Towards OntoUML for Software Engineering:
Transformation of OntoUML into Relational Databases

by

Zdeněk Rybola

A dissertation thesis submitted to
the Faculty of Information Technology, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Dissertation degree study programme: Informatics

Prague, August 2017
Supervisor:

doc. Ing. Karel Richta, CSc.
Computer Science and Engineering Department
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Copyright © 2017 Zdeněk Rybola
Abstract and contributions

This dissertation thesis deals with incorporating OntoUML as a conceptual data modelling language into the MDD approach to software engineering. To achieve it, the concepts of OntoUML in the conceptual data modelling context are described and a method of the transformation of the OntoUML conceptual model into its realization in a relational database is proposed.

In particular, the main contributions of the dissertation thesis are as follows:

1. Complex overview of OntoUML and UFO-A concepts, illustrated on simple examples.

2. Example of a complex OntoUML conceptual model containing most of the OntoUML and UFO-A concepts. This model defines the structure of information processed by a library information system.

3. Proposition of the transformation of OntoUML PIM into UML PIM, preserving all the semantics and domain-specific constraints defined by various universal and relation types in the OntoUML PIM model.

4. Proposition of the transformation of the resulting UML PIM into RDB PSM, focusing on the realization of the constraints derived from the original OntoUML PIM to preserve all the semantics and domain-specific constraints.

5. Proposition of the transformation of the resulting RDB PSM into SQL ISM, focusing on possible realizations of the constraints derived from the original OntoUML PIM.

Keywords:
constraints, MDD, OCL, OntoUML, relational database, software engineering, SQL, transformation, UML.
First of all, I would like to express my gratitude to my dissertation thesis supervisor, Dr. Karel Richta. He has been a constant source of encouragement and helped me to initiate my research and finish this thesis.

I would also like to thank Dr. Robert Pergl, who inspired my interest in OntoUML and conceptual modelling and helped me to understand the OntoUML language. I also greatly appreciate that he let me join him in his research on the transformations of OntoUML models into object-oriented UML models, which we continued together. Similarly, I would like to thank Mr. Jiří Mlejnek for his deep knowledge of UML and frequent consultations to various UML patterns. Furthermore, I would like to thank Dr. Michal Valenta for his expertise in relational databases and consultations to various SQL constructs used for realizing the constraints in our approach.

Special thanks go to the staff of the Department of Software Engineering, who maintained a pleasant and flexible environment for my research. I would like to express special thanks to the department management for providing my research with most of the funding.

My research has also been partially supported by the Ministry of Education, Youth, and Sport of the Czech Republic under the research program RVO18000, the grant project of the Czech Grant Agency No. GA201/09/0990 and the grant projects of the Student Grant Comtention of CTU in Prague No. SGS11/087/OHK3/1T/18, SGS12/093/OHK3/1T/18, SGS13/099/OHK3/1T/18 and SGS16/120/OHK3/1T/18.

I would also like to express thanks to Mgr. Naděžda Němcová for her valuable comments and proofreading.

Finally, my greatest thanks go to my family, for their infinite patience and faith that I am really able to finish this thesis successfully and complete my studies.
Dedication

To my little PampEliška
Contents

1 Introduction .......................................................... 1
   1.1 Motivation ....................................................... 2
   1.2 Problem Statement ............................................ 3
   1.3 Related Work and Previous Results .......................... 4
   1.4 Contributions of the Thesis .................................. 5
   1.5 Structure of the Thesis ...................................... 6

2 Background and State of the Art ................................. 7
   2.1 Model Driven Development ..................................... 7
      2.1.1 PIM of Application Data ............................... 11
      2.1.2 PSM of Relational Database ........................... 11
      2.1.3 ISM of Relational Database ........................... 12
   2.2 UML ................................................................... 12
      2.2.1 UML Class Diagram ....................................... 13
      2.2.2 UML Data Modelling Profile ........................... 17
   2.3 OCL .................................................................. 19
   2.4 UML and OCL Tools .............................................. 20
   2.5 Transformation of Conceptual Model into Relational Model ...... 22

3 OntoUML ................................................................. 25
   3.1 Universals, individuals, types and instances .................. 26
   3.2 Identity and Identity Criterion ................................ 28
   3.3 Rigidity .................................................................. 29
   3.4 Generalization and Specialization ............................... 30
   3.5 Substantial Universals .......................................... 30
      3.5.1 Kinds and Subkinds ........................................ 32
      3.5.2 Roles .......................................................... 33
      3.5.3 Phases ........................................................ 33
      3.5.4 Mixins ........................................................ 34
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Moment Universals</td>
<td>36</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Quality</td>
<td>36</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Mode</td>
<td>38</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Relations and Relator</td>
<td>38</td>
</tr>
<tr>
<td>3.7</td>
<td>Whole-Part Relations</td>
<td>39</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Properties of Part-Whole Relations</td>
<td>40</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Functional Complex</td>
<td>43</td>
</tr>
<tr>
<td>3.7.3</td>
<td>Collective</td>
<td>44</td>
</tr>
<tr>
<td>3.7.4</td>
<td>Quantity</td>
<td>46</td>
</tr>
<tr>
<td>3.8</td>
<td>Tools</td>
<td>47</td>
</tr>
<tr>
<td>4</td>
<td>Overview of Our Approach</td>
<td>49</td>
</tr>
<tr>
<td>4.1</td>
<td>Running Example</td>
<td>51</td>
</tr>
<tr>
<td>5</td>
<td>Transformation of OntoUML PIM into UML PIM</td>
<td>59</td>
</tr>
<tr>
<td>5.1</td>
<td>Transformation of OntoUML Universal Types</td>
<td>59</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Kinds and Subkinds</td>
<td>60</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Roles</td>
<td>61</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Phases</td>
<td>62</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Mixins</td>
<td>65</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Moments</td>
<td>65</td>
</tr>
<tr>
<td>5.1.6</td>
<td>Part-Whole Relations</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Discussion</td>
<td>69</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Optimization of Rigid Generalization Sets</td>
<td>70</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Optimization of Phases</td>
<td>72</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Optimization of Roles</td>
<td>73</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Optimization of Relators</td>
<td>74</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Immutable Parts and Wholes</td>
<td>76</td>
</tr>
<tr>
<td>6</td>
<td>Transformation of UML PIM into RDB PSM</td>
<td>77</td>
</tr>
<tr>
<td>6.1</td>
<td>Transformation of UML PIM Constructs</td>
<td>77</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Classes and Attributes</td>
<td>78</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Associations</td>
<td>80</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Generalization</td>
<td>84</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Special Multiplicities</td>
<td>93</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Exclusivity Constraints</td>
<td>96</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Enumeration Constraints</td>
<td>97</td>
</tr>
<tr>
<td>6.1.7</td>
<td>Immutability</td>
<td>98</td>
</tr>
<tr>
<td>6.2</td>
<td>Discussion</td>
<td>103</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Generalization</td>
<td>103</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Mutually Dependent References for One-To-One Associations</td>
<td>106</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Single Table Realization of One-to-One Associations</td>
<td>107</td>
</tr>
</tbody>
</table>
C Additional Examples 213
C.1 Transformation of Part-Whole Relations from OntoUML PIM into UML PIM 213
C.2 Transformation of Generalization from UML PIM into RDB PSM 215
C.2.1 Single Table Realization 215
C.2.2 Referencing Tables Realization 217
## List of Figures

2.1 Models of MDA ........................................... 9
2.2 Example of the transformation of PIM into PSM for XML Schema [1] .......... 10
2.3 Example of a UML class diagram with classes Person and Book ............. 14
2.4 Example of a UML class diagram with an association between classes Book and BookEdition ........................................... 15
2.5 Example of a UML class diagram with the generalization between the Man and Woman subclasses and the Person class .............................. 17
2.6 Example of a relational PSM created using the UML Data Modelling profile . 19

3.1 Example of instantiation in OntoUML ..................................... 27
3.2 Example of instantiation with multiple generalization sets and multiple inheritance ................................................................. 31
3.3 Hierarchy of Substantial universals representing domain types in OntoUML. Inspired by [2, p. 106] and Guizzardi’s presentation at OntoUML course at University of Economics, Prague, September 2011 .......................... 31
3.4 Example of an OntoUML model with Sortal universal types ................. 32
3.5 Example of an OntoUML model with Mixin universals .......................... 35
3.6 Example of an OntoUML model with a Mixin universal. Inspired by [2, p. 113] 36
3.7 Hierarchy of Moment universals. Inspired by [2] ................................. 37
3.8 Example of an OntoUML model with Moment universals .................... 38
3.9 Example of an OntoUML model with Formal and Material relations .......... 40
3.10 Example of an OntoUML model with a shareable and exclusive parts. Inspired by [2, p.162] ................................................................. 41
3.11 Example of an OntoUML model of an aggregate with parts of various optionalities 43
3.12 Example of an OntoUML model of immutable part of a whole ............... 44
3.13 Example of an OntoUML model of collectives .................................. 45
3.14 Example of an OntoUML model of quantities ................................... 47

4.1 Overview of the transformation process ....................................... 50
4.2 OntoUML PIM of the Library - Works ....................................... 52
List of Figures

4.3 OntoUML PIM of the Library - Authorship ........................................... 53
4.4 OntoUML PIM of the Library - Book and periodical editions .................... 54
4.5 OntoUML PIM of the Library - Edition translations and publication .......... 54
4.6 OntoUML PIM of the Library - Copies ................................................. 55
4.7 OntoUML PIM of the Library - Clients .................................................. 56
4.8 OntoUML PIM of the Library - Loans ..................................................... 56
4.9 OntoUML PIM of the Library - Loan and copy states ............................... 57
4.10 OntoUML PIM of the Library - Evidence .............................................. 58

5.1 Excerpt of the UML PIM with transformed Kind and Subkinds .................. 61
5.2 Excerpt of the UML PIM with transformed Roles ..................................... 62
5.3 Excerpt of the UML PIM with transformed Phases with exclusive associations 63
5.4 Excerpt of the UML PIM with transformed Phases with an abstract phase class 65
5.5 Excerpt of the UML PIM with transformed Mixins .................................. 66
5.6 Excerpt of the UML PIM with transformed Quality .................................. 67
5.7 Excerpt of the UML PIM with transformed Modes ................................... 67
5.8 UML PIM with transformed generalization set of Subkinds with empty subclasses 70
5.9 UML PIM with transformed generalization set of Subkinds realized by the discriminator attribute ................................................................. 71
5.10 UML PIM with transformed phase partition realized by the phase attribute .... 73
5.11 UML PIM with transformed Roles optimized into an optional relation between the identity bearer and the truthmaker ........................................... 74
5.12 UML PIM with transformed Relator reduced into optional association between the related classes ................................................................. 75

6.1 Transformation of a PIM class into a database table in the RDB PSM ........... 80
6.2 Decomposition of many-to-many association ........................................... 81
6.3 Transformation of many-to-one relations ............................................... 83
6.4 Transformation of one-to-one relation .................................................... 84
6.5 Transformation of generalization into a single table realization ................... 87
6.6 Transformation of generalization into its realization by separate tables .......... 90
6.7 Transformation of generalization into its realization by referencing tables .... 93
6.8 Transformation of association with special multiplicities .......................... 95
6.9 Transformation of association with mandatory source multiplicity ............... 96
6.10 RDB PSM with transformed exclusive Phases ......................................... 97
6.11 RDB PSM with transformed class Person with an immutable discriminator attribute ................................................................. 99
6.12 RDB PSM with transformed immutable association .................................. 100
6.13 RDB PSM with transformed immutable association restricting the source table 101
6.14 RDB PSM with mutual references realizing a one-to-one association .......... 107
6.15 RDB PSM with a single table realizing a mandatory one-to-one association .... 108
6.16 UML PIM with a one-side optional one-to-one association ....................... 109
6.17 RDB PSM with a single table realizing a one-side optional one-to-one association .... 109
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.18</td>
<td>RDB PSM with the <em>phase partition</em> realized by <em>abstract phase</em> class and referencing tables</td>
<td>111</td>
</tr>
<tr>
<td>6.19</td>
<td>RDB PSM with the optimized <em>phase partition</em> realized by <em>abstract phase</em> class and referencing tables</td>
<td>111</td>
</tr>
<tr>
<td>7.1</td>
<td>RDB PSM with the mutually dependent mandatory references</td>
<td>179</td>
</tr>
<tr>
<td>7.2</td>
<td>RDB PSM with updated model solving the problem of mutually dependent mandatory references</td>
<td>179</td>
</tr>
<tr>
<td>A.1</td>
<td>Structure of the attached CD</td>
<td>203</td>
</tr>
<tr>
<td>B.1</td>
<td>Structure of the attached Running Example document</td>
<td>205</td>
</tr>
<tr>
<td>B.2</td>
<td>OntoUML PIM of the Library</td>
<td>207</td>
</tr>
<tr>
<td>B.3</td>
<td>UML PIM of the Library</td>
<td>209</td>
</tr>
<tr>
<td>B.4</td>
<td>RDB PSM of the Library</td>
<td>211</td>
</tr>
<tr>
<td>C.1</td>
<td>UML PIM with transformed functional whole and its components</td>
<td>213</td>
</tr>
<tr>
<td>C.2</td>
<td>UML PIM with transformed Collectives</td>
<td>214</td>
</tr>
<tr>
<td>C.3</td>
<td>UML PIM with transformed Quantities</td>
<td>214</td>
</tr>
</tbody>
</table>
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Overview of the support of constraints in the most popular RDBMSs</td>
<td>166</td>
</tr>
<tr>
<td>7.2</td>
<td>Overview of the support of triggers in the most popular RDBMSs</td>
<td>167</td>
</tr>
<tr>
<td>7.3</td>
<td>Overview of possible realizations of various constraints in a RDB PSM model</td>
<td>169</td>
</tr>
<tr>
<td>B.1</td>
<td>Library OntoUML PIM - counts of the elements of the model</td>
<td>206</td>
</tr>
<tr>
<td>B.2</td>
<td>Library UML PIM - counts of elements of the model</td>
<td>208</td>
</tr>
<tr>
<td>B.3</td>
<td>Library RDB PSM - counts of elements of the model</td>
<td>210</td>
</tr>
<tr>
<td>B.4</td>
<td>Library SQL ISM - counts of elements of the model</td>
<td>212</td>
</tr>
</tbody>
</table>
# List of Constraints

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example of an OCL invariant for the class <code>Book</code></td>
<td>20</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of an OCL post-condition for the method <code>update()</code> of the class <code>Book</code></td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Exclusivity constraint for the exclusivity of the phases of a <code>Copy</code> defined as</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>OCL invariant for the possible discriminator values</td>
<td>71</td>
</tr>
<tr>
<td>5.3</td>
<td>OCL invariant for the possible phase values</td>
<td>73</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of the OCL invariant for a <code>{disjoint,complete}</code> generalization set realized by a single table</td>
<td>88</td>
</tr>
<tr>
<td>6.2</td>
<td>Example of the OCL invariants for distributed unique constraint</td>
<td>90</td>
</tr>
<tr>
<td>6.3</td>
<td>Example of the OCL invariants for a <code>{complete,disjoint}</code> generalization set realized by related tables</td>
<td>92</td>
</tr>
<tr>
<td>6.4</td>
<td>Example of the OCL invariant for a special multiplicity constraint in RDB PSM</td>
<td>95</td>
</tr>
<tr>
<td>6.5</td>
<td>Example of the OCL invariant for a mandatory multiplicity constraint in RDB PSM</td>
<td>95</td>
</tr>
<tr>
<td>6.6</td>
<td>OCL invariant for the transformed exclusivity constraint</td>
<td>98</td>
</tr>
<tr>
<td>6.7</td>
<td>OCL invariant for the transformed enumeration constraint</td>
<td>99</td>
</tr>
<tr>
<td>6.8</td>
<td>OCL postcondition constraint for the immutability constraint for the target table <code>BOOK EDITION</code></td>
<td>101</td>
</tr>
<tr>
<td>6.9</td>
<td>OCL postcondition constraint for the immutability constraint for the target table <code>BOOK EDITION</code></td>
<td>101</td>
</tr>
<tr>
<td>6.10</td>
<td>OCL precondition constraint for the DELETE operation realizing the immutable meta-property of an association defined at the side of the source class</td>
<td>102</td>
</tr>
<tr>
<td>6.11</td>
<td>OCL postcondition constraint for the UPDATE operation realizing the immutable meta-property of an attribute</td>
<td>103</td>
</tr>
<tr>
<td>6.12</td>
<td>OCL constraint for the mutually dependent references</td>
<td>107</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.13</td>
<td>OCL constraint for the mandatoriness of attributes of the optional class in the single table realization of a <em>one-to-one</em> association</td>
<td>109</td>
</tr>
<tr>
<td>6.14</td>
<td>OCL constraint for the optimized <em>phase partition</em> realized by <em>abstract phase</em> class and referencing tables</td>
<td>112</td>
</tr>
<tr>
<td>C.1</td>
<td>Example of the OCL invariants for a <em>{overlapping,complete}</em> generalization set realized by a single table</td>
<td>215</td>
</tr>
<tr>
<td>C.2</td>
<td>Example of the OCL invariants for a <em>{disjoint,incomplete}</em> generalization set realized by a single table</td>
<td>215</td>
</tr>
<tr>
<td>C.3</td>
<td>Example of the OCL invariants for a <em>{overlapping,incomplete}</em> generalization set realized by a single table</td>
<td>216</td>
</tr>
<tr>
<td>C.4</td>
<td>Example of the OCL invariants for a <em>{complete,overlapping}</em> generalization set realized by related tables</td>
<td>217</td>
</tr>
<tr>
<td>C.5</td>
<td>Example of the OCL invariants for a <em>{incomplete,disjoint}</em> generalization set realized by related tables</td>
<td>217</td>
</tr>
<tr>
<td>C.6</td>
<td>Example of the OCL invariants for a <em>{incomplete,overlapping}</em> generalization set realized by related tables</td>
<td>218</td>
</tr>
</tbody>
</table>
## List of SQLs

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>SQL ISM with the CREATE statements for a table</td>
</tr>
<tr>
<td>7.2</td>
<td>SQL ISM with the ADD CONSTRAINT statement for a FOREIGN KEY constraint</td>
</tr>
<tr>
<td>7.3</td>
<td>SQL ISM with the ADD CONSTRAINT statement for the FOREIGN KEY constraint</td>
</tr>
<tr>
<td>7.4</td>
<td>SQL ISM with the CREATE VIEW statement for the view definition</td>
</tr>
<tr>
<td>7.5</td>
<td>SQL ISM with the CREATE VIEW statement for the updatable view definition</td>
</tr>
<tr>
<td>7.6</td>
<td>Error thrown by the Oracle database when inserting data violating the WHERE clause</td>
</tr>
<tr>
<td>7.7</td>
<td>SQL ISM with the ADD CONSTRAINT statement to define a CHECK constraint</td>
</tr>
<tr>
<td>7.8</td>
<td>Error thrown by the Oracle database when inserting data violating the CHECK constraint</td>
</tr>
<tr>
<td>7.9</td>
<td>Error thrown by the Oracle database when creating a CHECK constraint with subqueries</td>
</tr>
<tr>
<td>7.10</td>
<td>SQL ISM with a trigger definition</td>
</tr>
<tr>
<td>7.11</td>
<td>SQL ISM with the CREATE VIEW statement for the checked updatable database view definition realizing a <em>generalization set constraint</em></td>
</tr>
<tr>
<td>7.12</td>
<td>SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing a <em>generalization set constraint</em></td>
</tr>
<tr>
<td>7.13</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition for a <em>generalization set constraint</em></td>
</tr>
<tr>
<td>7.14</td>
<td>SQL ISM with the CREATE VIEW statement for the database view definition realizing a <em>generalization set constraint</em></td>
</tr>
<tr>
<td>7.15</td>
<td>SQL ISM with the CREATE VIEW statement for the database view definition for a subclass table realizing a <em>generalization set constraint</em></td>
</tr>
<tr>
<td>7.16</td>
<td>SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing a <em>generalization set constraint</em></td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>7.17</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition on the superclass table realizing a generalization set constraint</td>
</tr>
<tr>
<td>7.18</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition for the INSERT operation on a subclass realizing a generalization set constraint</td>
</tr>
<tr>
<td>7.19</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition for the UPDATE operation on a subclass realizing a generalization set constraint</td>
</tr>
<tr>
<td>7.20</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition for the DELETE operation on a subclass realizing a generalization set constraint</td>
</tr>
<tr>
<td>7.21</td>
<td>SQL ISM with the CREATE VIEW statement for the database view definition realizing a distributed unique constraint</td>
</tr>
<tr>
<td>7.22</td>
<td>SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing a distributed unique constraint</td>
</tr>
<tr>
<td>7.23</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing a distributed uniqueness constraint</td>
</tr>
<tr>
<td>7.24</td>
<td>SQL ISM with the CREATE VIEW statement for the view realizing a special multiplicity constraint</td>
</tr>
<tr>
<td>7.25</td>
<td>SQL ISM with the CREATE VIEW statement for the view realizing a mandatory multiplicity constraint</td>
</tr>
<tr>
<td>7.26</td>
<td>SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing special multiplicity constraint</td>
</tr>
<tr>
<td>7.27</td>
<td>SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing simplified special multiplicity constraint</td>
</tr>
<tr>
<td>7.28</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition on the constrained table realizing a special multiplicity constraint</td>
</tr>
<tr>
<td>7.29</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition on the constrained table realizing a mandatory multiplicity constraint</td>
</tr>
<tr>
<td>7.30</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition on the related table realizing the special multiplicity constraint</td>
</tr>
<tr>
<td>7.31</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition on the related table realizing the mandatory multiplicity constraint</td>
</tr>
<tr>
<td>7.32</td>
<td>SQL ISM with the CREATE VIEW statement for the view realizing an exclusivity constraint</td>
</tr>
<tr>
<td>7.33</td>
<td>SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing an exclusivity constraint</td>
</tr>
<tr>
<td>7.34</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition on the constrained table realizing an exclusivity constraint</td>
</tr>
<tr>
<td>7.35</td>
<td>SQL ISM with the CREATE TRIGGER statement for the trigger definition for the INSERT operation on a related table realizing an exclusivity constraint</td>
</tr>
</tbody>
</table>
7.36 SQL ISM with the CREATE TRIGGER statement for the trigger definition for the UPDATE and DELETE operations on a related table realizing an exclusivity constraint ........................................... 158
7.37 SQL ISM with the CREATE VIEW statement for the view definition realizing an enumeration constraint .......................................................... 160
7.38 SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing an enumeration constraint . . . 161
7.39 SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing an enumeration constraint ............................................. 162
7.40 SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing an immutability constraint checking the UPDATE operation . . 164
7.41 SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing an immutability constraint checking the DELETE operation . . 164
7.42 SQL ISM with the CREATE VIEW statement for the enumeration constraint on the CONDITION column of the table COPY ............................. 176
7.43 SQL ISM with the CREATE VIEW statement for the exclusivity constraint on the table COPY ................................................................. 177
7.44 SQL ISM with the CREATE VIEW statement for the exclusivity constraint on the table COPY ................................................................. 177
7.45 Error thrown by the Oracle database when inserting into the combined view 177
Abbreviations

Modelling Techniques and Models

CIM Computation Independent Model
DOLCE Descriptive Ontology for Linguistic and Cognitive Engineering
GFO General Formal Ontology
GOL General Ontology Language
ISM Implementation Specific Model
MDA Model-Driven Architecture
MDD Model-Driven Development
MDE Model-Driven Engineering
MDSD Model-Driven Software Development
MDSE Model-Driven Software Engineering
MOF Meta Object Facility
OCL Object Constraint Language
PIM Platform Independent Model
PSM Platform Specific Model
RDB PSM Platform Specific Model of a relational database
SQL ISM Implementation Specific Model consisting of SQL scripts
UFO Unified Foundational Ontology
UML Unified Modeling Language

Programming Languages

DDL Data Definition Language
DML Data Manipulation Language
SQL Structured Query Language

Model Elements and Constructs
List of Tables

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB</td>
<td>Database</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>FK</td>
<td>Foreign key</td>
</tr>
<tr>
<td>PK</td>
<td>Primary key</td>
</tr>
<tr>
<td>RDB</td>
<td>Relational database</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational Database Management System</td>
</tr>
</tbody>
</table>

Organizations and Research Groups

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCMi</td>
<td>Centre for Conceptual Modelling and Implementations</td>
</tr>
<tr>
<td>CTU</td>
<td>Czech Technical University in Prague, Czech Republic</td>
</tr>
<tr>
<td>FIT</td>
<td>Faculty of Information Technology</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
</tbody>
</table>

Tools and Programs

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE</td>
<td>Computer-Aided Software Engineering</td>
</tr>
<tr>
<td>EA</td>
<td>Enterprise Architect</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>OCLE</td>
<td>Object Constraint Language Environment</td>
</tr>
<tr>
<td>OLED</td>
<td>OntoUML lightweight editor</td>
</tr>
<tr>
<td>USE</td>
<td>UML-based Specification Environment</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Software engineering is a demanding discipline that deals with complex systems [3]. The goal of software engineering is to ensure high-quality software implementation of these complex systems. To achieve it, various software development approaches have been developed.

