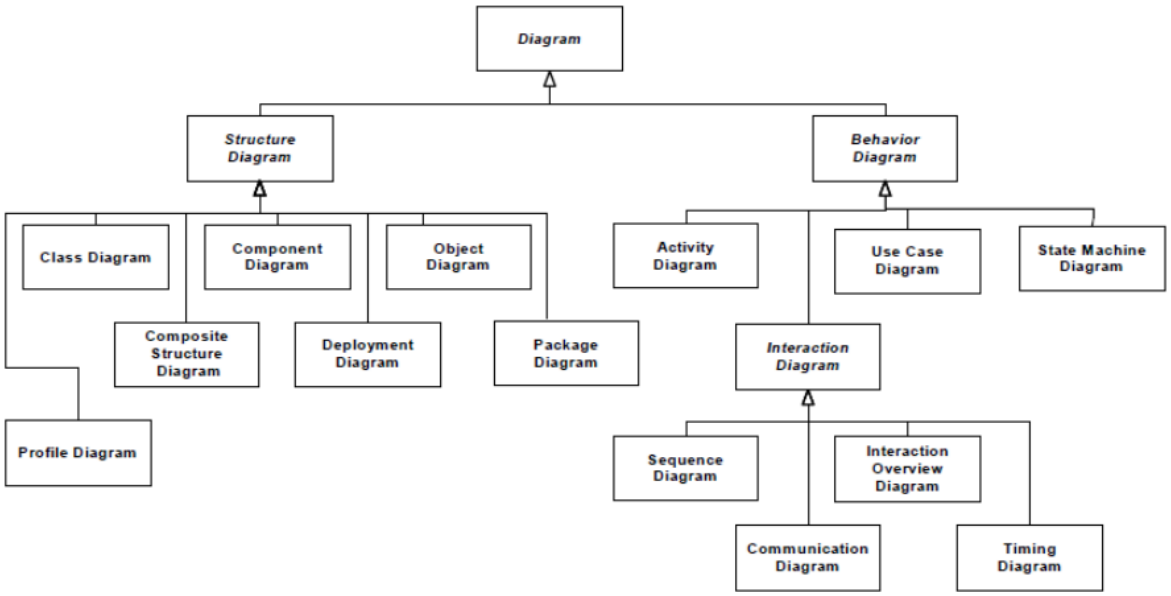


Figure 2.1: An overview of all UML 2.5 diagrams [2].

A class diagram consists of classes and associations between classes. Every class is depicted as a rectangular box with inside the information of its attributes and methods. An example of a class that could be found in a class diagram is given in Figure 2.2. This class is called Employee and contains two attributes: name (which is of type String) and salary (which is of type Integer). The class contains two methods: getSalary, with no parameters and returns an Integer and setSalary, which has to be called with one parameter salary, which is an integer, and returns nothing (void).

Employee
name : String salary : Integer
getSalary() : Integer setSalary(salary : Integer) : void

Figure 2.2: A class in a class diagram

An example of a class diagram consisting of both classes and the relations between classes can be found in Figure 2.3. The relations are depicted as lines between classes. All the relations in this example are association relations, which include the name of the association and the multiplicity. For example, the worksIn association between Employee and Department is an association that goes to both sides. The worksOn association from Employee to Project has a

multiplicity *, which means that an employee works on 0 or more projects. If the multiplicity is higher than 1 then we are talking about collections. UML defines four types of collections; bags, sets, ordered sets and sequences, each with their own characteristics.

Another important relation is the inheritance relation, which is denoted with an arrow head that points to the most general class. An example of an inheritance relation can be found in Figure 2.1 where the component diagram inherits from the structure diagram.

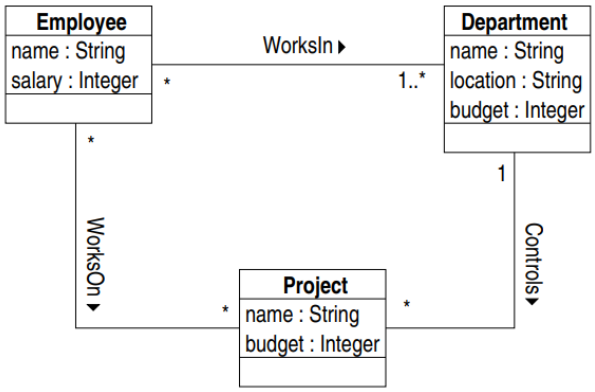


Figure 2.3: An example of a UML class diagram [4].

2.4 Object Constraint Language

In the past, UML class diagrams had a shortcoming since it was impossible to add constraints to variables; for example, there was no way to add a constraint that for every instance of Employee, the employee should have a positive salary. The Object Constraint Language (OCL) [3] is a declarative language that describes constraints in the form or rules that apply to UML models. A few examples of OCL constraints can be found in Example 3.

Example 3. A few OCL examples that add constraints to the class diagram in Figure 2.3. The comment above the constraint explains what the constraint does.

```
1    // every department must have a budget of at least 0
2    context Department inv: self.budget >= 0
3
4    // all employees that a project includes must also be included in the
5    // department of this project
6    context Project inv: self.department.employee->includesAll(self.employee)
```

OCL has a type system that uses seven predefined types and the types that are defined in the UML class diagram. The seven predefined types, with examples, are:

- **OclInvalid**, such as, *invalid*.
- **OclVoid**, such as *null* and *invalid*.
- **Boolean**, such as *true* and *false*.
- **Integer**, such as $-5, 0, 7$, etc.
- **Real**, such as $-2.1, 1.5$, etc.
- **String**, such as "*Just a string*".
- **UnlimitedNatural**, such as, $0, 1, 42$, etc.

From the seven predefined types, the *null* and *invalid* are interesting values. *null* occurs if a

value is not instantiated and therefore still *null*. *invalid* occurs if the OCL constraint contains an operation call on a value that is not instantiated, so an operation call on the *null* value.

A machine readable definition of OCL 2.0 and upwards is not given in the OCL version specification. The last machine readable OCL grammar definition is of OCL 1.5 and can be found in Appendix A (taken from [8]). As it can be seen in the appendix, the grammar definition does not define the methods that can be used on OCL objects or sets. For example, given the grammar definition, it is not possible to know if `isEmpty()` is a method that is part of the OCL syntax.

2.5 Decidability

The property of decidability means there exists an algorithm that gives the correct answer given a question. This algorithm has to be executable in a finite amount of time. For example, decidability can say something whether an OCL constraint is satisfied given an instance of a UML class diagram. Whether this is decidable or not will be discussed in Section 3.2.

In logic, decidability of a problem tells us if it is possible to determine the correct answer. If a theory is decidable then there exists an effective method to determine whether an arbitrary formula is included in or excluded from the theory.

Besides decidable, a theory can also be semidecidable, which is a weaker statement than decidability. A theory θ is semidecidable if there exists an effective method, given an arbitrary formula ϕ, that always gives the correct answer if ϕ is included in θ, but the algorithm may give no answer at all if ϕ is not included in θ.

2.6 Graph Theory Introduction

In [9], König et al. have given an introduction about graph transformation, which is used in the definitions of this chapter, except for the definitions involving (nested) graph conditions, constraints and clans. For these definitions, the work of Habel et al. [10] is used. These definitions are necessary to understand the transformations from OCL to nested graph constraints that will be explained in Section 3.3.

Graph A graph G is a tuple $G = (V, E, s, t, l)$ where

- V is a set of nodes (or vertices),
- E is a set of edges,
- s: $E \to V$ is the source function,
- t: $E \to V$ is the target function,
- l: $E \to L$ is the labelling function and
  L is the fixed set of labels, such that $l \in L$.

Graph morphism Let G, H be two graphs. A morphism $\varphi : G \to H$ is a mapping that maps a graph to another graph. This mapping uses two functions: one that maps nodes to nodes ($\varphi_V : V_G \to V_H$) and one that maps edges to edges ($\varphi_E : E_G \to E_H$). If φ_E maps an edge to an edge, then there must be a mapping between the source and target nodes of the two edges. A mapping between the labels is not necessary, because the labels of the graph must be preserved. This means that for all $e \in E_G$ the following conditions must hold:

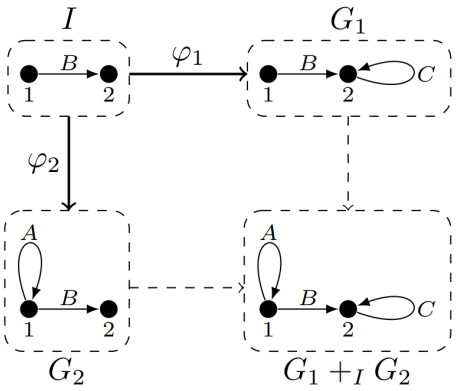
- $s_H(\varphi_E(e)) = \varphi_V(s_G(e))$
- $t_H(\varphi_E(e)) = \varphi_V(t_G(e))$

- $l_H(\varphi_E(e)) = l_G(e)$

Injective morphism A morphism $\varphi : G \to H$ is injective (or one-to-one) if each element in H is the image of at most one element in G. If φ is an injective morphism, it can be denoted as $\varphi = G \hookrightarrow H$

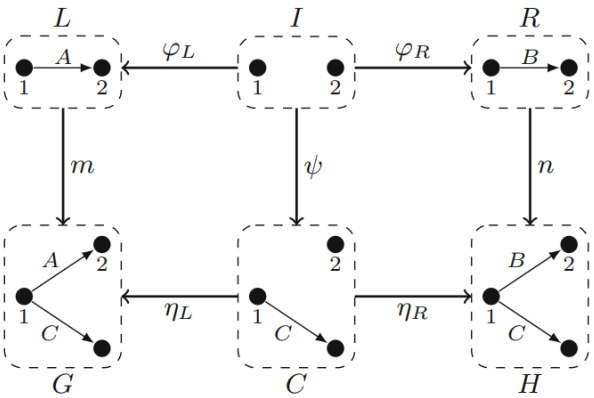
Graph gluing consists of combining two graphs (G_1, G_2) with an intermediate graph I that has overlap with both G_1 and G_2. The resulting graph is denoted as $G_1 +_I G_2$. In the literature graph gluing is also called pushout. An example of graph gluing can be found in Example 4.

Example 4. The following graph gluing example is given in [9], where you can see that the graphs G_1 and G_2 are glued to I, which results in $G_1 +_I G_2$.



Graph transformation rule transforms a graph in two steps, with a morphism for every step. The first step only removes nodes and edges with morphism φ_L, this morphism is from the intermediate graph I to the left-hand side graph L. The second step only adds nodes and edges with morphism φ_R, this morphism is from I to the right-hand side graph R. The two steps together are denoted as $L \xleftarrow{\varphi_L} I \xrightarrow{\varphi_R} R$. The two step technique prevents dangling edges. A dangling edge is an edge from which the source or the target node has been removed.

Example 5. The following graph transformation example is given in [9]. On the first row the graph transformation is defined and it is applied on the second row. As you can see on the first row, φ_L removes an A edge between two nodes and φ_R adds a B edge between those two nodes. On the second row this transformation rule is applied to graph G. In graph C the A edge is removed and in graph H the B edge is added.



Nested graph condition Given a graph G, a (nested) graph condition over G is defined to be of the form *true* or $\exists(a, c)$, where $a : G \to C$ is a morphism and c is a condition over C. This means that a nested graph condition is defined recursively. An example of a nested graph condition is given in Example 6. If the graph condition is built by injective morphisms, then the graph condition is injective. Graph conditions can be denoted in a more compact form: $\exists a$ abbreviates $\exists(a, true)$ and $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$. Nested graph conditions over the empty graph $\emptyset$ are called nested graph constraints.

Definition 1. To be able to denote nested graph conditions, you need a way to denote the graph part. Nodes are depicted as rectangles: $\boxed{\text{v:T}}$, in which v is the node name and T is its type. If we are talking about a set of nodes, for example {u, v}, then the node is depicted as $\boxed{\text{u=v:T}}$ which says that the node of type T has both the names u and v. Edges are drawn by arrows from the source node to the target node and their labels are put on top of the edge, such as $\boxed{\text{v:T}} \xrightarrow{l} \boxed{\text{u:T}}$. If a node contains an attribute, this is depicted as an additional node with the name of the attribute as the label of the edge. For example, if an instance of type T has an attribute budget with the value 1, this is depicted as: $\boxed{\text{v:T}} \xrightarrow{budget} \boxed{\text{1:int}}$. Operations are depicted as a relation between two nodes, the symmetrical equality operations, $\texttt{eq} \in \{=, <>\}$ are depicted as: $\boxed{\text{v:T}} \overset{\texttt{eq}}{—} \boxed{\text{v':T'}}$. The asymmetrical compare operations, $\texttt{op} \in \{<, \leq, \geq, >\}$ are depicted as: $\boxed{\text{v:T}} \xrightarrow{\texttt{op}} \boxed{\text{v':T'}}$. For example: $\boxed{\text{v:T}} \xrightarrow{\geq} \boxed{\text{v':T'}}$ means that $\boxed{\text{v:T}}$ is greater or equal than $\boxed{\text{v':T'}}$.

Example 6. Below we will give an example of a nested graph constraint and with an example we show when the condition is satisfied and when it is not satisfied. The following nested graph condition, which can be applied on the class diagram, given in Figure 2.3, checks if all budgets are positive integers.

$$\forall(\emptyset \to \boxed{\text{self:Department}}, \exists(\boxed{\text{self:Department}} \to (\boxed{\text{self:Department}} \xrightarrow{budget} \boxed{\text{v:int}} \overset{\geq}{—} \boxed{\text{0:int}})))$$

An instance of the class diagram is given in Figure 2.4. For this instance there exists a morphism from every morphism that is part of the nested graph condition to the instance graph, therefore the instance graph matches the condition. So the condition is satisfied, because every node of the type Department contains a budget that is greater or equal to 0. If one of the department instances had a negative budget then there would not exist a morphism from the nested graph constraint to the instance graph, therefore there would be no matching and the condition would not be satisfied.

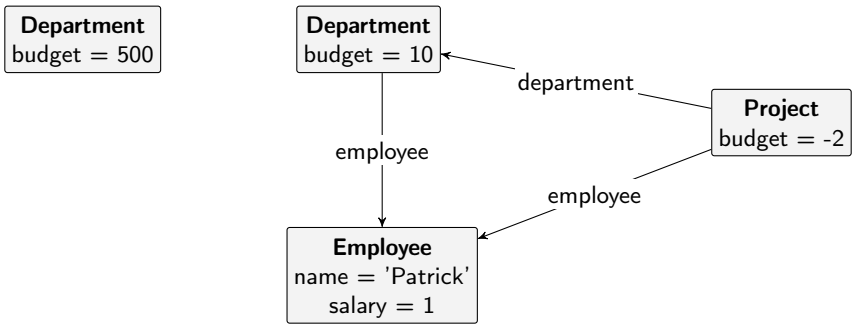


Figure 2.4: An instance of the class diagram in Figure 2.3.

Lax Condition Lax conditions are introduced by Radke et al. in [11]. Lax conditions are based on nested graph conditions. The big difference is that the morphisms are replaced with

graphs. This is allowed because the morphisms of the nested graph constraints do only extend the source morphism graph.To achieve this, the domain of the morphisms is removed and only the resulting graphs of a morphism are used. Therefore a lax condition is of the form *true* or $\exists(C, c)$, where C is a graph and c is *true* or another lax condition. For a more compact notation $\exists(C)$ abbreviates $\exists(C, true)$.

Lax conditions are based on nested graph constraints. To make the connection between those two more clear we will give a small example. Let C_1, C_2 and C_3 be graphs for which $C_2 \subseteq C_1$ holds. On the left side the nested graph constraint is given and on the right side the lax condition is given:

$$\forall(\emptyset \to C_1, \exists(C_2 \to C_3, true)) \equiv \forall(C_1, \exists(C_3))$$

Clan Given a node v, clan(v) is the set of all supertypes of v, including the type of v itself.

Example 7. The class diagram in Figure 2.3 does not contain inheritance. So for this example we will use the class diagram in Figure 2.5. In this example clan(v:B) = $\{B, A\}$ and clan(v:A) = $\{A\}$.

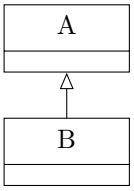


Figure 2.5: A small class diagram that contains inheritance to support Example 7.

clan-disjoint (union) Two graphs C_1 and C_2 are clan-disjoint if the clans of the types of C_1 and C_2 are disjoint, so if clan(C_1) $\cap$ clan(C_2) = $\emptyset$. The union of two clan-disjoint graphs is denoted as $C_1 + C_2$.

Example 8. To give an example of a clan-disjoint union we will use the class diagram given in Figure 2.3.

Let C_1 = e:Employee and C_2 = d:Department
Then $clan(C_1)$ = {Employee} and $clan(C_2)$ = {Department}
$clan(C_1) \cap clan(C_2) \equiv$ {Employee} $\cap$ {Department} = $\emptyset$
So C_1 and C_2 are clan-disjoint and thus the clan-disjoint union is allowed
$C_1 + C_2$ = e:Employee d:Department
$C_1 + C_2$ is one graph consisting of two nodes.

2.7 GROOVE

GROOVE [5] is one of the many existing graph transformation tools, other existing tools are Henshin [12], GReAT [13] and even more can be found in [14]. GROOVE stands for GRaph-based Object-Oriented VErification and is developed at the Formal Methods and Tools research group at the University of Twente. As the name suggests, the main goal of the tool is to support model checking of object-oriented systems by using graph transformations. States are encoded in graphs and transitions between states are encoded with graph transformations. GROOVE is able to use graph transformations to generate the state space that can be used for model

checking. This combination makes GROOVE an interesting tool that combines the areas of model checking and graph transformations.

GROOVE does not follow all the predefined definitions completely. First the edges in GROOVE are defined as a tuple of its source, label and target function. This has as a result that if two edges have the same source, label and target then GROOVE treats it as one edge and thus duplicate edges cannot exist. Second in the definition of morphisms we stated that morphisms have to preserve all the labels. This is true in GROOVE with the exception of type labels, in the case of type labels it is allowed to have a label that is of a subtype, as defined in the type graph. If this is not the desired behaviour, then you can use sharp typing. This is done by prepending a hashtag (#) to the type label and this label has to be preserved by any morphism.

Within GROOVE it is possible to define operations on values of primary attributes with the so called product nodes, these are depicted as diamonds in graphs. The rule graph in Figure 2.7 contains such a product node. In this example we make use of the `ge` product, that stands for greater than or equal, $\pi 0$ and $\pi 1$ are the two input parameters and thus the operation should be read as: $\pi 0 \geq \pi 1$ which should evaluate to *true*. Besides basic compare operations, GROOVE has support for more complex operations such as calculating the minimum or maximum value, given a set of numbers. These operations are called `bigmin` and `bigmax`.

The quantifiers in GROOVE have an additional role, these quantifiers are also used to encode logical *and* ($\wedge$) and *or* ($\vee$) operations. Two incoming *in* edges at an existential quantifier encode the $\wedge$ operator and two incoming *in* edges at a universal quantifier encode the $\vee$ operator. Figure 2.6 gives an example of two incoming *in* edges at a universal quantifier and so it encodes the logical *or* operation.

In [15], Rensink describes with formal definitions how it is possible to represent first-order logic with graphs. This theory is applied within GROOVE.

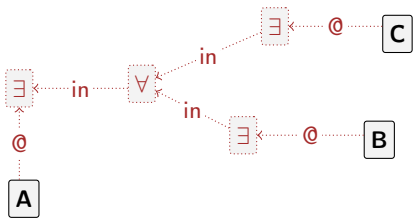


Figure 2.6: The way to encode $\exists(\boxed{\text{A}}, \exists(\boxed{\text{B}}) \vee \exists(\boxed{\text{C}}))$ in GROOVE.

Example 9. At the moment we have not shown how to rewrite an OCL constraint to a nested graph constraint. For now we assume that `context Department inv: self.budget >= 0` corresponds to $\forall(\boxed{\text{self:Department}}, \exists(\boxed{\text{self:Department}} \xrightarrow{budget} \boxed{\text{v:int}} \xrightarrow{\geq} \boxed{\text{0:int}}))$. Now we can apply the next step, converting the nested graph constraint to GROOVE, resulting in the graph given in Figure 2.7, this constraint will only apply if all departments have a budget that is greater than or equal to 0. Figure 2.8 gives an instance model, this model does satisfy the given constraint because there exists a morphism from the given nested graph condition to this model. If one of the departments had a negative budget, then the morphism would not exist and thus the model would not satisfy the condition.

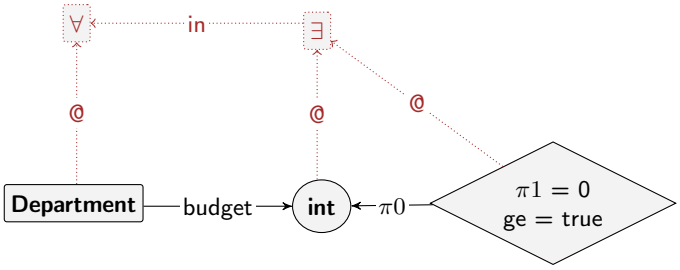


Figure 2.7: The rule graph that corresponds to: `context Department inv: self.budget >= 0`.

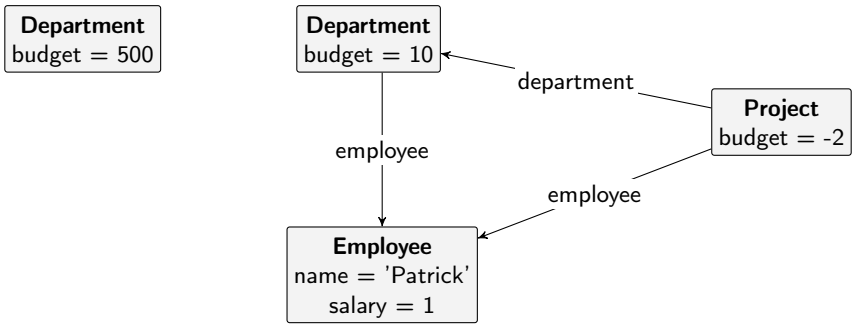


Figure 2.8: An instance model where every department has a positive budget.

2.8 Parsing

Parsing is the process that takes text as input and builds a data structure, which is often some kind of parse tree, Concrete Syntax Tree (CST) or Abstract Syntax Tree (AST). A CST is an ordered, rooted tree that represents the syntactic structure of the textual input according to some grammar. An AST is a more compact form of the CST, since in most cases parsing results in a CST and then users are able to create their own AST. In our case, the OCL constraints will be the textual input and the syntax tree should represent the syntactic structure according to the OCL definition, which can be found in Appendix A.

A parser for a given language does not always exist. If the parser does not exist yet then there are two ways to create it: by developing it by hand or by generating it with a parser generator.

Given a defined grammar, a parser generator is able to generate the source code of a parser that can parse the input that satisfies the defined grammar. In most cases, using a parser generator takes less effort than developing a complete parser by hand. So using a parser generator is the recommended way to obtain a parser.

A few examples of parser generators are: ANTLR [16], YACC [17] and SableCC [18], but there are many more. In this thesis we will use SableCC because the OCL grammar already exists for SableCC [19], which is not the case for the other parser generators.

Chapter 3

Related Work

In the past there has been some research about transforming OCL constraints to a different logic such that it is easier to reason about these constraints. In this related work section, the different approaches to transform OCL constraints are discussed.

3.1 OCL to FOL

In [20], Beckert et al. present a way to transform a UML class diagram with corresponding OCL constraints into FOL. The transformation of the OCL constraints depends on the transformation of the UML class diagram. The transformation of a class diagram consists of three parts, where each part creates its own axioms. Additional axioms may be added during the OCL constraint transformations, but this depends on the OCL constraints. The three parts that create axioms are:

1. A part independent of the class diagram that contains the abstract data types and abstract data collection types. Examples are Integers, Booleans, Sets and Bags.
2. The inter-dependencies among the function and relation symbols that can be extracted from the class diagram, for instance, the symmetrical aspects of associations in the class diagram. If we apply this to the association WorksIn of our class diagram in Figure 2.3, this will result in the following three axioms:
   $\forall e : Employee\ \forall d : Department(d \in department(e) \iff e \in employees(d))$
   $\forall e : Employee\ \forall d : Department(d \in department(e) \iff worksIn(e, d))$
   $\forall e : Employee\ \forall d : Department(e \in employees(d) \iff worksIn(e, d))$
3. The axioms that represent the restrictions on system states in the class diagram, such as the multiplicity constraints and subtyping restrictions.

In the last section of [20], a start is given to get rid of the additional axioms at the OCL transformation part. This topic is not explained in the paper and only a reference to another paper is given, but sadly this paper cannot be found anymore and we were not able to make contact with the author, therefore this explanation is missing.

3.2 OCL to Relational Algebra

In [21], Franconi et al. have proved that in full OCL, checking whether an OCL constraint is satisfied in an arbitrary UML diagram is not decidable, and not even semidecidable. This means

that it is possible to have an arbitrary model of which we do not know if it satisfies the OCL constraint.

The proof that OCL is not semidecidable is based on the problem of word acceptance in a type-0 grammar. In this problem, it is checked whether a word is produced by a type-0 grammar or not, and exactly this is undecidable. Since it is possible to define the problem of word acceptance in an OCL constraint, it is possible to create an undecidable OCL constraint, therefore OCL is undecidable. Thereafter to prove non semidecidability the type-0 acceptance problem is reduced to checking whether some word is not produced by the type-0 grammar. This can be defined by negating the undecidable OCL constraint and therefore OCL is even non-semidecidable. The full proof can be found in [21].