In the recent years, the Model-Driven Development (MDD) approach became very popular. It is an approach to software development based on creating models of various abstraction levels and various perspectives of the intended system. Transformations are used to transform individual models into other models [4]. The most popular part of this approach in software engineering is the process called forward engineering – the process of transforming more abstract models into more specific ones. The most usual use-case for this approach is the creation of conceptual data models of the domain and their transformation into source codes or database scripts.

For this approach to be effective, it is necessary to use high-quality expressive models that should define the requirements for the system on all levels as precisely as possible [3]. To successfully use them in the MDD approach, these models should define all requirements and constraints of the system. Moreover, all these requirements should be carried across all the models to realize them correctly in the final application.

To create such a high-quality expressive conceptual model, OntoUML seems to be very suitable. It is an ontologically well-founded conceptual modelling language based on the Unified Modeling Language (UML) and Unified Foundational Ontology (UFO). Its elements are based on cognitive science, philosophy and mathematical theory of sets and relations. Thanks to that, it offers precise semantics to the model primitives as well as the methodological guidelines for classification and categorization of the model elements [5].

This thesis deals with incorporating OntoUML into the MDD approach to software engineering as a conceptual data modelling language. It presents a new method of the transformation of such OntoUML conceptual models into relational database models and their realization in a relation database. During these transformations, various aspects and constraints derived from the elements of the OntoUML model must be handled to maintain the semantics and constraints defined by the model. These constraints must
1. **Introduction**

be realized in the relational database model as well as in the actual relational database. Without the realization of the constraints, the application would be able to store data different from their definition in the initial OntoUML conceptual model.

1.1 **Motivation**

In the last decade, the MDD approach became very popular in software development, especially in data modelling. Conceptual platform-independent models are created to define the objects of the domain, their attributes and relations. Such models are then used to generate relational database models and also the SQL scripts to create the actual database schemas.

To achieve high-quality software, such models should be as precise as possible, i.e., they should define all the data objects, their properties and constraints. Therefore, a well-founded and precise modelling language is necessary. As UML is designed as an implementation language, it does not provide enough constructs for precise domain modelling [2]. In contrast, OntoUML is a conceptual modelling language based on cognitive science, psychology and mathematical theories of modal logic. Thanks to these well-stated principles, OntoUML helps to create ontologically well-founded conceptual models with much more precise classification of the domain objects and their relations and properties. For the reason, it seems very suitable for the conceptual data modelling in the MDD approach. Therefore, we discuss the potential usage of OntoUML for this purpose.

Although new types and architectures of database management systems (DBMSs) emerged to react to current requirements of practice, including NoSQL, NewSQL and Hadoop database systems [6], the DB-Engines Ranking [7] shows that relational databases (RDBs) are still the most popular and common DBMSs used in practice. From the list of top 100 of popular DBMSs in March 2017, including relational databases, object databases, graph databases, key-value stores, document databases, XML databases, and many other types, 44 of them are standard relational database management systems (RDBMSs). From the top 10 most popular DBMSs, it is 7 RDBMSs. Even the top 4 most popular DBMSs are relational, namely Oracle, MySQL, Microsoft SQL Server and PostgreSQL. This is why we aim at the transformations of the OntoUML conceptual models into relational database models and the actual relational database.

However, as the OntoUML model contains a lot of special semantics and domain-specific constraints defined by the OntoUML universal and relation types, these special constraints must be transformed properly into the realization in the RDB to prevent losing them and storing the invalid data. Therefore, we focus on the proper implementation of the constraints derived from the initial OntoUML model.
1.2 Problem Statement

This thesis aims at incorporating OntoUML into the MDD approach for software development as a conceptual data modelling language. However, although OntoUML is designed as a light-weight extension of UML, which is used for conceptual data modelling in MDD very often, it distinguishes many special types of classes and relations, which provide special meaning and constraints to the model elements. To fully integrate OntoUML as the notation for Platform-Independent Models, these special aspects of the language need to be handled during the transformations.

In order to confirm the suitability of OntoUML for the conceptual data modelling, we formulate the following research questions:

**Q1** Is it possible to use OntoUML for conceptual data modelling?

**Q2** Is it possible to transform an OntoUML model into a relational database model and generate SQL scripts from it?

**Q3** Is it possible to realize all the implicit constraints defined by the types of universals and relations used in the OntoUML model in the relational database?

In order to answer question **Q1**, the OntoUML language and its constructs must be correctly understood. For that, we provide a complex overview of the language and its principles and constructs in the context with data modelling in [chapter 3]. Also, we provide an example of a complex OntoUML conceptual model containing most of the OntoUML and UFO-A concepts (see [section 4.1]). The model defines the structure of information used in a library institution.

In order to answer questions **Q2** and **Q3**, the proper transformation of various OntoUML types of universals and relations must be investigated. In this thesis, we propose a method for the transformation of these types, considering all the implicit constraints of each particular type. In our approach, the transformation is divided into three steps:

1. transformation of OntoUML PIM into UML PIM,

2. transformation of UML PIM into PSM of a relational database (RDB PSM),

3. and finally, transformation of RDB PSM into ISM consisting of SQL scripts (SQL ISM).

In each of the steps, we target certain aspects of the model that require special handling during the transformation. On the other hand, although the transformation might be done in a single step, the transformation division into the three steps allows: a) the reuse of the existing know-how for the transformation between a UML model and the relational database, b) model adaptation and optimization between the individual steps, and c) the reuse of the initial transformation of an OntoUML model into a UML model for other target platforms.
1. Introduction

Although the OntoUML notation is based on the UML notation, it also uses various types of universals and relations, determining certain additional features like rigidity, anti-rigidity, essentiality and immutability. Therefore, these individual types and special features are transformed into a standard UML model in step 1 in such a way that the semantics of the original OntoUML model should be preserved. If necessary, additional constraints are defined to preserve the semantics.

In step 2, the transformation of classes and their relations into database tables and references is performed. Also, the additional constraints derived from the OntoUML types must be accordingly transformed to restrict the valid data in the database. Moreover, in certain situations, special multiplicity values of a relation in the UML PIM need to be correctly realized in the database model to prevent the data violating such multiplicities.

Finally, in step 3, SQL scripts are generated to create the database schema and to realize the special constraints derived from the OntoUML types, as well as the special multiplicity values, to truly prevent creating invalid data in the database.

1.3 Related Work and Previous Results

OntoUML was established in 2005 by Giancarlo Guizzardi in his doctoral thesis [2]. Since then, many consecutive papers were published discussing various aspects of OntoUML and its use in various domains and contexts. To refer to some of the papers, we would like to recommend [8], [9], [5] or [10]. In the meantime, several efforts were made to create a complex OntoUML portal containing a detailed documentation of the language and its concepts, model repository and modelling tools, e.g., OLED [11], Menthor [12] or the OntoUML overview at the OMG wiki [13].

The idea of using OntoUML as the conceptual modelling language in context with the MDD approach to software development was introduced in [14]. In the master thesis, the author discusses the conceptual modelling in two levels – ontological and informational. He also proposes the transformation of an OntoUML conceptual model into an object-oriented implementation model in UML. However, the author limits himself only to a subset of OntoUML and UFO concepts, namely rigid Sortals (Kind, Subkind), Categories, Qualities and Roles (including Relators and RoleMixins). The author does not address other OntoUML and UFO concepts such as Phases, Modes and various parthood relation types. Also, the author’s approach to certain parts of the transformation vary from our approach, for instance, the transformation of Roles and their representation in the UML model. Finally, the author discusses only transformation into the implementation UML model, while we propose the full transformation (composed of three consecutive steps) of the OntoUML conceptual model into its realization in a relational database.

In our previous work [A.6], we proposed using OntoUML as the conceptual language in the MDD approach to pure object implementation of the data model. In the paper, we discussed the aspects of OntoUML Sortal types and their transformation into a pure object implementation model. We also discussed the model instantiation, as well as the approach based on the separation of the identity and state of the objects. In contrast to [A.6], in this
thesis, we focus on data models realized in a relational database, which leads to different constructs available in the model.

In [A.5], we proposed an approach for the realization of special multiplicity constraints in the relational database. The approach has been inspired by DresdenOCL Toolkit [I5], where OCL constraints are transformed into database views that allow the access only to the data satisfying the constraints. It has also been inspired by the realization of inverse referential integrity constraints used in IIS*Case [I6 I7].

In recent paper [A.7], we introduced our approach to the transformation of the OnToUML conceptual models into their realization in the relational databases based on the approach used for the special multiplicity constraints in [A.5]. This approach was discussed in more detail in the consecutive papers [A.8] and [A.10] for rigid Sortal types and in [A.9] for anti-rigid Sortal types, respectively.

1.4 Contributions of the Thesis

The contributions of this thesis can be summarized in the following list:

1. Complex overview of OntoUML and UFO-A concepts – including the distinction of universals and individuals, together with the classification of the object types, moments and part-whole relations – thoroughly explained and illustrated on simple examples.

2. Example of a complex OntoUML conceptual model from the library domain. This model contains most of the OntoUML and UFO-A concepts to demonstrate their gradual transformation.

3. Proposition of the transformation of OntoUML PIM into UML PIM, preserving the semantics defined by the OntoUML universal and relation types. The proposed transformation is illustrated on the running example of the Library OntoUML PIM.

4. Proposition of the transformation of UML PIM into RDB PSM, preserving all the semantics defined in the derived UML PIM model – focusing on the special multiplicity constraints and the constraints derived from the initial OntoUML model.

5. Proposition of the transformation of RDB PSM into ISM consisting of the SQL creation scripts, preserving the semantics derived from the OntoUML universal and relation types used in the initial OntoUML PIM, as well as the special multiplicity constraints. An overview of the available realizations of these constraints is provided and the advantages and disadvantages of each approach are discussed.

This thesis provides only an overview of the OntoUML language and a theoretical proposition of the transformation methods. No exact methodology is defined and no tooling support is implemented for the proposed transformations. The implementation of the tool supporting the proposed transformation, as well as an extensive experimental evaluation of the proposed realizations are subjects for the future research.
1. **Introduction**

1.5 **Structure of the Thesis**

The thesis is organized into eight chapters as follows:

1. **Introduction** The chapter introduces the topic, states our motivation, the overview of the related work and previous results and specifies our contributions.

2. **Background and State of the Art** The chapter introduces the reader into the background and current state of the art of the topics related to the topic of the thesis, namely the Model-Driven Development, UML, OCL and UML Data Modelling Profile. Also, an overview of other related approaches is provided, comparing them to our approach.

3. **OntoUML** The chapter provides a complex overview of UFO and OntoUML, explaining its various concepts and constructs, illustrated on simple examples.

4. **Overview of Our Approach** The chapter introduces our approach to the transformation of an OntoUML PIM into its realization in a relational database.

5. **Transformation of OntoUML PIM into UML PIM** The chapter discusses the details of the transformation of OntoUML PIM into UML PIM, while preserving all the semantics of the OntoUML universal and relation types used in the original OntoUML PIM.

6. **Transformation of UML PIM into RDB PSM** The chapter discusses the details of the transformation of the resulting UML PIM into RDB PSM, while preserving all the constraints derived from the OntoUML universal types used in the original OntoUML PIM, as well as the special multiplicity constraints of the relations in the model.

7. **Transformation of RDB PSM into ISM** The chapter discusses the details of the transformation of RDB PSM into ISM realized by the SQL scripts, while preserving all the constraints derived from the OntoUML universal types used in the OntoUML PIM and the special multiplicity constraints of the relations in the model.

8. **Conclusions** The chapter summarizes the results of our research, suggests possible topics for further research, and concludes the thesis.
Background and State of the Art

This chapter provides the information about the background of our research. It describes the approaches and technologies our research is based on, the current state of the art we use as the starting point of our research and also the related work.

The structure of the chapter matches the following outline:

- In section 2.1 we provide the introduction to the Model-Driven Development approach and we outline the basic process of forward engineering based on transformations of conceptual models into their realization.

- In section 2.2 we provide the introduction to the Unified Modeling Language (UML). We discuss its purpose, the notation of UML Class diagrams, the extension mechanism of UML profiles and also provide an overview of some tools for UML.

- In section 2.3 we discuss the Object Constraint Language (OCL) and its syntax and semantics in context of invariant constraints. We also provide an overview of some tools supporting the definition, validation and transformation of OCL constraints.

- In section 2.4 we provide an overview of some available tools for modelling in UML and OCL, discussing their support for model transformations and database script generation.

### 2.1 Model Driven Development

Model Driven Development (MDD) – also called Model-Driven Software Development (MDSD) – is a software development approach based on modelling and transformations [4]. The product to be developed is described using various types of models of various levels of abstraction specifying the requirements, functions, structure and deployment of the product. These models are used to support the development of the product by means of transformations between the models and code generation [18] [19].
The history of this approach can be tracked back to the CASE tools developed 1980’s. Because there was no support for a unified modelling notation to capture the complexity of the technologies of that time, it was not very widely adopted. However, in the past two decades, the development of object-oriented programming languages like C++, Java or C# and the establishment of the Unified Modeling Language (UML) as the general analytical and design notation highly increased the interest in the MDD approach. Many methodologies and approaches appeared that are often referred to as Model-Driven Engineering (MDE) or Model-Driven Software Engineering (MDSE) [20].

The formal foundations of the approach were established in 2001 by the Object Management Group (OMG), which developed a set of standards called Model Driven Architecture (MDA). This set includes specifications of many various technologies and languages including UML, MOF, CORBA or XMI. The basic definition of the MDA is defined in MDA Guide Revision 2.0 [1].

According to Arlow and Neustadt [21], the software development defined by MDA is based on models. The models serve as fully qualified development artefacts and not only as a documentation tool. Also, the source code of the software is considered only a very specific type of model. The software product is then constructed by a sequence of model transformations going from more abstract levels to more specific ones.

According to [1], MDA defines four levels of abstraction of the models (see Figure 2.1):

**Computation Independent Model (CIM)** A model describing the environment, business processes and business requirements for the product. This model is the most abstract specification dealing with the real business, abstracting from any concrete and specific implementation and technologies. The goal of the model is to capture what is expected of the product. CIM is usually referred to as the domain model.

**Platform Independent Model (PIM)** A model describing the requirements and specification of the system. The model usually consists of conceptual data models, use case models, description of system functions and processes. However, all the requirements are defined in a general form abstracting from a concrete technologies and platform. The goal of the model is to define the system functions and behaviour that can be applied to various technologies, platforms and environments. PIM is usually referred to as the conceptual or analytical model.

**Platform Specific Model (PSM)** A model describing the design of the system for a specific platform. This model captures the way how the system requirements defined in the PIM are realized using the specific technologies. Therefore, platform specific tools, constructs, libraries and objects can be used in the model. PSM is usually referred to as the logical model of the system and provides visualization and documentation of the final source code.

**Implementation Specific Model (ISM)** This level of abstraction is the actual code of the system with its implementation documentation (JavaDoc, PHPDoc, etc.). Sometimes ISM is also referred to as the physical model.
2.1. Model Driven Development

MDA is mostly based on UML (see section 2.2) for creating and describing the individual models. Various types of UML diagrams are used to capture the modelled system in different abstraction levels and different perspectives. However, some other languages are also needed to define the semantics of the model, since UML is not an action language. Arlow and Neustadt [21] mention ASL as an example of such an action language.

As already mentioned, MDA is based on the transformations between the models. These transformations may be defined between the models on the same abstraction level or between different abstraction levels. The latter transformations are parts of the forward engineering and reverse engineering processes – processes of transforming abstract models (e.g. PIM) into more specific ones (e.g. PSM and source code), and specific models into more abstract ones, respectively.

During the transformations to more specific models, additional information is required to decide which of the many possible options should be used for the specific situation – for instance, what programming language and platform will be used in the PSM model when transforming the PIM model, or how the inheritance between the classes should be realized in the PSM model. An example is shown in Figure 2.2 [1]. The example shows a transformation of a platform independent model of a person that is parametrized by both selecting the XML Schema as the platform and applying additional parameters of the transformation (e.g., which person attributes are realized as elements and which as attributes). The result is a PSM model of a person in an XML Schema.

Moreover, no information should be lost when transforming from a more abstract model into a more specific model [2]. When transforming a model, all information should be preserved between the consecutive software engineering project phases [22]. To achieve a high-quality software system, high-quality expressive models are necessary to define the
2. Background and State of the Art

Figure 2.2: Example of the transformation of PIM into PSM for XML Schema [1]

requirements for the system on any level [2]. Such models should contain as much information as possible to be able to use the forward engineering process and create a system as best suited to the needs as possible. Losing any information from the more abstract model when transforming it into the more specific one would necessarily lead to missing some of the requirements identified in the more abstract model.

The top desire of OMG and MDA is so called round-trip engineering – a combination of the forward and reverse engineering processes. In this approach, the system is developed using models and using transformations in both directions. Changes in the model are used to generate a new version of the system while persisting all the changes in the code. Also, changes in the code can be extracted and transformed to the model so new versions of the system can be generated [21].

The book Model-Driven Software Development: Technology, Engineering, Management by Stahl et al. [23] provides a great overview of the MDD approach including the terminology, specifications, transformations and case studies. Another book Model-Driven Software Engineering in Practice by Brambilla et al. [24] presents the foundations of MDSE approach and also deals with the technical aspects of MDSE including the basics of domain-specific languages, transformations and tools. Also, the survey by da Silva [25] provides a good overview of the MDD approach and terminology related to MDE, MDD and MDA. Another survey was published by Whittle et al. [26] that focused on the support of the MDE approach in tools and provides a taxonomy of tool-related considerations.

Currently the most widely used concept of the Model-Driven Development approach is the forward engineering process. The intended product is defined using various conceptual models which are then transformed into the logical model and source code or database scripts are generated from such logical model. Also, most of the current CASE tools and
IDEs offer various transformation and code generation functions. In this thesis, we focus on the transformation of a conceptual data PIM created using OntoUML into its realization in relational databases in context with the forward engineering process.

### 2.1.1 PIM of Application Data

The most often use case of the MDD approach is data modelling. Conceptual PIMs are created to describe the domain of interest, its entities and their properties and relations. The goal of the PIM of application data is to define what types of information are required to handle in the system. The model defines classes – in some models called *entities* – that define the types of objects in the domain. Each of the classes defines common properties of all objects of that type – the *instances* of the class – including the attributes and relations [21].

Various modelling languages use different notations to capture the domain classes, their attributes and relations. The notation used most often is probably the UML Class diagram described in subsection 2.2.1. In our approach, we utilize OntoUML as the primary language for creating the conceptual data model. However, when transforming such OntoUML PIM into its realization in a relational database, we use UML PIM as the intermediate model (see chapter 5).

### 2.1.2 PSM of Relational Database

In a relational database, data are stored as rows in database tables [27]. Usually, each row in a table represents an object – in databases often called *entity*. The individual characteristics of the objects are stored in separate columns of various types, each row having the specific values of the specific object.

The goal of PSM for a relational database (RDB PSM) is to define the structure of the database. The model consists of the definition of the individual tables, their columns and various types of constraints. NOT NULL constraints are defined to enforce having a non-empty value in the column for all stored objects. UNIQUE constraints are defined to enforce storing unique values in the constrained column or combination of columns. PRIMARY KEY constraints are defined for columns containing values uniquely identifying each stored object. FOREIGN KEY constraints are defined for references between tables to enforce the reference value stored in the source table to exist in the target table.

For the creation of a RDB PSM model, various notations are used, including ER diagrams using the Chen notation [28] or Barker notation [29]. There are also approaches for using the Extended ER models (EER), e.g. [30]. In this thesis, we use the UML Data Modelling Profile for the UML Class diagrams used by Enterprise Architect [31] for modelling the RDB PSM models (see subsection 2.2.2). In our approach, the RDB PSM model is created in the second step by the transformation of UML PIM, defining the realization of the initial OntoUML PIM in the relational database (see chapter 6).
2. Background and State of the Art

2.1.3 ISM of Relational Database

In context with the database modelling, ISM constitutes of the actual code of the database. Therefore, the ISM model consists of the SQL statements and scripts realizing the individual constructs in the database. Usually, the ISM model consists of CREATE statements that can be used to create the individual database tables, their columns, constraints and other constructs like triggers, procedures and functions.

Having a RDB PSM model, various modelling tools can be used to generate the SQL scripts, including Enterprise Architect [32] or DresdenOCL Toolkit [15]. In our approach, the ISM model consisting of the SQL scripts (SQL ISM) is the final result of the transformation of the original OntoUML PIM into the relational database (see chapter 7).

2.2 UML

Unified Modeling Language (UML) is a language for creating and maintaining variety of models using diagrams and additional components [33][21]. UML is generally known notation by analysts, designers and developers and it is taught as the required knowledge base at most software engineering schools and universities including FIT CTU in Prague.

The Unified Modeling Language was in the beginning developed by Grady Booch and Jim Rumbaugh in the company Rational Corporation in 1990’s. In 1995, Booch and Rumbaugh were joined by Ivar Jacobson and together they created the specification RFP containing the UML as the standard for object-oriented visual modelling. In 1997, the proposed modelling language was accepted by OMG as the industry standard UML 1.1 [21]. Since then, many versions of UML were introduced. Among them, the following versions were important [34]:

UML 1.4 Released in 2001, the addition of UML profiles as the extension mechanism for UML.

UML 1.5 Released in 2003, the addition of actions – executable actions and procedures, including their run-time semantics, defined the concept of a data flow to carry data between actions, etc.

UML 2.0 Released in 2005, a lot of new diagrams and elements were introduced or enhanced, including the package diagrams, object diagrams, activity diagram and sequence diagrams.

UML 2.4.1 Released in 2011, it is considered the current version of UML, revising previous version with various fixes and clarifications [35].
UML 2.5 Released in 2015, a big revision of the UML specification, which simplified the specification document and changed some classifications and compliances. Also, it changed the defaults for generalization sets (see subsection 2.2.1.3) [36].

UML defines a set of building blocks from which a model is created, mechanics of modelling in the language and a general architecture of the model [21]. The building blocks contain various types of elements (i.e. classes, use cases, components, etc.), relations (i.e. association, generalization, dependency, etc.) and diagrams (class diagram, use case diagram, sequence diagram, etc.). For our research, the UML Class diagram is important, as we use it for one of the intermediate models in the transformation of OntoUML PIM into its realization in a relational database. Also, the OntoUML notation is based on the UML Class diagram notation. Therefore, it is necessary to summarize the basic UML Class diagram notation in the next subsection.

2.2.1 UML Class Diagram

In context with the MDD approach, the UML Class diagram is the notation mostly used to define the PIM models of application data [25]. It provides the constructs to define various types of objects, their properties and relations. This section provides an overview of the UML Class diagram notation in context with MDD and PIM.

All the examples of the UML Class diagrams are created in Enterprise Architect (see section 2.4).

2.2.1.1 Objects and Classes

"The purpose of a Class is to specify a classification of objects and to specify the Features that characterize the structure and behavior of those objects" [36]. Therefore, a class can be understood as a template for objects of the same type defining the common features of all its instances. These features include attributes, relations and methods.

An object, on the other hand, "represents a particular instance of a class. It has identity and attribute values" [37]. It means that the objects are instances of the classes in the model holding specific values for the attributes defined by the classes and being related to other objects according to the relations defined between the classes.

According to [21], in UML, a class is depicted by a rectangle with up to three compartments:

**Title compartment** First compartment containing name of the class. It can also define stereotypes to distinguish various types of classes.

**Attribute compartment** The second compartment define attributes of the class. Each attribute has its name. It can also define the data type of the attribute depicted after the attribute name, separated by the colon symbol. Moreover, the multiplicity of the attribute can be defined after the data type to restrict the minimum and maximum number of values of that attribute the instance of the class can have. When the
2. Background and State of the Art

![Diagram of Person and Book classes](image)

**Figure 2.3:** Example of a UML class diagram with classes Person and Book

 multiplicity of the attribute is not defined in the diagram, the value 1..1 is used as the default. According to [38], attribute of a class can also be set constant or immutable by adding expression \{immutable\}, which means that it cannot change its value after the initialization of the instance.

**Operation compartment** The third compartment define operations and methods of the class. As PIM of application data is solely focused on data structures and their relations, this compartment is usually not shown in conceptual models [37].

In Figure 2.3 an example of a UML Class diagram is shown with two classes – class Person with optional attribute firstName, mandatory attribute lastName and immutable mandatory attribute gender, and class Book with mandatory attribute title, optional attribute description and multi-value attribute genres.

2.2.1.2 Associations

To express the situation that an instance of a class can be related to other instances, associations are used in UML [21]. Usually, the association defines bidirectional relation between instances of two distinct classes, but it can also define relations between instances of the same class. According to [21], the notation of associations in UML contain the following features:

**Multiplicity** The multiplicity is a constraint which defines how many instances of the target class can be related to a single instance of the source class. It should be defined always for both sides of the association to restrict the allowed number of related instances for instances of both classes. The multiplicity constraint is very important for transformation of a PIM into a PSM and therefore it is discussed in more details in the following text.

**Name** The association name defines the meaning of the relation from the point of view of one of the related instances.

**Names of roles** Each side of the association can define a name of the role the instances of that class play for the instances of the other class.

---

[3] Enterprise Architect uses expression \{readOnly\} to denote the immutability of the attribute.
Multiplicity is defined as a list of comma-separated intervals \[21\]. However, only one interval is usually defined for each multiplicity value. Each interval has a minimal and maximal multiplicity value defined. The \textit{minimal multiplicity} defines the minimal number of instances of one class that can be related to a single instance of the other class. The \textit{maximal multiplicity} \[4\] defines the maximal number of instances of one class that can be related to a single instance of the other class. To define unlimited number, the symbol of asterisk (\(\ast\)) is used. When both the minimal and maximal multiplicities are equal, only one value can be shown.

In Figure 2.4 an example of a UML Class diagram is shown with the association between classes \texttt{Book} and \texttt{BookEdition}. It represents the fact that instances of the class \texttt{BookEdition} are edition of certain book represented by an instance of the class \texttt{Book}. According to the defined multiplicities, each instance of the \texttt{BookEdition} class is related to exactly one instance of the \texttt{Book} class, while each instance of the \texttt{Book} class can be related to unrestricted number of instances of the class \texttt{BookEdition}. Moreover, the instance of the class \texttt{Book} related to the instances of the class \texttt{BookEdition} is \textit{immutable} and cannot be changed.

Usually, the minimal multiplicity of 0 or 1 is used to define the optionality or mandatory of the relation, respectively. For the maximal multiplicity, the values 1 or \(\ast\) are usually used to restrict the number of related instances to a single instance or keep it unrestricted to allow relation to multiple instances, respectively. However, in general, the minimal and maximal multiplicities can be any positive numbers. If the value is different from the standard 0, 1 or \(\ast\), we call them \textit{special multiplicity values}. Because this is not used very often, many CASE tools do not take such values into consideration in their transformations and also these values are usually not checked in relational databases. As the transformation of OntoUML PIMs often results in such special multiplicities – especially minimal multiplicities of 1 on both sides of the relation, we discuss the possibilities of their realization, among others, in \texttt{Chapter 6} and \texttt{Chapter 7}.

Furthermore, alike the attributes, also the associations can be set \textit{immutable}. However, as there are two sides of each association, the immutability needs to be defined for the individual ends of the association. Setting one end of the association as \textit{immutable} by

\[\text{\texttt{Book}}\]
- title: String
- description: String \([0..1]\)
- genres: String \([0..\ast]\)

\[\text{\texttt{BookEdition}}\]
- ISBN: String
- edition: String
- language: String

\(1\)
\{frozen\}
of
\(1..\ast\)

Figure 2.4: Example of a UML class diagram with an association between classes \texttt{Book} and \texttt{BookEdition}.

\[\text{\texttt{Book}}\]
- title: String
- description: String \([0..1]\)
- genres: String \([0..\ast]\)

\[\text{\texttt{BookEdition}}\]
- ISBN: String
- edition: String
- language: String

\(1\)
\{frozen\}
of
\(1..\ast\)
adding \{immutable\} property to the role name means, that the set of instances of the class at this side of the association related to an instance of the other class cannot be changed\[5\].

### 2.2.1.3 Generalization in UML

Generalization is a taxonomic relation between a more general class and a more specific class \[36\]. It is used in situations when there are multiple special cases of the more general class with additional features or special meaning. The more general class is often called *superclass* or *parent* of the more specific class, on the other hand, the more specific class is usually called *subclass* or *child* of the more general class. Also, the relation is often called *generalization* in the direction from the more specific class to the more general class. For the other direction, the term *specialization* is often used. Both generalization and specialization represent the same relation. In the UML Class diagrams, the generalization relation is depicted as an arrow line with a hollow triangle as the arrowhead between the symbols representing the involved classes, pointing to the superclass. Multiple specializations of the same class can be modelled by separate relations or using so called a *shared target style* \[36\] (see Figure 2.5).

The subclass inherits all the features – attributes, relations and methods – from the superclass, adding them to its own features. Thanks to this inheritance, the instances of the subclass also have features defined by the superclass, and therefore it is also considered to be instance of the superclass \[21\].

In UML, the generalization relation is rigid. That means that it holds in any situation. As the fact that an object is instance of a class is part of its identity – changing this relation would change the identity of the object and thus it would be a different object – an object cannot cease to be instance of neither the subclass nor the superclass. This aspect of the generalization in UML is in contrast to the view of generalization utilized in OntoUML (see section 3.4).

Although not very common in UML models, the subclasses may form generalization sets to define a partition of subclasses with common sense \[36\]. The generalization set is defined by its name for each of the generalization relations or for the shared generalization tree. If no generalization name is defined for a superclass, all subclasses are considered to form a single generalization set regardless the target style (separate or shared) \[21\].

For each generalization set, the constraint consisting of the *isCovering* and *isDisjoint* properties should be set to restrict the relation of an instance to the individual subclasses \[36\]. The *isCovering* property specifies whether the generalization set is complete – if *true*, every instance of the superclass is always also an instance of some of the subclasses; if *false*, instance of the superclass may not be instance of any of the subclasses. The *isDisjoint* property specifies whether the subclasses of the generalization set can overlap – if an instance of the superclass can be instance of only one subclass (if *true*) or if it can be an

\[5\]Enterprise Architect uses expression \{frozen\} to denote the immutability of the association end.
2.2. UML

Figure 2.5: Example of a UML class diagram with the generalization between the Man and Woman subclasses and the Person class

instance of multiple subclasses at the same time (if false). In the UML Class diagram, the constraint is defined next to the generalization relation in curly brackets (see Figure 2.5). When no constraint is defined for a generalization set, the default constraint is used. However, the default constraint differs in the various versions of UML: up until UML 2.4.1 [35], the \{incomplete, disjoint\} was declared as the default constraint, while UML 2.5 [36] defines the \{incomplete, overlapping\} as the default.

When multiple generalization sets are defined for the same superclass, an instance of the superclass may be an instance of multiple subclasses from the same generalization set as well as from multiple generalization sets. This is made possible by the isCovering property individual for each of the generalization sets. This is in contrast to the common perception of generalization in UML Class diagrams where the generalization sets are not very often used but it is very important in context of the generalization utilization in OntoUML (see section 3.4).

An example of the generalization relation is shown in Figure 2.5. The class Person is specialized by subclasses Man and Woman, forming a generalization set gender with the \{complete, disjoint\} generalization set constraints.

2.2.2 UML Data Modelling Profile

The UML Class diagram notation can also be used for modelling PSM of a relational database. The UML Data Modelling profile [31] as an extension to the UML Class diagram may be used to describe the structure of a relational database – i.e. tables, columns, constraints, etc. – in UML.

The notation is made as an extension of the standard UML Class diagram. The individual database tables are depicted by classes with the stereotype ≪table≫. The

\(^6\)In Enterprise Architect, this stereotype is usually depicted by the table symbol in the top-right corner.
columns of such a table are defined as attributes of the class with the stereotype "column". Also, database-specific data types and restrictions can be defined as additional properties of the class attributes. As discussed in [40], unlike many other approaches for database modelling in UML, this approach elevates standard UML features without the need of extending the UML notation.

The constraints are defined by operations with stereotypes defining the type of constraint. Parameters of these operations identify the columns constituting the constraint, including their order:

**NOT NULL constraint** The NOT NULL constraint is depicted by the asterisk symbol (*) before the column name.

**UNIQUE constraint** The columns included in a UNIQUE constraint are depicted with their names underlined. The constraint itself is defined as an operation with the stereotype ≪ UNIQUE ≫ and the columns constituting the constraint as its ordered parameters.

**PRIMARY KEY constraint** The columns constituting the PRIMARY KEY constraint are marked by the PK symbol before the column name and the constraint itself is defined as an operation with the title of the constraint, the columns constituting the constraint as its parameters and the stereotype ≪ PK ≫.

**FOREIGN KEY constraint** The FOREIGN KEY constraint is defined by an operation with the stereotype ≪ FK ≫ and the columns constituting the constraint as parameters. Also, the individual columns constituting the FOREIGN KEY constraint are marked by the FK symbol before the column names.

Furthermore, as the FK constraint realizes the reference between the two tables, an association is depicted in the diagram. This association is uni-directional in the direction from the source table – the table with the FK constraint – to the target table – the table that is referenced by the FK constraint. Beside the multiplicities of the association, the name of the FK constraint is used as the source table role name and the name of the PK constraint of the target table is used as the role name of the target table in that association. Also, the mapping between the FK columns and PK columns composing the FK constraint is shown as the name of the association with the ≪ FK ≫ stereotype.

In Figure 2.6 an example of the database diagram is shown using the UML Data Modelling Profile with two tables BOOK and BOOK_EDITION. In the BOOK table, three columns are defined (BOOK_ID, TITLE and DESCRIPTION) and the PRIMARY KEY constraint PK_BOOK constituted by the BOOK_ID column. In the table BOOK_EDITION, four columns are defined (EDITION, BOOK_ID, ISBN and LANGUAGE, PRIMARY KEY constraint PK_BOOK_EDITION constituted by the column EDITION, FOREIGN KEY constraint FK_BOOK_EDITION_BOOK of the element.
Figure 2.6: Example of a relational PSM created using the UML Data Modelling profile referencing the PRIMARY KEY in the BOOK table constituted by the column BOOK_ID and the UNIQUE constraint UQ_ISBN constituted by the ISBN column.

2.3 OCL

As not all aspects of the domain can be expressed solely by the UML elements and diagrams, additional constraints must be usually defined to accompany the UML model to express these aspects, restrictions and limitations. For such purpose, natural language, various textual representations or a formal language can be used. One of the most usually used formal language in context of UML models is the Object Constraint Language (OCL) [21].

OCL is a formal language used to define constraints in a UML model [41]. The first version of the language was developed at IBM in 1995 as ”a business engineering language” inspired by Syntropy and it became part of the UML standard by OMG since version 1.1 [42]. OCL 2.4 [41] is the most recent version of the language.

According to [21], OCL can be used to write queries to access the model elements and their values, to define constraints and restrictions for the model elements and their values, and to define query operations.

In our research, we deal with the realization of special constraints and restrictions on the data model and therefore we use OCL to express these constraints in context of the individual classes. In OCL, such constraints can be defined as invariants, pre-conditions and post-conditions [41]):

Invariants  An invariant is defined in context of certain class in the model and it determines a boolean condition that must be satisfied by all instances of the contextual class.

Pre-conditions  A pre-condition is defined in context of certain method of certain class in the model and it determines a boolean condition that must be satisfied before executing that method. Therefore it can be used to restrict the values of the input parameters or properties of the instance on which the method can be executed.
2. Background and State of the Art

**Constraint 2.1** Example of an OCL invariant for the class Book

context b: Book inv BookTitleLength:
  b.title.size() > 101

**Constraint 2.2** Example of an OCL post-condition for the method update() of the class Book

context BookEdition::update() post BookEditionImmutability:
  self.edition = self.edition@pre

**Post-conditions** A post-conditions is defined in context of certain method of certain class in the model and it determines a boolean condition that must be satisfied after executing that method. Therefore it can be used to restrict the valid result of the method or the properties of the instance after executing the method.

Each of the OCL constraints can have a unique name for simple referencing. In the body of the constraint, the actually evaluated instance can be referenced using the keyword `self` or using the name defined in the context of the constraint. Also, various navigation, aggregation and iteration functions can be used in the body of the constraint. In the pre- and post-conditions, the values of the parameters can also be accessed. Moreover, in the post-conditions, the former values of the contextual instance can be accessed using the postfix `@pre`. The detailed syntax of the OCL constraints can be found in [41] and [43]. Also, in [44], the transformations between equivalent forms of various OCL expressions are proposed in order to support their definition and realization.

In **Constraint 2.1** an example of an OCL invariant is shown, restricting the size of the value of the attribute `title` of all instances of the class `Book` to less than 101. In **Constraint 2.2** an example of an OCL post-condition is shown, restricting the method `update()` of class `BookEdition`, for which the value of the `edition` attribute must be the same before and after the method execution.

2.4 UML and OCL Tools

On the market, there is a lot of various tools supporting the MDD techniques and modelling languages like UML or OCL. Currently, almost every modelling and CASE tool provides support for model transformation and source code generation. Because there is really many available tools I will mention only few examples.

*Enterprise Architect* [32] (EA) is a complex commercial CASE tool for maintaining models, transformation of models, source code generation and reverse engineering process from a source code to a PSM. Beside others, it provides a transformation from a data PIM into a PSM model for a specific database and generation of SQL source code from such a

---

PSM model. In the current version 11.1, EA also supports definition of OCL constraints. However, the constraints are validated only on the meta-model level and cannot be used to really validate the actual model of the system and its instances.

ArgoUML \[45\] is an open-source UML modelling tool written in Java. It supports all 9 UML 1.4 diagrams, forward engineering and code generation for Java, C++, C# and PHP languages and reverse engineering from Java source code files. It also supports creation of OCL constraints and enables integration with DresdenOCL toolkit for OCL constraints syntax and type checking.

DresdenOCL \[15\] is a project researched at the Technical University of Dresden. After loading a model and its instance along with a set of OCL constraints, the tool provides OCL syntax checking and OCL constraints evaluation. It also provides generation of SQL tables and views according to the model. Each OCL constraint is transformed into a database view selecting records violating the particular constraint. The tool also offers transformation of the model with constraints to AspectJ \[46\] for a Java source code. The tool is also integrated as a toolkit to the ArgoUML tool and as a plugin to the Eclipse IDE \[47\].

The Database Systems Group at the University of Bremen develops the project USE: UML-based Specification Environment \[48\]. USE is a system for the specification of information systems. It is based on a subset of UML and OCL. A model can be animated to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot the OCL constraints are automatically checked. OCL expressions can be entered and evaluated to query detailed information about a system state. This tool uses a special language USE as an intermediate language for the internal model representation.

OCLE (Object Constraint Language Environment) \[49\] is a UML and OCL modelling tool. It supports UML 1.5 and OCL 2.0. The tool provides UML syntax checking against Well-Formedness rules, Profile Rules and Methodological Rules written as OCL constraints on the meta-model level. It also support code generation for both UML models and OCL constraints.

Although there are many UML and OCL tools, most of them lack the possibility to define certain types of constraints or ignore them when generating the source codes. In \[50\], the authors presents the results of a survey comparing various UML and OCL tools with regards to their support for constraints in the source code generation process. Unfortunately, since then, not much have changed and the tools still have difficulties in realizing certain types of constraints defined in the UML model. For instance, the current versions of both the Dresden OCL toolkit and the Enterprise Architect ignore the special multiplicities or mandatory source multiplicity when generating references and FOREIGN KEY constraints realizing a binary relationship defined in the UML model. Also, no realization is generated for other types of constraints, such as the meta-properties of the generalization sets in UML. Therefore, we focus on the proper realization of such constraints in our approach and we use OCL for the intermediate definition of the constraints before transforming them into their realization in the database.
2. Background and State of the Art

2.5 Transformation of Conceptual Model into Relational Model

We may distinguish several efforts dealing with the differences of conceptual models and the relational model, usually referred as the Impedance Mismatch Problem (IMP) [51]. These are:

1. Transformation of the traditional Entity-Relationship (ER) models into the relational model.
2. Using the UML notation to express the relational models.
3. Object-relational mapping technologies (ORM).
4. Transformation from (Onto)UML/OCL into a relational model: UML PIM into RDB PSM

Ad 1, it is a long-studied and well-established approach documented, e.g., in [27] or [52]. There are also approaches for the transformation of the Extended ER models (EER), see, e.g., [30]. However, our focus is using OntoUML and UML for the conceptual models instead of ER models. Moreover, these traditional approaches usually neglect checking of certain types of constraints. For instance, when realizing generalization sets in RDB, the existence of referencing records in other tables should be checked to correctly realize its meta-properties isDisjoint and isCovering. Similarly, certain special multiplicity values of relationships are usually not checked in the RDB, although defined in the conceptual model. In [A.4] and [A.5], we discuss the proper implementation of special multiplicity constraints RDB using the views, CHECK constraints and triggers. The same problem is also addressed in [10] and [17] as the inverse referential integrity constraints (IRICs). The authors present an approach to the automated implementation of the IRICs by combinations of database triggers, procedures and functions in a tool called IIS*Case. This tool was designed to provide a complete support for platform-independent design and automated prototyping of information systems, including, e.g., checking of the consistency of constraints embedded into the DB [53] or the integration of subschemas into a relational DB schema [54]. Besides the IRICs, the tool also handles other types of constraints, e.g., the check constraints [53] [55].

Ad 2, there are several approaches for using UML for database modelling. One of them is the UML Data Modelling Profile created by Geoffrey Sparks [31], which is used in Enterprise Architect CASE tool [32] and which we utilize in our approach (see subsection 2.2.2). Another example is the UML Data Modelling Profile designed by Scott W. Ambler in [56].