Given this knowledge, we know that it is not possible to transform an arbitrary OCL constraint to a nested graph constraint. In [21], Franconi et al. have identified an OCL fragment that has the same expressiveness as FOL. This fragment is called OCL_{FO} and its syntax can be found in Appendix B. The OCL_{FO} fragment has the following limitations:

1. Operations that are not FOL are not part of the OCL_{FO} syntax, such as, (i) operation `closure`, or (ii) aggregation functions as `count`. With respect to (i) `Closure`, or better known as transitive closure has the transitivity property, to encode this in FOL you will need an infinite formula to be able to encode all possible options of transitivity, which makes it impossible. With respect to (ii), it is possible to use the function `Size` and compare the size of a set with a fixed integer. Because of the fixed integer you know exactly the size of the FOL formula. However it is not possible to compare the size of two sets because this will result, again, in a possible infinite formula.
2. The following operations are kept out of the syntax of OCL_{FO}, even though these operations can be transformed to FOL. The reason Franconi et al. gave is that their OCL_{FO} fragment is based on two case studies, DBLP[7], and osCommerce [22], in neither of them there exists an OCL constraint that uses one of the following operations:
   - `let ... in`
   - `if ... then ... else ... endif`
   - `including`
   - `excluding`

Besides giving a proof that full OCL is not semidecidable and defining the OCL_{FO} fragment, [21] gives a transformation from OCL_{FO} to relational algebra. However, the paper does not give a full description of how to transform OCL_{FO} to relational algebra, since the transformation is only given in short pieces of pseudocode.

3.3 OCL to Nested Graph Constraints

In [23], Arendt et al. have made a beginning of transforming core OCL constraints to nested graph constraints. Although the definition of the core OCL fragment is not given in the paper, we can deduce the definition from the rewrite rules and determine in this way which operations are considered to be part of the core OCL fragment. One of the limitations is that of all the available collection types within OCL, only the set collections are considered, and only the following boolean-typed set operations are allowed: `isEmpty, notEmpty, exists` and `forAll`.

In [6], Radke et al. have extended the work in [23] in which they have focused on set operations like `select`, `collect`, `union` and `size`. The complete list of transformation rules can be found in Appendix C. Besides the transformation rules, Radke et al. have defined some equivalence

rules to make the nested graph constraint more compact, which results in a constraint that is easier to read and understand. Radke's equivalences rules can be found in Appendix C.1.

In our research, a broader fragment of OCL will be addressed. To be more specific, we have tried to create transformation rules for the OCL_{FO} fragment, including the operations that [21] kept out of the fragment, so that it results in a more complete set of transformation rules that can handle all OCL constraints that are FOL compatible.

Chapter 4

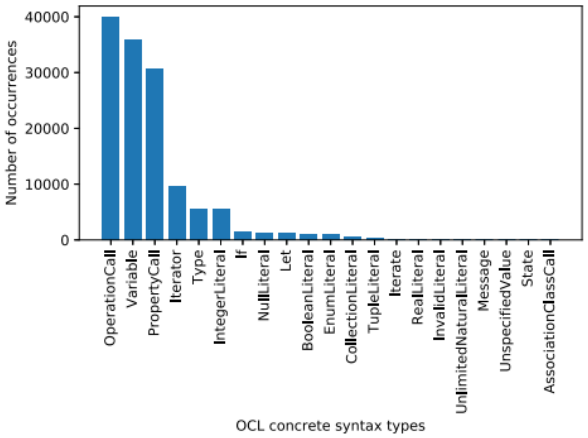
Transforming OCL constraints to graphs

In this chapter, we explain our proposal for transforming OCL constraints to nested graph constraints. First in Section 4.1 the OCL fragment supported by our transformation is given and in Section 4.2 we focus on the collection type. In Section 4.3 we explain why we match non-injectively. Finally, in Section 4.4, our transformation and equivalence rules are given, and in Section 4.5 we focus on our contribution and changes relative to the transformation rules of Radke et al. given in Appendix C.

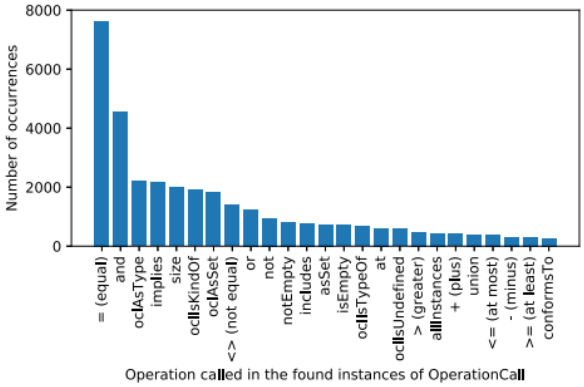
4.1 OCL fragment

It is known that full OCL is non-semidecidable, as discussed in Section 3.2. We decided to select a fragment of OCL that is decidable because our transformation should be consistent with the existing first-order logic semantics.

In [24], Noten et al. have performed an empirical research based on 9188 OCL constraints gathered from 504 EMF (Eclipse modeling Framework) metamodels in 245 systematically selected GitHub repositories. Based on all the gathered data, two figures are created, which can be found in Figure 4.1. By analysing these figures, we can conclude that there are 3 most common OCL constructs and the others appear much less often. One of these constructs is an operation call, which is split in the top 25 operation calls in Figure 4.1b.



(a) Frequency of OCL constructs.



(b) Frequency of top 25 operations.

Figure 4.1: Empirical research results of [24].

OCL_{FO} is defined in Appendix B and contains the part of OCL that can be transformed to relational algebra, which in turn is known to be equivalent to (domain-independent) first-order logic [21]. Through the empirical research in [24], we know which syntax types and operation calls of OCL have been used most often in the considered case studies. By comparing the OCL_{FO} fragment with the results of the empirical research, it is possible to conclude that 21 of the top 25 operations are part of the OCL_{FO} fragment. The following four operations are not part of the OCL_{FO} fragment:

- **oclAsSet**: converts a non-collection value to a collection value.
- **at**: has the following two implementations:
  - Returns the i-th element of the sorted list.
  - Returns the character of the string at position i.
- **oclIsUndefined**: returns true if `self` is equal to *invalid* or equal to *null*.
- **conformsTo**: is a double type check, it checks both the collection and the elements in that collection. For example, `Set(Bicycle) conforms to Collection(Transport)`, is true if `set` is a subtype of `Collection` and `Bicycle` is a subtype of `Transport`.

It would be useful to extend the OCL_{FO} fragment with these four operations, such that the fragment contains the top 25 most used OCL operations, according to the empirical research. We will indicate for each operation if it is possible to extend the OCL_{FO} fragment with the operation or why it is not possible.

- **oclAsSet** will be included in the OCL_{FO} fragment.
- **at** has two implementations but the OCL_{FO} fragment will not be extended with either of them because:
  - The collections we will focus on in graphs do not have any kind of ordering, so there is no i-th element. The restriction on collection types will be explained in Section 4.2.
  - Within GROOVE it is not possible to do a substring or character selection operation on a string and therefore we will not be able to implement it.
- **oclIsUndefined** is not possible to include because it is not possible to encode the invalid part since it is unknown which part of the expression is invalid. Therefore we will not include this operation in the OCL_{FO} fragment.
- **conformsTo** is not possible to include because it type checks on two levels: the type of the elements within the set and the type of the collection itself. Since we do not have a

type hierarchy of collection types it is not possible to implement this operation.

4.2 Collection types

UML defines four types of collections; bags, sets, ordered sets and sequences, each with their own characteristics. In graphs, it is much more difficult to differentiate these collection types. For example, ordered sets and sequences encode some sort of ordering in their collections, but since graphs do not represent any sort of ordering, it is not straightforward to encode these collections in graphs.

The difference between bags and sets is that all elements in a set are unique and thus that bags may have duplicate elements. The difficulty lies in creating a collection in a GROOVE graph with duplicate elements because creating two edges with the same names between two nodes has the same effect as creating one edge. This behaviour of a GROOVE graph is different from the behaviour of a graph defined in Section 2.6. To be able to encode bags you will need additional nodes in between types to represent the 'slots' of a bag, which would result in something like Figure 4.2. Creating a bag collection is more difficult than creating a set collection, so bags are not often represented in graphs. So in this thesis we will focus on the set collection type.

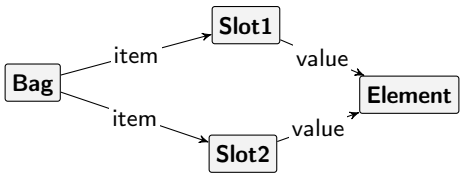


Figure 4.2: Representing a bag in a graph.

4.3 Injective matching

In [11], Radke et al. have defined that the matching should always happen injectively. This means that the found morphism from a nested graph constraint to an instance graph, as discussed as the matching in Example 6 in Section 2.6, should be an injective morphism.

Matching injectively makes some transformation rules easier and others more complex. For example transformation rule xviii (`expr1.attr1 op expr2.attr2`) in Appendix C will become easier. The part before the $\vee$ is necessary because Radke et al. match injectively. If you match non-injectively the part before the $\vee$ can be removed and the rule embodies the same constraint. On the other side if we take a look at transformation rule xiv (`expr->size()`) in Appendix C. This transformation rule will become more complex because this rule works based on the concept that the matching is injective. If you stop matching injectively, you have to add the constraint that every node has to be different from each other. Otherwise every v_i could be matched to one and the same node, which is not the intention.

We are going to change from matching injectively to matching non-injectively, this reduces the complexity of multiple transformation rules and only increases the complexity of one transformation rule, the transformation of the `size()` operation. This will result in some changes in transformation rules, as will be discussed in Section 4.5.

4.4 Transforming OCL to Lax Conditions

In [11], Radke et al. have defined transformation rules to transform OCL constraints to nested graph constraints in the format of lax conditions. We have used their definitions as a starting point and have continued by improving and extending their definitions. The complete overview with all our transformation rules is given in Section 4.4.1 and our equivalences are given in Section 4.4.2. The transformation rules are categorized in the following four types of transformations:

1. Invariant transformation tr_I, this category is the starting point of a transformation and transforms the complete OCL invariant.
2. Expression transformation tr_E, this category transforms boolean expressions, such as comparisons.
3. Navigation transformation tr_N, this category is responsible for creating the path to the variable that is created in a previous transformation. To be able to create this connection to an existing variable, this category has as input the remaining OCL expression and the existing variable.
4. Set transformation tr_S, this category is responsible for navigation expressions of sets, so this category has the same input parameters as the previous category.

4.4.1 Transformation Rules

Let `expr`, `expr1` and `expr2` be OCL expressions, v, v‘ names of nodes (i.e. variables), T = t(v) denotes the type of v and likewise T‘=t(v‘), attr1 and attr2 be attribute names, eq $\in \{=, <>\}$, op $\in \{<, \leq, \geq, >\}$ a comparison operator, `attr` be an attribute of a class and `N` a constant value.

Group	#	OCL	Nested Graph
$\mathbf{tr}_I$	1	`context C inv: expr`	$\forall(\boxed{\text{self:C}}, tr_E(\texttt{expr}))$
	2	`context var:C inv: expr`	$\forall(\boxed{\text{var:C}}, tr_E(\texttt{expr}))$
$\mathbf{tr}_E$	3	`True`	`True`
	4	`not expr`	$\neg tr_E(\texttt{expr})$
	5	`expr1 and expr2`	$tr_E(\texttt{expr1}) \wedge tr_E(\texttt{expr2})$
	6	`expr1 or expr2`	$tr_E(\texttt{expr1}) \vee tr_E(\texttt{expr2})$
	7	`expr1 xor expr2`	$(tr_E(\texttt{expr1}) \vee tr_E(\texttt{expr2})) \wedge \neg(tr_E(\texttt{expr1}) \wedge tr_E(\texttt{expr2}))$
	8	`expr1 implies expr2`	$tr_E(\texttt{expr1}) \implies tr_E(\texttt{expr})$
	9	`if cond then expr1 else expr2 endif`	$(tr_E(\texttt{cond}) \wedge tr_E(\texttt{expr1})) \vee (\neg tr_E(\texttt{cond}) \wedge tr_E(\texttt{expr2}))$
	10	`expr = null`	$tr_E(\texttt{expr->isEmpty()})$
	11	`expr <> null`	$tr_E(\texttt{expr->notEmpty()})$
	12	`expr1 eq expr2` t(expr1)=t(expr2)=Set(T)	$\forall(\boxed{\text{v1:T}}, \exists(\boxed{\text{v1:T}} \xrightarrow{\texttt{op}} \boxed{\text{v2:T}}, tr_S(\texttt{expr2}, \boxed{\text{v2:T}})) \wedge tr_S(\texttt{expr1}, \boxed{\text{v1:T}})) \wedge \forall(\boxed{\text{v3:T}}, \exists(\boxed{\text{v3:T}} \xrightarrow{\texttt{op}} \boxed{\text{v4:T}}, tr_S(\texttt{expr1}, \boxed{\text{v4:T}})) \wedge tr_S(\texttt{expr2}, \boxed{\text{v3:T}}))$
	13	`expr1 eq expr2`	$\exists(\boxed{\text{v:T}} \xrightarrow{\texttt{eq}} \boxed{\text{v‘:T‘}}, tr_N(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_N(\texttt{expr2}, \boxed{\text{v‘:T‘}}))$
	14	`expr1 op expr2`	$\exists(\boxed{\text{v:T}} \xrightarrow{\texttt{op}} \boxed{\text{v‘:T‘}}, tr_N(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_N(\texttt{expr2}, \boxed{\text{v‘:T‘}}))$
	15	`expr1->exists(v:T|expr2)`	$\exists(\boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \wedge tr_E(\texttt{expr2}))$
	16	`expr1->forAll(v:T|expr2)`	$\forall(\boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \implies tr_E(\texttt{expr2}))$
	17	`expr1->forAll(`v_1`,...,`v_n`:T|expr2)`	$\forall(\boxed{v_1\text{:T}} ... \boxed{v_n\text{:T}}, (tr_S(\texttt{expr1}, \boxed{v_1\text{:T}}) \wedge ... \wedge tr_S(\texttt{expr1}, \boxed{v_n\text{:T}})) \implies tr_E(\texttt{expr2}))$

	18	`expr1->includesAll(expr2)`	$\forall(\boxed{\text{v:T}}, tr_S(\texttt{expr2}, \boxed{\text{v:T}}) \implies tr_S(\texttt{expr1}, \boxed{\text{v:T}}))$
	19	`expr1->excludesAll(expr2)`	$\forall(\boxed{\text{v:T}}, tr_S(\texttt{expr2}, \boxed{\text{v:T}}) \implies \neg tr_S(\texttt{expr1}, \boxed{\text{v:T}}))$
	20	`expr1->includes(expr2)`	$tr_E(\texttt{expr1->includesAll(Set\{expr2\})})$
	21	`expr1->excludes(expr2)`	$tr_E(\texttt{expr1->excludesAll(Set\{expr2\})})$
	22	`expr->notEmpty()`	$\exists(\boxed{\text{v:T}}, tr_S(\texttt{expr}, \boxed{\text{v:T}}))$
	23	`expr->isEmpty()`	$\neg tr_E(\texttt{expr->notEmpty()})$
	24	`expr->size() >= N`	$\exists(\boxed{\text{v}_1\text{:T}}...\boxed{\text{v}_N\text{:T}}, \bigwedge_{i\in\{1..N\}} tr_S(\texttt{expr}, \boxed{\text{v}_i\text{:T}}) \bigwedge_{j=\{1..N\}}^{i\neq j} \boxed{\text{v}_i\text{:T}} \neq \boxed{\text{v}_j\text{:T}})$
	25	`expr->size() > N`	$tr_E(\texttt{expr->size() >= N+1})$
	26	`expr->size() = N`	$tr_E(\texttt{expr->size() >= N}) \wedge \neg tr_E(\texttt{expr->size() >= N+1})$
	27	`expr->size() <= N`	$\neg tr_E(\texttt{expr->size() >= N+1})$
	28	`expr->size() < N`	$\neg tr_E(\texttt{expr->size() >= N})$
	29	`expr->size() <> N`	$\neg tr_E(\texttt{expr->size() = N})$
	30	`expr1->one(v:T|expr2)`	$tr_E(\texttt{expr1->select(v:T\|expr2)->size() = 1})$
	31	`expr->isUnique(attr)`	$tr_E(\texttt{expr->forAll(v1,v2:T\|v1 <> v2} \implies \texttt{v1.attr <> v2.attr})$
	32	`expr.oclIsKindOf(T`)`	$\exists(\boxed{\text{v:T}} \hookrightarrow \boxed{\text{v:T`}}, tr_N(\texttt{expr}, \boxed{\text{v:T}}))$
	33	`expr.oclIsTypeOf(T`)`	$\exists(\boxed{\text{v:T}} \hookrightarrow \boxed{\text{v:T`}}, \bigwedge_{T``\in clan(T`)}^{T``\neq T`} \neg\exists(\boxed{\text{v:T`}} \hookrightarrow \boxed{\text{v:T``}}) \wedge tr_N(\texttt{expr}, \boxed{\text{v:T}}))$
$\mathbf{tr}_N$	34	`(v,` $\boxed{\text{v`:T}}$`)`	$\exists(\boxed{\text{v=v`:T}})$
	35	`(N,` $\boxed{\text{v:T}}$`)`	$\exists(\boxed{\text{N=v:T}})$
	36	`(expr.oclAsType(T`),` $\boxed{\text{v:T`}}$`)`	$\exists(\boxed{\text{v:T}} \hookrightarrow \boxed{\text{v:T`}}, tr_N(\texttt{expr}, \boxed{\text{v:T}}))$
	37	`(expr.attr,` $\boxed{\text{v:T}}$`)`	$\exists(\boxed{\text{v`:T`}} \xrightarrow{attr} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v`:T`}}))$
	38	`(expr.attr->min(),` $\boxed{\text{x:T}}$`)`	$\exists(\boxed{\text{v1:T`}} \xrightarrow{attr} \boxed{\text{x:T}}, tr_N(\texttt{expr}, \boxed{\text{v1:T`}}) \wedge \forall(\boxed{\text{v2:T`}} \xrightarrow{attr} \boxed{\text{v3:T}}, tr_N(\texttt{expr}, \boxed{\text{v2:T`}}) \implies \exists(\boxed{\text{x:T}} \xrightarrow{\leq} \boxed{\text{v3:T}})))$
	39	`(expr.attr->max(),` $\boxed{\text{x:T}}$`)`	$\exists(\boxed{\text{v1:T`}} \xrightarrow{attr} \boxed{\text{x:T}}, tr_N(\texttt{expr}, \boxed{\text{v1:T`}}) \wedge \forall(\boxed{\text{v2:T`}} \xrightarrow{attr} \boxed{\text{v3:T}}, tr_N(\texttt{expr}, \boxed{\text{v2:T`}}) \implies \exists(\boxed{\text{x:T}} \xrightarrow{\geq} \boxed{\text{v3:T}})))$
$\mathbf{tr}_S$	40	`(expr.attr,` $\boxed{\text{v:T}}$`)`	$\exists(\boxed{\text{v`:T`}} \xrightarrow{attr} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v`:T`}}))$
	41	`(T.allInstances(),` $\boxed{\text{v:T}}$`)`	$\exists(\boxed{\text{v:T}})$
	42	`(Set{expr`$_1$`,...,expr`$_n$`},` $\boxed{\text{v:T}}$`)`	$tr_N(\texttt{expr}_1, \boxed{\text{v:T}}) \vee ... \vee tr_N(\texttt{expr}_n, \boxed{\text{v:T}})$
	43	`(expr1->union(expr2),` $\boxed{\text{v:T}}$`)`	$\exists(\boxed{\text{v`:T`}} \xrightarrow{=} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v`:T`}})) \vee \exists(\boxed{\text{v``:T``}} \xrightarrow{=} \boxed{\text{v:T}}, tr_S(\texttt{expr2}, \boxed{\text{v``:T``}}))$
	44	`(expr1->intersection(expr2),` $\boxed{\text{v:T}}$`)`	$\exists(\boxed{\text{v`:T`}} \xrightarrow{=} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v`:T`}})) \wedge \exists(\boxed{\text{v``:T``}} \xrightarrow{=} \boxed{\text{v:T}}, tr_S(\texttt{expr2}, \boxed{\text{v``:T``}}))$
	45	`(expr1 - expr2,` $\boxed{\text{v:T}}$`)`	$\exists(\boxed{\text{v`:T`}} \xrightarrow{=} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v`:T`}})) \wedge \neg\exists(\boxed{\text{v``:T``}} \xrightarrow{=} \boxed{\text{v:T}}, tr_S(\texttt{expr2}, \boxed{\text{v``:T``}}))$
	46	`(expr1->symmetricDifference(expr2),` $\boxed{\text{v:T}}$`)`	$tr_S(\texttt{(expr1->union(expr2)}) - \texttt{(expr1->intersection(expr2))}, \boxed{\text{v:T}})$
	47	`(expr1->including(expr2),` $\boxed{\text{v:T}}$`)`	$tr_S(\texttt{expr1->union(Set\{expr2\})}, \boxed{\text{v:T}})$
	48	`(expr1->excluding(expr2),` $\boxed{\text{v:T}}$`)`	$tr_S(\texttt{expr1 - Set\{expr2\}}, \boxed{\text{v:T}})$
	49	`(expr1->select(v:T|expr2),` $\boxed{\text{v`:T}}$`)`	$tr_S(\texttt{expr1}, \boxed{\text{v`:T}}) \wedge tr_E(\texttt{expr2})\{v`/v\}$
	50	`(expr1->reject(v:T|expr2),` $\boxed{\text{v`:T}}$`)`	$tr_S(\texttt{expr1}, \boxed{\text{v`:T}}) \wedge \neg tr_E(\texttt{expr2})\{v`/v\}$
	51	`(expr->selectByKind(T),` $\boxed{\text{v:T`}}$`)`	$\exists(\boxed{\text{v:T`}} \hookrightarrow \boxed{\text{v:T}}) \wedge tr_S(\texttt{expr}, \boxed{\text{v:T`}})$

	52	`(expr->selectByType(T),` $\boxed{\text{v:T‘}}$`)`	$\exists(\boxed{\text{v:T‘}} \hookrightarrow \boxed{\text{v:T}}, \bigwedge_{T\text{“}\in clan(T)}^{T\text{“}\neq T} \neg\exists(\boxed{\text{v:T}} \hookrightarrow \boxed{\text{v:T“}}))$ $\wedge\, tr_S($`expr,` $\boxed{\text{v:T‘}}$`)`
	53	`(expr->oclAsSet(),` $\boxed{\text{v:T}}$`)`	$tr_N($`expr,` $\boxed{\text{v:T}}$`)`

4.4.2 Lax Condition Equivalences

After creating the lax condition it is important to simplify the condition. In [11] Radke et al. have also created a starting point for these equivalences and we have improved and extended these equivalences. We will explain these improvement and extensions in Section 4.5. Our equivalence rules are defined in Definition 2.

Definition 2. Let $C_1 \oplus_P C_2$ denote the gluing or pushout of C_1 and C_2 along P and let $\mathcal{P}$ denote the set of all intersections of C_1 and C_2:

(E1) Given two nested lax conditions that are existentially quantified, we are able to resolve these conditions into one single lax condition with the following two equivalence rules:
  (a) $\exists(C_1, \exists(C_2)) \equiv \exists(C_1 + C_2)$ if C_1 and C_2 are clan-disjoint.
  (b) $\exists(C_1, \exists(C_2)) \equiv \exists(C_2)$ if $C_1 \subseteq C_2$ and $\equiv \exists(C_1)$ if $C_2 \subseteq C_1$.

(E2) The implies operation can be rewritten with the following equivalence rule:
  $(\exists(a) \implies \exists(b)) \equiv \forall(a, \exists(b))$

(E3) Given a lax condition that includes a logical and ($\wedge$) operation, we are able to merge the conditions such that we can rewrite the $\wedge$ with the following two equivalence rules:
  (a) $\exists(C_1, \exists(C_2) \wedge \exists(C_3)) \equiv \exists(C_1, \vee_{P\in\mathcal{P}} \exists(C_2 \oplus_P C_3))$, if for all node names occurring in both C_2 and C_3, a node with that name already exists in C_1.
  (b) $\exists(C_1) \wedge \exists(C_2) \equiv \exists(C_1 + C_2)$ if C_1 and C_2 are clan-disjoint and have disjoint sets of node names.

(E4) Given a node with multiple names, such as $\boxed{\text{v=u:T}}$, we are able to rename all the existing u nodes to v, with the following two equivalence rules:
  (a) $\exists(\boxed{\text{u:T}}, \exists(C) \wedge \exists(\boxed{\text{v=u:T}})) \equiv \exists(\boxed{\text{v:T}}, \exists(C\{v/u\}))$ provided that v does not exist in C and $C\{v/u\}$ is the graph obtained from C by renaming u to v.
  (b) $\exists(\boxed{\text{u:T}}, \exists(\boxed{\text{v=u:T}})) \equiv \exists(\boxed{\text{v:T}})$

4.5 Contribution

Our transformation extends the defined transformation rules of Radke et al. which have been introduced in Section 3.3 and can be found in Appendix C. In this section we look into our contribution to the changes and improvements we have made relative to the implementation of Radke et al.

The definitions of Radke et al. in [11] contain mistakes and in the included examples were intermediate steps missing. We have fixed the mistakes and recreated the missing intermediate steps such that we could implement the transformation from OCL to graph theory in GROOVE. The reason why the transformation rules did not work, how we have fixed it and an example to show that the new definition works, is given in Appendix D. Due to the limited time, we have chosen not to create a complete formal proof for the new definitions. Below we will give a short summary of the mistake that the transformation rule contained.