Ad 3, there are technologies offered by libraries of various object-oriented languages (such as Java, Smalltalk, Ruby and C#) to overcome the object-relational Impedance Mismatch Problem (IMP). The library routines perform automatic run-time transformations between the object model used in the programming languages and the relational model of the database. An extensive study of the current leading ORM solutions is presented by
2.5. Transformation of Conceptual Model into Relational Model

Torres et al. in [51]. Nevertheless, ORM works at an application level, while our goal is to introduce richer semantics to the database level. Also, as noted by Torres et al., ORM provides the tools to work with the IMP, but not a complete methodology to solve them.

Ad 4, there are several approaches most similar to our effort. In [57], the authors describe the tool called Dresden OCL Toolkit [15] that is able to validate a model instance against the defined UML model and OCL constraints. The tool can be also used to generate SQL code from the UML model and attached OCL constraints, realizing the constraints by means of database views that can be used to query the data violating the constraints. A similar approach is also proposed in [58], where the authors present their approach to checking the constraints by incremental SQL queries that select the violating data. In their approach, only the queries affected by the data update are needed to be evaluated and only affected data are checked, improving the efficiency of the constraint checking.

We have been inspired by these approaches in one of our proposed realizations of the constraints derived from the initial OntoUML PIM. However, in our approach, we aim at automatically ensuring the satisfaction of the defined constraints and querying only valid data, hiding the potential invalid data.

In [59], the author describes an extension plugin for Enterprise Architect that generates the SQL code realizing OCL constraints. His approach is based on translating the OCL expressions to SQL queries and realizing the constraints by database functions used to detect the constraint violation. Such functions can be then used by CHECK constraints or triggers. In our approach, we prefer the direct implementation of the constraints in the views, CHECK constraints or triggers.

Another related work can be found in [60], where the authors transform the OCL constraints into stored procedures. However, we prefer using other techniques, since procedures must be invoked explicitly by the application, while the suggested CHECK constraints and triggers are executed automatically when performing the standard DML operations. Also, we present the realization specifically for Oracle Database 12g, while the authors of [60] discuss MariaDB, PostgreSQL and SQL server. Another approach to the realization of UML PIM is discussed, e.g., in [61] or [62], where the authors present the transformation of UML PIM into an Object-Relational database. They also do not discuss realization of the meta-properties of generalization sets nor other types of constraints, which can be derived from the OntoUML PIM model and which is the focus of our research.

As regards the transformation of OntoUML into its realization in RDB specifically, based on the literature reviews and personal confirmation by the author of OntoUML, Dr. Giancarlo Guizzardi, there has been no published method for transformation into UML, apart from our previous work, so far. On the other hand, there are works dealing with the transformation of OntoUML into different languages, such as OWL [63] and Alloy [64], or into an object-oriented implementation model in UML [A.6].
OntoUML is a graphical conceptual modelling language aimed at constructing ontologically well-founded conceptual models. It is based on Unified Foundational Ontology (UFO), created by Guizzardi and Wagner as the effort to unify several existing ontologies \[65\]. The first results of this effort were published already in 2002 in \[37\] and completely presented in Guizzardi’s thesis in 2005 \[2\].

The effort is based on the notion of foundational ontology: "A foundational ontology ... defines a range of top-level domain-independent ontological categories, which form a general foundation for more elaborated domain-specific ontologies" \[66\]. Based on this notion, the authors created a unified foundational ontology as the synthesis of (i) the General Formal Ontology (GFO) underlying the General Ontology Language (GOL) and (ii) the OntoClean ontology and the Descriptive Ontology for Linguistic and Cognitive Engineering (DOLCE) \[66, 67\]. Beside this cognitive base, UFO and its elements are defined using modal logic and related mathematical foundations such as sets and relations, providing the theory with strict formal foundations.

According to \[67\], UFO consists of several parts describing different aspects of the domain:

**UFO-A** deals with the theory of various types of things – objects and other endurants ("something which persists in time while keeping its identity").

**UFO-B** deals with the theory of events and processes – perdurants (perdurant is an "individual that is composed of temporal parts").

**UFO-C** deals with the theory of intentional and social aspects of business processes based on the notion of actions, activities and agents.

In parallel to definition of UFO itself, an effort to create a graphical conceptual language based on UFO to enable construction of ontologically well-founded models arouse. This effort was first discussed in \[68\] where a UML profile for Ontology Representation and Conceptual Modeling was proposed. Later, this effort was finalized in 2005 in the
3. OntoUML

Guizzardi’s thesis [2]. In this thesis, OntoUML was established as a light-weight extension of the Unified Modeling Language using the notion of profiles.

Unlike other extensions of UML, OntoUML is not based on the ontologically vague notion of class. Instead, it works with the notions of the universals and the individuals (see section 3.1). It uses the basic notation of the UML Class Diagram like classes, associations and generalization/specialization relations together with various stereotypes and meta-attributes to define the nature of the individual elements more specifically. On the other hand, it omits a set of other problematic concepts (for instance, the aggregation and composition) and replaces them with its own ontologically correct concepts. Meta-model of OntoUML "has been designed to comply with the ontological distinctions and axiomatic theories [of UFO]" [5].

Together, UFO and OntoUML solve many problems in conceptual modelling, such as part-whole relations [9] or roles and the counting problem [8]. The language has been successfully applied in different domains such as: interoperability for medical protocols in electrophysiology [69]; the evaluation of an ITU-T standard for transport networks [70]; the domain of Oil and Gas [10]; the domain of news information management [71]; the University Campus Management [72]; Logistics [73]. The language has also been used to define the Data Modelling Guide of the U.S. Department of Defence [74].

As OntoUML is based on UFO, which is in turn based on cognitive science and mathematical apparatus, it supports the creation of precise and expressive conceptual models. OntoUML diagrams provide higher expressive power and more precise semantics to the model primitives then pure UML. Also, these ontological meta-properties of the primitives also serve as a guideline to identify the most suitable ontological category of various domain entities [5].

Being domain-agnostic, we believe that OntoUML may be suitable for conceptual modelling of application data in the context with MDD to define more precise conceptual models that might be then transformed into the more specific models and even the relational database schema.

3.1 Universals, individuals, types and instances

In [66], the authors identify things to be the building stones of the whole UFO. Thing is defined as "anything perceivable and conceivable". The things, then, can be either sets having other things as members or entities where the entities are either types or individuals.

The type is defined as an "entity that has an extension (being a set of entities that are instances of it) and an intension, which includes an applicability criterion for determining if an entity is an instance of it" [66]. According to this definition, types can be understood as a template defining the features for all its instances. These features are defined by a set of axioms that may involve other types as its parts.

The individual is defined as an "entity that is not a type" [66]. From this definition we can derive that individuals are the real objects and other entities we perceive in the
3.1. Universals, individuals, types and instances

"The relation between individual and type is one of classification" [66]. It defines the fact that an individual is an instance of a type and that we perceive that individual to be a particular representative of the type – the individual is a realization of the template defined by the type (e.g. Mark is a Person, therefore he has all the features defined by the type Person). Important feature of UFO is the fact that an individual may instantiate multiple types at the same time, combining features of all those types. For instance, Mark is a Person and a Student at the same time (see Figure 3.1). However, all the types an individual instantiates must carry the same identity criterion (see section 3.2 for more information about identity criterion).

In [2], the author builds the theory on the ontological notion of the universals and the individuals. According to the author, the individuals are the real objects and other entities we perceive in the real world around us. This matches the definition of individuals presented in [66], as well as a similar definition for the particulars presented in [75]. The universals, on the other hand, represent general classification of those individuals. They define certain features, which can be realized in a number of individuals/particulars [37, 2, 75]. Moreover, all the universals carry the principle of application in accordance with which we are able to distinguish if the features defined by the universal apply to an individual. This matches the definition of types presented in [66]. However, in [2], the author uses the term type as the representation of a universal in conceptual modelling. Furthermore, the author also identifies instances of these types in a conceptual model to be representatives of the individuals and the principle of application carried by the universals to be realized by the instantiation relation between the instances and the types they instantiate.

Since OntoUML was established in [2], we will stick to its terminology: individuals standing for the real object of the domain, universals standing for the general ontological distinction of the features of the individuals, types standing for the classifiers in the conceptual model representing the individual universals, and instances standing for the individuals being representatives of the universal represented by their respective types.

Since OntoUML is defined as a light-weight extension of standard UML, it uses the notation of the UML Class Diagrams for the conceptual models. The types are represented by classes with their attributes, relations and, if-need-be, constraints. However, since UFO
OntoUML distinguishes many different kinds of universals depending on their features and constraints, OntoUML utilizes stereotypes of the classes to identify the kind of universal represented by the type. The individual kinds of types are identified by various stereotypes of these classes.

3.2 Identity and Identity Criterion

Identity and Identity criterion are two of the key features of UFO. "Identity can be defined as the fact of being who or what a person or thing is" [74]. It is the nature of an individual of being exactly the concrete individual, different from all other individuals. In conceptual modelling, every entity existing in the domain of interest always has its identity, and also all individuals according to UFO always have their unique identity [2].

On the other hand, "an identity criterion supports the judgment whether two instances are the same" [66]. In [74], the identity criterion is called the principle of identity and it is defined as "the method(s) or rule(s) by which we judge if two (apparent) instances of some concept are identically the same instance of the thing." In other words, it defines the way how we can determine if two instances represent the same individual or not. In the literature, as well as in the remaining of this thesis, the terms identity criterion, identity principle and principle of identity are used as synonyms.

The identity criterion is carried only by some kinds of universals defined in UFO – they are called Sortal universals. Types representing the Sortal universals provide the identity criterion to all their instances, and therefore we are able to distinguish them one from each other. Examples of the Sortal types are, e.g., Person, University or Student.

On the other hand, there are also kinds of universals that do not carry the identity criterion – such universals are called Non-Sortal universals [2], Mixin universals [2] or Dispersive universals [3]. The instances of a type representing a Non-Sortal universal follow various principles of identity provided by their respective Sortal universal. Such universals define the properties common to the different Sortal universals. An example of a Non-Sortal type is Customer, whose instances may be either persons (instances of the type Person) or companies (instances of the type Company) – instances of types with two different identity principles.

Certain types of Sortal universals define the identity criterion for distinguishing their instances (e.g. Kind, Subkind), while other Sortal types do not (e.g. Role, Phase). Instead, they inherit this criterion from some of their predecessors and provide this inherited identity criterion to their instances.

According to [66], "Every instance in a conceptual model must have an identity and, hence, must be an instance of Sortal type.” As the types in an OntoUML model represent universals, it also means that each individual must be an instance of a Sortal universal. On the other hand, being an instance of a Sortal type means having an identity. This identity is determined at the time the individual comes to existence, based on the identity criterion of the universal it is instance of. Furthermore, the identity of the individual is immutable – it can never be changed (e.g. Mark will always be Mark, even when he changes his name).
As stated in section 3.1, an individual may be an instance of multiple types at the same time. However, all these types must have the same identity criterion, otherwise there would be a conflict of the identity criteria. This restriction is defined in \cite{2} as The Restriction Principle and The Uniqueness Principle. These principles define that if several different Sortal universals apply to a single individual in the course of its existence, then there must be a unique ultimate Sortal universal which is a supertype of all these Sortal universals. Such ultimate Sortal universal is called Substance Sortal universal.

3.3 Rigidity

UFO and OntoUML are built on the notion of worlds coming from Modal Logic – various configurations of instances of the types representing various individuals in the domain in various circumstances and contexts of time and space. The theory is based on modal operands necessity and possibility.

Rigidity is a meta-property of the OntoUML universals which defines the fact that the extension of the respective type, i.e., the set of all its instances, is world invariant \cite{68}. In other words, it defines if it is possible that one individual is an instance of a type in one world but it is not the instance of that type in a different world. UFO distinguishes rigid, anti-rigid and semi-rigid universals:

**Rigid universals** are such universals which apply their principle of application to their instances necessarily (in the modal sense of necessity) \cite{2} – the extension of their respective types is rigid \cite{66}. It means, that any instance of a rigid type cannot cease to be its instance without ceasing to exist. For instance, Mark is always perceived to be a Person. As this applies to all instances of the Person type, it is rigid.

**Anti-rigid universals** are such kind of universals which apply to all its instances possibly (in the modal sense of possibility) \cite{2}. In other words, the extension of an anti-rigid type contains, in some world, instances which are not contained in that extension in another world. As a consequence, an individual may start or cease to be an instance of an anti-rigid type without affecting their existence and identity. For instance, Mark is a Student now\footnote{Supposing being a student means attending a school at the time.} but he was not a Student when he was born nor will be a Student 50 years later. Therefore, the Student type is anti-rigid.

**Semi-rigid universal** is such kind of universal which apply necessarily to some of its instances but only possibly to some other of its instances. Therefore its extension contains a combination of instances of anti-rigid and rigid types. This means that some of the instances in the extension will always be contained in that extension (these are instances of a rigid type) while the other instances may not be contained in that extension in certain worlds (these are instance of an anti-rigid type). For instance, a chair – an individual which is an instance of the Chair type – is always seatable –
OntoUML being instance of the Seatable type, but a crate – individual which is an instance of the Crate type – is seatable only when it is in the solid state – being instance of the Solid create phased anti-rigid type (see description of Phases in subsection 3.5.3).

3.4 Generalization and Specialization

In UFO and OntoUML, the generalization relation between various types – and not strictly a more general type and a more specific type – defines the inheritance of the identity criterion between the types (if any defined by the supertype) as well as the inheritance of some common features [66]. The important difference between UML and OntoUML is the rigidity of the relation. In UML, the generalization/specialization relation is rigid: when an object comes to existence, it is either an instance of the subclass – and thus it is also considered to be an instance of the superclass – or it is an instance of only the superclass. But, this fact cannot be changed in the lifetime of the object. In contrast to that, in UFO and OntoUML, the relation may be rigid or non-rigid. The individuals are always instances of some Substance Sortal universal defining the identity criterion of the individual, but may become instances of any other type with the same identity criterion, which is inherited from it. Thus, in one world, the individual may be an instance of both the supertype and subtype, and in another world, they can be an instance of only the supertype, while still keeping the same identity, as the identity criterion of the individual is not changed.

Also, the generalization sets are much more common in UFO and OntoUML than in UML. It is so because the individuals are instances of many types at the same time and the generalization set define the rules for what combinations of the types an individual may be instance of at a single time. The generalization sets are defined the same way they are in UML (see subsubsection 2.2.1.3). The default values for the generalization set constraint is \{incomplete, overlapping\} which is different from the defaults in UML up to version 2.4.1, however, in UML 2.5 it was finally unified.

Furthermore, UFO and OntoUML support multiple inheritance. A single type can be a subtype of multiple other types. It means that the individual which is an instance of the subtype is also instance of all its supertypes. Moreover, when the supertype has some other generalization set, the individual may be an instance of a subtype from that generalization set as well – if it is complete, it even must be. An example is shown in Figure 3.2, where Mark is a Personal customer and therefore he is also a Person and a Customer. Moreover, as Person is specialized by a complete generalization set, Mark must be an instance of either Man or Woman – in this case he is a Man.

3.5 Substantial Universals

UFO and OntoUML distinguishes various kinds of universals with different characteristics and constraints. "Types (e.g., Person or Car), attributes (e.g., being colored, or being
3.5. Substantial Universals

Figure 3.2: Example of instantiation with multiple generalization sets and multiple inheritance

Figure 3.3: Hierarchy of Substantial universals representing domain types in OntoUML. Inspired by [2, p. 106] and Guizzardi’s presentation at OntoUML course at University of Economics, Prague, September 2011

happy), and associations (e.g., being married to, being enrolled at) are all considered sorts of universals” [2, p. 201]. In OntoUML model, types are represented by various kinds of Substantial universals – universals, whose instances have clear identity independent of any other individual. This is in contrast to Moment universals whose instances have an identity dependent on another individual (for more details about Moment universals see section 3.6).

The Substantial universals are divided into Sortal and Non-Sortal universals, which in turn are divided into rigid, anti-rigid and semi-rigid ones. The hierarchy of the kinds of Substantial universals used in OntoUML is shown in Figure 3.3. The individual kinds of Substantial universals are introduced in the sequel.
3. OntoUML

Figure 3.4: Example of an OntoUML model with Sortal universal types

3.5.1 Kinds and Subkinds

The backbone of an OntoUML conceptual model is created by Kinds. Kind is a Rigid Sortal universal which defines the identity principle supplied to its instances [2]. Because of the rigidity, all instances of a Kind type receives their identity derived from this identity principle and keep it for all their life. Because of the identity the Kind defines, it can never be a subtype of another Sortal universal, otherwise there would be a clash of identity principles.

In OntoUML, Kind types are depicted as classes with the \(\llbracket\text{Kind}\rrbracket\) stereotype. Examples of types representing Kind universals are a Person and a University types as seen in Figure 3.4.

Subkind is another Rigid Sortal universal. It defines a special case of a Kind universal or another Subkind universal. Subkind types form generalization sets of other Kind or Subkind types with the root in a Kind type. This inheritance may have any possible meta-properties regarding the completeness and disjointness.

A Subkind universal does not define its own identity principle but it inherits it through the generalization relation from the ancestral Kind or Subkind universal and extends it by its own features. Thanks to this inheritance, an instance of a Subkind type is always also an instance of its rigid ancestor type. As both the Kind and Subkind universals are rigid, also the generalization/specialization relation between them is rigid. It means, that when an individual is an instance of a Subkind type, it cannot cease to be its instance as it is part of its identity.

In OntoUML, a Subkind type is depicted as a class with the \(\llbracket\text{Subkind}\rrbracket\) stereotype and generalization relation to its identity ancestor. Examples of Subkind types may be a
3.5. Substantial Universals

**Man** and a **Woman** as Subkinds of a **Person** as shown in [Figure 3.4](#).

### 3.5.2 Roles

*Role* is an Anti-rigid Sortal universal. It is used to define certain facts and properties of individuals when they are related to some other individuals – i.e. they play a role in the context of their relation to the other individual.

A Role universal does not define its own identity principle but it inherits it through the generalization relation from another Sortal universal having it. This inherited identity principle is then provided to the instances of the Role. However, unlike Subkinds, Roles are anti-rigid which means that an individual may change its instantiation of the Role. As the identity of an individual is immutable, Roles do not extend the inherited identity principle – being an instance of a Role is not part of the individual’s identity. Instead, the generalization relation to the type providing the identity principle – the *identity bearer* – rather defines the required identity principle of the individuals that are instances of the Role – this identity bearer defines the universal who may play the Role in the relation.

According to [2], the fact if an individual is perceived to be an instance of a Role depends on the extrinsic (relational) properties of the Role. In other words, Role universals are *relational-dependent*. It means that for each Role universal there must be a mandatory relation to another universal – a *truthmaker* – which makes the role valid. This relation always has the target minimal multiplicity at least 1 – so that all instances of the Role type are related to some instances of the other type making them to really play the role. Without such relation, the individual in fact does not play the role and therefore it cannot be instance of the Role universal. However, the Role universal does not represent the relation to any particular instance of the truthmaker, but the fact of having the extrinsic properties when there exists such a relation. It only defines that the individual is perceived to play the role, regardless of the actual number of the related instances of the truthmaker (as long as there is at least one).

Moreover, Roles do not form generalization sets specializing the identity bearer as a single instance of the identity bearer may be instance of multiple Roles at the same time. On the other hand, thanks to the inheritance of the identity principle, a single individual cannot be instance of the same role multiple times as the inherited identity principle would be the same, thus necessarily leading to the same instance.

In OntoUML, a type representing a Role universal is depicted by a class with the $\ll$Role$\gg$ stereotype with a generalization relation to the type representing its identity bearer. Example of a Role type may be a **Student** which is a role of a **Person** studying at a **University**, as shown in [Figure 3.4](#).

### 3.5.3 Phases

*Phase* is another Anti-rigid Sortal universal. Phases "constitute possible stages in the history of a substance sortal" [2]. These stages represent possible states the instances of a universal may be in, varying in properties or meaning.
Phase universals do not define their own identity principle but they inherit it from another Sortal universal through a generalization relation. Similar to Roles, Phases are also anti-rigid, therefore an individual may change the fact of being instance of a Phase universal. Therefore, in parallel to Roles, the generalization relation to the identity bearer rather defines the identity principle the instance of the Phase universal must have.

Unlike Roles, whose instantiation depends on extrinsic properties, instantiation of Phases depends on intrinsic properties – the fact that an individual is an instance of the Phase universal means that in that particular context we perceive the individual to have the properties or meaning of the Phase [2].

Phase universals form Phase partitions – complete and disjoint generalization sets, defining all possible states an instance of the Sortal universal may be in [2]. It means that each instance of the Sortal universal is automatically – through the specialization relation – an instance of one of the Phase universals in the generalization set as well. Moreover, there can be multiple Phase partitions for a single universal and the instance of this universal is always an instance of one Phase universal from each of the Phase partitions.

In OntoUML, a type representing a Phase universal is depicted as a class with the ≪Phase≫ stereotype and a generalization relation to the type representing its identity bearer forming the Phase partition with the other Phase types. Examples of the Phase types may be Junior and Senior states of a Student shown in Figure 3.4 – the distinction between junior and senior students depends on the number of years they already study, influencing the payment for the studies.

3.5.4 Mixins

"Conceptual modeling types classified as Mixins represent the dispersive universals" [2] p. 112. Dispersive universals, also called Non-Sortal universals or Mixin universals, are such kinds of universals whose instances may follow various different principles of identity. This is in contrast to Sortal universals, whose all instances must always follow the same identity principle.

Mixin universals are used to define common features shared by multiple other universals with disjunct identities. Therefore they are always defined as a supertype of other types in the OntoUML model. Moreover, asMixin universals do not provide any identity principle, there cannot be any direct instances of Mixin universals – each individual being instance of a Mixin universal is always an instance of some ultimate Sortal universal at the same time. Therefore, Mixin universals are always abstract. Also, the generalization set of the universals extending the Mixin universal is always \{disjoint, complete\}, as otherwise it would be possible to have a direct instance of the Mixin universal, or to have an instance mixing the identities of the extending universals.

Moreover, the generalization relation between a Mixin and another universal specializing it is always rigid, regardless of the rigidity of the universals it generalizes. On the other hand, UFO and OntoUML distinguishes several kinds of Mixin universals, based on the rigidity of the universals it abstracts.
3.5. Substantial Universals

Category is a Mixin universal defining common shared features of multiple rigid universals, usually Kinds. In OntoUML model, the types representing Category universals are marked by the ≪ Category ≫ stereotype. An example of a Category type is shown in Figure 3.5 where the Category NamedEntity defines the common property of having a name shared by the Kinds Person and Company, entities with different identity principles.

The Mixin universal defining common features of anti-rigid Roles is called RoleMixin. Important restriction for RoleMixins is that, as a Mixin universal, it defines common features of Roles following various distinct identity principles. Therefore, the Roles specializing the RoleMixin must specialize (recursively) different Substance Sortal universals. Otherwise, the identity principle of the Roles would be the same, and the common features might be defined by a Role supertype.

In OntoUML model, types representing the RoleMixin universals are marked by the stereotype ≪ RoleMixin ≫. An example of a RoleMixin type is shown in Figure 3.5 where both the persons in the role of PersonalCustomer and the companies in the role CompanyCustomer are perceived to be customers, expressed by the RoleMixin Customer.

Moreover, there are also Mixins which "represent properties that are essential to some of its instances and accidental to others" [2, p. 113]. Such Mixins define common features of both rigid universals – individuals being instance of such rigid universal are its instance in all possible worlds and therefore they have the Mixin features in all possible worlds – and anti-rigid universals – individuals being instance of such anti-rigid universal have the Mixin features only when they are instance of that anti-rigid universal.

Types representing such Mixin universals are marked simply by the stereotype ≪ Mixin ≫ in OntoUML models. Example of a type representing a Mixin universal is shown in Figure 3.6 where a Chair is always perceived to be a SeatableThing while a Crate is seatable only in the SolidCrate state.

Although not defined in the original version of OntoUML, the practice has shown that
one more Mixin universal is missing – \textit{PhaseMixin} – defining properties which are common to multiple Phases of different identity principles. This makes perfect sense as the anti-rigid RoleMixin covers only a part of the existing anti-rigid universals – Roles. The concept of PhaseMixin was first mentioned in [5].

In [70], the PhaseMixin universal is discussed together with other types of Mixin universals based on various combinations of other universal types (e.g. ModeMixin, RelatorMixin, ModePhaseMixin, etc.). These types also make perfect sense, as the Mixin universals help to improve readability of the diagrams and remove redundancy and duplicities in the model structures. Moreover, as Mixin universals define properties common to universals following different identity principles, they can be supertypes of any universal type having the identity principles – even Modes, Qualities, Relators, etc.

3.6 Moment Universals

Beside substantials – individuals which are instances of a substantial universal – there can also exist such individuals whose identity is dependent on some other individual. In UFO, such individuals are called \textit{Moments} [75]. Moments are existentially dependent on other individuals and they inhere in them – the individual they depend on is part of their own identity. The individual a Moment inhere in is called \textit{bearer} [2].

The universals the Moments are instances of are called \textit{Moment universals}. These universals provide – similar to Substantial universals – identity principle to their instances. However, they are existentially dependent on another universal which is part of their own identity principle.

The hierarchy of Moment universals in OntoUML is shown in Figure 3.7. In the sequel, the individual Moment universals used in OntoUML are introduced.

3.6.1 Quality

"A \textit{quality universal} is an intrinsic Moment universal that is associated with a quality structure" [2 p. 224]. Being an intrinsic Moment universal means that all its instances
are dependent only on intrinsic properties of the individuals they inhere in\cite{75}. Being associated with a quality structure means that Quality universals define certain property of the bearer with a structured value from certain domain. In other words, Quality universal defines "particular type of intrinsic property which has an structured value"\cite{77}.

UFO distinguishes three types of Quality universals:

- **Perceivable quality** is a quality that can be measured with a tool or instrument (e.g., weight, size, speed),
- **Non-Perceivable quality** is a quality that cannot be directly measured with a tool or instrument (e.g., currency),
- **Nominal quality** representing properties without measurable value but still perceived as a quality domain (e.g., one’s name, company VAT number)\cite{77}.

The Quality universal is connected to its bearer universal by a *characterization* relation. This association is always *one-to-one* as a single Quality individual may inhere only in a single bearer individual, and the bearer individual may have only a single value in the domain represented by the Quality. Moreover, the minimal multiplicity of the bearer is always 1 as the Quality individual is existentially dependent on the bearer, and thus cannot exist without it.

In OntoUML, types representing Quality universals are depicted as classes with the $\ll$Quality$\gg$ stereotype. Sometimes even the distinction between various types of Quality universals is defined using the appropriate stereotype. However, for this thesis, this distinction is not important. The *characterization* relation to the bearer is depicted by an association with the stereotype $\ll$characterization$\gg$.

An example of a type representing a Quality universal is shown in Figure 3.8 where the MaximalSpeed type defines the fact, that the maximal speed of a car model is defined as a value which can be measured in multiple scales.
3.6.2 Mode

According to [2, p. 237], "modes are intrinsic moments that are not directly related to quality structures". Similar to Qualities, Modes are also existentially dependent on their bearer in which they inhere. However, unlike Qualities which represent a measurable value from a single quality domain, Modes represent an independently identified property of their bearers. Like Substantials, Modes can have properties from multiple domains. Also, Modes have their own identity, however, the identity of their bearer is part of this identity. Usually, Modes define various versions or variants of their bearers (e.g., versions of a specific software) or individuals existing only because of another individual (e.g., a hole in a wall).

The universal which the Modes are instances of is called Moment universal. In OntoUML model, types representing Mode universals are depicted as classes with the ≪Mode≫ stereotype. Like for Qualities, the inherence relation of the Moment universal to its bearer universal is defined by an association with the stereotype ≪characterization≫. However, unlike for Qualities, this relation may be one-to-many as well as one-to-one, as a single bearer may bear many Modes of the same type. Still, the minimal multiplicity of the bearer is 1 because of the inherence and existential dependency.

An example of a type representing a Mode universal is shown in Figure 3.8 where the CarModel is defined as a Mode of CarBrand type.

3.6.3 Relations and Relator

"Relations are entities that glue together other entities" [2, p. 236]. In other words, relations are entities that connect other entities together and provide meaning to those particular connections. The number of entities connected together by a single relation is called its arity.

UFO distinguishes two types of relations - formal and material. "Formal relations hold between two or more entities directly without any further intervening individual" [2, p. 236]. It means that formal relations are based on intrinsic properties of the connected individuals, it is part of their identity and existence. Various comparative relations (heavier-than, older-than, etc.) and part-of relations belong among the formal relations.

"Material relations, conversely, have material structure of their own" [2, p. 236]. It means that there exists a special individual that relates the other individuals together – it mediates the material relation between the individuals. Sometimes it is also called the truthmaker as it makes the relation to exist. In UFO, such mediating individual is called
Relator. Relators might be physical objects such as an employment contract between an employee and an employer, as well as pure abstract ones such as a kiss or a call.

Relators may connect any number of individuals, mediating the material relation between all pairs of participants. An example of a ternary material relation is a purchase which connects a good, a person and a shop, and mediates the material relations of a good bought by a person, a good bought in a shop and a person bought in a shop.

As relators are individuals, the universal they are instance of is called Relator universal. Such universal defines the mediation relations to all those universals which its instances may mediate. Also, as mentioned in subsection 3.5.2 these connected universals are usually Roles, as these are the universals based on relations to other entities.

In OntoUML, formal relations are defined by standard associations with the stereotype ≪ Formal ≫. When the formal relation defines a comparison of qualities of the related individuals, its name is prefixed with \ as it is de facto derived from the qualities. For parthood formal relations, various more specific stereotypes are used as discussed in section 3.7.

Types representing the Relator universals are defined as classes with the ≪ Relator ≫ stereotype. These Relator types are then linked to all the universals participating in the n-ary material relation by an association with the ≪ Mediation ≫ stereotype and with the corresponding multiplicities. If the material relation between two universals, which is mediated by the Relator, is worth to be shown in the model, it is marked with the ≪ Material ≫ stereotype and its name is marked with \, as it is derived from the Relator. Also, the Material relation is linked to the mediating Relator by a dashed line in the diagram [2].

An example of a formal and a material relation is shown in Figure 3.9. A Person is perceived to be older than some other persons based on its quality (or property) age. Also, one person (a man) can be married to another person (a woman) – a man becomes a Husband and a woman becomes a Wife, related together by the material relation being husband to mediated by the Relator Marriage.

3.7 Whole-Part Relations

Besides the theory of types, attributes and relations, UFO also addresses the theory of parts and wholes. In [2], Guizzardi states that the part-whole relations are crucial for conceptual modelling, used in most of the modelling languages, however, "the concepts of part and whole are understood only intuitively". Therefore, Guizzardi and his colleagues aimed at building well-founded theory of conceptual part-whole relations based on philosophy and cognitive science.

UFO considers various constraints for the part-whole relations, such as optionality, mandatoriness and essentiality of the relation, as well as the different roles the parts may play within the whole [2][78]. In the sequel sections, these aspects of UFO and OntoUML are introduced.
3. OntoUML

![OntoUML Diagram](image)

Figure 3.9: Example of an OntoUML model with Formal and Material relations

3.7.1 Properties of Part-Whole Relations

As mentioned in [subsection 3.6.3](#), the Part-Whole relations belong to the group of formal relations, as they are based on the nature of the entities and there is no mediator constituting the relation [2]. Therefore, the relation is defined directly between the part and the whole. However, as there are multiple types of part-whole relations, these are not marked with the ≪Formal≫ stereotype but rather more specific stereotypes as explained later.

**Shareability.** In OntoUML, the part-whole relation are depicted as an association between the two types representing the related universals with an empty or full diamond at the whole end. However, the meaning of this diamond is different than it is in standard UML. In UML, the association with the empty diamond represents aggregation – a relation between the whole and its parts that can exist on their own without being part of the whole. Contrary, the association with the full diamond represents composition – a much stronger relation relating a fully dependent part of the whole which cannot exist without being part of the whole [21].

In contrast to UML, in OntoUML, the diamond is related to the notion of shareability (or exclusivity) defined in UFO – the ability of an individual to be part of multiple wholes of the same type at the same time [37]. In the OntoUML notation, the full diamond association is used for exclusive parts (the individual cannot be part of multiple wholes of the same type at the same time), while the empty diamond is used for shareable parts (the individual can be part of multiple wholes of the same type at the same time). However, the exclusivity of a part does not restrict the same individual to be a part of multiple wholes of different types [2]. An example of a model with a shareable and exclusive parts is shown.
### 3.7. Whole-Part Relations

Figure 3.10: Example of an OntoUML model with a shareable and exclusive parts. Inspired by [2, p.162]

in Figure 3.10 where a Person can be a FamilyMember only in a single Family, but he/she can be a Researcher in multiple ResearchGroups.

**Optionality and mandatoriness of parts.** Besides shareability, UFO also addresses various modes of optionality and mandatoriness of wholes and parts [2]. In UML, this is defined by the multiplicities of the relation. However, in UFO, also essentiality and inseparability of the parts is considered. In total, UFO distinguishes the following situations for the optionality of parts:

- **Optional part:** The whole can exists without any instance of the part. Moreover, it does not matter what instances of the part compose the whole. These instances can be exchanged or removed without afflicting the whole’s identity. For instance, a Car can have Airbags but they are not mandatory. Also, it does not matter what instances of Airbag are installed. In OntoUML, the optionality is defined by the minimum multiplicity of the part equal to 0.

- **Mandatory part:** The whole requires an instance (or instances) of the part. However, the instances of the part can be exchanged for other ones, without changing the identity of the whole, so far as the minimum number of part instances is greater or equal to the minimum multiplicity of the part. For instance, a Car must have an Engine, otherwise it is not perceived to be a car. In OntoUML, the mandatory part is defined by the minimum multiplicity of the part greater than 0.

---

2If we do not consider types of airbags or their location.
3. OntoUML

- **Essential part**: The whole requires specific instance of the part as its part. Moreover, this individual cannot be exchanged without afflicting the identity of the whole. For instance, a Car must have chassis, however, the chassis cannot be exchanged as it constitutes the identity of the car. In OntoUML, the essential part is defined by the essential meta-property of the relation. Moreover, the minimum multiplicity of the part must be greater than 0.

**Optionality and mandatoriness of wholes.** Similarly, the optionality and mandatoriness of the whole composing the part is addressed in UFO. In total, three levels are distinguished:

- **Optional whole**: The instance of the part can exist without being part of the whole. Moreover, the instance may change the whole it is part of or even start or stop being part of the whole at all. For instance, an Engine can be part of a Car but it can be moved to another car or just be removed and kept in a garage. In OntoUML, the optionality is defined by the minimum multiplicity of the whole equal to 0.

- **Mandatory whole**: The part instance requires to be a part of some whole. However, the instance of the part can change the whole it is part of, as long as it is not on its own without being part of a whole. For instance, a FamilyMember must have a Family it is member of, otherwise he/she is not perceived to be a family member. In OntoUML, the mandatory whole is defined by the minimum multiplicity of the whole greater than 0.

- **Inseparable part**: The part instance requires to be part of a specific whole. It is not possible for the part instance to change the whole it is part of. For instance, Chassis is always part of a Car, however, it participates on its identity and therefore cannot be simply replaced to a different car. In OntoUML, the inseparable part is defined by the inseparable meta-property of the relation. Moreover, the minimum multiplicity of the whole must be greater than 0.

The discussed example of the Car and its parts is shown in Figure 3.11. The example of a mandatory whole is shown in Figure 3.10.

**Immutability.** UFO also addresses the parthood essenciality of anti-rigid types. As an individual may change its instantiation of an anti-rigid universal, it is important to understand if that individual must be always related to the same whole/part when it is instance of that anti-rigid universal. This fact is defined by the immutability of the whole or the part, respectively. In other words, immutability is a property of the whole-part relation between an anti-rigid universal and a rigid universal.

---

3From the cognitive point of view, regardless the fact that the car VIN is located on the chassis.
3.7. Whole-Part Relations

The situation, when an anti-rigid whole individual must be related to the same specific part individual, is defined as *immutable part*. Contrary, the situation, when the anti-rigid part individual must be part of the same specific whole individual, is called *immutable whole*. In OntoUML, these constraints are defined as the meta-property of the relation between the two universals [2].

For instance, whenever a certain *Wheel* is attached to a *Car*, it requires some *Tyre* (which can be replaced by any other tyre) and certain *SafetyScrew* (which is unique to that particular wheel – without this particular screw, the wheel cannot be properly attached). This means that the *SafetyScrew* is immutable for the *Wheel*. Model for this example is shown in Figure 3.12.

### 3.7.2 Functional Complex

*Functional complex* is the most common case of wholes. It represents a whole consisting of various objects – components. Such whole is heterogeneous and complex – the components are of various types and play various roles in the complex [2, 72]. Also, the components may have various optionalities as discussed above, including the essentiality, inseparability and immutability.

Functional complexes are standard objects we perceive around us (in contrast to quantities, see below). Therefore they can be represented by any type of universal, although, functional complexes are usually Kinds. Also the components of the complexes are real objects – they are perceived as unique individuals in the domain of interest – and therefore they can also be of any universal kind. Because of this, even a functional complex can be a component of another functional complex. However, even when a component is essential and inseparable, thus not able to exists without being part of the complex, it is still perceived as a unique individual – which is in contrast to Modes, which are always perceived
only when inhering in their bearer.

As both the functional complex and its components may be various types of universals, there is no special stereotype used to identify their types in the OntoUML model. Only the parthood relation between the functional complex type and the component type is marked by a special stereotype – ≪ ComponentOf ≫.

An example of a functional complex is shown in Figure 3.11 where a Kind Car is defined as a functional complex composed of a mandatory Engine, optional Airbags and essential and inseparable Chassis.

3.7.3 Collective

Collective is another type of a whole. Similar to functional complexes, collective is composed by real objects of the domain of interest – its members. However, unlike components of functional complexes, these members are not distinguished in their type nor role [78]. "For example, if all ships of a fleet are conceptualized as playing solely the role of "member of a fleet" then it can be said to be a collection. Contrariwise, if this role is further specialized in "leading ship", "defense ship", "storage ship" and so forth, the fleet must be conceived as a functional complex." [2, p.183].

Collectives are closure systems – there is a unifying relation between all the members which gives them common shared cognitive meaning [2]. Moreover, collectives are maximal, and thus all instances satisfying this unifying relation are necessarily members of the collective. Thanks to this, collective has its own unique identity derived from the unifying...
The relation between a collective and its member is that of membership. This relation is derived from the unifying relation rather than defined by the extrinsic properties of the individuals. Also, this relation is not transitive – when collection A is a member of another collection B, the members of the collection A are not perceived as members of the collection B. For instance, even though John is a member of a sport club and this sport club is member of an international body, John is not member of the international body.

On the other hand, there is a relation called subcollection. This relation holds between two collectives where the unifying relation of one collective is a specialization of the unifying relation of the other collective. In other words, one collection is a subset of members of the other collection. For this relation, the following applies: all members of a collective A are also members of any collective B the collective A is subcollection of. For instance, John is a member of the defense players in his club, therefore he is also a member of the club as the defense players are subcollection of all club players.

The universal whose instances are collectives is called Collective universal. In OntoUML, the type representing such universal is marked by the stereotype «Collective». As members of a collective can be individuals of any kind, there is no special stereotype for them. The relation between the type of collective and the type of its (potential) members is defined by a parthood relation with the stereotype «MemberOf». Alternatively, a symbol M can be shown inside or beside the diamond of the parthood relation. The subcollection relation between the collectives is defined by a parthood relation between the Collective types with the «SubcollectionOf» stereotype. Alternatively, a symbol C can be shown inside or beside the diamond of the parthood relation.

An example of an OntoUML model with collectives is shown in Figure 3.13 where a Person is defined as a member of a DefensePlayers collective, which is a subcollection of SportClub collective. This collective is then a member of an InternationalBody.
3. OntoUML

3.7.4 Quantity

*Quantity* is the last type of wholes used in UFO. Unlike collectives and functional complexes, whose parts are objects, quantities represent *amounts of matter* (e.g. sand, clay, water, etc.) \[80\]. As any amount of matter consists of infinite number of subamounts of the same matter, quantities are defined in the following way: "*quantity K is a maximally self-connected object constituted by portions of K*" \[2, p. 179\]. According to this definition, it means that a quantity individual has its identity derived from the exact amount of the matter and changing the amount changes its identity – it is simply a different quantity.

A *Quantity universal* is such universal whose instances are qualities. In fact, Quantity universal is a specialization of the Kind universal, as it is also an ultimate Substance Sortal defining the identity principle for its instances. However, in contrast to Kind, its instances are not objects but quantities. In OntoUML, types representing Quantity universals are marked by the ≪Quantity≫ stereotype.

Furthermore, each quantity is always instantiated in certain form or container (e.g. a statue of clay, a bottle of water, etc.) \[80\]. It is not possible to think about a maximally self-connected object without some object defining its borders: "There is no mass, except the mass of a certain object. There is no stuff except the stuff a certain thing consists of" \[2, p. 177\]. However, the relation between the quantity and its container or form, called *containment*, is not essential – the quantity can change its form or container.

In OntoUML, the containment relation between the Quantity universal and the universal of its container is defined as an association between the particular types with the ≪Containment≫ stereotype. Moreover, as each quantity individual is maximal self-connected object, this relation is always *one-to-one* with the mandatory multiplicity of the container – the instance of the container can always contain only a single instance of the quantity, while the quantity exists only when it is contained in some instance of the container.

Although quantities are maximally self-connected objects, they still can be composed of other quantities in the same sense of quantity (e.g. alcohol in wine, sand in concrete, etc.) \[2\]. This way, we are able to define the fact, that certain quantity contains some other maximally self-connected object as its self-connected part. This relation is called *subquantity-of* and it is always essential. In OntoUML, this relation is defined as a parthood relation between the types representing the two Quantity universals with the ≪SubquantityOf≫ stereotype, or alternatively, as a parthood relation with the symbol $Q$ inside or beside the diamond of the parthood relation. Moreover, the relation is always strictly *one-to-one* and not shareable.

Moreover, UFO also uses relation of *constitution*. In context of quantities, it is the relation between a quantity representing amount of some matter and the matter itself (e.g. the relation between the vintage and certain quantity of wine). In such situation, the matter is an instance of some ultimate Substance Sortal other than Quantity (typically a Kind). In OntoUML, this constitution relation is defined by an association between the Quantity type and the other type with the ≪Constitution≫ stereotype.

An example of quantities is shown in Figure 3.14 where a model of Bottles contain-
3.8. Tools

Beside the tools used for UML modelling, there is also a few of tools supporting the OntoUML notation. OntoUML lightweight editor (OLED) [11, 81] is an environment for modelling with OntoUML, which also offers functions for model evaluation, validation or transformations into OWL, Alloy and SBVR. It also support imports from Enterprise Architect. However, it does not offer transformations into UML nor relational databases.

Menthor Editor [12] is a successor of OLED, providing more convenient environment for modelling. It supports model OntoUML validation, including the syntax of attached OCL constraints. It also provides model transformation into OWL and export into UML for Eclipse IDE or Ecore model. However, the tools does not support transformations into relational database models, nor generates additional constraints needed to define all the implicit constraints defined by the OntoUML universal types used in the model.

There is also an Enterprise Architect plugin [82]. This plugin adds a new profile, diagram and several patterns, allowing to create OntoUML diagrams and model elements. Unfortunately, this plugin does not provide any functionality regarding transformations or validations.

Additionally, there also exists a palette for UMLet editor [83], which can be used for creation of simple diagrams. However, UMLet is not a proper CASE tool, as it does not allow creating a complex model consisting of multiple diagrams reusing the same model elements.
Overview of Our Approach

As discussed in section 1.1, our goal is to encourage software engineers to use OntoUML as the conceptual language when modelling the data models of the domain. To support this goal, we define the rules for the transformation of such a model into its realization in a relational database, as it is the most common way of persisting application data. With such transformations, the OntoUML conceptual model may be used as the input artefact for the Model-Driven Development approach to define and generate the script for the creation of the database.

As mentioned in section 1.2, our approach to the transformation of an OntoUML PIM into its realization in a relational database consists of the following three steps (see Figure 4.1):

1. transformation of the initial OntoUML PIM model into the corresponding UML PIM model,
2. transformation of the UML PIM model from the previous step into the corresponding RDB PSM model,
3. transformation of the RDB PSM model from the previous step into the corresponding SQL ISM model.