- The renaming Equivalence rule (E4), the problem with the definition in [11] was that variable was not renamed everywhere. The desired result is that the old variable does not occur in the condition anymore. We have changed the definition such that the equivalence rule has the desired result.

- `expr1 op expr2` transformation rule(s) had several problems with the given definition in [11], such as variables did disappear by applying equivalence rules and it was difficult to extend with additional operators. To overcome these problems we have created a complete new definition. The new transformation rules are 12, 13 and 14 (and additional corresponding navigation rules).
- The implication transformation rule vii that is defined in [11] did not correspond with the given examples. We have analysed the examples and determined that the examples had the desired result. We have changed the implication transformation rule such that it gives a logical implication ($\Longrightarrow$) and added a new equivalence rule (E2) that has the corresponding result to the given examples.
- Radke et al. have defined two variants of transformation rule (xxiii and xxiv): $tr_N(\texttt{expr.attr}, \boxed{\text{v:T}})$. Since we have decided to match non-injectively, as discussed in Section 4.3, the variant with a self loop is not necessary anymore and is thus removed.
- The transformation of `size` had to be extended ad previously indicated in Section 4.3. Because we mathc non-injectively the restriction that every node is different is added to the transformation rule. This resulted in the new transformation rule 24.

Below we will give a short summary of the new transformation rules defined that are and extension of the work of Radke et al.

- The transformation of `forAll` is extended such that multiple variables are allowed, resulting in the new transformation rule 17. Since we could use the definition of `forall` with one variable it was easy to extend the transformation rule for multiple variables.
- And lastly the following additional operations that we support relative to the work of Radke et al. in [11]:
  - `xor`; transformation rule 7. The definition was straightforward and is based on the logical equivalent of the `xor`.
  - `isUnique`; transformation rule 31. The definition was straightforward because we have used the `isUnique` definition that is found in [21].
  - `min`; transformation rule 38. The definition of this transformation rule was not straightforward and is based on the idea that there exists a number for which we know that all the other numbers are smaller or equal to that given number. With this in mind the definition took shape.
  - `max`; transformation rule 39. After the definition of `min` this transformation rule was straightforward.
  - `selectByKind`; transformation rule 51. The definition is based on the `oclIsKindOf` transformation rule, except that it has to be connected to the given context node. Given that transformation rule the definition was straightforward.
  - `selectByType`; transformation rule 52. The definition is based on the `oclIsTypeOf` transformation rule, except that it has to be connected to the given context node. Given that transformation rule the definition was straightforward.
  - `oclAsSet`; transformation rule 53. This definition mostly syntactic sugar and thus the definition was straightforward.

Chapter 5

Graph constraints to GROOVE

In the previous chapter we have defined our transformation rules from OCL to lax conditions. This chapter explains how the lax conditions are represented in GROOVE. First of all Section 5.1 explains why a type graph is mandatory. Section 5.2 explains how it is possible to encode conjunctions, disjunctions, negations and type checking in GROOVE. Section 5.3 explains in which steps a lax condition is transformed into a GROOVE graph.

5.1 Type graphs

To be able to use the OCL functionality in GROOVE it is mandatory to have an enabled type graph. The type graph is necessary to determine the type of expressions. For example consider the OCL expression `context Employee inv: self.department.notEmpty()`, that is applicable to the given UML class diagram in Figure 2.3. From the expression `self.department`, it is not possible to know the type, you only know that there exists an association from `Employee` that is called `department`. If we take a look in the class diagram, we can see that the type is `Department`. This information is also available in the GROOVE type graphs and therefore it is mandatory to have an enabled type graph. The corresponding type graph of the UML class diagram in Figure 2.3 is given in Figure 5.1.

In Appendix D.8 we explain how and why we have changed the definition of the operations `union`, `intersection` and `minus`$(-)$. The new implementation is based on the fact that all classes have one common supertype, the `Object` type. So when you want to use one of these three set based operations then it is mandatory to have the supertype `Object` defined in your type graph. That is why we have defined the `Object` class also in Figure 5.1.

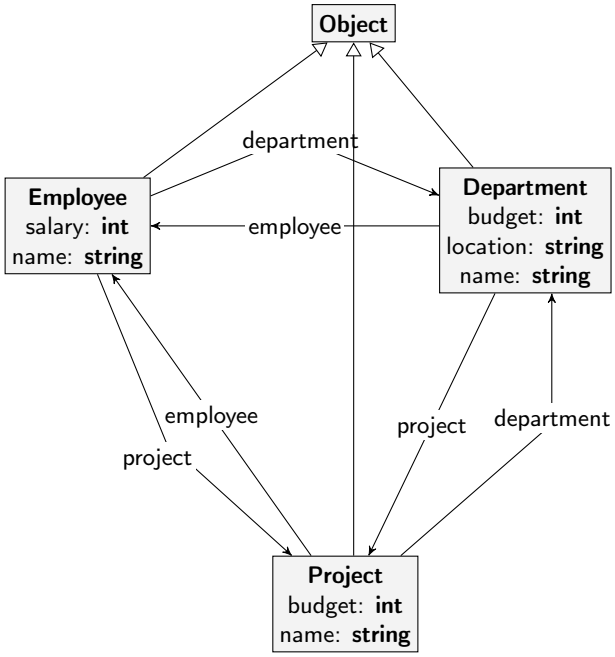


Figure 5.1: GROOVE type graph corresponding to the UML class diagram in Figure 2.3.

5.2 Logic in GROOVE

It is not completely trivial how to encode all expressions in GROOVE. That is why we explain in Section 5.2.1 how to encode the logical operators $\wedge$ and $\vee$ in GROOVE. Section 5.2.2 explains how to encode negations of a lax condition in GROOVE and in Section 5.2.3 we explain how we make use of the type checking functionality in GROOVE. Lastly in Section 5.2.4 we explain why we do not make use of the of the GROOVE definitions of the `min` and `max` operations, but instead we follow the transformation definitions as given in Section 4.4.

5.2.1 AND and OR in GROOVE

As discussed in Section 2.7, GROOVE has a unique way to encode the logical *and* ($\wedge$) and *or* ($\vee$) operators. So it is important to make sure that we get this form also in our lax condition before we transform the lax condition to a GROOVE graph. Figure 5.2 shows how to encode a logical *or* operation, which needs an additional universal quantifier. Without the additional universal quantifier the double existential quantifiers are treated as an $\wedge$ instead of an $\vee$. The same line of thought applies to the $\wedge$, although most $\wedge$ constraints are merged together through the three E1 equivalence rules.

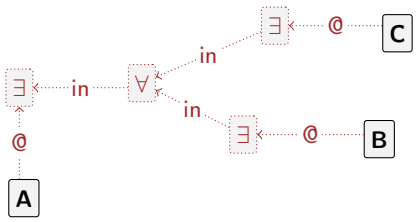


Figure 5.2: The way to encode $\exists(\boxed{\text{A}}, \exists(\boxed{\text{B}}) \vee \exists(\boxed{\text{C}}))$ in GROOVE.

5.2.2 Negations in GROOVE

Negating a lax condition starts with negating the logical dual operators. For these negations we can use the following default negation rules in which a and b are lax conditions:

$$\neg(a \wedge b) \equiv \neg a \vee \neg b$$
$$\neg(a \vee b) \equiv \neg a \wedge \neg b$$
$$\neg(a \implies b) \equiv a \wedge \neg b$$

The remaining lax condition is either of the form $\neg\exists(a)$ or $\neg\forall(a)$. The negation of such a lax condition is defined by Radke et al. as:

$$\neg\exists(a) \equiv \forall(a, false) \qquad\qquad \neg\forall(a) \equiv \exists(a, false)$$

The problem with this definition is that it is impossible in GROOVE to implement the $false$ part. So we have to find a way to mimic this behaviour. First of all we identify the different kind of graphs within a lax condition. Graphs within a lax condition can have the following four forms:

1. It can be a node: $\boxed{\text{v:T}}$.
2. It can be two nodes with an edge in between : $\boxed{\text{v:T}} \xrightarrow{attr} \boxed{\text{u:T‘}}$.
3. It can be two nodes with a comparison operator in between: $\boxed{\text{v:T}} \ \underline{\texttt{op}} \ \boxed{\text{v‘:T}}$.
4. It can be a combination of the above, nodes with an edge (2) and nodes with a comparison operator (3).

To be able to negate a lax condition, it is necessary to be able to negate these four options. Since the fourth option is a combination of two other types we do not have to look into the fourth option. In GROOVE it is pretty easy to negate a node or an edge, just add `not:` and it's negated. In the definitions of negations we will use the GROOVE syntax.

1. $\neg(\boxed{\text{type:T}}) \equiv \boxed{\begin{array}{l}\text{not:}\\ \text{type:T}\end{array}}$
2. $\neg(\boxed{\text{type:T}} \xrightarrow{\texttt{attr}} \boxed{\text{type:T‘}}) \equiv \boxed{\text{type:T}} \xrightarrow{\texttt{not:attr}} \boxed{\text{type:T‘}}$
3. This one is a bit different because depending on the type `T` this one has two implementations within GROOVE.
   (a) `T` $\in \{bool, int, real, string\}$, in this case it is implemented with a production rule in which we can negate the result, this results in the following equality in which we have added one intermediate step in which GROOVE checks the comparison with an boolean:
   $\neg(\boxed{\text{v:T}} \ \underline{\texttt{op}} \ \boxed{\text{v‘:T}}) \equiv \neg(\boxed{\text{v:T}} \ \underline{\texttt{op}} \ \boxed{\text{v‘:T}} = true) \equiv (\boxed{\text{v:T}} \ \underline{\texttt{op}} \ \boxed{\text{v‘:T}} = false)$
   (b) `T` $\notin \{bool, int, real, string\}$ in this case `op` is an equality or an inequality and we can negate the edge, this results in the following equality:
   $\neg(\boxed{\text{v:T}} \ \underline{\texttt{op}} \ \boxed{\text{v‘:T}}) \equiv \boxed{\text{v:T}} \ \underline{\texttt{!op}} \ \boxed{\text{v‘:T}}$

5.2.3 Type checking in GROOVE

In Section 4.4 we have defined quite some transformation rules from OCL to graph theory, 3 of the rules are about type checking or type casting. This are the following three operations: `oclIsKindOf`, `oclIsTypeOf` and `oclAsType`.

By default type checking within GROOVE is implemented as checking whether there exists an injective morphism between the current type and the desired type. This corresponds to the defined transformation rules of `oclIsKindOf` and `oclAsType`. The rule for `oclIsTypeOf` does an additional check to make sure that the type is exactly the type we want and not some kind of subtype. In GROOVE this can be implemented more easily with a sharp type check. In Figure 5.3 an example is given of a sharp type check, to be more specific this is the way to encode `context Person inv: self.authoredPublication.oclIsTypeOf(BookChapter)` in GROOVE. This checks whether the `AuthoredPublication` of a `Person` is a `BookChapter` and not some kind of super or sub-type of `BookChapter`.

Figure 5.3: An example of a sharp type check in GROOVE.

5.2.4 Min and Max constructs in GROOVE

The min and max definitions are quite complex, especially since GROOVE has the possibility to use operators like the `bigmin` and `bigmax`. These operators are optimised in frequently used constructs that are beyond first order logic and are introduced by Rensink in [25]. The usage of the big operations within lax conditions is however even more troublesome because of all the restrictions on the usage. An example of a `bigmin` is given in Figure 5.4 the following restrictions are weighed in the decision and can be found in the given example:

- The resulting int node has to be defined within a universal quantifier.
- The bigmin (bigmax) production rule is not allowed to be within this universal quantifier.
- The bigmin (bigmax) production rule has to be defined in a special universal quantifier ($\boxed{\forall^{>0}}$), which is not discussed in this paper.
- Between these two universal quantifiers it is not allowed to have an additional existential quantifier.

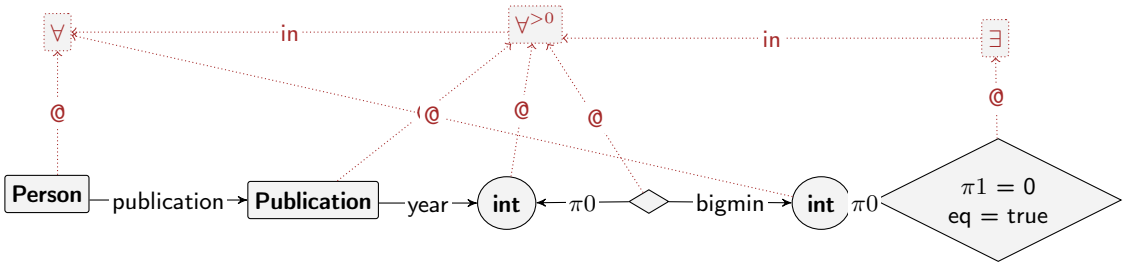


Figure 5.4: The way to encode the `min` operation with a `bigmin` in GROOVE.

5.3 Lax Condition to Graph

To convert a lax condition to a graph in GROOVE, the multiple sub graphs inside each lax condition, have to be merged into one graph including their quantifications. This happens in three steps; first two steps that will transform the lax condition to some kind of normal

form such that the final transformation, transforming the lax condition to one single GROOVE graph, becomes easier. Section 5.3.1 explains the first transformation: renaming node names to prevent node name collisions, this process is called α-conversion in logic. Section 5.3.2 explains the second transformation: adding additional quantifiers to encode $\wedge$ and $\vee$ as discussed in Section 5.2.1. Section 5.3.3 explained the final transformation from the formatted lax condition to a single GROOVE graph.

5.3.1 Node names

This process began to become tricky with implementing `if cond then expr1 else expr2 endif`. As you can see in Section 4.4, this is transformed to $(\texttt{cond} \wedge \texttt{expr1}) \vee (\neg\texttt{cond} \wedge \texttt{expr2})$. So `cond` happens to be there twice, once if it applies and once if the negation applies. The best way of creating the negated condition is cloning the condition and then negating it. Because of the clone step, the node names in the condition and in the negated condition are the same, which is no problem until the lax conditions are merged into one graph. So we have to validate if there are nodes in our lax condition that exist both in the left and in the right side of an operation condition but do not exist in the parents yet. In that case the nodes in the negated condition have to be renamed. This process is known as α-conversion in logic. For example we have the following OCL constraint:

```
context EditedBook inv ifthenelseNotEmpty: if self.conferenceEdition->notEmpty()
then self.year > 0 else self.publicationYear > 0 endif
```

$$\equiv \forall(\boxed{\text{self:EB}}, (\exists(\boxed{\text{v:CE}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}})) \wedge \exists(\boxed{\text{self:EB}} \xrightarrow{year} \boxed{\text{v1:int}} \xrightarrow{\geq} \boxed{\text{0:int}}))$$
$$\vee (\neg\exists(\boxed{\text{v:CE}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}})) \wedge \exists(\boxed{\text{self:EB}} \xrightarrow{publicationYear} \boxed{\text{v1:int}} \xrightarrow{\geq} \boxed{\text{0:int}})))$$

As you can see in both the if as the else branch there is an $\exists(\boxed{\text{v:CE}})$, with in both cases the node name v, which is not declared in its parent yet. Therefore the nodes in the negated condition should be renamed such that both initiated nodes are not mapped to the same node after merging the lax conditions. So after using the validate method this example becomes:

$$\equiv \forall(\boxed{\text{self:EB}}, (\exists(\boxed{\text{v:CE}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}})) \wedge \exists(\boxed{\text{self:EB}} \xrightarrow{year} \boxed{\text{v1:int}} \xrightarrow{\geq} \boxed{\text{0:int}}))$$
$$\vee (\neg\exists(\boxed{\text{v2:CE}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v2:CE}})) \wedge \exists(\boxed{\text{self:EB}} \xrightarrow{publicationYear} \boxed{\text{v3:int}} \xrightarrow{\geq} \boxed{\text{0:int}})))$$

5.3.2 Quantification

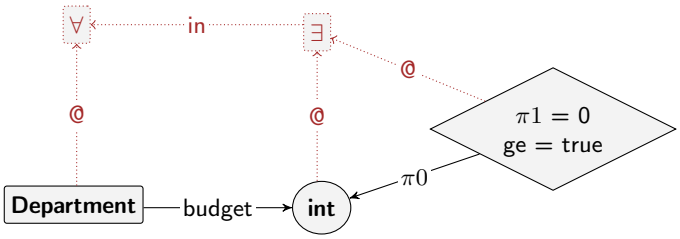
As discussed in Section 5.2.1 the quantification is important in the way to encode the logical operators and ($\wedge$) and or ($\vee$). To make sure that the right quantification is present we validate the lax condition and insert, if necessary, an additional lax condition. For example if we have a lax condition that contains an `OR` statement inside an existential quantification, we insert an additional universal quantified lax condition such that GROOVE will interpret the lax condition in the right way. This results in the following insertion:

$$\exists(\boxed{\text{A}}, \exists(\boxed{\text{B}}) \vee \exists(\boxed{\text{C}})) = \exists(\boxed{\text{A}}, \forall(\emptyset, \exists(\boxed{\text{B}}) \vee \exists(\boxed{\text{C}})))$$

5.3.3 GROOVE graph

In this step we create, from the lax condition, the nested graph condition in GROOVE, which happens in one single graph. Therefore each level of the lax condition has to be merged with its parent and be connected with the already existing nodes. If we take the OCL constraint and

GROOVE graph from Example 9 we see that the lax condition ∀(self:Department, ∃(self:Department $\xrightarrow{budget}$ v:int $\geq$ 0:int)) is equal to the GROOVE graph below.



This GROOVE graph is created by merging lax conditions with their parent. So we start at the most outer lax condition: ∀(self:Department). We add the universal quantifier to the graph and connect the node self with the quantifier. In GROOVE this connection is done by an -edge. The next lax condition is ∃(self:Department $\xrightarrow{budget}$ v:int $\geq$ 0:int). The node self:Department already exists in the universal quantifier, so this node is not created again but the current graph is connected with the already existing graph. Therefore only the budget attribute is created, and connected with the already existing node and the production rule is created. These two are created within an existential quantifier and are connected. Since the node self:Department is already created within a higher level quantifier, this node is not connected with the existential quantifier. Because the existential quantifier exists within the universal quantifier these two quantifiers are also connected, which is done in GROOVE with an in-edge. In this way more complex and bigger lax conditions are also transformed to a single GROOVE graph.

Pseudocode 5.1 gives an idea how the transformation from the formatted lax condition to the single GROOVE graph works. The transformation starts with an empty graph in which the GROOVE graph is built. With `OperatorCondition` we mean conditions that are concatenated with an operation such as ∧ and ∨.

```
function laxToGraph(Graph G, LaxCondition L) {
    GNodes = G.nodes
    G = G ∪ L.graph
    quantifier = G.addNode(L.quantifier)

    for (node : L.graph.nodes) {
        if(node ∉ GNodes) {
            G.addEdge(node, "@", quantifier)
        }
    }
    G.addEdge(quantifier, "in", prevQuantifier)

    laxToGraph(L.condition)
}

function laxToGraph(Graph G, OperatorCondition L) {
    laxToGraph(G, L.expr1)
    laxToGraph(G, L.expr2)
}
```

Pseudocode 5.1: The way that a lax condition is transformed to a single GROOVE graph.

Chapter 6

Implementation

This chapter explains the current implementation. Section 6.1 explains how and why SableCC is used to create a Concrete Syntax Tree given an OCL constraint based on the OCL 1.5 grammar definition. Section 6.2 discusses limitations of our implementation. Section 6.3 shows how many lines of code GROOVE had at the start of the graduation project and how many lines were added during the project. Section 6.4 shows how we have integrated the OCL transformation rules with the existing GROOVE interface

6.1 OCL To Syntax Tree

There are several OCL parsers around that are able to parse OCL. There are two requirements for our OCL parser. 1) The concrete syntax tree (CST) or abstract syntax tree (AST) has to be accessible so that the transformation rules can be applied. 2) The parser is able to parse OCL constraints based only on the OCL grammar and without requiring a UML class diagram, we are going to use the GROOVE type graph for this purpose, this is explained in Section 5.1.

First we take a look at three different tools that can parse OCL; the famous Dresden OCL Toolkit in Section 6.1.1, Octopus in Section 6.1.2 and the Eclipse OCL Toolkit in Section 6.1.3. Because these tools do not meet the defined requirements we look into SableCC in Section 6.1.4. Section 6.1.5 explains how the OCL parser is generated. Lastly, Section 6.1.6 discusses the new features of OCL 2.0 that we do not support because our parser supports OCL 1.5.

6.1.1 Dresden OCL Toolkit

The Dresden University of Technology has developed multiple tools for OCL, which they bundled into the Dresden OCL Toolkit [26]. One tool in particular is interesting, the OCL 2.0 parser. In [27], Demuth et al. describe that OCL 2.0 is an L-attribute grammar. Which means that attribute evaluation is performed by letting a tree walker perform a single-sweep, depth-first, left-to-right tree walk on the syntax tree of an OCL constraint. In the paper they also describe how they have built their OCL 2.0 parser with an extended version of the SableCC parser generator. Sadly we did not manage to get the Dresden OCL toolkit working in Eclipse by following their tutorial. Neither was the extended version of the SableCC parser generator available.

6.1.2 Octopus

Another OCL tool that is able to parse OCL is Octopus [28]. Octopus is one of the tools that is able to transform a UML + OCL model into platform dependent source code, for example Java code. The syntax tree is not accessible in Octopus, therefore we did not look further into Octopus.

6.1.3 Eclipse OCL Toolkit

Eclipse also delivers their own OCL toolkit such that you can parse OCL expressions given the context. This will only work if you use an ecore file in which you can define the class diagram and generate the corresponding Java code. The ecore file has the same function as our type graph has, as discussed in Section 5.1. In this way the context can be found within the generated Java code and Eclipse can parse the OCL constraint. Therefore the Eclipse OCL toolkit [29] is not able to generate a syntax tree from only the OCL constraint given the OCL grammar.

6.1.4 Parser generator - SableCC

Given the three tools, we were not able to generate a syntax tree of an OCL constraint without a complete class diagram. Besides these three tools we did not manage to find other OCL tools that are able to parse OCL. The alternative to using an existing tool is using a parser generator, which is able to generate the source code of a parser, given a defined grammar. More information about parsing is given in Section 2.8.

In Appendix A of [19], Fadi Chabarek has defined the OCL 1.5 grammar for SableCC, one of the parser generators, which is able to create a parser in Java such that the CST of any OCL constraint is accessible without the need of the corresponding class diagram. An example of the result of parsing OCL constraints with the given OCL grammar for SableCC is given in Example 10.

The OCL 1.5 grammar definition in SableCC is used for our graduation project. Example 10 shows how the CST looks like and is therefore accessible. Besides the parser that SableCC generates, it generates also a tree walker that can be extended. By extending the tree walker we have been able to apply our transformation rules and create nested graph constraints.

Example 10. An example in which the OCL constraint: `context Deparment inv: self.budget >= 0` gets parsed by the given OCL 1.5 grammar for SableCC. This results in the following CST:

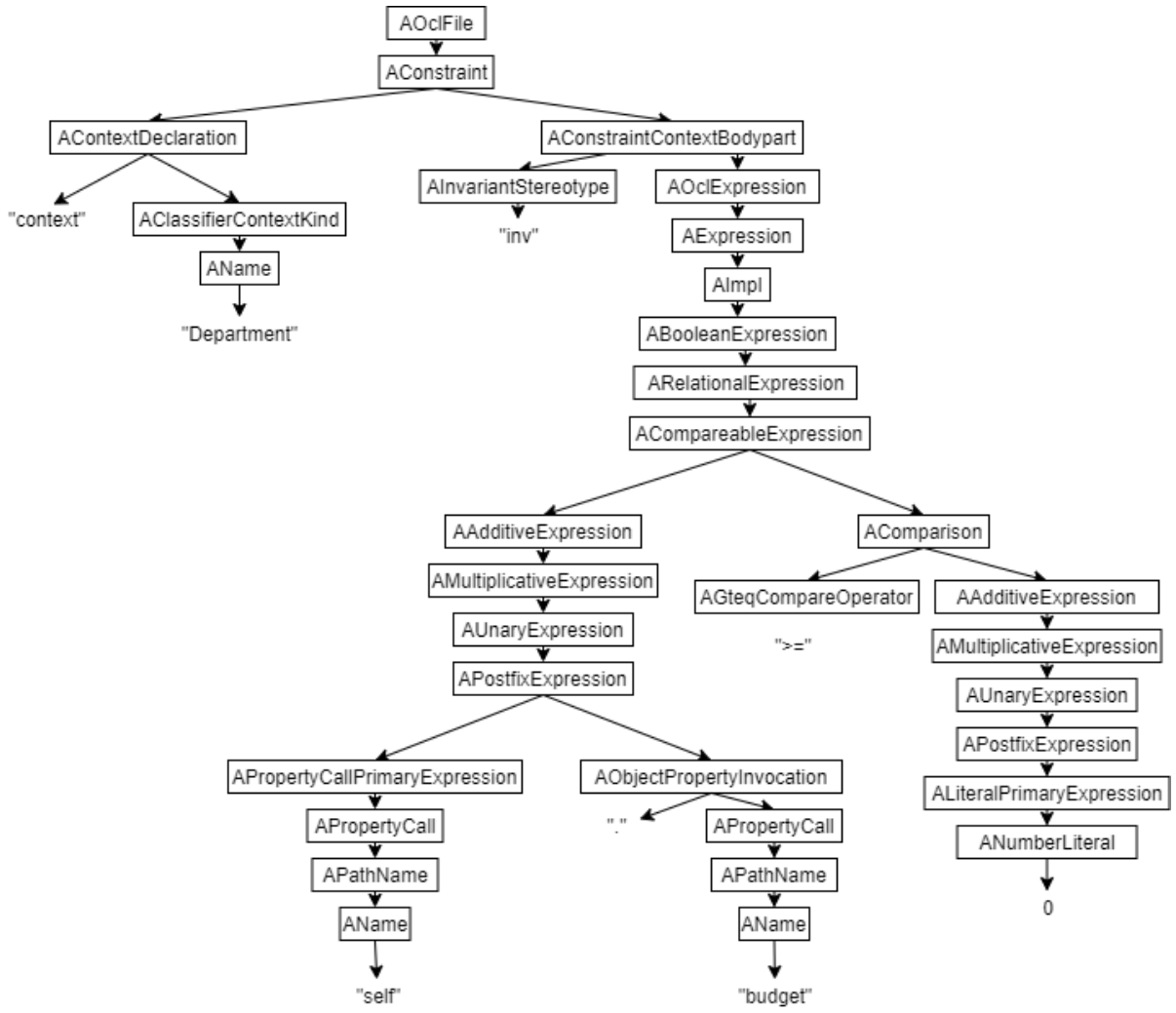


Figure 6.1: The CST of the OCL constraint: `context Department inv: self.budget >= 0`.

6.1.5 Generate OCL parser

In Appendix A of [19], Fadi Chabarek has defined the OCL 1.5 grammar for SableCC. To generate the Java code you have to run the following command: `java -jar sablecc.jar ocl15.grammar`, in which `sablecc.jar` is part of the downloadable SableCC tool and `ocl15.grammar` is the name of the defined OCL grammar. The OCL grammar can be downloaded from [30]. We prefer to refer to Appendix A of [19], because this grammar definition is easier to read.

We have made a small change to the OCL1.5 grammar definition for SableCC. In this definition an OCL constraint is obligated to start with `package ... endpackage`. We have removed this part because it had no value for us and in GROOVE you want to use OCL constraint of the format: `context C inv: expr` as defined in the transformation rules in Section 4.4.

By default SableCC generates a concrete syntax tree (CST) and not an abstract syntax tree (AST). It is possible to create an AST by walking through the CST, but since you are walking through the CST already we have chosen to start the transformation process immediately from

the CST. Otherwise you need to walk through the CST once to create the AST and then walk through the AST to apply the transformation. Now you only walk once through the CST and the transformation is applied immediately.

SableCC generates next to a parser also a tree walker, which makes use of a variant of the well known visitor design pattern. You can compare this with the well known switch statement, but more efficient. By walking the tree you have to check at every node what type of node it is, with a switch statement you have to compare the type with every existing node type which is in the worst case of O(n). With the visitor design pattern this can be reduced to O(1). The visitor mechanism is explained thoroughly in chapter 6 of the SableCC documentation [18].

6.1.6 OCL 2.0 features

We are using the available OCL 1.5 grammar for SableCC, as discussed in the previous section. Nowadays we are at OCL 2.4, so what features do we miss in our parser? First of all it is worth noting that the parser did not lack any features to parse the OCL_{FO} fragment. In [31], Hussmann et al. discuss the four major features of OCL 2.0.

1. **Metamodel**
   The concepts and the relationships have been expressed in a MOF metamodel, UML is also expressed in a MOF metamodel. This results in a better integration between OCL and UML. Older versions of OCL do not have this MOF metamodel.
2. **Query language**
   OCL 1.5 is a constraint language, from OCL 2.0 and onwards OCL has become a query language. To facilitate as a query language OCL 2.0 has introduced the tuple type to combine multiple values into one tuple value.
3. **Formal semantics**
   OCL 2.0 has defined and standardized semantics, which is necessary to create formal constraints for UML.
4. **OclMessages**
   OCL 2.0 introduces the concept of message expressions. These can be used to state that a certain message has been sent from a classifier during a certain period of time.

6.2 Limitations

Our implementation has some limitations, this section gives a summary of each limitation. In this way it is clear why these limitations exists and if it may be possible to remove the limitation in the future.

6.2.1 Mandatory type definition

Since it is difficult to determine the type of a variable inside a `exists` or `forall` operation, we have added the limitation that the type of the variable has to be given. For example if we have the following OCL constraint: `self.publication->exists(p:Publication | p.title = 'test')`. The `:Publication` part is mandatory, because this is the way to define the type of the variable. This syntax is part of the official OCL syntax, although the type definition is not mandatory within this syntax [3]. It is possible to remove this limitation with an additional parsing such that the type is known within these operations.

6.2.2 Negation limitations

The negation implementation, as discussed in Section 5.2.2, has some limitations. For example building further on a negated node is not possible. So for example the OCL constraint below is not possible, although this can be achieved by writing down the constraint differently, but simplifying OCL constraints is another research area. Building further on a negated node is not possible in GROOVE because it is not possible to encode the $false$ in a lax condition.

```
context Person inv selectIsEmpty: self.editedBook->select(e:EditedBook |
 e.bookSection->notEmpty())->isEmpty().
```

6.3 Statistics

The source code of GROOVE consisted of 231.911 lines of code at the beginning of the graduation project. At the end this was increased to 234.555 lines of code, this is an increase of 2654 lines of code. These lines of code are measured in the Java files with the IntelliJ plugin Statistic [32].

We have also the statistics available about the generated OCL parser and the newly created test package. The generated OCL parser consists of 29.691 lines of code and the test package consists of 1.862 lines of code.

6.4 Integration in GROOVE

The class `groove.gui.action.NewAction` is responsible for creating and executing different kind of actions within GROOVE. So in this class the connection between our OCL transformation and the already existing GROOVE implementation is created. This resulted in the action "New OCL" under edit, as shown in Figure 6.2. This button results in a window where you can insert your OCL constraint, as shown in Figure 6.3. If you click on the OK button this will transform the OCL invariant and generate the corresponding graph invariant. The corresponding graph invariant can be found in Figure 6.4.

Given the graph invariant we can create models to verify. Figure 6.5 shows a model that does not satisfy the *positiveSalary* constraint with the corresponding indication of GROOVE that the graph fails to satisfy the graph invariant. Figure 6.6 shows a model that does satisfy the *positiveSalary* constraint with the corresponding indication of GROOVE that the graph matches the *positiveSalary* constraint, and thus the graph satisfies the constraint.

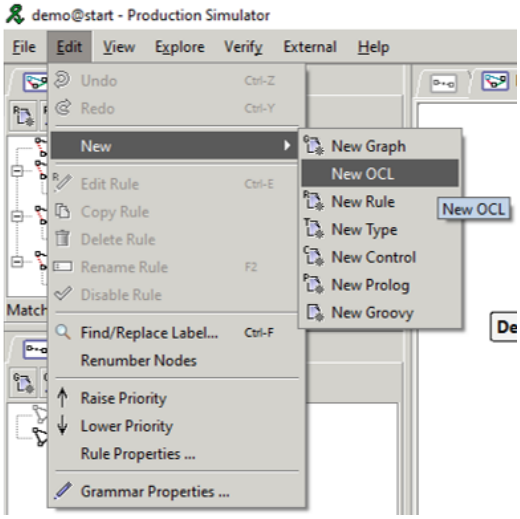


Figure 6.2: The location to create a new OCL constraint in GROOVE.

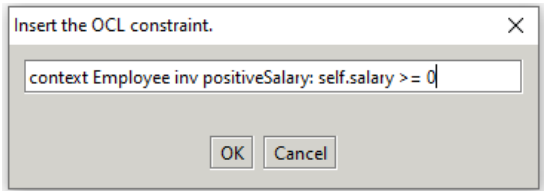


Figure 6.3: The way to create a new OCL constraint in GROOVE.

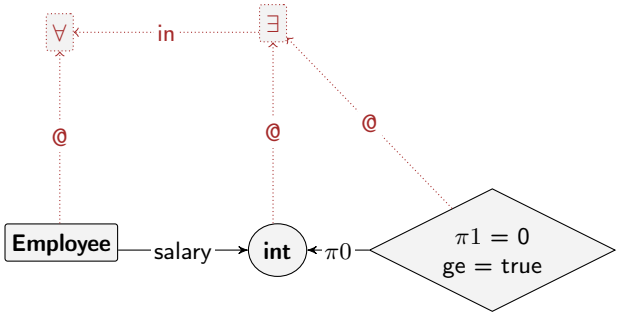


Figure 6.4: The positiveSalary invariant in GROOVE.

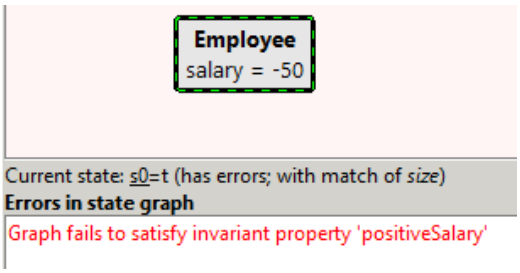


Figure 6.5: The location to create a new OCL constraint in GROOVE.

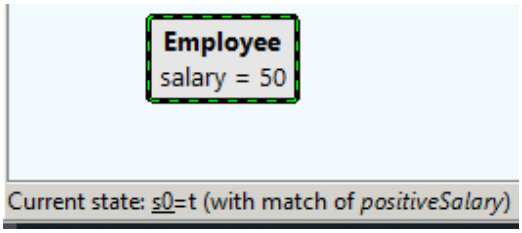


Figure 6.6: The way to create a new OCL constraint in GROOVE.

Chapter 7

Validation

In this chapter we describe the validation of our implementation in GROOVE. We show that our implementation does what it should do by using case studies. Two case studies have been rebuilt in both Eclipse OCL and in our implementation of GROOVE. In this way, we can check if our implementation accepts and rejects the same models as the existing Eclipse OCL and validate if our implementation enforces the constraints correctly. If the implementation works as desired, there should be no difference between the accepted and rejected models of Eclipse OCL and our implementation.

In Section 7.1, we validate our implementation based on the USE case study [4]. This case study contains a small class diagram and some basic OCL constraints. In Section 7.2, we validate our implementation based on the DBLP case study [7]. This case study is based on a bigger class diagram and also contains some set operations. In Section 7.3, we show that we support many more operations that are not part of one of these case studies. Lastly in Section 7.4, we give a summary of the two case studies and the coverage section.

7.1 USE case study

The USE case study is based on a small class diagram with three classes. This class diagram can be found in Figure 7.1.

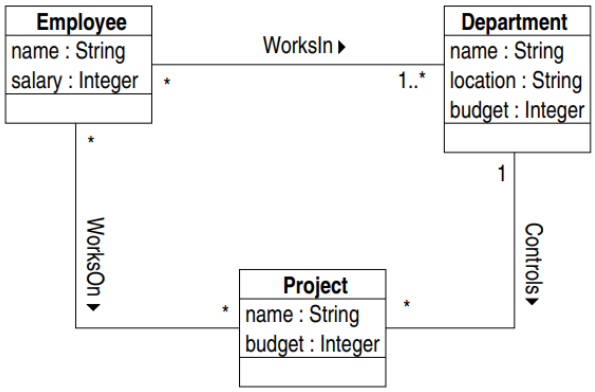
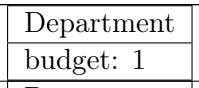
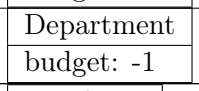
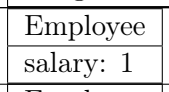
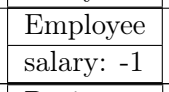
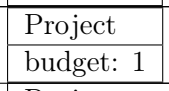
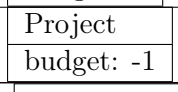
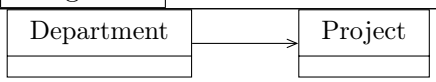


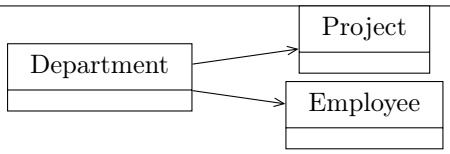
Figure 7.1: The USE case study class diagram [4].

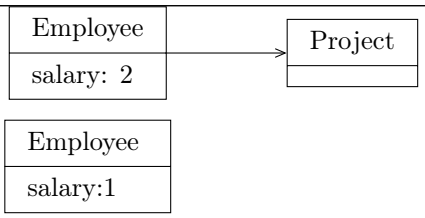
In the table below we show for every OCL constraint documented in the case study, our input

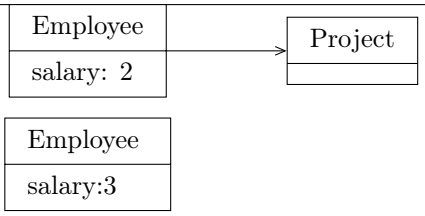
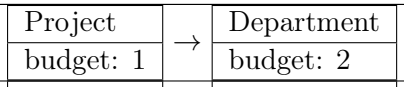
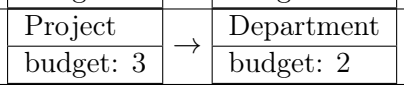
model, the result of Eclipse and the result of our GROOVE implementation. There are, in total, four results that do not match. This are test cases 5, 6, 7 and 8, that are based on two OCL constraints. Both these constraints compare the size of a set with the size of another set. This is one of the limitations of the transformation to graph theory, it is only possible to compare the size with a constant and so these OCL constraints are not supported and thus it is not possible to validate these constraints with our GROOVE implementation. The other test cases did succeed and so we could handle 5 out of the 7 OCL constraints defined by the USE case study.

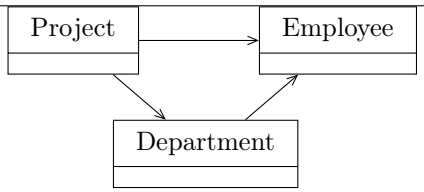
OCL constraint	#	**input**	**Eclipse**	**GROOVE**
`context Department inv i1a:` `self.budget >= 0`	1		`True`	`True`
	2		`False`	`False`
`context Employee inv i1b:` `self.salary >= 0`	3		`True`	`True`
	4		`False`	`False`
`context Project inv i1c:` `self.budget >= 0`	5		`True`	`True`
	6		`False`	`False`

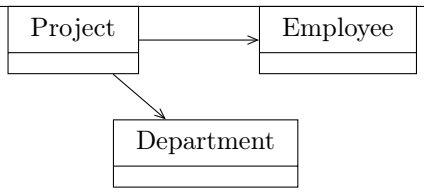
`context Department inv i2:`
`self.employee->size() >=`
`self.project->size()`
	7		`False`	-

	8		`True`	-

`context Employee inv i3:`
`Employee.allInstances()->forAll(e1, e2 |`
`e1.project->size() > e2.project->size()`
`implies e1.salary > e2.salary)`
	9		`True`	-

	10		`False`	-
`context Project inv i4:` `self.budget <= self.department.budget`	11		`True`	`True`
	12		`False`	`False`

`context Project inv i5:`
`self.department.employee`
`->includesAll(self.employee)`
	13		`True`	`True`

	14		`False`	`False`

7.2 DBLP case study

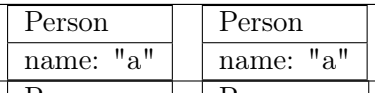
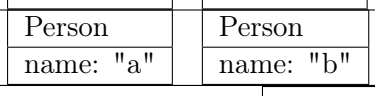
The DBLP case study is based on an extensive class diagram that can be found in Appendix E. In the table below we show for every OCL constraint documented in the case study, our input model, the result of Eclipse and the result of our GROOVE implementation. We had to make a small change in one of the OCL constraints, because of the limitation defined in Section 6.2.1. So we have added the type of the variables and have replaced the OCL constraint:

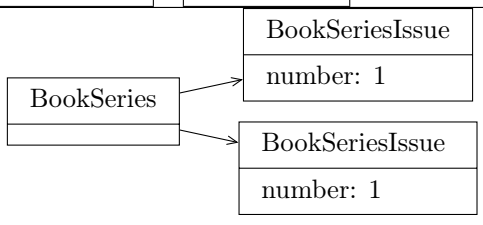
```
context EditedBook inv correctPagination: self.bookChapter->forAll(c1, c2
| c1 <> c2 implies c1.iniPage > c2.endPage or c2.iniPage > c1.endPage)
```

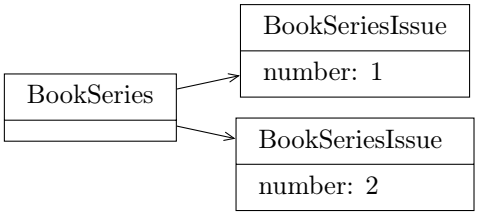
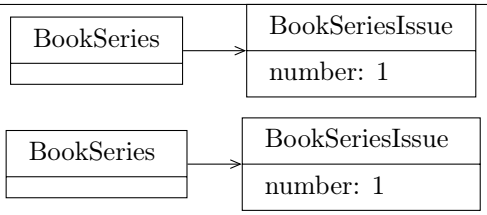
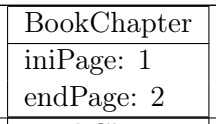
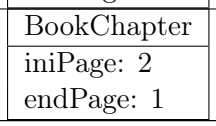
with the following OCL constraint:

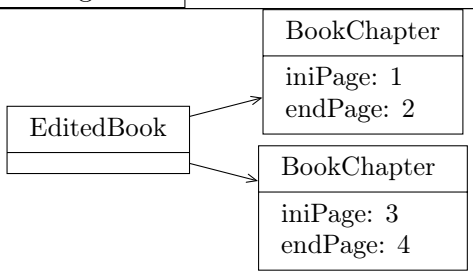
```
context EditedBook inv correctPagination: self.bookChapter->forAll(c1, c2:BookChapter
| c1 <> c2 implies c1.iniPage > c2.endPage or c2.iniPage > c1.endPage)
```

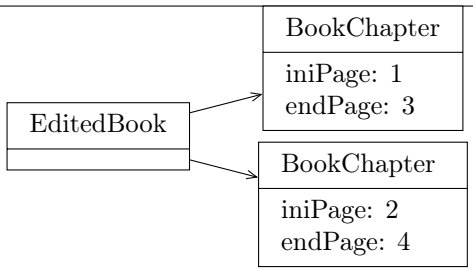
There are, in total, four results that do not match. This are test cases 24 and 25, that are based on one OCL constraint. This constraint is based on the operation `sortedBy`. In our implementation we do not have some kind of order, as explained in Section 4.2, therefore it is not possible to sort elements and thus it is not possible to validate this constraint with our GROOVE implementation. The other test cases did succeed and so we could handle 7 out of the 8 unique OCL constraints defined by the DBLP case study.

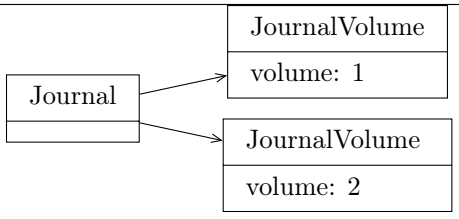
OCL constraint	#	**input**	**Eclipse**	**GROOVE**
`context Person inv nameIsKey:` `Person.allInstances()->isUnique(name)`	15		`False`	`False`
	16		`True`	`True`

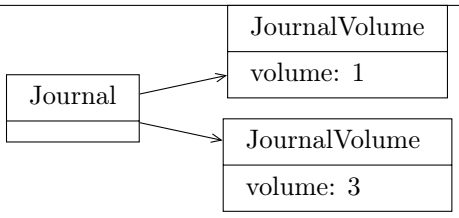
`context BookSeries inv BookSeries`
`AndNumberIdentifyBookSeriesIssue:`
`self.bookSeriesIssue->isUnique(number)`
	17		`False`	`False`

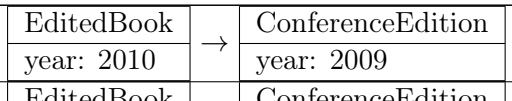
	18		`True`	`True`
	19		`True`	`True`
`context BookChapter inv correctPage:` `self.iniPage <= self.endPage`	20		`True`	`True`
	21		`False`	`False`

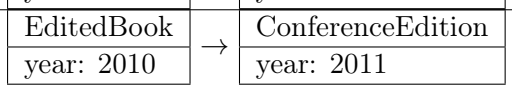
`context EditedBook inv correctPagination:`
`self.bookChapter->forAll(`
`c1, c2:BookChapter | c1 <> c2`
`implies c1.iniPage > c2.endPage`
`or c2.iniPage > c1.endPage)`
	22		`True`	`True`

	23		`False`	`False`

`context Journal inv consecutiveVolumes:`
`self.journalVolume->sortedBy(volume)`
`.volume =`
`Sequence{1..self.journalVolume->size()}`
	24		`True`	-

	25		`False`	-

`context EditedBook inv compatibleYear:`
`self.conferenceEdition->notEmpty()`
`implies self.year`
`>= self.conferenceEdition.year`
	26		`True`	`True`

	27		`False`	`False`

	28	EditedBook year: 2010	True	True

`context ConferenceEdition inv published:`
`self.editedBook->notEmpty()`
`or self.bookSeriesIssue->notEmpty()`
`or self.journalIssue->notEmpty()`
	29	ConferenceEdition	False	False

	30	ConferenceEdition → EditedBook	True	True
	31	ConferenceEdition → BookSeriesIssue	True	True
	32	ConferenceEdition → JournalIssue	True	True

`context Book inv theSamePublisher:`
`if self.oclIsTypeOf(BookSeriesIssue)`
`then self.publisher =`
`self.oclAsType(BookSeriesIssue)`
`.bookSeries.publisher`
`else true endif`
	33	BookSeriesIssue
		publisher: "a" → BookSeries
		publisher:"a"
			True	True

	34	BookSeriesIssue publisher: "a" → BookSeries publisher:"b"	False	False
	35	BookSeriesIssue publisher: "a"	False	False
	36	EditedBook publisher: "a"	True	True

7.3 Coverage

In this section we give an overview of additional supported OCL constraints to give a broader coverage. These OCL constraints are based on the DBLP case study class diagram, which can be found in Appendix E.

There are, in total, two results that do not match. This are test cases 61 and 73. Test case 61 is a special case of `including` and test case 73 is a special case of `excluding`. If we take test case 61 for example, the input of this case is Person and the OCL constraint is:

```
context Person inv including:
self.editedBook->including(self.authoredPublication)->notEmpty()
```

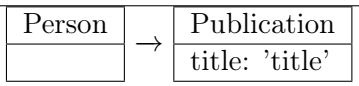
Eclipse said this was *true* and our implementation of GROOVE said this was *false*. Since the person does not have either an editedbook nor an authoredpublication, we believe that the result is an empty set and thus *false*. Given this reasoning we have accepted the difference in result between Eclipse and our implementation.

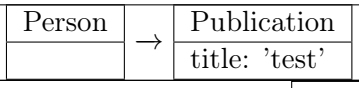
The same line of thought can be applied to test case 73, but then the opposite.