In the first step, the initial OntoUML PIM model is transformed into a pure UML PIM model. However, as mentioned in section 2.1, it should hold that no information should be lost when transforming from a more abstract model into a more specific one. Since OntoUML applies certain constraints to the types based on the kind of universal represented by the particular type, these constraints should be carried over to the other consecutive models. In our approach, these constraints are realized by a combination of the generalization, specific multiplicities of the associations and additional OCL constraints for those constraints, which cannot be expressed directly in the diagrams. The resulting UML PIM model presents the very same semantics as the former OntoUML PIM model, however, it is defined by means of a standard well-known notation. The details of this step of the transformation are discussed in chapter 5.
4. Overview of Our Approach

In the second step, the resulting UML PIM model with the constraints derived from the initial OntoUML PIM model is transformed into a RDB PSM model. In this transformation, the UML classes with attributes are transformed into database tables with columns and the relations are transformed into references. Furthermore, we also transform the OCL constraints from the UML PIM model into the OCL constraints defined on the database model. Moreover, to preserve the same restrictions in the resulting RDB PSM model, we also address the meta-properties of the generalizations sets and the multiplicity constraints of the associations. In certain cases, this leads to defining additional OCL constraints. The details of this step of the transformation can be found in chapter 6.

In the final step, the resulting RDB PSM model with the OCL constraints from the previous step is transformed into an SQL ISM model. This model consists of SQL DDL scripts for creating the database schema – the tables, columns and standard SQL constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, etc.). Additionally, we deal with the realization of the OCL constraints defined on the PSM model to preserve the database consistency in context with the constraints derived from the initial OntoUML PIM model. The details of this transformation step are discussed in chapter 7.

Our approach was introduced in [A.7], where we outlined our approach to the transformation and the challenges of such transformation. In the sequel papers [A.8], [A.9] and [A.10], we discussed the transformation of Rigid (Kinds and Subkinds) and Anti-rigid (Roles, Phases) Sortal types, respectively, in more detail.

Although the transformation could be done in a single step consisting, i.e., by generating the SQL DDL scripts directly from the OntoUML model, our approach brings several advantages. First, the existing know-how for the transformation of UML models into relational databases may be utilized (see, e.g., [52] [84] [A.5]), as well as the existing tools supporting this transformation (e.g., Enterprise Architect [32]). Second, after each transformation step, the model may be analysed and refactored, on basis of the resulting model and the application domain; this refactoring may greatly simplify the model. In each of the chapters describing the individual steps of the transformation, these optim-
izations and variations of our approach are discussed. And third, the first step of the transformation may be used as a part of the transformation into any other platform, such as a pure object model of Smalltalk, an object-oriented data model of EJB\textsuperscript{1}, etc., since it is the transformation between the models on the same platform-independent level.

Furthermore, in OntoUML, only binary relations are used. When more entities are interrelated together, it is always through a Relator, which mediates the n-ary relation. Moreover, any n-ary relation can be always transformed into a set of binary relations \cite{27}. Therefore, in the approach discussed in this dissertation thesis, we address only binary relations between the entities.

In section 4.1, a complex example of an OntoUML PIM is presented. This example is used to illustrate our approach to the transformation in the individual chapters discussing the individual transformation steps.

4.1 Running Example

For the demonstration of our approach, a running example of a complex OntoUML PIM is used. This running example aims at creating a database model used by an information system of a library. The process starts with the domain analysis and the creation of the conceptual OntoUML model describing the key elements of the domain, their properties and relations. This model is then transformed into the other consecutive models applying our approach discussed in this dissertation thesis. The complete description of all the models is provided in the attached Running Example document (see Appendix B).

In the following part of the section, the individual parts of the OntoUML PIM model are discussed and explained. The complete conceptual OntoUML PIM model is shown in Figure B.2.

The key element of the domain is Work. This entity defines the piece of literature which is available in the library in various copies which can be borrowed by the clients. As it is the basic element having its own identity, it is classified as a Kind. For each work, its original title and the description of its contents is important.

Two types of works are kept in the library - books and periodicals. The Book entity represents standard books and monographs, while the Periodical entity represents various magazines, journals and other periodically published works. As these two entities define distinct types of works, they are classified as Subkinds of Work. Moreover, as there can be no combination of a periodical and a book, and no other types of works are considered, the generalization set of books and periodicals is defined \{disjoint,complete\}. Also, because of the completeness of the generalization set, the Work entity is defined abstract. For a periodical, its language and ISSN is important, if available, while for a book, its authors are important.

\footnote{1\textit{Enterprise Java Beans}, http://www.oracle.com/technetwork/java/javaee/ejb/index.html}
4. Overview of Our Approach

The works may be organized in series – a set of books or periodicals usually sharing the topic or story. The series is represented by the Collective entity Series, collecting two or more individual works as its members and defining the title of the series.

The excerpt of the conceptual OntoUML model of the works and its types is shown in Figure 4.2.

Author of a book is always a person represented by the Person entity. For each person, its first name (optionally) and last name (mandatory) are significant. The authorship of a book by an author is a material relation between the written book represented by the Role WrittenBook of the Book and the author represented by the Role Writer of the Person. This material relation is mediated by the Relator Authorship connecting the two Roles. As a single author can write many books and a single book can be written by many authors, the multiplicity of the Relator Authorship is 1..* to both Roles. The same multiplicities are also defined for the actual material relation. Moreover, the ratio of authorship of an author is significant for some books, represented by the ratio attributes of the Authorship entity.

The excerpt of the conceptual OntoUML model of the authorship is shown in Figure 4.3.

For each book, the book editions are significant – they are represented by the entity BookEdition. As the book edition is existentially dependent on the book itself, defining a variation of the book, it is classified as Mode. Similarly, for a periodical, its individual issues are significant, represented by the Mode entity Issue.

For each periodical issue, the number of the issue and the main topics are significant. For a book edition, the marking of the edition, ISBN and language are necessary.

As for both the periodical issue and the book edition the content, translation and publication are significant, a common supertype Edition was identified to define these shared properties. However, as both the periodical issue and the book editions have different identity principles derived from the types they characterize, and both of them are rigid, this supertype must be classified as Category. Furthermore, as the distinction
4.1. Running Example

between Issue and BookEdition is disjoint and complete, the generalization set was set \{disjoint,complete\}.

The translation of a book edition by a translator is represented by the Relator entity Translation which connects the person who participated on the translation represented by the Role entity Translator – another Role of a Person beside Writer – and the translated book edition represented by the Role entity TranslatedEdition. A single book edition may be translated by multiple translators as well as a single person can translate multiple book editions, which leads to the multiplicity 1..* of the Translation to the both Roles.

The publisher of a book edition is always a legal entity – a company or other type of a legal group of persons – represented by the Kind entity LegalEntity. For each such legal entity, its title and VAT are necessary. The publication of a book edition or a periodical issue is then defined by the Relator entity Publication. It defines the year of the publication and connects the Role entity Publisher of the entity LegalEntity and the Role entity PublishedEdition of the entity Edition. As a single edition always has only a single publisher, the multiplicity of the Publication entity in the relation to the PublishedEdition is set to 1. On the other hand, a single publisher may publish many books, therefore the multiplicity of the Publication entity in the relation to the Publisher is set to 1..*.

The excerpt of the conceptual OntoUML model of the editions is shown in Figure 4.4. The excerpt of the model showing the translation and publication of an edition is shown in Figure 4.5.

To be able to lend the books and periodicals to the readers and other clients, there must be some actual copies of them. In the conceptual OntoUML model, the copies are represented by the entity Copy. As a copy has its identity dependent on a book edition or a periodical issue – it represents the materialization of the respective edition – it is classified as Mode characterizing the Edition entity. Although the Edition is classified as Category, and therefore it does not define the identity principle required by the characterizing Mode, it is abstract, and therefore the real characterized type is one of its subtypes – either the
4. Overview of Our Approach

![OntoUML PIM of the Library - Book and periodical editions](image1)

**Figure 4.4:** OntoUML PIM of the Library - Book and periodical editions

![OntoUML PIM of the Library - Edition translations and publication](image2)

**Figure 4.5:** OntoUML PIM of the Library - Edition translations and publication
BookEdition or the Issue, with their own identity principles based on the types they characterize.

Beside this, also the condition of the copy is important. There are three states used for a copy in the library regarding its condition: undamaged copy when there is no apparent damage; damaged copy when there is some damage but the copy can still be lent; and destroyed copy when the damage is too big and the copy cannot be lent any more. As each copy is always in one of these states, they are defined as Phase entities Undamaged, Damaged and Destroyed, respectively, forming a phase partition of the Copy entity.

The excerpt of the conceptual OntoUML model of the copies and their condition is shown in Figure 4.6.

The copies of books and periodicals can be lent to clients of the library. As the copies may be lent to a person as well as a company or other legal subject, the Client entity is defined as a RoleMixin, defining the common ability to borrow copies of the Role entity RegisteredReader of the entity Person and the Role entity RegisteredLegalEntity of the entity LegalEntity.

Moreover, it is important to register contact information for the clients. Therefore the Client entity defines attributes phone and e-mail, together with a Quality PostalAddress with attributes of postal address of the client.

Beside that, regardless of the roles of both persons and legal entities, it is also necessary to distinguish the gender of the persons, defined by the Subkinds Man and Woman. Also, it is needed to register a unique name of the persons and legal entities, by which they are known in the library, and the place of activity of the subject, defined by the attributes city and country of the Category Subject, which is the supertype of entities Person and LegalEntity.

The excerpt of the conceptual OntoUML model of the clients is shown in Figure 4.7.

The fact that a copy of a work can be lent is defined by the entity Loan. It represents a single loan of a number of copies – represented by the Role entity LentCopy of the entity Copy – to a single client – represented by the RoleMixin entity Client. For each loan, the
4. Overview of Our Approach

Figure 4.7: OntoUML PIM of the Library - Clients

Figure 4.8: OntoUML PIM of the Library - Loans

date of borrowing and the deadline to return the lent copies are significant, represented by the respective attributes of the Loan entity.

The excerpt of the conceptual OntoUML model of the loans is shown in Figure 4.8.

To effectively deal with loans, it is useful to easily distinguish ongoing and finished loans. In the model, this distinction is defined by the phase partition of Phase entities.
4.1. Running Example

Ongoing and Finished. For finished loans, the returning date of the copies is important, represented by the corresponding attribute of the entity Finished.

Also the copy goes through several distinct states regarding the loans. It can be either available in the library for anyone to borrow it; or it is borrowed by someone at the moment; or it is discarded and is not available to be borrowed any more. A copy is always in one of these states and they are disjoint, therefore they are defined in the model as another phase partition of the Copy entity, consisting of the Phase entities Available, Borrowed and Discarded, respectively. For a borrowed copy, the active loan is significant, represented by the Formal relation between the Borrowed entity and the Ongoing entity. For discarded copies, the discarding date is important for evidence records.

The excerpt of the conceptual OntoUML model of the loan and copy states is shown in Figure 4.9.

For evidence purposes, all copies of books and periodicals must be marked by an evidence number. However, as in the evidence also various pieces of furniture is recorded, they also bear the evidence number. In the conceptual model, this is represented by the entity Furniture and the common supertype entity of Furniture and Copy entities called EvidenceItem. As furniture represent a clearly identified (and new) concept, it is classified as Kind. Then, as the EvidenceItem entity defines common properties of multiple rigid types with disjunct identity principles – in this case multiple a Kind and a Mode – it is classified as Category, defining the property of having the evidenceNumber value.

Moreover, it is useful to know, where a copy or a piece of furniture from the evidence is positioned. It is a common property of the entity Furniture and the entity Available – only available copies are stored in some shelf or bookcase – defined by the entity StoredItem. As this entity defines common properties of a rigid type – Kind entity Furniture – and an anti-rigid type – Phase entity Available – it is classified as Mixin. The position of a copy or piece of furniture is described by the number of the room and the description of exact location in that room, represented by the respective attributes of the StoredItem entity.

The excerpt of the conceptual OntoUML model of the evidence is shown in Figure 4.10.
4. Overview of Our Approach

Figure 4.10: OntoUML PIM of the Library - Evidence
In this chapter, we discuss the details of the first step of the transformation – the transformation of OntoUML PIM into UML PIM. In this transformation, the types representing various kinds of universals and relations defined in the OntoUML model are transformed into the standard UML classes and relations. However, the kinds of universals and relations used in OntoUML define certain semantics to the model elements. This semantics must be preserved during the transformation. Therefore, it is required to handle different OntoUML universal types in a different manner.

The chapter is structured in the following way:

- in section 5.1, the transformation of various OntoUML universal types from OntoUML PIM into their representation in UML PIM is discussed;
- in section 5.2, the discussion to our approach is provided, discussing various limitations of our approach and possible optimizations of the resulting UML PIM.

The complete UML PIM model created by applying the proposed transformations on the OntoUML PIM shown in section 4.1 can be found in Figure B.3.

### 5.1 Transformation of OntoUML Universal Types

As OntoUML is a light-weight extension of standard UML based on a profile of the UML Class diagram, the basics of the transformation are simple: each type in the OntoUML model is transformed into a standard UML class simply by removing the stereotype. All the attributes of the types are transformed into attributes of the classes, preserving their datatypes and multiplicities. The semantics defined by the kinds of universals defined for the individual types in the OntoUML model are then realized in the UML model by certain variants of generalization or association with specific multiplicities. When the semantics
cannot be expressed by these means, additional special constraints are defined to define the semantics. For this purpose, we use OCL constraints, as OCL is part of the UML standard and is well-known.

Our approach to the transformation of Sortal universal types from the OntoUML PIM into their ontologically correct representation in the UML PIM was introduced in [A.7]. In [A.8] and [A.10], more details about the transformation of Rigid Sortal universal types into their representation in the UML PIM, as well as their realization in the relational database, were discussed. In [A.9], the details of the transformation of Anti-rigid Sortal universal types into their representation in the UML PIM, as well as their realization in the relational database, were discussed.

In the sequel sections, the details of the transformation of the individual universal and relation types used in an OntoUML PIM are discussed. Also, the transformation is illustrated on the excerpts of the OntoUML PIM model of the Running Example presented in section 4.1.

5.1.1 Kinds and Subkinds

As Kind universal is rigid and defines the identity principle for its instances, its instances cannot change its instantiation without ceasing to exist. The same applies in UML: every object is always an instance of a class and this instantiation relation cannot be changed in the lifetime of the object. There, the transformation of Kinds is simple: each type classified as a Kind in the OntoUML PIM is transformed into a standard UML class in the UML PIM without any other constraints, keeping all its attributes.

Similarly, also the Subkind universal is rigid and provides the identity principle to its instances. Therefore, also the types classified as Subkinds in the OntoUML PIM are transformed into standard UML classes in the UML PIM, keeping all their attributes.

Finally, the generalization relation between the Subkind type and its identity ancestor (another Kind or Subkind type), defining the inheritance of properties and the identity principle in the OntoUML PIM, is also rigid, meaning that an instance of the subtype is also always instance of the supertype. As in UML the generalization relation has the same meaning, this generalization in the OntoUML PIM is realized in the UML PIM by a generalization as well.

When the generalization between the transformed Subkind type and its identity ancestor in the OntoUML PIM is part of a generalization set, it is part of the same generalization set of the transformed generalizations in the UML PIM as well, preserving the values of the meta-properties isCovering and isDisjoint of the set. Moreover, when any of the generalization sets (including phase partitions) is complete, meaning that instance of the supertype is always also instance of some of the subtypes from that generalization set, the class representing the supertype is defined abstract as it cannot have any direct instances.

An example of the applied transformation of Kinds and Subkinds can be found in Figure 5.1, where the excerpt of the UML PIM is shown with the transformed Kind Work and
5.1. Transformation of OntoUML Universal Types

Figure 5.1: Excerpt of the UML PIM with transformed Kind and Subkinds

Subkinds Book and Periodical, including the generalization set between them (see Figure 4.2 for the original OntoUML PIM excerpt).

5.1.2 Roles

As Role universals are also defined in the OntoUML PIM as types, defining the properties of its instances, they are transformed into the UML PIM as standard UML classes with the appropriate attributes. However, as the Role universal is anti-rigid, defining properties of the instances of its identity bearer only when related to other individuals, these instances can change the fact of instantiation of the Role universal. As this is not possible in UML, it cannot be represented by generalization in the UML PIM. Instead, the generalization relation between the Role type and the type representing its identity bearer in the OntoUML PIM is transformed into an association between the class representing the transformed identity bearer type and the class representing the transformed Role type.

To preserve the meaning of the identity inheritance of the relation between the Role universal and its identity bearer defined in the OntoUML PIM, the association between the Role class and the identity bearer class must have specific constraints:

- The multiplicity at the side of the identity bearer class is $1..1$, as the instance of the Role class can exist only when related to the (exactly one) instance bearing the identity of the individual – this is the entity playing the role.
- The multiplicity at the side of the Role class is $0..1$, as the instance of the identity bearer representing the individual may or may not have the role, depending on the existence of the relation to the truthmaker in the reality. On the other hand, the individual either plays the role or not, regardless of the actual number of the related instances. Therefore, the maximal multiplicity is restricted to 1.
5. Transformation of OntoUML PIM into UML PIM

- The side of the identity bearer class must be immutable, as the instance of the class representing the Role cannot change its bearer – the bearer is part of its identity.

With this association, the individual which is an instance of the Role universal are represented by the pair of related instances of the identity bearer class and the Role class, while the individual which is not an instance of the Role universal is represented only by the instance of the identity bearer class. Therefore, the association has the meaning of is a, as both instances represent the same individual, while the identity is defined from the identity bearer class.

Furthermore, the type representing the Role universal in OntoUML PIM is always connected by a mandatory relation to the type representing the truthmaker universal. However, as this relation can be either formal or material, we discuss it in more detail in subsection 5.1.5.

The example of the applied transformation of Role types can be found in Figure 5.2, where the excerpt of the UML PIM with the transformed Roles Writer of Kind Person and WrittenBook of Subkind Book is shown (the original OntoUML PIM is shown in Figure 4.3).

5.1.3 Phases

Similarly to Roles, Phase universals are also defined in the OntoUML PIM as types, defining the properties of its instances. Therefore, they are transformed into the UML PIM as standard UML classes with the appropriate attributes. As the Phase universal is also anti-rigid, defining properties of the instances of its identity bearer only when in certain state or phase of its existence, the individuals who are instances of the identity bearer can change the fact of instantiation of the individual Phase universals. As this is not possible in UML, it cannot be represented by generalization in the UML PIM. Instead, the whole generalization sets of the Phase types and the type representing its identity bearer in the
5.1. Transformation of OntoUML Universal Types

OntoUML PIM – called *phase partitions* – must be realized in a more complicated way: either by *exclusive phase associations* or by an *abstract phase*.

### 5.1.3.1 Exclusive Phase Associations

One possible realization of the *phase partition* in the UML PIM is based on exclusive associations. Each of the generalization relations between the *Phase* type and the identity bearer type is transformed into a one-to-one association between the classes representing the respective types. As this association represents the same type of relation as in the case of *Roles*, the same constraints must be met:

- The multiplicity at the side of the identity bearer class is $1..1$, as the instance of the *Phase* class can exist only when related to the instance bearing the identity of the individual – this is the entity being in that phase.

- The multiplicity at the side of the *Phase* class is $0..1$, as the instance of the identity bearer representing the individual may or may not be in the phase represented by an instance of that particular *Phase* class.

- The side of the identity bearer class must be *immutable*, as the instance of the class representing the *Phase* cannot change its bearer – the bearer is part of its identity.

The example of application of this transformation is shown in Figure 5.3, where the excerpt of the UML PIM is shown with the transformed *Phases* *Undamaged*, *Damaged* and *Destroyed* of the *Kind Copy* defined in Figure 4.6.

Moreover, a special constraint must be defined to preserve the phase partition meta-properties \{disjoint,complete\}. This constraint defines the restriction, that each instance of the identity bearer class is related exclusively to an instance of exactly one of the related *Phase* classes from the phase partition – therefore we call it *exclusivity constraint*.

![Figure 5.3: Excerpt of the UML PIM with transformed Phases with exclusive associations](image-url)
5. Transformation of OntoUML PIM into UML PIM

**Constraint 5.1** Exclusivity constraint for the exclusivity of the phases of a Copy defined as OCL invariant

context Copy inv EX_Copy.Condition:
self.undamaged <> OclVoid XOR self.damaged <> OclVoid XOR self.destroyed <> OclVoid

As this exclusivity is based on the individual Phase types, not the number of related instances of that type, the constraint is based on checking existence. The potential existence of only a single unique related instance is ensured by the maximal multiplicity of 1 for the Phase classes. As discussed in Chapter 4, we use OCL invariants to define such restrictions to persist the semantics of the OntoUML universal types. Therefore, the OCL invariant is defined in the following form:

- The constraint is defined in context of the identity bearer class.
- The name of the constraint is generated by concatenating prefix EX, the name of the class and the name of the generalization set, or the name identity bearer class and the meaning of the phase partition.
- In the body of the constraint, each of the exclusive associations to the Phase classes is compared with the OclVoid using the <> operator and all of them are connected using the XOR operator – only one association can be non-empty.

An example of the exclusivity constraint for the phase partition defined in Figure 4.6 is shown in Constraint 5.1

5.1.3.2 Abstract Phase

Another possible realization is definition of a special abstract Phase class. Each of the real phases defined by the Phase types in the OntoUML PIM is transformed into a UML class specializing this abstract Phase class. Together, all the transformed Phase classes form a {disjoint,complete} generalization set corresponding to the phase partition in the OntoUML PIM.

The relation between the identity bearer and its phases is then realized by a one-to-one association between the class of the identity bearer and the abstract Phase class. The multiplicities of this relation are 1..1 at the both ends, as an individual with the identity of the identity bearer is always in some phase and the phase instance cannot exist without its identity bearer. Thanks to the abstraction of the special phase and the completeness and disjointness of the generalization set, the instance of the identity bearer is always related to an instance of one of the actual Phase classes, exactly as defined in the OntoUML PIM. Moreover, with this variant, no special exclusivity constraint is needed as the exclusivity is defined by the special abstract phase generalization set. However, as in the case of exclusive phase associations, the association between the Phase class and the identity bearer class must be immutable at the side of the identity bearer class.
5.1. Transformation of OntoUML Universal Types

![Diagram](image)

Figure 5.4: Excerpt of the UML PIM with transformed Phases with an abstract phase class

In Figure 5.4, the excerpt of the UML PIM of the running example is shown with the transformed Phases Undamaged, Damaged and Destroyed of the Kind Copy. The phase partition is realized by a special abstract Phase and a \{disjoint, complete\} generalization set of the actual Phase classes.