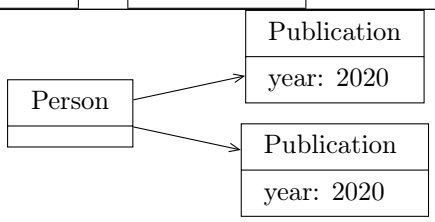
OCL constraint	#	**input**	**Eclipse**	**GROOVE**

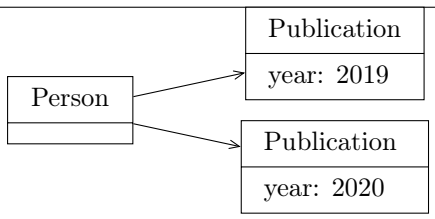
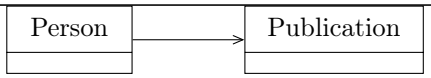
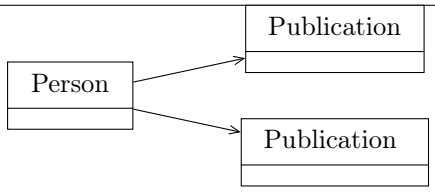
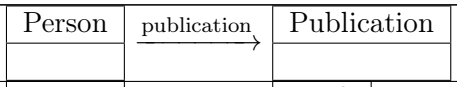
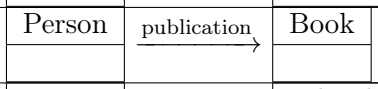
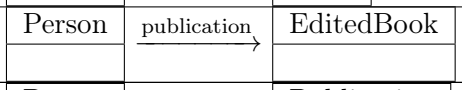
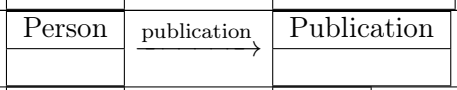
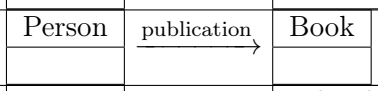
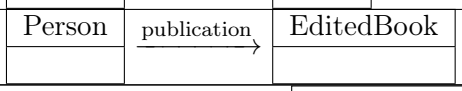
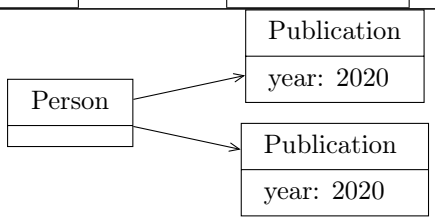
`context ConferenceEdition inv xor:`
`self.editedBook->notEmpty()`
`xor self.journalIssue->notEmpty()`
	37	ConferenceEdition	False	False

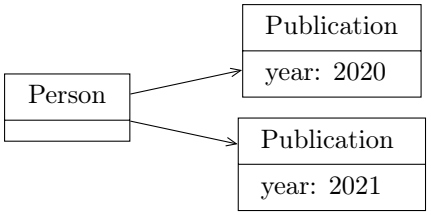
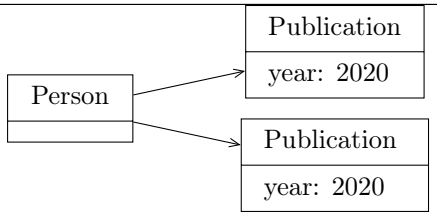
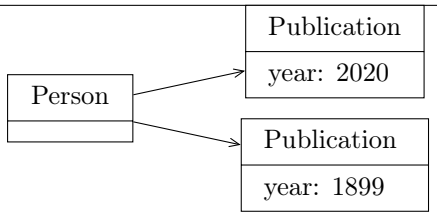
	38	ConferenceEdition → EditedBook	True	True
	39	ConferenceEdition → JournalIssue	True	True
	40	ConferenceEdition → EditedBook ConferenceEdition → JournalIssue	False	False

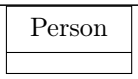
`context Person inv exists:`
`self.publication->exists(p:Publication |`
`p.title = 'test')`
	41		`False`	`False`

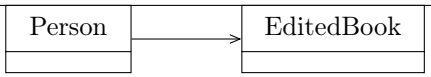
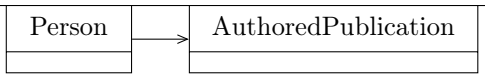
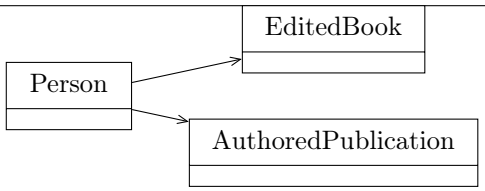
	42		`True`	`True`

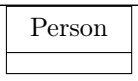
`context Person inv forAll:`
`self.publication->forAll(`
`p1,p2:Publication | p1.year = p2.year)`
	43		`True`	`True`

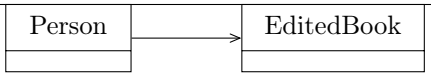
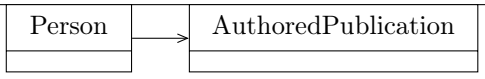
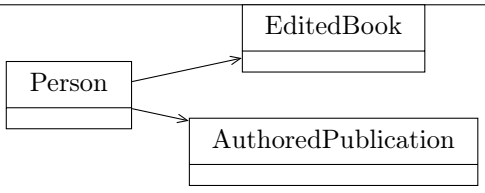
	44		`False`	`False`
`context Person inv size:` `self.publication->size() >= 2`	45		`False`	`False`
	46		`True`	`True`
`context Person inv oclIsKindOf:` `self.publication.oclIsKindOf(Book)`	47		`False`	`False`
	48		`True`	`True`
	49		`True`	`True`
`context Person inv oclIsTypeOf:` `self.publication.oclIsTypeOf(Book)`	50		`False`	`False`
	51		`True`	`True`
	52		`False`	`False`
`context Person inv max:` `self.publication.year->max() <= 2020`	53		`True`	`True`

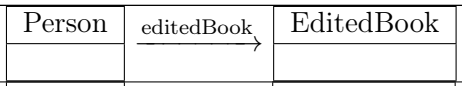
	54		`False`	`False`
`context Person inv min:` `self.publication.year->min() >= 1900`	55		`True`	`True`
	56		`False`	`False`

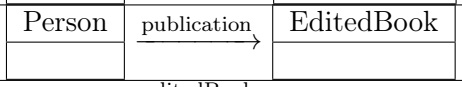
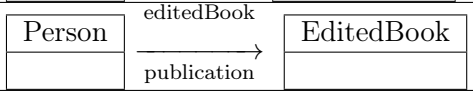
`context Person inv union:`
`self.editedBook->union(`
`  self.authoredPublication)->notEmpty()`
	57		`False`	`False`

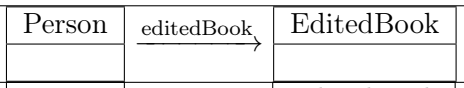
	58		`True`	`True`
	59		`True`	`True`
	60		`True`	`True`

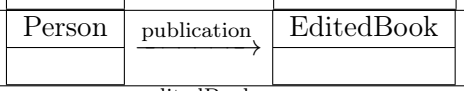
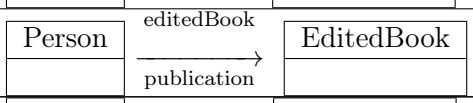
`context Person inv including:`
`self.editedBook->including(`
`  self.authoredPublication)->notEmpty()`
	61		`True`	`False`

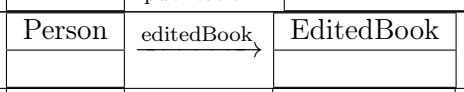
	62		`True`	`True`
	63		`True`	`True`
	64		`True`	`True`

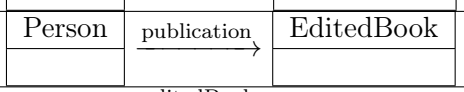
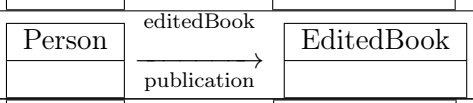
`context Person inv intersection:`
`self.editedBook->intersection(`
`  self.publication)->notEmpty()`
	65		`False`	`False`

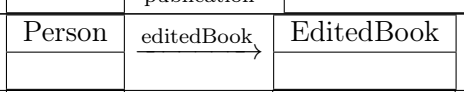
	66		`False`	`False`
	67		`True`	`True`

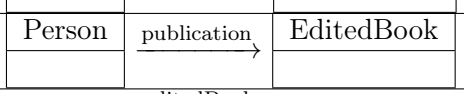
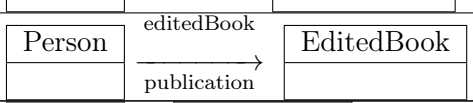
`context Person inv minus:`
`(self.editedBook - self.publication)`
`->notEmpty()`
	68		`True`	`True`

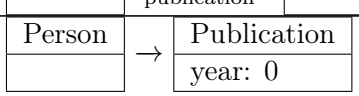
	69		`False`	`False`
	70		`False`	`False`

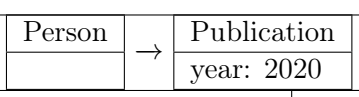
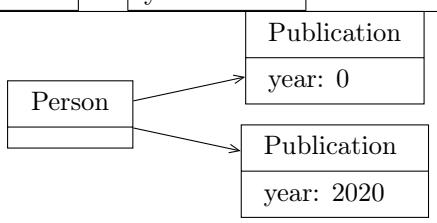
`context Person inv excluding:`
`self.editedBook->excluding(`
`self.publication)->notEmpty()`
	71		`True`	`True`

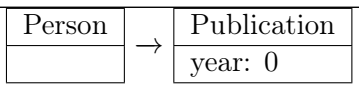
	72		`False`	`False`
	73		`True`	`False`

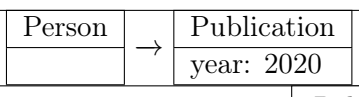
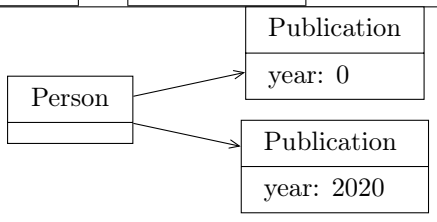
`context Person inv symmetricDifference:`
`self.editedBook->symmetricDifference(`
`self.publication)->notEmpty()`
	74		`True`	`True`

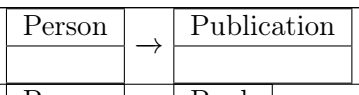
	75		`True`	`True`
	76		`False`	`False`

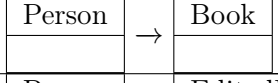
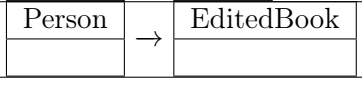
`context Person inv select:`
`self.publication->select(p:Publication |`
`p.year > 0)->notEmpty()`
	77		`False`	`False`

	78		`True`	`True`
	79		`True`	`True`

`context Person inv reject:`
`self.publication->reject(p:Publication |`
`p.year > 0)->notEmpty()`
	80		`True`	`True`

	81		`False`	`False`
	82		`True`	`True`

`context Person inv selectByKind:`
`self.publication->selectByKind(Book)`
`->notEmpty()`
	83		`False`	`False`

	84		`True`	`True`
	85		`True`	`True`

`context Person inv selectByType:`
`self.publication->selectByType(Book)`
`->notEmpty()`
	86	Person → Publication	`False`	`False`

	87	Person → Book	`True`	`True`
	88	Person → EditedBook	`False`	`False`

7.4 Summary

In this chapter we have described the validation of our implementation in GROOVE on the basis of two case studies and a coverage section. Table 7.4 gives the summary of our validation in the sense of amount of constraints we could parse and the amount of test cases in which Eclipse OCL and our implementation had the same result.

Section	**OCL constraints**	**Parsed**	**Test cases**	**Same result as Eclipse**
USE case study	7	5	10	10
DBLP case study	8	7	20	20
Coverage	18	18	52	50

Table 7.4: Summary of the two case studies and the coverage section.

As we can see in Table 7.4, we could not parse three OCL constraints. These constraints were not part of OCL_{FO} as defined in Section 4.1, and thus this was expected given the OCL constraints. We have two test cases of which our result was different than the result of Eclipse OCL. We have accepted these differences and have explained this in Section 7.3.

We have not dived into the performance of our implementation. But we know that the transformation of OCL constraints to a graph constraint is almost instantly for the constraints we have used in our validation process. After the transformation we have a graph constraint that has the same performance as GROOVE had before our implementation. In [33], Zambon et al. have documented the performance of GROOVE on the N-queens problem, with N ranging from 2 to 9. We can assume that our graph invariants have the same performance. The performance of the N-queens problem in GROOVE can be found in Figure 7.2.

N	State space size		Time	Distinct
	States	Transitions	(s)	solutions
2	13	12	< 1	0
3	36	43	< 1	0
4	123	220	< 1	2
5	578	1,411	1	10
6	3,331	9,676	3	4
7	20,594	72,427	15	40
8	139,345	585,256	131	92
9	1,030,642	5,098,357	1,241	352

Figure 7.2: The performance of GROOVE for the N-queens problem [33].

Chapter 8

Final Remarks

Our goal was to represent OCL constraints in GROOVE models, such that OCL constraints are enforced throughout the state space of a software system. In order to do this, we have defined 53 transformation rules that transform OCL to nested graph constraints and 8 equivalence rules to simplify the transformed nested graph constraint.

We have used the work of Radke et al. in [11] as starting point. From this point, we have improved and extended their definitions and added new transformation rules. Therefore our implementation supports a broader set of OCL operations. We have shown that our definitions have the desired result with examples in which we rewrite OCL constraints to nested graph constraints with the new transformation rules.

Our transformation rules result in nested graph constraints, which consist of multiple nested graphs. This is not the format that GROOVE supports. GROOVE has its own way to define a nested graph constraint, namely in a single graph. So we had to transform a nested graph constraint, with multiple graphs, to a single graph such that GROOVE can handle the nested graph constraint. We have explained this transformation in Chapter 5.

Given the transformation and equivalence rules and the knowledge of how to represent a nested graph constraint in GROOVE, we have implemented OCL support in GROOVE. The current implementation has two limitations and sadly we did not manage to implement all the defined transformation rules. The limitations are described in Section 8.1 and the missing implementations in Section 8.2.

In Chapter 7 we have demonstrated that our implementation accept the intended valid models. We have rebuilt two case studies in both Eclipse OCL and in our implementation of GROOVE. In both systems, we have created the same instance models to determine if the instance models do satisfy the OCL constraints. Those instance models are created manually, but it may be better to be able to transform instance models from Eclipse OCL to GROOVE. In this way it is ensured that the same instance model is used in both Eclipse OCL and in our implementation of GROOVE.

8.1 Limitations

Our current implementation has the following limitations:

- Currently it is obligated that the type of a variable in, for example, `forall` is given. It is currently too difficult to determine its type during the transformation of the parse tree.

It is possible to solve this issue by adding a round of parsing in which the type of every expression is added to the nodes.

- In GROOVE it is impossible to create a nested graph constraint in which only a part of the constraint is negated. Therefore it is not possible to build further on a negated part of a condition. The best way to solve this is making it possible in GROOVE to encode that a nested graph constraint can be $false$.

8.2 Missing Implementations

We did not manage to implement all the transformation rules that are defined in Section 4.4, the missing transformation rules are:

- Transformation rule 12: `expr1 eq expr2`, where the type of `expr1` and `expr2` is a `Set`.
  The difficulty of implementing this operation lays in the fact to know whether you have to apply the set equality (rule 12) or a value equality (rule 13). This is possible by using multiplicities, but to take multiplicities into account of determining the type means that we have to change one of the fundamental parts of our implementation, which is being able to determine the type of an expression.
- Transformation rule 30: `expr1->one(v:T | expr2)`.
  The difficulty of implementing this operation lays in the fact how negations work in GROOVE. It is common with this operation to build further on a negated part of a nested graph constraint, which is not possible in GROOVE. So to be able to implement this operation, it has to be possible to build further upon a negated nested graph constraint.

8.3 Future Work

This thesis has defined a way to represent OCL constraints in GROOVE models. Due to the limited time we had the following remains for future work:

- We have improved and extended the definitions of Radke et al. in [11]. We have shown that our definitions have the desired result, but a formal proof is missing, so creating formal proofs to show that our definitions are completely correct remains for future work.
- We have added the limitation that it is obligated to define the type of variables in, for example, `forall`. It is possible to remove this limitation by addin a round of parsing where you add the types to the syntax tree. In that case you know the types of every variable and sub expression at the moment you apply the transformation rules. In the current implementation you have to determine the type at the current point in the syntax tree. And thus this will make the application of the transformation rules easier. Adding the round of parsing to determine the types remains for future work.
- We have explained the limitation of our current implementation about negations, since we had no way to encode that a nested graph constraint is *false*. We have found two solution directions; 1) implement a way to let a nested graph constraint be *false* in GROOVE, or 2) we may have found a way to encode a *false* nested graph constraint in GROOVE by applying the following rule in which `False` is a type defined in the type graph that will not appear in any instance graph. Therefore the existential part will always evaluate to *false*. The `False` type is specifically defined to be able to encode the *false* nested graph constraint.

$$\exists(C, false) \stackrel{?}{\equiv} \forall(C, \exists(\boxed{\text{False}}))$$

Section 6.2.2 gives an OCL constraint that could not be encoded in GROOVE due to the negation limitation, this is the `selectIsEmpty` constraint that can also be found below. Figure 8.1 seems to give the way to encode the *false* nested graph constraint by applying the rule above. This may be a solution direction to solve the negation limitation, due to the limited time this remains for future work.

```
context Person inv selectIsEmpty: self.editedBook->select(e:EditedBook |
 e.bookSection->notEmpty())->isEmpty().
```

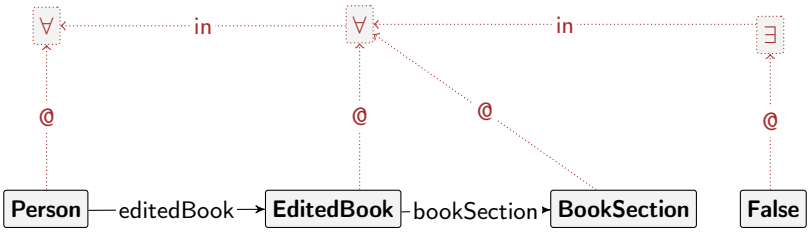


Figure 8.1: A way to encode a *false* nested graph constraint.

- We did not manage to implement all the defined transformation rules, as discussed in Section 8.2. The two missing transformation rules remain for future work.
- We have created the instance models in both Eclipse OCL and GROOVE manually for the validation. It may be better to be able to translate instance models from Eclipse OCL to GROOVE, such that it is ensured that the same instance model is used in both Eclipse OCL and in GROOVE. Since this translation is not available yet, this is potential future work.

8.4 Evaluation

At the beginning of my graduation project I had not much experience with parser generators and their tree walkers. With the current knowledge it is possible to increase the efficiency of my implementation. The most profit can be made by preprocessing the tree before applying the transformation rules. In the current implementation the transformation rules are applied immediately on the generated parse tree. In some cases additional tree walking is needed, for example if we transform the following OCL constraint:

```
context Department inv: self.employee->size() > 2
```

The syntax tree of this constraint will look like the tree we showed earlier in Figure 6.1, and thus at the node where the `size` operation is defined we do not have the compare operator and the constant value available. So it is not possible to transform the `size` operation at that node. Currently this is fixed by determining if we need to apply the `size` transformation rule at the `ACompareableExpression`, because we do have all the necessary information available at that node. With preprocessing we could add the necessary information to the `size` node such that the `size` transformation rule can be applied at its node.

Bibliography

[1] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, p. 9–36, Mar. 1976.

[2] Object Management Group, "About the Unified Modeling Language Specification Version 2.5.1," 2017.

[3] Object Management Group, "About the Object Constraint Language Specification Version 2.4," 2014.

[4] M. Richters and M. Gogolla, "Validating UML Models and OCL Constraints," in *<<UML>> 2000 — The Unified Modeling Language* (A. Evans, S. Kent, and B. Selic, eds.), (Berlin, Heidelberg), pp. 265–277, Springer Berlin Heidelberg, 2000.

[5] A. Rensink, M. de Mol, and E. Zambon, "Groove homepage." `https://groove.ewi.utwente.nl`.

[6] H. Radke, T. Arendt, J. S. Becker, A. Habel, and G. Taentzer, "Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations," in *Graph Transformation* (F. Parisi-Presicce and B. Westfechtel, eds.), (Cham), pp. 155–170, Springer International Publishing, 2015.

[7] E. Planas and A. Olive, "The DBLP case study," 2006. `http://www-pagines.fib.upc.es/~modeling/DBLP.pdf`.

[8] Object Management Group, "OMG Unified Modeling Language Specification version 1.5." `https://www.omg.org/spec/UML/1.5/PDF`.

[9] B. König, D. Nolte, J. Padberg, and A. Rensink, *A tutorial on graph transformation*, pp. 83–104. Lecture Notes in Computer Science, Springer, 2018.

[10] A. HABEL and K.-H. PENNEMANN, "Correctness of high-level transformation systems relative to nested conditions," *Mathematical Structures in Computer Science*, vol. 19, no. 2, p. 245–296, 2009.

[11] H. Radke, T. Arendt, J. S. Becker, A. Habel, and G. Taentzer, "Translating Essential OCL Invariants to Nested Graph Constraints Focusing on Set Operations: Long version (2015)," in *Graph Transformation*, 2015. `https://www.uni-marburg.de/fb12/forschung/berichte/berichteinformtk/pdfbi/bi2015-01.pdf`.

[12] "Henshin." `https://www.eclipse.org/henshin/`.

[13] "GReAT: Graph Rewriting and Transformation." `https://www.isis.vanderbilt.edu/tools/GReAT`.

[14] "Graph Transformation & Graph Grammars." https://www.cs.le.ac.uk/people/rh122/gratra/applications.html.

[15] A. Rensink, "Representing First-Order Logic Using Graphs," in *Graph Transformations* (H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, eds.), (Berlin, Heidelberg), pp. 319–335, Springer Berlin Heidelberg, 2004.

[16] "ANTLR Homepage." https://www.antlr.org.

[17] "YACC docs." http://dinosaur.compilertools.net/yacc/index.html.

[18] E. Gagnon and L. Hendren, "SableCC An Object-Oriented Compiler Framework," *Proceedings of TOOLS 1998*, 04 1998.

[19] F. Chabarek, "Development of an OCL-parser for UML-Extensions," *Master's thesis, Technical University of Berlin*, 2004.

[20] B. Beckert, U. Keller, and P. H. Schmitt, "Translating the Object Constraint Language into First-order Predicate Logic," in *In Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC)*, pp. 113–123, 2001.

[21] E. Franconi, A. Mosca, X. Oriol, G. Rull, and E. Teniente, "OCL FO : first-order expressive OCL constraints for efficient integrity checking," *Software and Systems Modeling*, pp. 1–24, 2018.

[22] A. Tort, "The osCommerce case study." http://www-pagines.fib.upc.es/~modeling/osCommerce_cs.pdf.

[23] T. Arendt, A. Habel, H. Radke, and G. Taentzer, "From Core OCL Invariants to Nested Graph Constraints," in *Graph Transformation* (H. Giese and B. König, eds.), (Cham), pp. 97–112, Springer International Publishing, 2014.

[24] J. Noten, J. G. M. Mengerink, and A. Serebrenik, "A Data Set of OCL Expressions on GitHub," in *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, (Piscataway, NJ, USA), pp. 531–534, IEEE Press, 2017.

[25] A. Rensink, *How Much Are Your Geraniums? Taking Graph Conditions Beyond First Order*, pp. 191–213. Cham: Springer International Publishing, 2017.

[26] B. Demuth, "The Dresden OCL toolkit and its role in Information Systems development," *Int. Conf. on Information Systems Development*, 2004.

[27] B. Demuth, H. Hussmann, and A. Konermann, "Generation of an OCL 2.0 Parser," *Proceedings of the MoDELS'05 Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends*, pp. 48–62, 2005.

[28] A. Kleppe and J. Warmer, "Octopus: OCL Tool for Precise Uml Specifications." http://octopus.sourceforge.net.

[29] Eclipse, "Classic Ecore/UML Programmers Guide." https://help.eclipse.org/kepler/topic/org.eclipse.ocl.doc/help/ProgrammersGuide.html?cp=45_5.

[30] "ocl15.grammar." http://sablecc.sourceforge.net/grammars/ocl15.grammar.html.

[31] H. Hussmann and S. Zschaler, "The Object Constraint Language for UML 2.0 – Overview and Assessment.," *Upgrade*, 2004.

[32] "Statistic - plugin for IntelliJ IDEs." `https://plugins.jetbrains.com/plugin/4509-statistic`.

[33] Eduardo Zambon and Arend Rensink, "Solving the N-Queens Problem with GROOVE - Towards a Compendium of Best Practices," in *Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014)* (F. Hermann and S. Sauer, eds.), no. 938 in Electronic Communications of the EASST, pp. –, European Association of Software Science and Technology, 4 2014.