5.1.4 Mixins

As discussed in subsection 3.5.4, all types of Mixin universals define properties shared by multiple universals with different principles of identity. Therefore, similar to other Substance universals, also types representing Mixin universals are transformed into standard UML classes with the attributes representing its properties. As Mixin universals are always abstract, also the classes realizing them are always abstract.

Also the generalization sets specializing the types representing Mixin universals in the OntoUML PIM are transformed into standard generalization sets in the UML PIM, as the generalization relation between the Mixin universals and the universals specializing them is always rigid.

This transformation can lead to multiple inheritance – a single class specializing multiple Mixin classes or a Mixin class and a class representing its identity bearer. However, UML on the conceptual level allows usage of multiple generalizations for a single class, combining the properties inherited from multiple superclasses [36].

An example of the applied transformation of types representing Mixin universals can be found in Figure 5.5, where the excerpt of the UML PIM is shown with the transformed Category Subject of Kinds Person and LegalEntity, and RoleMixin Client of the Roles RegisteredReader and RegisteredLegalEntity, shown originally in Figure 4.7.

5.1.5 Moments

All Moment universals – Qualities, Modes and Relators – define properties of individuals whose identity is dependent on other individuals. Therefore, in the OntoUML PIM, they
5. Transformation of OntoUML PIM into UML PIM

are represented as types with a mandatory relation or relations to other types. However, as the Moment universals provide the identity principle to their instances (similarly to Sortal universals), they can be transformed into the UML PIM as standard UML classes.

As the relation between the type representing the Moment universal and its bearer represents de facto a simple relation between the instances of the related types, only with certain mandatory multiplicities, these relations can be transformed into the UML PIM as standard associations between the classes representing the transformed types.

However, as the relation to particular specific instance of the other universal is part of the identity of the instance of the Moment, the association must be immutable at the end of the class representing the bearer of the Moment (see subsubsection 2.2.1.2 for the notion of immutable associations). Moreover, in the case of Qualities, both ends of the association must be defined immutable, as the particular instance of the Quality type represents certain particular structured property of the bearer, and thus cannot be replaced.

The only exception to this proposed transformation is the actual material relation between universals, which is mediated by a Relator. As this relation is derived from the Relator and its mediation relations to the individual related universals, realizing such relation in the UML PIM would lead to data redundancy and a risk of inconsistency. Therefore, the actual material relations are not transformed into the UML PIM, as they can be derived from the relations between the classes representing the transformed related types and the class representing the Relator.

Furthermore, formal relations can be also directly transformed into the UML PIM as standard associations, as they only define existing relation between instances of the related types based on their intrinsic properties and constitute no structural constraints.

Example of this applied transformation can be found in Figure 5.2, where the Relator Authorship with the mediation relation shown in Figure 4.3 is transformed into the class

![Figure 5.5: Excerpt of the UML PIM with transformed Mixins](image-url)
5.1. Transformation of OntoUML Universal Types

5.1.6 Part-Whole Relations

As discussed in section 3.7, although the Part-Whole relations in the OntoUML PIM are defined using the same visual notation as composition and aggregation in UML, OntoUML uses different semantics for the relations – the full or empty diamond of the relation defines the shareability of the parts instead of the existential dependency. Therefore, in our approach, when transforming the Part-Whole relations from the OntoUML PIM into the UML PIM, they are not transformed into composition nor aggregation to prevent misunderstanding of the relations. Instead, the Part-Whole relations are transformed into standard UML associations between the classes representing the related types with specific multiplicities and other constraints depending on the properties of the Part-Whole relation.

5.1.6.1 Properties of Part-Whole Relations

Although OntoUML uses several distinct kinds of Part-Whole relations and kinds of Wholes, all of them use the same basic properties.

Shareability. The shareability of the parts defines the fact, that a single instance of the part class can be part of multiple whole at the same time. However, it is de facto also expressed by the maximal multiplicity at the side of the whole. Therefore, when
transforming the Part-Whole relation from the OntoUML PIM into the UML PIM, the shareability can be simply expressed in the UML PIM by the maximal multiplicity of the association at the side of the class representing the whole type without any special constraints.

**Optionality and mandatoriness.** Also the optionality and mandatoriness of the parts to be part of a whole is defined by the multiplicities, and therefore require no special constraints in the UML PIM. The same also applies for the optionality and mandatoriness of the whole to have the parts.

**Essentiality and inseparability.** The essentiality and inseparability expresses the existential dependency of the parts on a specific instance of the whole type and the whole on specific parts, respectively. Therefore, it is realized by setting the appropriate side of the association immutable.

**Immutability.** Finally, as the immutability of parts and wholes in the OntoUML PIM restricts immutability of the relations of anti-rigid types, which are represented by creating or destroying instances of the classes representing the particular anti-rigid types, they cannot be simply realized in the UML PIM. The discussion to this constraint and suggestion of possible solutions is provided in subsection 5.2.5.

### 5.1.6.2 Types of Wholes, Parts and their Relations

In the OntoUML PIM, various types of wholes, their parts and relations between them are used. In the following paragraphs, the transformation of these types is discussed.

**Functional complex.** As discussed in subsection 3.7.2, functional complex is a whole composed of components of potentially distinct types playing different roles in the functioning of the complex. As the type representing the functional complex can be of any universal type and it defines properties of the instances of the complex, it is transformed into a UML class according to its universal type. The component-of relation between the complex type and the type of its parts is also transformed into standard UML association with appropriate multiplicities and immutability according to the properties of the Part-Whole relation, as it simply defines relation between the instances of the particular types.

An example of this applied transformation can be found in Figure C.1.

**Collective.** Collective is another type of a whole distinguished in OntoUML, whose all members are of the same type and play the same role. The type classified as Collective in the OntoUML PIM defines the properties of the collection of the members. Therefore, the Collective type is transformed into a standard UML class, whose instances represent the Collectives, with the relation to the class representing the type of its members.
As discussed in subsection 3.7.3, the actual membership of the members in the Collective is based on a unifying relation. However, in the OntoUML PIM, the unifying relation is not defined – only the member-of relation defining the type of members of the Collective is defined. Therefore, this relation can be transformed into the UML PIM as standard UML association between the classes representing the Collective type and the type of its members and the unifying relation determining the relations between the actual instances must be realized on the application level.

Also the subcollection-of relation between various Collective types can be simply transformed into standard UML association between the classes representing the Collectives, as it defines the type of subcollectives.

An example of this applied transformation can be found in Figure C.2.

**Quantity.** As discussed in subsection 3.7.4, a Quantity is a maximally self-connected amount of matter contained in certain container or constituting certain form. Types classified as Quantities in the OntoUML PIM define properties of such amount of matter contained in instances of the other type. Therefore, these Quantity types are transformed into standard UML classes, whose instances represent the particular Quantity.

The relation of containment connects the type of the Quantities to the type of the container in the OntoUML PIM. As this relation defines relation between instances of different types, it can be realized as standard UML one-to-one association. Moreover, as the relation is always essential, the side of the container is defined as immutable.

Also the subquantity-of relation between two types classified as Quantities is transformed into standard UML one-to-one association between the classes representing the related Quantity types, as both quantities have their own independent identity. The essentiality of the relation is then realized by the immutability of the relation at the side of the containing Quantity type.

The constitution relation between the Quantity and the matter itself is defined in the OntoUML PIM as a relation between the type of the Quantity and the type representing the matter. Therefore, also the containment relation is transformed into standard UML association. However, as it participates on the identity of the Quantity, the association end of the class representing the type of the matter is defined as immutable.

An example of this applied transformation can be found in Figure C.3.

**5.2 Discussion**

In this section, we discuss various limitations of our approach and possible optimizations of the UML PIM created by the transformation of the OntoUML PIM in this section. These optimizations are usually based on the knowledge of the domain of discourse, so no important information is lost during the refactoring of the model. Therefore, these optimizations cannot be done by any automatic tool but must be done manually after the whole model is transformed. Nevertheless, these optimizations are not mandatory and
5. Transformation of OntoUML PIM into UML PIM

necessary for successful realization of the initial OntoUML model in a relational database. These optimizations just offer a way to simplify the model if possible.

5.2.1 Optimization of Rigid Generalization Sets

As discussed in section 3.4, generalization is used in OntoUML to define that certain types are special cases of a more general type. It is used to define properties (identity principle, attributes and relations) common to several distinct types by defining these properties in their supertype.

In the case of rigid generalization sets (and generalization relations in general) in the OntoUML PIM, e.g. between Kinds and Subkinds as discussed in subsection 5.1.1 or between Non-Sortal type and a Sortal type, it is transformed into standard generalization even in the UML PIM. However, as OntoUML is based on cognitive science, the subtypes often define only certain meaning of the individuals or certain distinction of possible variants, without any specializing properties and any further use for this distinction\footnote{However, from the cognitive point of view, this distinction is important}. Transformation of such generalization leads to empty classes specializing the superclass defining the important properties.

An example of such generalization set is shown in Figure 4.7, where the Kind Person is divided into two Subkinds Man and Woman without any additional properties. This classification is used just to define, that we need to distinguish between men and women in the domain. When applying the standard transformation as discussed in subsection 5.1.1, two empty classes specializing the superclass a created in the UML PIM as shown in Figure 5.8.

In such cases, the individual subclasses forming the same generalization set may be reduced to an enumeration values of a special discriminator attribute defined in the superclass. Then, instances of a particular subtype from such generalization set in the OntoUML PIM are represented in the UML PIM by instances of the superclass with the discriminator value matching the particular subtype. This optimization has the following limitations:
5.2. Discussion

Person
- firstName: String [0..1]
- lastName: String
- gender: String {readOnly}

Figure 5.9: UML PIM with transformed generalization set of *Subkinds* realized by the discriminator attribute

**Constraint 5.2** OCL invariant for the possible discriminator values

```
context Person inv EN_Person_Gender:
self.gender = 'Man' OR self.gender = 'Woman'
```

- The *discriminator* attribute is constrained by *enumeration constraint*, restricting the possible values to represent the reduced subclasses according to the meta-properties *isDisjoint* and *isCovering* of the generalization set.

- The special *discriminator* attribute is defined as *immutable* footnoteEA marks the attribute by \{readOnly\} in the diagrams., as the reduced generalization set is rigid and no instance of the superclass can change the subtype it represents.

As the name of the *discriminator* attribute, the name of the reduced generalization set can be used, or according to the meaning of the generalization set. The enumeration constraint used to restrict the possible values of the *discriminator* attribute is defined as an OCL constraint in the following form:

- The constraint is defined in context of the superclass of the generalization set.

- The name of the constraint is generated by concatenating prefix EN, the name of the class and the name of the *discriminator* attribute.

- In the body of the constraint, the value of the *discriminator* attribute is compared with the names of the individual subclasses using the equality operator (=). In the case of an *incomplete* generalization set, the name of the superclass is included. In the case of a *overlapping* generalization set, all combinations of the subclass names are included. All the comparisons are joined using the OR operator.

An example of such optimization of the rigid generalization set shown in Figure 5.8 is shown in Figure 5.9. From the knowledge of the domain, we can derive that the distinction between men and women is based on the gender of the particular person. Therefore, the discriminator attribute is named *gender*. The enumeration constraint for the *gender* attribute can be found in Constraint 5.2.

Although this approach can be used also for generalization sets where the subtypes define additional attributes or relations for the instances of the individual subtypes, we do
5. Transformation of OntoUML PIM into UML PIM

recommend it. It is because these attributes and relations would be needed to be defined in the superclass, but the multiplicity constraints would be much more difficult to realize, because they would depend on the subtype, which the instance of the superclass represent. This also applies for the generalization sets of various types specializing various Mixin universal types (e.g. Categories, RoleMixins, Mixins, and others) where the actual types usually define their own attributes and relations, and therefore, this optimization is not useful.

Nevertheless, this proposed optimization of rigid generalization sets requires careful consideration. Reducing the subclasses into values of the discriminator attribute of the superclass leads to losing the concept of the particular subclasses in the model. This can be a problem for instance in regards to evolution of the model, when there is no actual class to carry new attributes identified for the subtypes in the OntoUML PIM.

5.2.2 Optimization of Phases

As discussed in subsection 3.5.3, Phases define properties (attributes and relations) of individuals with certain identity principle in distinct stages in their existence. However, in many cases, they just define the distinction of the stages without any specific properties. In such cases, transformation of the Phases and their phase partition leads to a set of empty classes, related either by exclusive associations to the class of the identity bearer, or forming a generalization set of an abstract Phase class.

In such cases, similarly as for the rigid generalization sets, the whole phase partition can be reduced into values of a special phase attribute defined in the class of the identity bearer:

○ In the case of exclusive associations, it is reduced into a phase attribute of the class related to all the phases by the one-to-one associations and restricted by the exclusivity constraint.

○ In the case of abstract Phase class, the generalization set representing the phase partition is rigid, and therefore according to subsection 5.2.1, it can be reduced into a single attribute of the abstract Phase class. However, as this class is related to the class of the identity bearer by strictly mandatory one-to-one relation and does not define anything else, both these classes can be joined into a single class with the properties of the identity bearer and the special phase attribute realizing the phase partition.

Similarly to the optimization of rigid generalization sets discussed in subsection 5.2.1, the special phase attribute is constrained by enumeration constraint, restricting the possible values to match the reduced Phase classes. The enumeration constraint has the very same form as in the case of rigid generalization sets. However, as the phase partition is always \{disjoint, complete\} expressed by the meta-properties of the generalization set specializing the abstract Phase class or the exclusivity constraint of the exclusive associations, the value of the special phase attribute cannot match the name of the identity
5.2. Discussion

Figure 5.10: UML PIM with transformed phase partition realized by the phase attribute

Constraint 5.3 OCL invariant for the possible phase values

```oclan
context Copy inv EN_Copy.Condition:
  self.condition = 'Undamaged' OR self.condition = 'Damaged'
  OR self.condition = 'Destroyed'
```

bearer class nor a combination of the Phase classes. Furthermore, the special phase attribute is not restricted by the immutability constraint, as the Phases are anti-rigid and the original associations between the class of the identity bearer and the Phase classes allow changes of the Phase instance. For the name of the enumeration constraint, the name of the original phase partition can be used or the name of the exclusivity constraint.

An example of the applied optimization can be found in Figure 5.10 where the reduced phase partition shown in Figure 4.6 is shown. The original UML PIM showing the result of the standard transformation using the exclusive associations can be found in Figure 5.3 and the result of the standard transformation using the abstract Phase class can be found in Figure 5.4. The enumeration constraint for the special attribute condition can be found in Constraint 5.3.

Although this optimization can be also used in the cases when the Phases define certain specific properties of the individuals in the particular phases, it would lead to complicated multiplicity constraints for the individual attributes, as they would be dependent on the actual value of the phase attribute of the instance. Therefore, we do not recommend using this optimization in such cases.

In any case, this proposed optimization of phase partitions always requires careful consideration of the analyst or designer. It is dependent on the knowledge of the domain of discourse, as by the reduction of the Phases into values of the phase attribute, the concepts of the individual Phases are lost. This may cause problems in context of evolution of the model, for instance, when a new property of certain Phase is identified.

5.2.3 Optimization of Roles

As discussed in subsection 3.5.2, Roles are used to define properties of certain individuals when they are related to some other individual – called truthmaker of the role – in the OntoUML PIM. However, in many cases, the Roles define only the actual relation to the truthmaker, without any additional properties. Therefore, when applying the standard transformation discussed in subsection 5.1.2, it leads to an empty class related by an
5. **Transformation of OntoUML PIM into UML PIM**

![Diagram](image)

Figure 5.11: UML PIM with transformed *Roles* optimized into an optional relation between the identity bearer and the truthmaker

immutable mandatory one-to-one association to the class representing the identity bearer of the *Role* and the mandatory association to the class representing the *truthmaker*.

In such cases, thanks to the one-to-one relation between them, the class representing the identity bearer and the class representing the *Role* can be joined together into a single class combining the properties of the identity bearer and the mandatory association to the class representing the *truthmaker*, including the maximal multiplicities and eventual immutability constraints. However, as the association between the identity bearer class and the *Role* class is optional – the individual with the identity of the identity bearer may or may not play the role – the association to the *truthmaker* must be defined optional (minimal multiplicity of 0). Then, the instance of the *identity bearer* class related to no instances of the *truthmaker* class represents a simple instance of the *identity bearer* type and the *Role* type.

Moreover, as the type of the *identity bearer* can be related to multiple disjunct *Roles* representing different relations to the same type of *truthmaker*, the name of the *Role* class in a particular relation can be expressed in the UML PIM by the name of the role of the identity bearer class in the particular association.

An example of the application of this optimization can be found in Figure 5.11, where the *Writer* and *WrittenBook* *Role* classes shown in Figure 5.2 are reduced.

Even this type of optimization requires careful consideration of the analyst or designer, as by reducing the *Role* class, the concept of the *Role* type is lost. This may cause problems in context when new properties of the *Role* are identified. Also, this optimization is not recommended in cases when the *Role* class define some properties for the related individuals, as the multiplicity constraints for such properties would have to be dependent on the existence of the relation to an instance of the *truthmaker* class.

### 5.2.4 Optimization of Relators

As discussed in subsection 3.6.3, types classified as *Relators* mediate *material* relations between other types in the OntoUML PIM, defining properties of this relation. When transforming the OntoUML PIM into the UML PIM, the *Relator* types are transformed into standard UML classes defining the properties of the *Relator* and the associations to
5.2. Discussion

the classes representing the types related by the *material* relation. However, in many cases, the *Relator* is defined in the OntoUML PIM only to define the actual mediator of the *material relation* without any properties. In such cases, it results into an empty class defining only the mapping of related instances of the two types.

In such cases, the *Relator* class can be removed and replaced by direct association between the two classes representing the types related by the *material* relation in the OntoUML PIM. The multiplicities of this association can be derived from the multiplicities of the removed associations to the removed *Relator* class. However, as the association in fact represents the actual original *material* relation from the OntoUML PIM, its multiplicities can be used. Furthermore, the name of the removed *Relator* class can be used as the name of the association to distinguish it from other relations.

An example of this applied optimization can be found in Figure 5.12, where the *Relator* shown in Figure 5.2 is removed and substituted for a direct association between the classes *Writer* and *WrittenBook*.

Such reduction is possible only in cases when the *Relator* class does not define any properties of the *material* relation it represents but the relations to the related classes. Otherwise, these properties would be lost. Furthermore, this reduction is appropriate only in situations, when it is not needed to preserve the concept of the relation *truthmaker*, which the *Relator* class represents, for instance because of more precise definition of multiplicities of the related instances. The lost of the concept can also cause problems when evolving the model, for instance by identifying new properties of the relation. Therefore, even this optimization cannot be used automatically and requires careful consideration of the analyst or designer.

Figure 5.12: UML PIM with transformed *Relator* reduced into optional association between the related classes

---

2This is important from the cognitive point of view and correct understanding of the domain according to UFO.
5. Transformation of OntoUML PIM into UML PIM

5.2.5 Immutable Parts and Wholes

As discussed in section 3.7, UFO and OntoUML also address the Part-Whole relations. Besides the optionality, mandatoriness, essentiality and inseparability of parts and whole, UFO also utilizes the concept of immutability of the Part-Whole relation. This property is used when the Part universal or the Whole universal is anti-rigid and it restricts the relation to certain specific individuals in the case, when the other individual is instance of the particular anti-rigid type. The property immutable part defines, that the whole individual contains the other individual as its part only in certain situation (for instance, in certain Phase or when having certain Role), but whenever in such situation, it always composes the same individual. On the other hand, immutable whole defines the fact, that the individual is part of a whole individual only in certain situation, but it is part of the same whole individual whenever it is in that particular situation. An example of such immutable Part-Whole relation is shown in Figure 3.12.

However, the anti-rigidity is realized in the UML PIM by creating and destroying the instances of the classes representing the anti-rigid types and relating them with the instance of the class defining the identity principle of the individuals. Therefore, it is not possible to preserve the information about the instance of the part class, to which an instance of the identity bearer class was related through the instance of the class representing the anti-rigid type, when this anti-rigid instance is deleted.

To preserve such information when changing the instantiation of the class representing an anti-rigid type, the data manipulation would require special treatment. Instead of deleting the anti-rigid instances when loosing the properties, it would require to only invalidate such instances and ignore them in further data operations. On the other hand, instead of creating new instances directly, the existing invalid instances would need to be searched and, if found, made valid again to restore the previously existing relation.

However, this aspect of the data modelling is out of scope of this thesis and is not discussed further. Therefore, in the current state of our approach, we are not able to define such immutability constraints, in respect of the non-rigid types, in the UML PIM and the constraints must handled on the application level.
In this chapter, we discuss the details of the second step of the transformation of an OntoUML conceptual model into its realization in a relational database – the transformation of UML PIM into RDB PSM.

The input for this transformation is the UML PIM model containing the classes representing the individual domain entities, their attributes and relations. Together with the model elements, the UML PIM model also contains various OCL constraints defined to preserve the semantics defined by the OntoUML universal and relation types defined in the initial OntoUML PIM model. When transforming the UML PIM model into the RDB PSM model, all the model elements and their properties, as well as the additional constraints need to be processed and transformed into their representation in the PSM model.

The chapter is structured as follows:

- **section 6.1** the transformation of various constructs from UML PIM is discussed;
- **section 6.2** the discussion to our approach for this step of the transformation is provided.

The complete RDB PSM model created by applying the proposed transformations to the UML PIM model shown in Figure B.3 can be found in Figure B.4.

### 6.1 Transformation of UML PIM Constructs

In this section, the details of the transformation of various constructs used in the UML PIM into their representation in the RDB PSM are discussed. During this transformation, the UML classes of the UML PIM need to be transformed into database tables, the classes’ attributes into the tables’ columns and the relations between the classes into references and FOREIGN KEY constraints.
6. Transformation of UML PIM into RDB PSM

Along with the classes, attributes and relations, also the OCL constraints defined for various model elements should be transformed. Although OCL allows to create a wide variety of constraints, we focus here on the transformation of the OCL constraints, which were generated during the transformation of the OntoUML PIM into the UML PIM and which are defined only to preserve the same semantics defined by the universal types used in the OntoUML PIM. As the RDB PSM consists of stereotypes classes representing the individual tables, the instances of such classes in the model represent the individual records in the tables. Therefore, standard OCL invariants can be used to define the constraints for the data in the tables.