Appendices

Appendix A

OCL 1.5 grammar

```
1  oclFile                 :=  ("package" packageName
2                                  oclExpressions
3                                  "endpackage"
4                              )+
5  packageName             :=  pathName
6  oclExpressions          :=  ( constraint )*
7  constraint              :=  contextDeclaration
8                              (   ( "def" name? ":" letExpression* )
9                                  |
10                                 ( stereotype name? ":" oclExpression)
11                             )+
12 contextDeclaration      :=  "context"
13                             ( operationContext | classifierContext )
14 classifierContext       :=  ( name ":" name )
15                             | name
16 operationContext        :=  name "::" operationName
17                             "(" formalParameterList ")"
18                             ( ":" returnType )?
19 stereotype              :=  ( "pre" | "post" | "inv" )
20 operationName           :=  name | "=" | "+" | "-" | "<" | "<=" |
21                             ">=" | ">" | "/" | "*" | "<>" |
22                             "implies" | "not" | "or" | "xor" | "and"
23 formalParameterList     :=  ( name ":" typeSpecifier
24                                 ("," name ":" typeSpecifier )*
25                             )?
26 typeSpecifier           :=  simpleTypeSpecifier
27                             | collectionType
28 collectionType          :=  collectionKind
29                             "(" simpleTypeSpecifier ")"
30 oclExpression           :=  (letExpression* "in")? expression
31 returnType              :=  typeSpecifier
32 expression              :=  logicalExpression
33 letExpression           :=  "let" name
34                             ( "(" formalParameterList ")" )?
```

```
                                ( ":" typeSpecifier )?
                                "=" expression
ifExpression            :=  "if" expression
                                "then" expression
                                "else" expression
                                "endif"
logicalExpression       :=  relationalExpression
                                ( logicalOperator
                                    relationalExpression
                                )*
relationalExpression    :=  additiveExpression
                                ( relationalOperator
                                    additiveExpression
                                )?
additiveExpression      :=  multiplicativeExpression
                                ( addOperator
                                    multiplicativeExpression
                                )*
multiplicativeExpression:=  unaryExpression
                                ( multiplyOperator
                                    unaryExpression
                                )*
unaryExpression         :=  ( unaryOperator
                                    postfixExpression
                                )
                                | postfixExpression
postfixExpression       :=  primaryExpression
                                ( ("." | "->")propertyCall )*
primaryExpression       :=  literalCollection
                                | literal
                                | propertyCall
                                | "(" expression ")"
                                | ifExpression
propertyCallParameters  :=  "(" ( declarator )?
                                ( actualParameterList )? ")"
literal                 :=  string
                                | number
                                | enumLiteral
enumLiteral             :=  name "::" name ( "::" name )*
simpleTypeSpecifier     :=  pathName
literalCollection       :=  collectionKind "{"
                                ( collectionItem
                                    ("," collectionItem )*
                                )?
                                "}"
collectionItem          :=  expression (".." expression )?
propertyCall            :=  pathName
                                ( timeExpression )?
                                ( qualifiers )?
```

```
                              ( propertyCallParameters )?
qualifiers                :=  "[" actualParameterList "]"
declarator                :=  name ( "," name )*
                              ( ":" simpleTypeSpecifier )?
                              ( ";" name ":" typeSpecifier "="
                                  expression
                              )?
                              "|"
pathName                  :=  name ( "::" name )*
timeExpression            :=  "@" "pre"
actualParameterList       :=  expression ("," expression)*
logicalOperator           :=  "and" | "or" | "xor" | "implies"
collectionKind            :=  "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator        :=  "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator               :=  "+" | "-"
multiplyOperator          :=  "*" | "/"
unaryOperator             :=  "-" | "not"
typeName                  :=  charForNameTop charForName*
name                      :=  charForNameTop charForName*
charForNameTop            :=  /* Characters except inhibitedChar
                                 and ["0"-"9"]; the available
                                 characters shall be determined by
                                 the tool implementers ultimately.*/
charForName               :=  /* Characters except inhibitedChar; the
                                 available characters shall be determined
                                 by the tool implementers ultimately.*/
inhibitedChar             :=  " " | "\"" | "#" | "\'" | "(" | ")" |
                              "*" | "+" | "," | "-" | "." | "/" |
                              ":" | ";" | "<" | "=" | ">" | "@" |
                              "[" | "\\" | "]" | "{" | "|" | "}"
number                    :=  ["0"-"9"] (["0"-"9"])*
                              ( "." ["0"-"9"] (["0"-"9"])* )?
                              ( ("e" | "E") ( "+" | "-" )? ["0"-"9"]
                              (["0"-"9"])*
                              )?
string                    :=  "'"
                              (( ~["'","\\","\n","\r"] )
                               |("\\"
                                  ( ["n","t","b","r","f","\\","'","\""]
                                    | ["0"-"7"]
                                      ( ["0"-"7"] ( ["0"-"7"] )? )?
                                  )
                                 )
                              )*
                              "'"
```

Appendix B

Franconi's OCL_{FO} fragment

```
OCL-Bool        ::= OCL-Bool BoolOp OCL-Bool | not OCL-Bool |
                    OCL-Set ->includesAll(OCL-Set) | OCL-Set ->excludesAll(OCL-Set) |
                    OCL-Set ->includes(OCL-Single) | OCL-Set ->excludes(OCL-Single) |
                    OCL-Set ->Forall(VarList | OCL-Bool) |
                    OCL-Set ->Exists(VarList | OCL-Bool) |
                    OCL-Set ->isEmpty() | OCL-Set ->notEmpty() |
                    OCL-Set ->size() CompOp Integer | OCL-Set ->one(Var | OCL-Bool) |
                    OCL-Set ->isUnique(attr) |
                    OCL-Object.oclIsKindOf(Class) | OCL-Object.oclIsTypeOf(Class) |
                    OCL-Object = null | OCL-Object <> null |
                    OCL-Navigation = OCL-Navigation | OCL-Navigation <> OCL-Navigation |
                    OCL-Value CompOp OCL-Value |
                    OCL-Object.bAttr | Var
OCL-Navigation  ::= OCL-Set | OCL-Single
OCL-Set         ::= OCL-Set ->union(OCL-Set) | OCL-Set ->intersection(OCL-Set) |
                    OCL-Set ->symmetricDifference(OCL-Set) | OCL-Set - OCL-Set |
                    OCL-Set ->select(Var | OCL-Bool) | OCL-Set ->reject(Var | OCL-Bool) |
                    OCL-Set ->selectByKind(Class) | OCL-Set ->selectByType(Class) |
                    OCL-Set.role[role] | OCL-Set.assoClass[role] |
                    OCL-Object.nfRole[role] | OCL-Object.nfAssoClass[role] |
                    OCL-Set.attr | OCL-Set.nfAttr |
                    Class.allInstances() | OCL-Single
OCL-Single      ::= OCL-Object | OCL-Value
OCL-Object      ::= OCL-Object.oclAsType(Class) |
                    OCL-Object.fRole | OCL-Object.fAssoClass |
                    Var | self
OCL-Value       ::= Constant | Var |
                    OCL-Object.fAttr |
                    OCL-Set ->min() | OCL-Set ->max()
BoolOp          ::= and | or | xor | implies
CompOp          ::= < | <= | = | >= | > | <>
VarList         ::= Var (,Var)*
Var             ::= <a variable name>
Class           ::= <a class name>
```

```
assoClass       ::= <an association class name>
fAssoClass      ::= <an association class name of a functional role>
nfAssoClass     ::= <an association class name of a non functional role>
role            ::= <a role name>
fRole           ::= <a functional role name>
nfRole          ::= <a non functional role name>
attr            ::= <an attribute name>
bAttr           ::= <a boolean attribute name>
fAttr           ::= <a functional attribute name>
nfAttr          ::= <a non functional attribute name>
Integer         ::= <an integer number>
Constant        ::= <a constant name>
```

Appendix C

Radke's OCL to graph transformation

Radke et al. use a slightly different notation for a node with an attribute. So where we write $\boxed{\text{v1:T}} \xrightarrow{attr1} \boxed{\text{v2:int}} \xrightarrow{\geq} \boxed{\text{0:int}}$ Radke et al. write $\boxed{\begin{array}{|c|}\hline \text{v1:T} \\ \hline \text{attr1} \geq 0 \\ \hline \end{array}}$. The difference in notation was necessary for our implementation of `expr1 op expr2`, which will be explained more thoroughly in Appendix D.2.

Let `expr`, `expr1` and `expr2` be OCL expressions, u, x, v1, v2, v3, v4 names of nodes (i.e. variables), T= t(v) denote the type of v and likewise T‘ = t(v‘), `attr1` and `attr2` be attribute names, op $\in \{<, >, \leq, \geq, =, <>\}$ a comparison operator, and role be a role of a class. Then

Gr	#	**OCL**	**Nested Graph**
$\mathbf{tr}_I$	i	‘context‘ C ‘inv:‘ expr	$\forall($ $\boxed{\text{self:C}}$, $\text{tr}_E(\text{expr}))$
	ii	‘context‘ var:C ‘inv:‘ expr	$\forall($ $\boxed{\text{var:C}}$, $\text{tr}_E(\text{expr}))$
$\mathbf{tr}_E$	iii	true	true
	iv	‘not‘ expr	$\neg\ \text{tr}_E(\text{expr})$
	v	expr1 ‘and‘ expr2	$\text{tr}_E(\text{expr1}) \wedge \text{tr}_E(\text{expr2})$
	vi	expr1 ‘or‘ expr2	$\text{tr}_E(\text{expr1}) \vee \text{tr}_E(\text{expr2})$
	vii	expr1 ‘implies‘ expr2	$\neg\text{tr}_E(\text{expr1}) \wedge \text{tr}_E(\text{expr2})$
	viii	‘if‘ cond ‘then‘ expr1 ‘else‘ expr2 ‘endif‘	$(\text{tr}_E(\text{cond}) \wedge \text{tr}_E(\text{expr1})) \vee (\neg\ \text{tr}_E(\text{cond}) \wedge \text{tr}_E(\text{expr2}))$
	ix	expr1‘->exists‘(v:T|expr2)	$\exists($ $\boxed{\text{v:T}}$, $\text{tr}_S(\text{expr1},$ $\boxed{\text{v:T}}$ $) \wedge \text{tr}_E(\text{expr2}))$
	x	expr1‘->forall‘(v:T|expr2)	$\forall($ $\boxed{\text{v:T}}$, $\text{tr}_S(\text{expr1},$ $\boxed{\text{v:T}}$ $) \implies \text{tr}_E(\text{expr2}))$
	xi	expr1‘->includesAll‘(v:T|expr2)	$\forall($ $\boxed{\text{v:T}}$, $\text{tr}_S(\text{expr2},$ $\boxed{\text{v:T}}$ $) \implies \text{tr}_S(\text{expr1},$ $\boxed{\text{v:T}}$ $))$
	xii	expr1‘->excludesAll‘(v:T|expr2)	$\forall($ $\boxed{\text{v:T}}$, $\text{tr}_S(\text{expr2},$ $\boxed{\text{v:T}}$ $) \implies \neg\ \text{tr}_S(\text{expr1},$ $\boxed{\text{v:T}}$ $))$
	xiii	expr‘->notEmpty()‘	$\exists($ $\boxed{\text{v:T}}$, $\text{tr}_S(\text{expr},$ $\boxed{\text{v:T}}$ $))$
	xiv	expr‘->size() >=‘ n	$\exists($ $\boxed{\text{v}_1\text{:T}}$ ... $\boxed{\text{v}_n\text{:T}}$, $\wedge_{i=1}^{n}\ \text{tr}_S(\text{expr},$ $\boxed{\text{v}_i\text{:T}}$ $))$
	xv	expr1 ‘=‘ expr2 *t(expr1)=t(expr2)=T*	$\exists($ $\boxed{\text{v:T}}$, $\text{tr}_N(\text{expr1},$ $\boxed{\text{v:T}}$ $) \wedge \text{tr}_N(\text{expr2},$ $\boxed{\text{v:T}}$ $))$
	xvi	expr1 ‘=‘ expr2 *t(expr1)=t(expr2)=Set(T)*	$\forall($ $\boxed{\text{v:T}}$, $\text{tr}_S(\text{expr1},$ $\boxed{\text{v:T}}$ $) \iff \text{tr}_S(\text{expr2},$ $\boxed{\text{v:T}}$ $))$
	xvii	expr.attr1 op n *t(expr) = T*	$\exists($ $\boxed{\text{v:T}}$, $\text{tr}_N(\text{expr},$ $\boxed{\text{v:T}}$ $) \wedge \exists($ $\boxed{\begin{array}{c}\text{v:T} \\ \hline \text{attr1 op n}\end{array}}$ $))$

	xviii	expr1.attr1 op expr2.attr2	$\exists($ v1:T , $\mathrm{tr}_N($expr1, v1:T1 | attr1 op x$)$ $\wedge \mathrm{tr}_N($expr2, v1:T1 | attr2 = x$)) \vee^* \exists($ v1:T1 v2:T2 , $\mathrm{tr}_N($expr1, v1:T1 | attr1 op x$)$ $\wedge \mathrm{tr}_N($expr2, v2:T2 | attr2 = x$)) $ *The part before $\vee^*$ is omitted if clan(t(expr1)) $\cap$ clan(t(expr2)) $= \emptyset$, and the part after $\vee^*$ is omitted if expr1=expr2*
	xix	expr‘->oclIsKindOf‘(T) *T‘=t(expr) and T $\in$clan(T‘)*	$\exists($ v:T‘ $\hookrightarrow$ v:T , $\mathrm{tr}_N($expr, v:T‘$))$
	xx	expr‘->oclIsTypeOf‘(T) *T‘=t(expr) and T $\in$clan(T‘)*	$\exists($ v:T‘ $\hookrightarrow$ v:T , $\bigwedge_{T“\in clan(T)}^{T“\neq T} \neg\exists($ v:T $\hookrightarrow$ v:T“$)$ $\wedge \mathrm{tr}_N($expr, v:T“$))$
$\mathbf{tr}_N$	xxi	(expr‘.oclAsType‘(T), v:T) *T‘=t(expr) and T $\in$clan(T‘)*	$\exists($ v:T‘ $\hookrightarrow$ v:T , $\mathrm{tr}_N($expr, v:T‘$))$
	xxii	(v, v‘:T)	$\exists($ v=v‘:T $)$
	xxiii	(expr.role, v:T) *T‘=t(expr) $\wedge$ T‘ $\notin$ clan(T)*	$\exists($ v‘:T‘ $\xrightarrow{role}$ v:T , $\mathrm{tr}_N($expr, v‘:T‘$))$
	xxiv	(expr.role, v:T) *T‘=t(expr) $\wedge$ T‘ $\in$ clan(T)*	$\exists($ v‘:T‘ $\xrightarrow{role}$ v:T , $\mathrm{tr}_N($expr, v‘:T‘$))\vee$ $\exists($ v:T $\hookleftarrow$ role , $\mathrm{tr}_N($expr, v:T$))$
$\mathbf{tr}_S$	xxv	(expr.role, v:T) *T‘=t(expr) $\wedge$ T‘ $\notin$ clan(T)*	$\exists($ v‘:T‘ $\xrightarrow{role}$ v:T , $\mathrm{tr}_N($expr, v‘:T‘$))$
	xxvi	(expr.role, v:T) *T‘=t(expr) $\wedge$ T‘ $\in$ clan(T)*	$\exists($ v‘:T‘ $\xrightarrow{role}$ v:T , $\mathrm{tr}_N($expr, v‘:T‘$))\vee$ $\exists($ v:T $\hookleftarrow$ role , $\mathrm{tr}_N($expr, v:T$))$
	xxvii	(expr1‘->select‘(v:T | expr2), v‘:T)	$\mathrm{tr}_S($expr1, v‘:T $) \wedge \mathrm{tr}_E($expr2$)\{$v‘/v$\}$
	xxviii	(expr1‘->reject‘(v:T| expr2), v‘:T)	$\mathrm{tr}_S($expr1, v‘:T $) \wedge\neg \mathrm{tr}_E($expr2$)\{$v‘/v$\}$
	xxix	expr1‘->collect‘(v:T | expr2), v‘:T‘ *if expr2 yields a set*	$\exists($ v:T , $\mathrm{tr}_S($expr1, v:T $)\wedge\mathrm{tr}_S($expr2, v‘:T‘$))$
	xxx	expr1‘->collect‘(v:T | expr2), v‘:T‘ *if expr2 yields an object*	$\exists($ v:T , $\mathrm{tr}_S($expr1, v:T $)\wedge\mathrm{tr}_N($expr2, v‘:T‘$))$
	xxxi	(expr1‘->union‘(expr2), v:T)	$\mathrm{tr}_S($expr1, v:T $) \vee \mathrm{tr}_S($expr2, v:T $)$
	xxxii	(expr1‘->intersection‘(expr2), v:T)	$\mathrm{tr}_S($expr1, v:T $) \wedge \mathrm{tr}_S($expr2, v:T $)$
	xxxiii	(expr1‘-‘expr2, v:T)	$\mathrm{tr}_S($expr1, v:T $) \wedge\neg \mathrm{tr}_S($expr2, v:T $)$
	xxxiv	(expr1‘-> symmetricDifference‘(expr2), v:T)	$\mathrm{tr}_S($expr1, v:T $) \veebar \mathrm{tr}_S($expr2, v:T $)$
	xxxv	(T‘.allInstances()‘, v:T)	$\exists($ v:T $)$
	xxxvi	(Set{expr1,...,exprN}, v:T)	$\mathrm{tr}_N($expr1, v:T $) \vee ... \vee \mathrm{tr}_N($exprN, v:T $)$

C.1 Radke's Equivalence Rules

The following equivalences are defined by Radke et al. in [6]. Let $C_1 \oplus_P C_2$ denote the gluing or pushout of C_1 and C_2 along P and let $\mathcal{P}$ denote the set of all intersections of C_1 and C_2:
(RE1)(a) $\exists(C_1, \exists(C_2)) \equiv \bigvee_{P\in\mathcal{P}} \exists(C_1 \oplus_P C_2)$.
(b) $\exists(C_1, \exists(C_2)) \equiv \exists(C_1 + C_2)$ if C_1 and C_2 are clan-disjoint.
(c) $\exists(C_1, \exists(C_2)) \equiv \exists(C_2)$ if $C_1 \subseteq C_2$ and $\equiv \exists(C_1)$ if $C_2 \subseteq C_1$.

(RE2)(a) $\exists(C_1, \exists(C_2) \wedge \exists(C_3)) \equiv \exists(C_1, \vee_{P \in \mathcal{P}} \exists(C_2 \oplus_P C_3))$, if for all node names occurring in both C_2 and C_3, a node with that name already exists in C_1.

(b) $\exists(C_1) \wedge \exists(C_2) \equiv \exists(C_1 + C_2)$ if C_1 and C_2 are clan-disjoint and have disjoint sets of node names.

(RE3) $\exists(\boxed{\text{u:T}}, \exists(C) \wedge \exists(\boxed{\text{u=v:T}})) \equiv \exists(\boxed{\text{u:T}}, \exists(C[u = v]))$ provided that either u or v does not exist in C and $C[u = v]$ is the graph obtained from C by renaming u by $u = v$.

Appendix D

Improvements on Radke's rules

This appendix explains our contributions and therefore our changes to the transformation rules as defined by Radke et al. which can be found in Appendix C. To be able to show what goes wrong and thus why we had to create new transformation rules we will use the transformation rules defined by Radke et al. The transformation rule of Radke et al. can be recognized by the Roman numbers and their equivalence rules start with an R. For example "RE1" is the first equivalence rule defined by Radke et al.

D.1 Renaming Equivalence Rule

Radke et al. have defined the renaming equivalence rule RE3 as:

$$\exists(\boxed{\text{u:T}}, \exists(C) \wedge \exists(\boxed{\text{u=v:T}})) \equiv \exists(\boxed{\text{u:T}}, \exists(C[u = v]))$$

provided that either u or v does not exist in C and $C[u = v]$ is the graph obtained from C by renaming u by $u = v$.

At the moment we started implementing the equivalence rules we realised that if you follow the defined equivalence rule strictly, the result is not the desired result. We show in Example 11 what happens if we follow the equivalence rule strictly.

Example 11. `context Department inv: self.budget >= 0`

$$\begin{aligned}
&=^{1} \forall(\boxed{\text{self:D}}, tr_E(\texttt{self.budget>=0})) \\
&=^{\text{xvii}} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:D}}, tr_N(\texttt{self}, \boxed{\text{v:D}}) \wedge \exists(\boxed{\begin{array}{|c|}\hline \text{v:D} \\ \hline \text{budget} \geq 0 \\ \hline\end{array}}))) \\
&=^{\text{xxii}} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:D}}, \exists(\boxed{\text{self=v:D}}) \wedge \exists(\boxed{\begin{array}{|c|}\hline \text{v:D} \\ \hline \text{budget} \geq 0 \\ \hline\end{array}}))) \\
&=^{RE3} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:D}}, \exists(\boxed{\begin{array}{|c|}\hline \text{v:D} \\ \hline \text{budget} \geq 0 \\ \hline\end{array}})))
\end{aligned}$$

At this moment we are stuck and the lax condition cannot be simplified further. If we read the current lax condition it reads; for every node of type Department, there exists another node of type Department that has an budget that is equal or greater than 0. What we want is that it reads; every node of type Department has a budget that is equal or greater than 0. Therefore we have to be able to do another simplification step such that we can combine the two existential

quantifiers with equivalence rule RE1c. Currently this is not possible because the equivalence $\exists(\boxed{\text{self=v:D}})$ is removed by applying the equivalence rule and now we are not sure we are allowed to apply RE1c.

Before we give a new definition of the renaming equivalence rule a deeper understanding of lax conditions is necessary. Within lax conditions a quantification goes over the graph and not over the name that is given to a node, in this example $\exists(\boxed{\text{self=v:D}})$ tells us that the node with the name self and the node with the name v are one and the same node. The node names that appear in $\boxed{\text{self=v:D}}$ are always defined already defined just like in the example where they are already defined with $\forall(\boxed{\text{self:D}})$ and $\exists(\boxed{\text{v:D}})$. To be able to keep simplifying you need to rename the remaining $\boxed{\text{v:D}}$ also to $\boxed{\text{self:D}}$, which is possible because $\exists(\boxed{\text{self=v:D}})$ told us that $\boxed{\text{self:D}}$ and $\boxed{\text{v:D}}$ are one and the same node.

Now that we know that we have to change the definition, we will also clarify the renaming part of the equivalence rule. Currently $C[u = v]$ could be read both ways and so you have no clue if u is renamed to v or the other way around. In the new definition we will denote a rename in the same way as they do in relational algebra. We write $C\{a/b\}$ as we want to rename every b in C to a. With these two changes in mind, the new renaming equivalence rule is defined in Definition 3 and is called E4a in our equivalence rules in Section 4.4.2.

Definition 3. $\exists(\boxed{\text{u:T}}, \exists(C) \wedge \exists(\boxed{\text{u=v:T}})) \equiv \exists(\boxed{\text{v:T}}, \exists(C\{v/u\}))$ provided that v does not exist in C and $C\{v/u\}$ is the graph obtained from C by renaming u to v.

Now if we replace RE3 with the new definition, E4a, we get the elaboration of Example 12 and as we can see we can apply RE1c now to get the intended result.

Example 12. `context Department inv: self.budget >= 0`

$$
\begin{aligned}
&=^{1} \forall(\boxed{\text{self:D}}, tr_E(\texttt{self.budget>=0})) \\
&=^{\text{xvii}} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:D}}, tr_N(\texttt{self}, \boxed{\text{v:D}}) \wedge \exists(\boxed{\begin{array}{c}\text{v:D}\\ \hline \text{budget} \geq 0\end{array}}))) \\
&=^{\text{xxii}} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:D}}, \exists(\boxed{\text{self=v:D}}) \wedge \exists(\boxed{\begin{array}{c}\text{v:D}\\ \hline \text{budget} \geq 0\end{array}}))) \\
&=^{E4a} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{self:D}}, \exists(\boxed{\begin{array}{c}\text{v:D}\\ \hline \text{budget} \geq 0\end{array}}))) \\
&=^{RE1c} \forall(\boxed{\text{self:D}}, \exists(\boxed{\begin{array}{c}\text{v:D}\\ \hline \text{budget} \geq 0\end{array}}))
\end{aligned}
$$

Renaming Equivalence Rule in Navigation

It is possible to get a $\boxed{\text{u=v:T}}$ without an $\wedge$. This happens when tr_N is called with a longer path instead of directly a variable on the node `self`. Example 13 shows in what scenario this happens.

Example 13. `context Department inv: self.project.budget >= 0`

$$
\begin{aligned}
&=^{\text{i}} \forall(\boxed{\text{self:D}}, tr_E(\texttt{self.project.budget>=0})) \\
&=^{\text{xvii}} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:P}}, tr_N(\texttt{self.project}, \boxed{\text{v:P}}) \wedge \exists(\boxed{\begin{array}{c}\text{v:P}\\ \hline \text{budget} \geq 0\end{array}})) \\
&=^{\text{xxiii}} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:P}}, \exists(\boxed{\text{v`:D}} \xrightarrow{p} \boxed{\text{v:P}}, tr_N(\texttt{self}, \boxed{\text{v`:D}})) \wedge \exists(\boxed{\begin{array}{c}\text{v:P}\\ \hline \text{budget} \geq 0\end{array}}))
\end{aligned}
$$

$$=^{\text{xxii}} \forall(\boxed{\text{self:D}}, \exists(\boxed{\text{v:P}}, \exists(\boxed{\text{v‘:D}} \xrightarrow{p} \boxed{\text{v:P}}, \exists(\boxed{\text{v‘=self:D}})) \wedge \exists(\boxed{\begin{array}{l}\text{v:P}\\ \hline \text{budget} \geq 0\end{array}}))$$

As you can see, in the end we have $\exists(\boxed{\text{u:T}}, \exists(\boxed{\text{u=v:T}}))$, which does not correspond to the defined renaming equivalence rule. To be able to simplify such a case we introduce a second renaming equivalence rule which does not obligate the $\wedge$, the new, additional, renaming equivalence rule is defined in Definition 4 and is called E4b since it is a special case of our E4a equivalence rule in Section 4.4.2.

Definition 4. $\exists(\boxed{\text{u:T}}, \exists(\boxed{\text{u=v:T}})) \equiv \exists(\boxed{\text{v:T}})$

D.2 Compare and equality operator

A lot has changed about the compare and equality operator transformation rules. To be more specific, we are talking about the following transformation rules: xv, xvi, xvii and xviii. In the upcoming paragraphs we will talk about what goes wrong with the current implementation and in Appendix D.2.4 we will give the new implementation.

D.2.1 Losing attributes

With the current implementation of transformation rule xviii we will lose attributes in some scenarios. transformation rule xviii is defined as:

$$\exists(\boxed{\text{v:T}}, tr_(\texttt{expr1}, \boxed{\begin{array}{l}\text{v:T}\\ \hline \text{attr1 op x}\end{array}}) \wedge tr_N(\texttt{expr2}, \boxed{\begin{array}{l}\text{v:T}\\ \hline \text{attr2} = \text{x}\end{array}}))\vee^*$$

$$\exists(\boxed{\text{v:T}}\ \boxed{\text{v‘:T‘}}, tr_N(\texttt{expr1}, \boxed{\begin{array}{l}\text{v:T}\\ \hline \text{attr1 op x}\end{array}}) \wedge tr_N(\texttt{expr2}, \boxed{\begin{array}{l}\text{v‘:T‘}\\ \hline \text{attr2} = \text{x}\end{array}}))$$

The part before $\vee^*$ *is omitted if* $\text{clan}(\text{t}(\texttt{expr1})) \cap \text{clan}(\text{t}(\texttt{expr2})) = \emptyset$*, and the part after* $\vee^*$ *is omitted if* `expr1=expr2`.

Example 14 shows how we lose an attribute in the given scenario, after the example we will explain why this is happening.

Example 14. `context Project inv: self.budget <= self.department.budget`

$$=^{\text{i}} \forall(\boxed{\text{self:P}}, tr_E(\texttt{self.budget <= self.department.budget}))$$

$$=^{\text{xviii}} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:P}}\ \boxed{\text{v‘:D}}, tr_N(\texttt{self}, \boxed{\begin{array}{l}\text{v:P}\\ \hline \text{budget} <= \text{x}\end{array}}) \wedge tr_N(\texttt{self.department}, \boxed{\begin{array}{l}\text{v‘:D}\\ \hline \text{budget} = \text{x}\end{array}})))$$

$$=^{\text{xxiii}} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:P}}\ \boxed{\text{v‘:D}}, tr_N(\texttt{self}, \boxed{\begin{array}{l}\text{v:P}\\ \hline \text{budget} <= \text{x}\end{array}}) \wedge \exists(\boxed{\text{v“:P}} \xrightarrow{d} \boxed{\begin{array}{l}\text{v‘:D}\\ \hline \text{budget} = \text{x}\end{array}}, tr_N(\texttt{self}, \boxed{\text{v“:P}}))))$$

$$=^{2\times\text{xxii}} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:P}}\ \boxed{\text{v‘:D}}, \exists(\boxed{\begin{array}{l}\text{v=self:P}\\ \hline \text{budget} <= \text{x}\end{array}}) \wedge \exists(\boxed{\text{v“:P}} \xrightarrow{d} \boxed{\begin{array}{l}\text{v‘:D}\\ \hline \text{budget} = \text{x}\end{array}}, \exists(\boxed{\text{v“=self:P}}))))$$

$$=^{2\times E3} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{self:P}}\ \boxed{\text{v‘:D}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\begin{array}{l}\text{v‘:D}\\ \hline \text{budget} = \text{x}\end{array}})))$$

$$=^{E1c} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\begin{array}{l}\text{v‘:D}\\ \hline \text{budget} = \text{x}\end{array}}))$$

As you can see in Example 14 we have lost the $\boxed{\begin{array}{l}\text{v:P}\\ \hline \text{budget} <= \text{x}\end{array}}$. This happens because of how transformation rule xviii works in combination with rule xxii and RE3.

Currently rule xviii defines the existence of two nodes and inside of the tr_N the attributes are defined. Important to know is that tr_N stands for navigation transformation. In this case the navigation transformation is called with a new attribute that is not defined within a quantifier yet. If you combine this with rule xviii, which creates an existence such that two nodes are the same, and rule RE3 which assumes that the node already exists and renames accordingly. You are going to lose the attribute, because RE3 only renames as you can see in the elaboration above.

D.2.2 Overlap in the clans

With the current implementation of transformation rule xviii something strange happens if both parts (before and after the $\vee^*$ are needed. This happens if `expr1` $\neq$ `expr2` and $clan(t(\texttt{expr1})) \cap clan(t(\texttt{expr2})) \neq \emptyset$. An example where this happens can be found below, in which EB stands for EditedBook and BC for BookChapter. This example can be applied to the class diagram of the DBLP case study [7] and can be found in Appendix E.

$$
\begin{aligned}
&\texttt{context EditedBook inv: self.year >= self.bookSection.bookChapter.year} \\
=^{\mathrm{i}}\ &\forall(\boxed{\text{self:EB}}, tr_E(\texttt{self.year >= self.bookSection.bookChapter.year})) \\
=^{\mathrm{xviii}}\ &\forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{v:EB}}, tr_N(\texttt{self}, \boxed{\begin{array}{c}\text{v:EB}\\ \hline \text{year} >= \text{x}\end{array}}) \wedge tr_N(\texttt{self.bS.bC}, \boxed{\begin{array}{c}\text{v:EB}\\ \hline \text{year} = \text{x}\end{array}})) \\
&\vee \exists(\boxed{\text{v:EB}}\ \boxed{\text{v`:BC}}, tr_N(\texttt{self}, \boxed{\begin{array}{c}\text{v:EB}\\ \hline \text{year} >= \text{x}\end{array}}) \wedge tr_N(\texttt{self.bS.bC}, \boxed{\begin{array}{c}\text{v`:BC}\\ \hline \text{year} = \text{x}\end{array}})))
\end{aligned}
$$

The problem is within the part before the $\vee$, where the existence of only one node $\boxed{\text{v:EB}}$ is used to check both attributes. In this case this is really strange because the type of `self.bS.bC (self.bookSection.b` is not of type EditedBook at all. Therefore $tr_N(\texttt{self.bS.bC}, \boxed{\begin{array}{c}\text{v:EB}\\ \hline \text{year} = \text{x}\end{array}})$ cannot be resolved. This issue is easily solved by the fact that we do not match injectively in constrast to Radke et al. Therefore the part before the $\vee^*$ is abundant. The cases where both attributes are one and the same node is also applied with the part after the $\vee^*$ because it is not matched injectively and so v and v‘ are allowed to be the same node.

D.2.3 Extensibility

We have extended the OCL fragment with some additional operations, such as `min()` and `max()`. Here a problem arises with the fact that every combination is defined as a new transformation rule. For example rule xvii `expr.attr1 op n` compares an expression with a constant, but this rule cannot handle `n op expr.attr1` where a constant is compared with an expression. At first this is not a really big issue, but we want to extend the comparison with two operations. Without a change this would have resulted in an exponential growth of transformation rules, which would have been unnecessary. For example for only the `min()` operation the following transformation rules were necessary, and the same line of thought would have applied to the `max()` operation. In our final solution we have resolved this issue.

1. `expr1.attr1->min() op n`
2. `expr1.attr1->min() op expr2.attr2`
3. `expr1.attr1->min() op expr1.attr1->max()`

D.2.4 Solution

In this section we will resolve the described problems one for one until we have the final solution, then we will show that our final solution works. This also has been a process during the development in which we have resolved these problems step by step, because without the intermediate solution steps the final solution is quite a big step from the definitions of Radke et al.

The solution to prevent losing the attributes in some scenarios as described in Appendix D.2.1, is to create the variables outside the navigation expression. Applying this to rule xviii results in the following transformation rule:

`expr1 op expr2` $\equiv$

$\exists($ v:T | attr1 op x, attr2 = x $, tr_($`expr1`$,$ v:T | attr1 op x $) \wedge tr_N($`expr2`$,$ v:T | attr2 = x $))\vee^*$

$\exists($ v:T | attr1 op x $,$ v‘:T‘ | attr2 = x $, tr_N($`expr1`$,$ v:T | attr1 op x $) \wedge tr_N($`expr2`$,$ v‘:T‘ | attr2 = x $))$

The solution to prevent a node with the wrong type name, which happens when the clans overlap and the expressions are not the same, as described in Appendix D.2.2, is to remove the part before the $\vee^*$. This is possible because we do not match injectively. Applying this to rule xviii results in the following transformation rule:

`expr1 op expr2` $\equiv$

$\exists($ v:T | attr1 op x $,$ v‘:T‘ | attr2 = x $, tr_N($`expr1`$,$ v:T | attr1 op x $) \wedge tr_N($`expr2`$,$ v‘:T‘ | attr2 = x $))$

The last problem described is the extensibility in Appendix D.2.3. To resolve this problem we have to be able to transform both expressions separate and connect them together with the defined operation. To make this possible we had to unpack the notation of nodes with attributes of Radke et al. because we need to access the attributes directly, such that we can create the comparison of the variables and then create the path to these variables with the navigation expressions. So the starting point of the new transformation step was:
`expr1 op expr2` $\equiv \exists($ v:T $\xrightarrow{\texttt{op}}$ v‘:T‘ $, tr_N($`expr1`$,$ v:T $) \wedge tr_N($`expr2`$,$ v‘:T‘ $))$

Notation-wise this is not completely correct because only the equals ($=$) and not equals ($<>$) are symmetrical operations and v:T $\xrightarrow{\texttt{op}}$ v‘:T‘ suggests that the operations are symmetrical. Therefore we create two variants of this rule, one for the two symmetrical operators and one for the remaining four comparison operators $\{<, \leq, \geq, >\}$. This is the following transformation rule:
`expr1 op expr2` $\equiv \exists($ v:T $\xrightarrow{\texttt{op}}$ v‘:T‘ $, tr_N($`expr1`$,$ v:T $) \wedge tr_N($`expr2`$,$ v‘:T‘ $))$

Of course we do need some additional navigation transformation rules to be able to connect all the possible expressions with the variable. First of all the constant value, this one is almost the same to transformation rule xxii except that v is a node name and not a constant, so we add an additional variant in which v is replaced for a constant, this results in the following transformation rule:
$tr_N(N,$ v:T $) \equiv \exists($ N=v:T $)$

To connect a variable with its class we can use transformation rule xxiii, since the name of the variable is placed on the edge, as described in Definition 1, this transformation rule does apply for this scenario too.

The way to determine if two sets are equal (transformation rule xvi) is different from the way to determine if two values are equal. So far we have focused on comparing two values, but what is the impact for comparing to sets? The new implementation is based on the value comparison definition, so you can recognize the two nodes with the compare operator in between. In this case, the node that is universally quantified is only connected to `expr1`, to solve this we have duplicated the statement, once for `expr1` and once for `expr2`. This results in the following transformation rule:

$$\begin{aligned}&\texttt{expr1 eq expr2} \equiv \\ &\forall(\boxed{\text{v1:T}}, \exists(\boxed{\text{v1:T}} \xrightarrow{\texttt{op}} \boxed{\text{v2:T}}, tr_S(\texttt{expr2}, \boxed{\text{v2:T}})) \wedge tr_S(\texttt{expr1}, \boxed{\text{v1:T}}))\wedge \\ &\forall(\boxed{\text{v3:T}}, \exists(\boxed{\text{v3:T}} \xrightarrow{\texttt{op}} \boxed{\text{v4:T}}, tr_S(\texttt{expr1}, \boxed{\text{v4:T}})) \wedge tr_S(\texttt{expr2}, \boxed{\text{v3:T}}))\end{aligned}$$

Lastly the two additional operations, `min()` and `max()` can be added quite easy now because we have changed the definitions in such a way that it is easier to extend with an additional transformation rule within the navigation transformations. The following two transformation rules are added to create support for the `min()` and `max()` operations.

$$\begin{aligned}tr_N(\texttt{expr.attr->min()}, \boxed{\text{x:T}}) \equiv &\exists(\boxed{\text{v1:T}} \xrightarrow{attr} \boxed{\text{x:T}}, tr_N(\texttt{expr}, \boxed{\text{v1:T}})\wedge \\ &\forall(\boxed{\text{v2:T}} \xrightarrow{attr} \boxed{\text{v3:T}}, tr_N(\texttt{expr}, \boxed{\text{v2:T}}) \implies \exists(\boxed{\text{x:T}} \xRightarrow{\leq} \boxed{\text{v3:T}}))) \\ tr_N(\texttt{expr.attr->max()}, \boxed{\text{x:T}}) \equiv &\exists(\boxed{\text{v1:T}} \xrightarrow{attr} \boxed{\text{x:T}}, tr_N(\texttt{expr}, \boxed{\text{v1:T}})\wedge \\ &\forall(\boxed{\text{v2:T}} \xrightarrow{attr} \boxed{\text{v3:T}}, tr_N(\texttt{expr}, \boxed{\text{v2:T}}) \implies \exists(\boxed{\text{x:T}} \xRightarrow{\geq} \boxed{\text{v3:T}})))\end{aligned}$$

Examples

Considering that this is a big change in definitions, we will give multiple examples in this paragraph to show that this implementation is able to handle both comparisons with constants, comparisons between variables but also comparisons with the minimal or maximal value. First in Example 15 we give an example with a comparison with a constant. In Example 16 we give an example where we compare two attributes. In Example 17 we give an example that compares the minimal value with a constant and in Example 18 we give an example where we compare the maximal value with an attribute. Lastly in Example 19, we give a set equality example.

Example 15. `context Employee inv: self.salary >= 1000`

$$\begin{aligned}&=^{1} \forall(\boxed{\text{self:E}}, tr_E(\texttt{self.salary >= 1000})) \\ &=^{14} \forall(\boxed{\text{self:E}}, \exists(\boxed{\text{v:int}} \xRightarrow{\geq} \boxed{\text{v`:int}}, tr_N(\texttt{self.salary}, \boxed{\text{v:int}}) \wedge tr_N(1000, \boxed{\text{v`:int}}))) \\ &=^{37,35} \forall(\boxed{\text{self:E}}, \exists(\boxed{\text{v:int}} \xRightarrow{\geq} \boxed{\text{v`:int}}, \exists(\boxed{\text{v``:E}} \xrightarrow{salary} \boxed{\text{v:int}}, tr_N(\texttt{self}, \boxed{\text{v``:E}})) \wedge \exists(\boxed{1000{=}\text{v`:int}}))) \\ &=^{34} \forall(\boxed{\text{self:E}}, \exists(\boxed{\text{v:int}} \xRightarrow{\geq} \boxed{\text{v`:int}}, \exists(\boxed{\text{v``:E}} \xrightarrow{salary} \boxed{\text{v:int}}, \exists(\boxed{\text{self}{=}\text{v``:E}})) \wedge \exists(\boxed{1000{=}\text{v`:int}}))) \\ &=^{E4} \forall(\boxed{\text{self:E}}, \exists(\boxed{\text{v:int}} \xRightarrow{\geq} \boxed{\text{1000:int}}, \exists(\boxed{\text{self:E}} \xrightarrow{salary} \boxed{\text{v:int}}))) \\ &=^{E1} \forall(\boxed{\text{self:E}}, \exists(\boxed{\text{self:E}} \xrightarrow{salary} \boxed{\text{v:int}} \xRightarrow{\geq} \boxed{\text{1000:int}}))\end{aligned}$$

Example 16. `context Project inv: self.budget <= self.department.budget`

$$
\begin{aligned}
&=^{1} \forall(\boxed{\text{self:P}}, tr_E(\texttt{self.budget <= self.department.budget})) \\
&=^{14} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, tr_N(\texttt{self.budget}, \boxed{\text{v:int}}) \wedge tr_N(\texttt{self.department.budget}, \boxed{\text{v1:int}})) \\
&=^{2\times 37} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:P}} \xrightarrow{b} \boxed{\text{v:int}}, tr_N(\texttt{self}, \boxed{\text{v2:P}})) \wedge \\
&\quad \exists(\boxed{\text{v3:D}} \xrightarrow{b} \boxed{\text{v1:int}}, tr_N(\texttt{self.department}, \boxed{\text{v3:D}})))) \\
&=^{37} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:P}} \xrightarrow{b} \boxed{\text{v:int}}, tr_N(\texttt{self}, \boxed{\text{v2:P}})) \wedge \\
&\quad \exists(\boxed{\text{v3:D}} \xrightarrow{b} \boxed{\text{v1:int}}, \exists(\boxed{\text{v4:P}} \xrightarrow{d} \boxed{\text{v3:D}}, tr_N(\texttt{self}, \boxed{\text{v4:P}}))))) \\
&=^{2\times 34} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:P}} \xrightarrow{b} \boxed{\text{v:int}}, \exists(\boxed{\text{self=v2:P}})) \wedge \\
&\quad \exists(\boxed{\text{v3:D}} \xrightarrow{b} \boxed{\text{v1:int}}, \exists(\boxed{\text{v4:P}} \xrightarrow{d} \boxed{\text{v3:D}}, \exists(\boxed{\text{self=v4:P}}))))) \\
&=^{2\times E4b} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{b} \boxed{\text{v:int}}) \wedge \exists(\boxed{\text{v3:D}} \xrightarrow{b} \boxed{\text{v1:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v3:D}})))) \\
&=^{E1} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{b} \boxed{\text{v:int}}) \wedge \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v3:D}} \xrightarrow{b} \boxed{\text{v1:int}})))
\end{aligned}
$$

Example 17. `context Project inv: self.employee.salary->min() = 1500`

$$
\begin{aligned}
&=^{1} \forall(\boxed{\text{self:P}}, tr_E(\texttt{self.employee.salary->min() = 1500})) \\
&=^{13} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{=} \boxed{\text{v1:int}}, tr_N(\texttt{self.employee.salary->min()}, \boxed{\text{v:int}}) \wedge tr_N(\texttt{1500}, \boxed{\text{v1:int}}))) \\
&=^{38,35} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{=} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, tr_N(\texttt{self.employee}, \boxed{\text{v2:E}}) \wedge \\
&\quad \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, tr_N(\texttt{self.employee}, \boxed{\text{v3:E}}) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v4:int}})) \wedge \exists(\boxed{\text{1500=v1:int}}))) \\
&=^{2\times 37} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{=} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{v5:P}} \xrightarrow{e} \boxed{\text{v2:E}}, tr_N(\texttt{self}, \boxed{\text{v5:P}})) \wedge \\
&\quad \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \exists(\boxed{\text{v6:P}} \xrightarrow{e} \boxed{\text{v3:E}}, tr_N(\texttt{self}, \boxed{\text{v6:P}})) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v4:int}})) \wedge \\
&\quad \exists(\boxed{\text{1500=v1:int}}))) \\
&=^{2\times 34} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{=} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{v5:P}} \xrightarrow{e} \boxed{\text{v2:E}}, \exists(\boxed{\text{self=v5:P}})) \wedge \\
&\quad \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \exists(\boxed{\text{v6:P}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{self=v6:P}})) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v4:int}})) \wedge \\
&\quad \exists(\boxed{\text{1500=v1:int}}))) \\
&=^{2\times E4b} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{=} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v2:E}}) \wedge \\
&\quad \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v3:E}}) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v4:int}})) \wedge \exists(\boxed{\text{1500=v1:int}}))) \\
&=^{E2} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{=} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v2:E}}) \wedge \\
&\quad \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \forall(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v4:int}}))) \wedge \exists(\boxed{\text{1500=v1:int}}))) \\
&=^{E4a} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{=} \boxed{\text{1500:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v2:E}}) \wedge \\
&\quad \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \forall(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v4:int}})))))
\end{aligned}
$$

Example 18. `context Project inv: self.employee.salary->max() < self.department.budget`

$$
\begin{aligned}
&=^{1} \forall(\boxed{\text{self:P}}, tr_E(\texttt{self.employee.salary->max() < self.department.budget})) \\
&=^{14} \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{<} \boxed{\text{v1:int}}, tr_N(\texttt{self.employee.salary->max()}, \boxed{\text{v:int}}) \wedge \\
&\quad tr_N(\texttt{self.department.budget}, \boxed{\text{v1:int}})))
\end{aligned}
$$

$$
\begin{aligned}
=^{39,37}\ & \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, tr_N(\texttt{self.employee}, \boxed{\text{v2:E}})\wedge \\
& \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, tr_N(\texttt{self.employee}, \boxed{\text{v3:E}}) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\geq} \boxed{\text{v4:int}}))\wedge \\
& \exists(\boxed{\text{v5:D}} \xrightarrow{b} \boxed{\text{v1:int}}, tr_N(\texttt{self.department}, \boxed{\text{v5:D}}))))\\
=^{3\times 37}\ & \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{v6:P}} \xrightarrow{e} \boxed{\text{v2:E}}, tr_N(\texttt{self}, \boxed{\text{v6:P}}))\wedge \\
& \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \exists(\boxed{\text{v7:P}} \xrightarrow{e} \boxed{\text{v3:E}}, tr_N(\texttt{self}, \boxed{\text{v7:P}})) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\geq} \boxed{\text{v4:int}}))\wedge \\
& \exists(\boxed{\text{v5:D}} \xrightarrow{b} \boxed{\text{v1:int}}, \exists(\boxed{\text{v8:P}} \xrightarrow{d} \boxed{\text{v5:D}}, tr_N(\texttt{self}, \boxed{\text{v8:P}})))))\\
=^{3\times 34}\ & \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{v6:P}} \xrightarrow{e} \boxed{\text{v2:E}}, \exists(\boxed{\text{self=v6:P}}))\wedge \\
& \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \exists(\boxed{\text{v7:P}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{self=v7:P}})) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\geq} \boxed{\text{v4:int}}))\wedge \\
& \exists(\boxed{\text{v5:D}} \xrightarrow{b} \boxed{\text{v1:int}}, \exists(\boxed{\text{v8:P}} \xrightarrow{d} \boxed{\text{v5:D}}, \exists(\boxed{\text{self=v8:P}})))))\\
=^{3\times E4b}\ & \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v2:E}})\wedge \\
& \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v3:E}}) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\geq} \boxed{\text{v4:int}}))\wedge \\
& \exists(\boxed{\text{v5:D}} \xrightarrow{b} \boxed{\text{v1:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v5:D}}))))\\
=^{E1}\ & \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v2:E}})\wedge \\
& \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v3:E}}) \implies \exists(\boxed{\text{v:int}} \xrightarrow{\geq} \boxed{\text{v4:int}}))\wedge \\
& \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v5:D}} \xrightarrow{b} \boxed{\text{v1:int}})))\\
=^{E2}\ & \forall(\boxed{\text{self:P}}, \exists(\boxed{\text{v:int}} \xrightarrow{\leq} \boxed{\text{v1:int}}, \exists(\boxed{\text{v2:E}} \xrightarrow{s} \boxed{\text{v:int}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v2:E}})\wedge \\
& \forall(\boxed{\text{v3:E}} \xrightarrow{s} \boxed{\text{v4:int}}, \forall(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{v:int}} \xrightarrow{\geq} \boxed{\text{v4:int}})))\wedge \\
& \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v5:D}} \xrightarrow{b} \boxed{\text{v1:int}})))
\end{aligned}
$$

Example 19. `context Project inv: self.employee = self.department.employee`

$$
\begin{aligned}
=^{1}\ & \forall(\boxed{\text{self:E}}, tr_E(\texttt{self.employee = self.department.employee}))\\
=^{12}\ & \forall(\boxed{\text{self:E}}, \forall(\boxed{\text{v1:E}}, \exists(\boxed{\text{v1:E}} \xrightarrow{=} \boxed{\text{v2:E}}, tr_S(\texttt{self.employee}, \boxed{\text{v1:E}})\\
& \wedge tr_S(\texttt{self.department.employee}, \boxed{\text{v2:E}}))\wedge \\
& \forall(\boxed{\text{v3:E}}, \exists(\boxed{\text{v3:E}} \xrightarrow{=} \boxed{\text{v4:E}}, tr_S(\texttt{self.department.employee}, \boxed{\text{v3:E}})\\
& \wedge tr_S(\texttt{self.employee}, \boxed{\text{v4:E}}))))\\
=^{4\times 40}\ & \forall(\boxed{\text{self:E}}, \forall(\boxed{\text{v1:E}}, \exists(\boxed{\text{v1:E}} \xrightarrow{=} \boxed{\text{v2:E}}, \exists(\boxed{\text{v5:P}} \xrightarrow{e} \boxed{\text{v1:E}}, tr_N(\texttt{self}, \boxed{\text{v5:P}}))\wedge \\
& \exists(\boxed{\text{v6:D}} \xrightarrow{e} \boxed{\text{v2:E}}, tr_N(\texttt{self.department}, \boxed{\text{v6:D}})))\wedge \\
& \forall(\boxed{\text{v3:E}}, \exists(\boxed{\text{v3:E}} \xrightarrow{=} \boxed{\text{v4:E}}, \exists(\boxed{\text{v7:D}} \xrightarrow{e} \boxed{\text{v3:E}}, tr_N(\texttt{self.department}, \boxed{\text{v7:D}}))\wedge \\
& \exists(\boxed{\text{v8:P}} \xrightarrow{e} \boxed{\text{v4:E}}, tr_N(\texttt{self}, \boxed{\text{v8:P}})))))\\
=^{2\times 37}\ & \forall(\boxed{\text{self:E}}, \forall(\boxed{\text{v1:E}}, \exists(\boxed{\text{v1:E}} \xrightarrow{=} \boxed{\text{v2:E}}, \exists(\boxed{\text{v5:P}} \xrightarrow{e} \boxed{\text{v1:E}}, tr_N(\texttt{self}, \boxed{\text{v5:P}}))\wedge \\
& \exists(\boxed{\text{v6:D}} \xrightarrow{e} \boxed{\text{v2:E}}, \exists(\boxed{\text{v9:P}} \xrightarrow{d} \boxed{\text{v6:D}}, tr_N(\texttt{self}, \boxed{\text{v9:P}})))))\wedge \\
& \forall(\boxed{\text{v3:E}}, \exists(\boxed{\text{v3:E}} \xrightarrow{=} \boxed{\text{v4:E}}, \exists(\boxed{\text{v7:D}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{v10:P}} \xrightarrow{d} \boxed{\text{v7:D}}, tr_N(\texttt{self}, \boxed{\text{v10:P}})))\wedge \\
& \exists(\boxed{\text{v8:P}} \xrightarrow{e} \boxed{\text{v4:E}}, tr_N(\texttt{self}, \boxed{\text{v8:P}})))))\\
=^{4\times 34}\ & \forall(\boxed{\text{self:E}}, \forall(\boxed{\text{v1:E}}, \exists(\boxed{\text{v1:E}} \xrightarrow{=} \boxed{\text{v2:E}}, \exists(\boxed{\text{v5:P}} \xrightarrow{e} \boxed{\text{v1:E}}, \exists(\boxed{\text{self=v5:P}}))\wedge
\end{aligned}
$$

$$
\begin{aligned}
&\exists(\boxed{\text{v6:D}} \xrightarrow{e} \boxed{\text{v2:E}}, \exists(\boxed{\text{v9:P}} \xrightarrow{d} \boxed{\text{v6:D}}, \exists(\boxed{\text{self=v9:P}})))))\wedge \\
&\forall(\boxed{\text{v3:E}}, \exists(\boxed{\text{v3:E}} \xrightarrow{=} \boxed{\text{v4:E}}, \exists(\boxed{\text{v7:D}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{v10:P}} \xrightarrow{d} \boxed{\text{v7:D}}, \exists(\boxed{\text{self=v10:P}})))\wedge \\
&\exists(\boxed{\text{v8:P}} \xrightarrow{e} \boxed{\text{v4:E}}, \exists(\boxed{\text{self=v8:P}})))) \\
=^{4\times E4a}\ &\forall(\boxed{\text{self:E}}, \forall(\boxed{\text{v1:E}}, \exists(\boxed{\text{v1:E}} \xrightarrow{=} \boxed{\text{v2:E}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v1:E}})\wedge \\
&\exists(\boxed{\text{v6:D}} \xrightarrow{e} \boxed{\text{v2:E}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v6:D}})))\wedge \\
&\forall(\boxed{\text{v3:E}}, \exists(\boxed{\text{v3:E}} \xrightarrow{=} \boxed{\text{v4:E}}, \exists(\boxed{\text{v7:D}} \xrightarrow{e} \boxed{\text{v3:E}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v7:D}}))\wedge \\
&\exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v4:E}})))) \\
=^{2\times E1a}\ &\forall(\boxed{\text{self:E}}, \forall(\boxed{\text{v1:E}}, \exists(\boxed{\text{v1:E}} \xrightarrow{=} \boxed{\text{v2:E}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v1:E}})\wedge \\
&\exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v6:D}} \xrightarrow{e} \boxed{\text{v2:E}})))\wedge \\
&\forall(\boxed{\text{v3:E}}, \exists(\boxed{\text{v3:E}} \xrightarrow{=} \boxed{\text{v4:E}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v7:D}} \xrightarrow{e} \boxed{\text{v3:E}})\wedge \\
&\exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v4:E}})))) \\
=^{2\times E3a}\ &\forall(\boxed{\text{self:E}}, \forall(\boxed{\text{v1:E}}, \exists(\boxed{\text{self:P}} \xrightarrow{e} \boxed{\text{v1:E}} \xrightarrow{=} \boxed{\text{v2:E}} \xleftarrow{e} \boxed{\text{v6:D}} \xleftarrow{d} \boxed{\text{self:P}}))\wedge \\
&\forall(\boxed{\text{v3:E}}, \exists(\boxed{\text{self:P}} \xrightarrow{d} \boxed{\text{v7:D}} \xrightarrow{e} \boxed{\text{v3:E}} \xrightarrow{=} \boxed{\text{v4:E}} \xleftarrow{e} \boxed{\text{self:P}})))
\end{aligned}
$$

D.3 Implication

Radke et al. have defined the implication transformation rule in transformation rule vii as:

$$tr_E(\texttt{expr1 implies expr2}) \equiv \neg tr_E(\texttt{expr1}) \vee \texttt{expr2}$$

But if we take a look at, for example, transformation rule xi, which is defined as:

$$tr_E(\texttt{expr1->includesAll(expr2)}) \equiv \forall(\boxed{\text{v:T}}, tr_S(\texttt{expr2}, \boxed{\text{v:T}}) \implies tr_S(\texttt{expr1}, \boxed{\text{v:T}}))$$

You see that transformation rule xi creates a logical implication, which is an inconsistency, therefore it would be more clear if transformation rule vii also creates a logical implication, this will result in the following definition of the implication transformation rule:

$$tr_E(\texttt{expr1 implies expr1}) \equiv tr_E(\texttt{expr1}) \implies tr_E(\texttt{expr2})$$

The next question is, is it possible to simplify such an implication and if so, how is this possible. For this answer we take a look at the second elaboration of example 6 in [11]. We will give the last simplification step below and omit the rest of the elaboration, because there is where the magic happens.

$$
\begin{aligned}
&\forall(\boxed{\text{self:PN}}, \forall(\boxed{\text{t:Tr}}, \exists(\boxed{\text{u:PN}}, \exists(\boxed{\text{u=self:PN}}) \wedge \exists(\boxed{\text{u:Pn}} \xrightarrow{tr} \boxed{\text{t:Tr}})) \implies \\
&\exists(\boxed{\text{v1:PTArc}}, \exists(\boxed{\text{w1:Tr}}, \exists(\boxed{\text{w1=t:Tr}}) \wedge \exists(\boxed{\text{w1:Tr}} \xrightarrow{preArc} \boxed{\text{v1:PTArc}}))) \vee \\
&\exists(\boxed{\text{v2:TPArc}}, \exists(\boxed{\text{w2:Tr}}, \exists(\boxed{\text{w2=t:Tr}}) \wedge \exists(\boxed{\text{w2:Tr}} \xrightarrow{postArc} \boxed{\text{v2:TPArc}})))) \\
=^{RE3}\ &\forall(\boxed{\text{self:PN}} \xrightarrow{tr} \boxed{\text{t:Tr}}, \exists(\boxed{\text{t:Tr}} \xrightarrow{preArc} \boxed{\text{v1:PTArc}}) \vee \exists(\boxed{\text{t:Tr}} \xrightarrow{postArc} \boxed{\text{v2:TPArc}}))
\end{aligned}
$$

In this simplification step Radke et al. indicate to use only E3. It is easy to see that they have omitted quite some intermediate steps, including the step where you could see what happens

with the implication. Below we will recreate the intermediate steps such that we are only left with the implication:

$$\begin{aligned}
&\forall(\boxed{\text{self:PN}},\ \forall(\boxed{\text{t:Tr}},\ \exists(\boxed{\text{u:PN}},\ \exists(\boxed{\text{u=self:PN}})\ \wedge \exists(\boxed{\text{u:PN}} \xrightarrow{tr} \boxed{\text{t:Tr}}))\ \Longrightarrow \\
&\exists(\boxed{\text{v1:PTArc}}, \exists(\boxed{\text{w1:Tr}}, \exists(\boxed{\text{w1=t:Tr}}) \wedge \exists(\boxed{\text{w1:Tr}} \xrightarrow{preArc} \boxed{\text{v1:PTArc}})))\vee \\
&\exists(\boxed{\text{v2:TPArc}}, \exists(\boxed{\text{w2:Tr}}, \exists(\boxed{\text{w2=t:Tr}}) \wedge \exists(\boxed{\text{w2:Tr}} \xrightarrow{postArc} \boxed{\text{v2:TPArc}}))))) \\
=^{RE3}\ &\forall(\boxed{\text{self:PN}},\ \forall(\boxed{\text{t:Tr}},\ \exists(\boxed{\text{self:PN}}, \exists(\boxed{\text{self:PN}} \xrightarrow{tr} \boxed{\text{t:Tr}}))\ \Longrightarrow \\
&\exists(\boxed{\text{v1:PTArc}}, \exists(\boxed{\text{t:Tr}}, \exists(\boxed{\text{t:Tr}} \xrightarrow{preArc} \boxed{\text{v1:PTArc}})))\vee \\
&\exists(\boxed{\text{v2:TPArc}}, \exists(\boxed{\text{t:Tr}}, \exists(\boxed{\text{t:Tr}} \xrightarrow{postArc} \boxed{\text{v2:TPArc}}))))) \\
=^{RE1c}\ &\forall(\boxed{\text{self:PN}}\ \boxed{\text{t:Tr}}, \exists(\boxed{\text{self:PN}} \xrightarrow{tr} \boxed{\text{t:Tr}})\ \Longrightarrow\ \exists(\boxed{\text{t:Tr}} \xrightarrow{preArc} \boxed{\text{v1:PTArc}})\vee \\
&\exists(\boxed{\text{t:Tr}} \xrightarrow{postArc} \boxed{\text{v2:TPArc}}))
\end{aligned}$$

If we compare the result of the above elaboration with the result in the paper of Radke et al. we can only guess what is happening since the paper does not tell us at all. This comes down to the following comparison:

$$\begin{aligned}
&\exists(\boxed{\text{self:PN}} \xrightarrow{tr} \boxed{\text{t:Tr}})\ \Longrightarrow\ \exists(\boxed{\text{t:Tr}} \xrightarrow{preArc} \boxed{\text{v1:PTArc}}) \vee \exists(\boxed{\text{t:Tr}} \xrightarrow{postArc} \boxed{\text{v2:TPArc}}) \\
=^{?}\ &\forall(\boxed{\text{self:PN}} \xrightarrow{tr} \boxed{\text{t:Tr}}, \exists(\boxed{\text{t:Tr}} \xrightarrow{preArc} \boxed{\text{v1:PTArc}}) \vee \exists(\boxed{\text{t:Tr}} \xrightarrow{postArc} \boxed{\text{v2:TPArc}}))
\end{aligned}$$

If we generalise what we see that is happening we can make the following assumption; Let A and B be graphs such that we are still talking about lax condition.

$$\exists(A)\ \Longrightarrow\ \exists(B) = \forall(A, \exists(B))$$

We can read the left lax condition as; if there exists a graph A then there exists a graph B. The right lax condition reads as; for all graph A's there exist a graph B. If we look further into the right lax condition you can also read it as; for all graph A's, if they exist, there exists a graph B, which can be read as; if there exists a graph A there exists a graph B. This sounds correct, but we do not have the correctness proof. To make sure that this works also in other cases we will try with another elaboration in Example 20. In this example we use EB for EditedBook, CE for ConferenceEdition and py for publicationYear.

Example 20. `context EB inv: self.CE->notEmpty() implies self.py >= self.CE.year`

$$\begin{aligned}
=^{\text{i}}\ &\forall(\boxed{\text{self:EB}}, tr_E(\texttt{self.CE->notEmpty() implies self.py >= self.CE.year})) \\
=^{\text{vii}}\ &\forall(\boxed{\text{self:EB}}, tr_E(\texttt{self.CE->notEmpty()})\ \Longrightarrow\ tr_E(\texttt{self.py >= self.CE.year})) \\
=^{\text{xii}}\ &\forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{v:CE}}, tr_S(\texttt{self.CE}, \boxed{\text{v:CE}}))\ \Longrightarrow\ tr_E(\texttt{self.py >= self.CE.year})) \\
=^{\text{xviii}}\ &\forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{v:CE}}, tr_S(\texttt{self.CE}, \boxed{\text{v:CE}}))\ \Longrightarrow\ \exists(\boxed{\text{v1:EB}} \xrightarrow{py} \boxed{\text{v2:int}} \xrightarrow{\geq} \boxed{\text{x:int}}\ \boxed{\begin{array}{c}\text{v2:CE}\\ year = x\end{array}}, \\
&tr_N(\texttt{self}, \boxed{\begin{array}{c}\text{v1:EB}\\ py \geq x\end{array}}) \wedge tr_N(\texttt{self.CE}, \boxed{\begin{array}{c}\text{v2:CE}\\ year = x\end{array}}))) \\
=^{2\times\text{xxiii}}\ &\forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{v:CE}}, \exists(\boxed{\text{v3:EB}} \xrightarrow{cE} \boxed{\text{v:CE}}, tr_N(\texttt{self}, \boxed{\text{v3:EB}})))\ \Longrightarrow \\
&\exists(\boxed{\begin{array}{c}\text{v1:EB}\\ py \geq x\end{array}}\ \boxed{\begin{array}{c}\text{v2:CE}\\ year = x\end{array}}, tr_N(\texttt{self}, \boxed{\begin{array}{c}\text{v1:EB}\\ py \geq x\end{array}}) \wedge \exists(\boxed{\text{v4:EB}} \xrightarrow{cE} \boxed{\begin{array}{c}\text{v2:CE}\\ year = x\end{array}}, tr_N(\texttt{self}, \boxed{\text{v4:EB}}))))
\end{aligned}$$

$$
\begin{aligned}
&=^{3\times xxii} \forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{v:CE}}, \exists(\boxed{\text{v3:EB}} \xrightarrow{cE} \boxed{\text{v:CE}}, \exists(\boxed{\text{v3=self:EB}}))) \implies \\
&\qquad \exists(\boxed{\begin{array}{l}\text{v1:EB}\\ py \geq x\end{array}} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}}, \exists(, \boxed{\begin{array}{l}\text{v1=self:EB}\\ py \geq x\end{array}}) \wedge \exists(\boxed{\text{v4:EB}} \xrightarrow{cE} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}}, \exists(\boxed{\text{v4=self:EB}})))) \\
&=^{3\times E3} \forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{v:CE}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}})) \implies \\
&\qquad \exists(\boxed{\begin{array}{l}\text{self:EB}\\ py \geq x\end{array}} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}}))) \\
&=^{RE1a} \forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{v:CE}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}})) \implies \exists(\boxed{\begin{array}{l}\text{self:EB}\\ py \geq x\end{array}} \xrightarrow{cE} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}})) \\
&=^{RE1c} \forall(\boxed{\text{self:EB}}, \exists(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}}) \implies \exists(\boxed{\begin{array}{l}\text{self:EB}\\ py \geq x\end{array}} \xrightarrow{cE} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}})) \\
&=^{E_{Imp}} \forall(\boxed{\text{self:EB}}, \forall(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}}, \exists(\boxed{\begin{array}{l}\text{self:EB}\\ py \geq x\end{array}} \xrightarrow{cE} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}}))) \\
&=^{RE1} \forall(\boxed{\text{self:EB}} \xrightarrow{cE} \boxed{\text{v:CE}}, \exists(\boxed{\begin{array}{l}\text{self:EB}\\ py \geq x\end{array}} \xrightarrow{cE} \boxed{\begin{array}{l}\text{v2:CE}\\ year = x\end{array}}))
\end{aligned}
$$

This reads as, for all EditedBook that have an ConferenceEdition, there exists an Conference edition for which the publicationYear of the EditedBook is greater than or equal to the year of ConferenceEdition. This is exactly what we want, because we do not match injectively. If we did match injectively, the nodes v and v2 were not allowed to be the same node, in which the rule would never apply with a model with one EditedBook and one ConferenceEdition. Not matching injectively does fix this problem. So the definition of transformation rule vii will be changed such that it will uses the logical implication and a new equivalence rule will be added to be able to simplify an implication. This new equivalence rule can be found in Definition 5.

Definition 5. Let A and B be graphs, then the following equivalence applies:

$$\exists(A) \implies \exists(B) \equiv \forall(A, \exists(B))$$

D.4 Navigation transformations

Radke et al. have defined the navigation transformation rules xxiii up to xxvi (two rules for both tr_N and tr_S) as:

$tr_N(\texttt{expr.role}, \boxed{\text{v:T}}) \equiv$

$\exists(\boxed{\text{v‘:T‘}} \xrightarrow{role} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v‘:T‘}}))$ if $T‘ \notin clan(T)$

$\exists(\boxed{\text{v‘:T‘}} \xrightarrow{role} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v‘:T‘}})) \vee \exists(\boxed{\text{v:T}} \hookleftarrow \text{role}, tr_N(\texttt{expr}, \boxed{\text{v:T}}))$ else

Since we do not match injectively anymore, these rules can be simplified. The check whether $T‘$ is in the clan of T is not necessary because the self loop option can be removed. If the self loop is needed, then $v‘ = v$ and the self loop is still possible. So the new definition of the navigation transformation rule can be found in Definition 6.

Definition 6. $tr_N(\texttt{expr.role}, \boxed{\text{v:T}}) \equiv$

$$\exists(\boxed{\text{v‘:T‘}} \xrightarrow{role} \boxed{\text{v:T}}, tr_N(\texttt{expr}, \boxed{\text{v‘:T‘}}))$$

D.5 Size

This is the one operation that increases in difficulty because of not matching injectively. Radke et al. have defined the transformation rule of the `size` operation in rule xiv as:

$$tr_E(\texttt{expr->size() >= n}) \equiv \exists(\boxed{\text{v}_1\text{:T}}...\boxed{\text{v}_n\text{:T}}, \bigwedge_{i=1}^{n} tr_S(\texttt{expr}, \boxed{\text{v}_i\text{:T}}))$$

This transformation rule did work because of the injective property, now that this property drops out, we have to add the restriction that every node, $\boxed{\text{v}_i\text{:T}}$ is not equal to every other node $\boxed{\text{v}_j\text{:T}}$ where $i \neq j$. This brings us to the new definition of this operation, which can be found in Definition 7.

Definition 7. $tr_E(\texttt{expr->size() >= n}) \equiv$

$$\exists(\boxed{\text{v}_1\text{:T}} \ ... \ \boxed{\text{v}_n\text{:T}}, \bigwedge_{i\in\{1..n\}} tr_S(\texttt{expr}, \boxed{\text{v}_i\text{:T}}) \bigwedge_{j=\{1..n\}}^{i\neq j} \boxed{\text{v}_i\text{:T}} \neq \boxed{\text{v}_j\text{:T}})$$

D.6 forAll

Radke et al. have defined one transformation rule for the `forAll` operation, in which they take one variable into account. This is transformation rule x and is defined as:

$$tr_E(\texttt{expr1->forAll(v:T|expr2)}) \equiv \forall(\boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v:T}}) \implies tr_E(\texttt{expr2}))$$

In Example 21 we have given two examples of OCL constraints based on the `forAll` operation. The defined transformation rule is able to handle the first constraint, because this one uses only one variable, but the second constraint is not possible. To be able to handle the second constraint too, the defined transformation rule has to be extended and this results in Definition 8.

Example 21. In this example we give two OCL constraints in which we have a `forAll` operation. In`forAll1` we have one variable and in `forAll2` we have two variables.

```
context Person inv forAll1: self.publication->forAll(p:Publication | p.title = 'test')
context Person inv forAll2: self.publication->forAll(p1,p2:Publication | p1.year = p2.year)
```

Definition 8. $tr_E(\texttt{expr1->forAll(v:T|expr2)}) \equiv$

$$\forall(\boxed{\text{v}_1\text{:T}} \ ... \ \boxed{\text{v}_n\text{:T}}, (tr_S(\texttt{expr1}, \boxed{\text{v}_1\text{:T}}) \wedge \ ... \ \wedge tr_S(\texttt{expr1}, \boxed{\text{v}_n\text{:T}})) \implies tr_E(\texttt{expr2}))$$

D.7 isUnique

The operation `isUnique` is not defined by Radke et al. This is one of the operations we have extended our OCL fragment with. In [21], Franconi et al. have defined the following equality on logic level to the `isUnique` operation:

`expr->isUnique(attr)` $\equiv$ `expr->forall(v1, v2 | v1 <> v2 implies v1.attr <> v2.attr)`

Within our definitions we included transformation definitions for the `forAll`, `implies` and the not equal operator (`<>`), so based on this logical equality we can define our transformation rule for `isUnique` as:

`expr->isUnique(attr)` $\equiv tr_E($`expr->forAll(v1,v2:T | v1 <> v2` $\implies$ `v1.attr <> v2.attr`$)$

D.8 Union, Intersection and Minus

The operations `union`, `intersection` and `minus`(−) are operations on sets that result in a new set. In OCL it is possible to create sets where the elements of the set do not have one common parent. For example, if we use the DBLP class diagram that is given in Appendix E, the following OCL constraint is valid:
`context ConferenceEdition inv: self.editedBook->union(self.JournalIssue)->notEmpty()`

In this chapter we will elaborate this OCL constraint, first in Example 22 we show what happens if we follow the transformation rules of Radke et al. if the types do not have one common parent.

Example 22. `context CE inv: self.editedBook->union(self.journalIssue)->notEmpty()`

$$
\begin{aligned}
&=^{\text{i}} \forall(\boxed{\text{self:CE}}, tr_E(\texttt{self.editedBook->union(self.journalIssue)->notEmpty()})) \\
&=^{\text{xiii}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:?}}, tr_S(\texttt{self.editedBook->union(self.journalIssue)}, \boxed{\text{v:?}})))
\end{aligned}
$$

The problem starts when we try to apply transformation rule xiii, the `notEmpty` transformation rule creates an existential lax condition with a node of the type of its expression. In this case that expression is `self.editedBook->union(self.JournalIssue)` and its type should be a generic type that is the parent of all types. If we look at Java there the class `Object` is the parent of every class. This kind of construction is what we need. So we define the type of a set as the most specific parent class of all elements of that set. If there is none, then the type is `Object`, just like in Java. With this in mind we continue our elaboration in Example 23.

Example 23. `context CE inv: self.editedBook->union(self.journalIssue)->notEmpty()`

$$
\begin{aligned}
&=^{\text{i}} \forall(\boxed{\text{self:CE}}, tr_E(\texttt{self.editedBook->union(self.journalIssue)->notEmpty()})) \\
&=^{\text{xiii}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, tr_S(\texttt{self.editedBook->union(self.journalIssue)}))) \\
&=^{\text{xxxi}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, tr_S(\texttt{self.editedBook}, \boxed{\text{v:Object}}) \vee tr_S(\texttt{self.journalIssue}, \boxed{\text{v:Object}}))) \\
&=^{\text{xxxvi}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, tr_N(\texttt{self.editedBook}, \boxed{\text{v:Object}}) \vee tr_N(\texttt{self.journalIssue}, \boxed{\text{v:Object}}))) \\
&=^{2\times\text{xxiii}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, \exists(\boxed{\text{v1:CE}} \xrightarrow{conferenceEdition} \boxed{\text{v:Object}}, tr_N(\texttt{self}, \boxed{\text{v1:CE}})) \vee \\
&\qquad \exists(\boxed{\text{v2:CE}} \xrightarrow{journalIssue} \boxed{\text{v:Object}}, tr_N(\texttt{self}, \boxed{\text{v2:CE}}))))
\end{aligned}
$$

At this point we have an invalid solution again, since the associations `conferenceEdition` and `journalIssue` from `ConferenceEdition` to `Object` do not exist. To solve this we have to change the transformation definitions of `union, intersection` and `minus` such that they create the actual type of the expression that is equal to the node $\boxed{\text{v:Object}}$. This action can be compared to a type cast in Java. This brings us to the new definitions of the three operations in Definition 9.

Definition 9. The new definitions of the operations `union`, `intersection` and `minus`(−) are:

$$
\begin{aligned}
tr_S(\texttt{expr1->union(expr2)}, \boxed{\text{v:T}}) \equiv\ & \exists(\boxed{\text{v‘:T‘}} \overset{=}{\text{---}} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v‘:T‘}})) \vee \\
& \exists(\boxed{\text{v“:T“}} \overset{=}{\text{---}} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v“:T“}})) \\
tr_S(\texttt{expr1->intersection(expr2)}, \boxed{\text{v:T}}) \equiv\ & \exists(\boxed{\text{v‘:T‘}} \overset{=}{\text{---}} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v‘:T‘}})) \wedge \\
& \exists(\boxed{\text{v“:T“}} \overset{=}{\text{---}} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v“:T“}})) \\
tr_S(\texttt{expr1 - expr2}, \boxed{\text{v:T}}) \equiv\ & \exists(\boxed{\text{v‘:T‘}} \overset{=}{\text{---}} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v‘:T‘}})) \wedge
\end{aligned}
$$

$$\neg\exists(\boxed{\text{v“:T“}} \xrightarrow{=} \boxed{\text{v:T}}, tr_S(\texttt{expr1}, \boxed{\text{v“:T“}}))$$

If we apply the new definition of the `union` transformation to the given example, instead of the definition of Radke et al. we can continue and complete the elaboration in Example 24. In this elaboration CE is used for ConferenceEdition, EB for EditedBook and JI for JournalIssue.

Example 24. `context CE inv: self.editedBook->union(self.journalIssue)->notEmpty()`

$$\begin{aligned}
&=^{\text{i}} \forall(\boxed{\text{self:CE}}, tr_E(\texttt{self.editedBook->union(self.journalIssue)->notEmpty()}))\\
&=^{\text{xiii}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, tr_S(\texttt{self.editedBook->union(self.journalIssue)})))\\
&=^{\text{union def.}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, \exists(\boxed{\text{v1:EB}} \xrightarrow{=} \boxed{\text{v:Object}}, tr_S(\texttt{self.editedBook}, \boxed{\text{v1:EB}}))\vee\\
&\quad \exists(\boxed{\text{v2:JI}} \xrightarrow{=} \boxed{\text{v:Object}}, tr_S(\texttt{self.journalIssue}, \boxed{\text{v2:JI}}))))\\
&=^{\text{xxxvi}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, \exists(\boxed{\text{v1:EB}} \xrightarrow{=} \boxed{\text{v:Object}}, tr_N(\texttt{self.editedBook}, \boxed{\text{v1:EB}}))\vee\\
&\quad \exists(\boxed{\text{v2:JI}} \xrightarrow{=} \boxed{\text{v:Object}}, tr_N(\texttt{self.journalIssue}, \boxed{\text{v2:JI}}))))\\
&=^{2\times\text{xxiii}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, \exists(\boxed{\text{v1:EB}} \xrightarrow{=} \boxed{\text{v:Object}}, \exists(\boxed{\text{v3:CE}} \xrightarrow{eB} \boxed{\text{v1:EB}}, tr_N(\texttt{self}, \boxed{\text{v3:CE}})))\vee\\
&\quad \exists(\boxed{\text{v2:JI}} \xrightarrow{=} \boxed{\text{v:Object}}, \exists(\boxed{\text{v4:CE}} \xrightarrow{jI} \boxed{\text{v2:JI}}, tr_N(\texttt{self}, \boxed{\text{v4:CE}})))))\\
&=^{2\times\text{xxii}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, \exists(\boxed{\text{v1:EB}} \xrightarrow{=} \boxed{\text{v:Object}}, \exists(\boxed{\text{v3:CE}} \xrightarrow{eB} \boxed{\text{v1:EB}}, \exists(\boxed{\text{self=v3:CE}})))\vee\\
&\quad \exists(\boxed{\text{v2:JI}} \xrightarrow{=} \boxed{\text{v:Object}}, \exists(\boxed{\text{v4:CE}} \xrightarrow{jI} \boxed{\text{v2:JI}}, \exists(\boxed{\text{self=v4:CE}})))))\\
&=^{2\times\text{E4b}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, \exists(\boxed{\text{v1:EB}} \xrightarrow{=} \boxed{\text{v:Object}}, \exists(\boxed{\text{self:CE}} \xrightarrow{eB} \boxed{\text{v1:EB}}))\vee\\
&\quad \exists(\boxed{\text{v2:JI}} \xrightarrow{=} \boxed{\text{v:Object}}, \exists(\boxed{\text{self:CE}} \xrightarrow{jI} \boxed{\text{v2:JI}}))))\\
&=^{2\times\text{E1}} \forall(\boxed{\text{self:CE}}, \exists(\boxed{\text{v:Object}}, \exists(\boxed{\text{self:CE}} \xrightarrow{eB} \boxed{\text{v1:EB}} \xrightarrow{=} \boxed{\text{v:Object}}))\vee\\
&\quad \exists(\boxed{\text{self:CE}} \xrightarrow{jI} \boxed{\text{v2:JI}} \xrightarrow{=} \boxed{\text{v:Object}})))
\end{aligned}$$

Appendix E

DBLP class diagram