Our approach to the transformation of the UML PIM into the RDB PSM was discussed for instance in [A.2, A.4, A.5]. In these papers, we focused on the transformation of binary relationships and special multiplicity values. Later, in [A.7, A.8, A.9, A.10], we discussed the transformation of the constructs of the UML PIM related to the representation of Rigid and Anti-rigid Sortal universal types used in an OntoUML PIM – the generalization sets and the exclusivity constraints.

In the following subsections, the transformation of the individual constructs of the UML PIM are discussed in details. To visualize the RDB PSM, we use the UML Data modelling profile of Enterprise Architect as discussed in subsection 2.2.2. Furthermore, as PSM is specific for concrete platform and technology and various database engines use different data types for storing the data, we use Oracle Database 12c [85] for the demonstration.

6.1.1 Classes and Attributes

The transformation of the classes and their attributes from the UML PIM into their representation in the RDB PSM is a well-known process, discussed for instance in [A.2] or [52].

In a relational database, the data are stored as records in rows of database tables. Actually, the data are stored in blocks in files in the file system of the database system, but they are handled using the non-procedural layer consisting of tables, views and rows, manipulated using SQL language [86]. A database table, in such approach, serves as a collection of the records as well as the definition of the structure of information contained by each of the records stored in that table, while each record contains values for each of the table’s columns.

In context of MDD approach to the software development, the goal of a RDB PSM is to define the structure of the database schema. Therefore, it consists of the definition of database tables, their columns and constraints.

As the classes in the UML PIM define the structure of the data objects in the domain of interest and the database tables in the RDB PSM define the structure of the records stored in the database, the transformation is straightforward: each class in the PIM is transformed into a database table in the PSM. Then, each instance of the class can be stored as a record in the corresponding database table. In our approach, we also transform the name of the class from the CamelCase notation into the UPPERCASE_UNDERSCORE notation of the
6.1. Transformation of UML PIM Constructs

Usually, each of a PIM class’s attributes is transformed into a column of the corresponding database table. During this transformation, the PIM data type of the attribute (e.g. String, int, Date) must be converted into a corresponding database data type (e.g. CHAR, VARCHAR2, NUMBER, INTEGER, DATE, etc.). Therefore, additional information – the datatype mapping – is required for such a transformation. This mapping is dependent on the data types used in the PIM, as well as the database engine used for the realization of the model.

Beside the data type, also the multiplicity of the attribute values needs to be handled (see subsection 2.2.1.1). When the maximal multiplicity is 1, it can be simply transformed into a table column. Depending on the minimal multiplicity, the column is defined with the NOT NULL constraint (for the value of 1) or not (for the value of 0). However, when the maximal multiplicity is not limited (value of *), it needs to allow to store multiple values for the same record. This can be achieved by refactoring the attribute into a separate class representing the value of the attribute, connected to the former class by a many-to-one association representing the possibility of the instance of the former class to have multiple values of the attribute. Such refactored class with the association then can be transformed into the RDB PSM in the standard way as discussed in subsection 6.1.2. According to the specification of SQL:2003 [87], the mechanism of variable arrays or nested tables might be also used. However, as many of the current database engines do not support this specification, we do not consider such realization of the attributes with multiple values.

Moreover, in general, both the minimal and maximal multiplicity can be also set to a value different from the standard values of 0, 1 or *. In such a case, these special multiplicities need to be handled in a special way discussed in subsection 6.1.4.

Furthermore, the implementation of a data model in the relational database needs unique identifiers for all records stored in each database table. However, because the PIM by definition does not contain any implementation specific characteristics, an additional special column in the corresponding table needs to be generated during the transformation to contain the record identification. In our approach, the name of this column is generated by concatenating the name of the table and the postfix "_ID". On this column, the PRIMARY KEY constraint is defined to ensure the unique values for all records in that table, and also to enable the possibility to refer to a record in this table by a FOREIGN KEY constraint using the value from this column (see subsection 6.1.2). Even if the class in the PIM contains some business attribute that uniquely identifies each object, it is not possible to automatically identify it during the transformation, and therefore, the special PRIMARY KEY column is generated.

An example of the basic transformation of a PIM class into a database table in the RDB PSM is shown in Figure 6.1. The class Work representing the original Kind Work is transformed into LENT COPY. The reason is that some database engines such as Oracle Database are case-insensitive and the names of the tables might get confusing.
6. Transformation of UML PIM into RDB PSM

Figure 6.1: Transformation of a PIM class into a database table in the RDB PSM

transformed into the table WORK with the columns TITLE and DESCRIPTION. Moreover, the column WORK_ID with the PRIMARY KEY constraint PK_WORK was generated to hold the unique identifier for each work stored in the table.

6.1.2 Associations

In UML PIM, associations are used to define relations between instances of the associated classes. These relations are bi-directional – the related instances are related to each other in both directions. On the other hand, in relational databases, the relations between records representing related entities are realized by references [52]. These references are based on the value of the identifier of the related record saved in a column of the other table. Unlike the associations in the PIM, the references in the database are always uni-directional. Therefore, when transforming the associations from the UML PIM into the RDB PSM, the direction of the reference realizing it is crucial. We discussed our approach to the transformation of binary relations based on the multiplicities of the transformed relation in [A.1], and later, we specialized this approach for the special multiplicities and their realization in [A.4].

In the context of the transformation of an OntoUML PIM, the associations in the UML PIM represent many different types of relations used in the OntoUML PIM. Beside the formal and material relations between the types, including the mediation of the actual material relation, also the Part-Whole relations of all the types used in OntoUML are transformed into standard associations. Moreover, because of the anti-rigidity, even the generalization relations between anti-rigid types like Roles and Phases are transformed into associations between the respective classes – we discussed this specific part of the transformation process in [A.9].

In the following subsections, the transformation of associations based on the multiplicities of the related classes is discussed.

6.1.2.1 Many-to-Many Associations

As a single record can hold only a single reference value in one column, referencing only a single record in the other table, the reference can only realize one-to-one or many-to-one associations. Although, according to SQL:2003 specification [87], variable arrays and
6.1. Transformation of UML PIM Constructs

nested tables are able to hold a collection of values, and thus a single record could refer multiple records in the other table, this specification is not fully implemented in many common database engines, and therefore we focus on using the standard means of SQL according to the SQL:1999 specification.

In the case of many-to-many associations, these are decomposed into two many-to-one associations first connected by an intermediating class representing the mapping between the instances of the two related classes [52]. An example of such decomposition is shown in Figure 6.2, where the decomposed many-to-many association between the Loan class and the Copy class presented in Figure B.3 is shown. The name of such intermediating class is generated by concatenating the names of the two related classes by "To". Then, the intermediating class can be simply transformed in the standard way discussed in subsection 6.1.1 and the two many-to-one associations can be transformed in the standard way as discussed in subsubsection 6.1.2.2. As the intermediating class defines only the mapping between the records in the related tables, no special ID column must be generated and the PRIMARY KEY constraint can be defined for the pair of the reference columns.

6.1.2.2 Many-to-One Associations

Because of the limitation of the reference mechanism to target only a single record, each of the many-to-one associations from the UML PIM is transformed into a reference located in the table representing the class containing the many instances – the source table and the source class, respectively – referencing a record in the table representing the class containing the one instance – the target table and the target class, respectively [A.1]. To hold the reference value, a special column is generated in the referencing table. For the name of the reference column, the name of the role of the target class in the association is used. In case of no role name, the name of the target table is concatenated with the postfix "_ID".

Moreover, to preserve the consistency of the data in the database, FOREIGN KEY (FK) constraints are defined for the references. The FK constraint is defined on a column of the referencing table and linked to the PRIMARY KEY (PK) column in the referenced table, restricting the possible values in the FK column only to the values existing in the PK column of the referenced table.

In general, the FOREIGN KEY constraint can be defined on a set of columns, referencing a composed key in the other table. However, in our approach we use only simple keys, and therefore we do not discuss such situation.
6. Transformation of UML PIM into RDB PSM

Beside the existence of the referenced record guaranteed by the FOREIGN KEY constraint, also the multiplicities of the referencing records should be constrained according to the multiplicities of the association in the UML PIM. As mentioned above, the reference mechanism can only target a single record in the target table. Therefore, the target maximal multiplicity of the reference is always restricted to 1. However, all the other multiplicity values are, by default, not restricted: the reference value can be NULL, causing the target minimal multiplicity to be 0; also, there can be any number of records in the source table referencing the same record in the target table, causing the source minimal multiplicity to be 0 and the maximal multiplicity to be *. Therefore, the following additional constraints are defined for the FK column to preserve the correct multiplicities when needed:

- When the target minimal multiplicity is 1, then the NOT NULL constraint is defined for the FK column of the source table, causing each record in the source table to have a reference value.

- When the source minimal multiplicity is 1, then a special multiplicity constraint needs to be defined as discussed in subsection 6.1.4. Similar additional constraint must be also defined when the multiplicities have special values, different from the standard ones 0, 1 and *.

An example of the applied transformation of many-to-one relations can be found in Figure 6.3, where the transformed model shown in Figure 6.2 is shown.

6.1.2.3 One-to-One Associations

The transformation of one-to-one associations from the UML PIM into the RDB PSM is very similar to the transformation of many-to-one associations. However, the direction of the reference between the tables is not determined by the maximal multiplicities, because both source and target maximal multiplicities are the same – 1. Instead, the direction is determined by the minimal multiplicities:

- When both minimal multiplicities are equal to 0, any of the directions can be used. In both cases, it is possible to restrict the multiplicities by the standard SQL constraints as discussed below.

- When only one of the minimal multiplicities is equal to 1, then the table representing the optional side of the association (the side with the 0 value) is the source table containing the reference, as the minimal multiplicity can be enforced by a NOT NULL constraint on the FK column.

- When both minimal multiplicities are equal to 1, any direction can be used, as in any case, a special multiplicity constraint to preserve the source minimal multiplicity of 1 is needed.
When the direction is determined, the association can be transformed into the reference in the same way as described in subsubsection 6.1.2.2. A special column for the reference value is generated in the source table and the FOREIGN KEY constraint is created for the reference column.

The same rules also apply for the multiplicity constraints. When the target minimal multiplicity is 1, the NOT NULL constraint is defined for the FK column. When the source minimal multiplicity is also 1, the special constraint discussed in subsection 6.1.4 is created. The same also applies in the case of special multiplicity values. However, contrary to many-to-one associations, the UNIQUE constraint is always defined for the FK column. Thanks to this constraint, the reference value of each record in the source table must be unique, and therefore, each of these records must refer to a different record in the target table, effectively limiting the source maximal multiplicity to 1.

In case of the target minimal multiplicity of 1, the uniqueness of the records in the source table referencing the same record in the target table can be also restricted by defining the reference on the column with the PRIMARY KEY constraint. Combining the FOREIGN KEY and PRIMARY KEY constraint on the same column forces each record to have a unique value, which is both the unique identification of the record in the source table and the reference to a record in the target table. This leads to so called weak entities – entities identified by identifier of a different entity [52]. However, a single class can have multiple one-to-one relations to other classes. Also from the cognitive point of view, entities of the domain represented by instances of certain class can have a different identifier independent of the relation to the other, although mandatory, entity. Therefore, in our approach, we
use separate columns for the identifier of the record and for the reference value.

An example of the applied transformation of a one-to-one association is shown in Figure 6.4, where the transformed UML PIM shown in Figure 5.2 is shown.

### 6.1.3 Generalization

In UML PIM, generalization is used to define shared properties of multiple types of objects. Such properties are defined in the superclass, which is specialized by the classes representing the more specific types. Moreover, the subclasses may form generalization sets with the meta-properties isDisjoint and isCovering. Then, instances of the subclasses are also instances of the superclass and they have properties defined by both the superclass and the particular subclasses. See subsubsection 2.2.1.3 for more details generalization sets in UML.

In context of the transformation of an OntoUML PIM into its realization in a relational database, generalization sets are used in the UML PIM as the result of the transformation of the hierarchies of Substance Sortal universals – the hierarchies of Kinds and the specializing Subkinds (see subsection 5.1.1). The generalization is also used in the UML PIM to represent the relations between Non-Sortal types – Categories, RoleMixins and Mixins – and their specializing Sortal types (see subsection 5.1.4).

However, as the concept of generalization is not present in relational databases, the generalization sets must be transformed into standard tables and references between them. This transformation is well-known and described in literature (e.g. [52], [27]). However, in context of the transformation of the OntoUML PIM, the correct realization of the meta-properties of the generalization set is important, but they are neglected in the standard approach. Therefore, we focus on the realization of these constraints in our approach. We discussed this approach in [A.8] and [A.10] in context with the rigid Sortal universals. However, this approach is generally applicable for all generalization sets in UML PIM, as in UML all generalization sets are rigid.

In general, there are three possible solutions for the realization of the generalization in relational database [52]:

---

3Sometimes, it is also called IS-A hierarchy
6.1. Transformation of UML PIM Constructs

◦ by a single table for the whole generalization set;
◦ by separate tables for each of the classes containing instances;
◦ by tables for each for each of the subclasses with reference to the table representing the superclass.

Each of these variants brings certain limitations and consequences. In the following subsections, the details of each of the possible realizations are discussed in context of the generalization set constraints and their realization, as well as other limitations and consequences.

6.1.3.1 Single Table

In this variant, the whole generalization set – the superclass and all its subclasses – is transformed into a single table. Such table contains the columns for all the attributes of the superclass and all its subclasses. Additionally, a single PK column is generated for identification of the records and an extra discriminator column is generated to identify the actual class of the represented instance.

In this realization, the multiplicity constraints of the superclass’s attributes can be realized easily by the NOT NULL constraint, as the constraints apply for instances of any of the subclasses as well. However, the constraints for the columns of the subclasses vary with the subclass of the represented instance. Instances of each subclass can provide values only to the attributes defined by their actual subclass and the superclass. The columns of the other subclasses always contain NULL values. Moreover, instances of the superclass have NULL values in all the columns of all the subclasses.

Instead, all the columns representing the attributes of all the subclasses must be defined NULLABLE and the NOT NULL constraints must be realized in context of the class of the actual stored instance identified by the discriminator value. Moreover, the meta-properties of the generalization set must be also implemented to respect the UML PIM definition and to prevent inserting records representing instances of invalid combinations of the subclasses. Therefore, the possible values of the discriminator column, as well as the derived NOT NULL constraints, are defined according to the meta-properties isCovering and isDisjoint of the generalization set.

In total, the following combinations of the values of meta-properties should be handled:

◦ \{disjoint, complete\}: The discriminator value must match one of the subclasses. Then, all the columns representing the mandatory attributes of the superclass and the subclass identified by the discriminator value must contain NOT NULL values and all the columns representing attributes of the other subclasses must contain NULL values.

◦ \{overlapping, complete\}: The discriminator value must match one the subclasses or their combination. Then, all the columns representing the mandatory attributes of
the superclass and the subclasses identified by the \textit{discriminator} value must contain NOT NULL values and all the columns representing attributes of the other subclasses must contain NULL values.

\textbullet \ \{\textit{disjoint, incomplete}\}: The \textit{discriminator} value must match the superclass or one of the subclasses. Then:

\begin{itemize}
  \item When the \textit{discriminator} value matches the superclass, all the columns representing the mandatory attributes of the superclass must contain NOT NULL values and all the columns representing attributes of the other subclasses must contain NULL values.
  \item When the \textit{discriminator} value matches one of the subclasses, all the columns representing the mandatory attributes of the superclass and the subclass identified by \textit{discriminator} value must contain NOT NULL values and all the columns representing attributes of the other subclasses must contain NULL values.
\end{itemize}

\textbullet \ \{\textit{overlapping, overlapping}\}: The \textit{discriminator} must match the superclass, a single subclass or a combination of the subclasses. Then:

\begin{itemize}
  \item When the \textit{discriminator} value matches the superclass, all the columns representing the mandatory attributes of the superclass must contain NOT NULL values and all the columns representing attributes of the other subclasses must contain NULL values.
  \item When the \textit{discriminator} value matches one of the subclasses or their combination, all the columns representing the mandatory attributes of the superclass and the subclasses identified by the \textit{discriminator} value must contain NOT NULL values and all the columns representing attributes of the other subclasses must contain NULL values.
\end{itemize}

Therefore, in our approach, we generate a table for the generalization set with the following properties:

\begin{itemize}
  \item The name of the superclass is used as the name of the table.
  \item The name of the particular subclass is used as prefix for the names of the columns representing the attributes of that subclass to prevent naming conflicts.
  \item The column \texttt{DISCRIMINATOR} is generated for identification of the represented class.
\end{itemize}

Furthermore, a special \textit{generalization set constraint} is generated as an OCL invariant to realize the NULLABILITY constraints of the subclasses’ columns depending on the meta-properties of the generalization set and the \textit{discriminator} value. This constraint has the following form:
The constraint is defined in context of the table representing the generalization set. The name of constraint is generated by concatenating prefix GS, the name of the superclass and the name of the generalization set. In the body of the constraint, for each possible value of the discriminator, a Boolean variable is defined using the def statement with the name generated by concatenating the value of the discriminator and postfix Instance. The value of this variable is determined as logical conjunction (AND) of the following comparisons:

- The value of the DISCRIMINATOR column is compared with the name of the class using the equality operator (=).
- Each of the columns representing the mandatory attributes of the subclasses identified by the discriminator value is compared with OclVoid using the equality operator (<>).
- Each of the columns representing the attributes of the other subclasses is compared with OclVoid using the inequality operator (=).

Finally, all the defined variables are combined together using logical disjunction (OR).

The example of application of this proposed transformation is shown in Figure 6.5 where the RDB PSM is shown with the table SUBJECT realizing the transformed generalization set of Person and LegalEntity classes specializing the superclass Subject shown in Figure 5.5. The generalization set constraint realizing the meta-properties of the generalization set and the conditional NULLABILITY of the columns is shown in Constraint 6.1. Examples of the constraints for other values of the meta-properties of the generalization set can be found in subsection C.2.1.

Although the realization of the NULLABILITY constraints and the generalization set meta-properties is complicated, the realization of UNIQUE constraints for the attributes
6. Transformation of UML PIM into RDB PSM

Constraint 6.1 Example of the OCL invariant for a \{disjoint,complete\} generalization set realized by a single table

context SUBJECT inv GS_Subject_Type:

def Person_Instance: Boolean = self.Discriminator = 'Person'
AND self.PERSON_LAST_NAME <> OclVoid AND self.PERSON_GENDER <> OclVoid
AND self.LEGAL_ENTITY_TITLE = OclVoid AND self.LEGAL_ENTITY_VAT = OclVoid

def LegalEntity_Instance: Boolean = self.Discriminator = 'LegalEntity'
AND self.PERSON_LAST_NAME = OclVoid AND self.PERSON_GENDER = OclVoid
AND self.LEGAL_ENTITY_TITLE <> OclVoid AND self.LEGAL_ENTITY_VAT <> OclVoid

Person_Instance OR LegalEntity_Instance

of any of the classes is simple (in contrast to the realization by the separate tables). As all data are stored in the same table, the column representing the particular attribute can be simply restricted by the UNIQUE constraint, such as shown in Figure 6.5 where the uniqueness of the column NAME is defined by the UQ_NAME constraint.

6.1.3.2 Separate Tables

In this variant, separate tables are created for each possible combination of classes of the instance. Each such table contains attributes defined by both the superclass and the particular subclass. Also, each of the table contains its own PK column used for identification of the records.

In most cases, it usually means creating a table for each of the subclasses containing the columns for the attributes of the particular subclass and its superclass. Then, instances of different subclasses are stored in different tables, but they can store values for all their attributes in that table. However, such situation reflects only a \{disjoint,complete\} generalization set. In the case of other values of the meta-properties, additional tables are created to hold the instances of only the superclass (in case of an incomplete generalization set) and instances of various combinations of the subclasses (in the case of overlapping generalization set).

Therefore, in our approach, we generate a table for each possible combination of the classes according to the meta-properties of the generalization set with the following properties:

- In the case of representing a single class, its name is used as the name of the table.
- In the case of representing multiple classes, the name of the table is generated by concatenating the names of the classes in alphabetical order.
- Each of the tables always contains the PK column, and the columns representing all the attributes of all the represented classes (including the superclass).

With this realization, the mandatory attributes can be realized by NOT NULL constraints defined for all the columns representing the mandatory attributes of both the
superclass and the particular subclasses, which the table represents, as only instances of
the same type are stored in a single table.

On the other hand, in the case of this variant, it is more complicated to ensure the
unique values for attributes, as the values are distributed in multiple separate tables. This
applies for the unique attributes of the superclass, and in the case of an overlapping
generalization set, also for the unique attributes of the subclasses, which are present in all
the tables representing the combinations with the particular subclass.

In such cases, the uniqueness cannot be defined only by using the standard UNIQUE
constraint for the particular column. Such constraint only enforces uniqueness in the
particular table. Instead, an additional distributed unique constraint is generated for each
of the tables containing the column representing the constrained attribute. The constraint
is defined as an OCL invariant in the following form:

- The constraint is defined in context of each of the tables with the particular column.
- The name of the constraint is generated by concatenating prefix UQ, the name of the
  represented class (or classes) and the name of the constrained attribute.
- In the body of the constraint, for each other table containing the constrained column
  (i.e. with the exception of the table in context of this particular OCL constraint), a
  Boolean variable is defined using the def statement with the following properties:
  - The name of the variable is generated by concatenating the name of the class
    (or classes) the table represents and the name of the constrained attribute.
  - The value is determined by negating the result of the operation exists on the
    collection of allInstances of the type of the table, where the values of the
    constrained column are equal.
- Finally, all the defined variables are combined together using logical conjunction
  (AND).

The example of the applied transformation using the approach of separate tables can be
found in Figure 6.6 where the RDB PSM for the transformed generalization set of classes
Person and LegalEntity specializing the Subject class shown in Figure 5.5 is shown.
Additionally, the constraint shown in Constraint 6.2 is defined to realize the distributed
unique constraint for the transformed attribute name defined in the superclass and realized
in both the tables as the column NAME.

6.1.3.3 Referencing Tables

In this variant, all the classes are transformed into their own respective tables. The table
representing the superclass contains only columns representing the attributes of the su-
perclass, a column with the PK value restricted by the PK constraint and the column
with the discriminator value to distinguish the class of the stored instance. The tables
Constraint 6.2 Example of the OCL invariants for distributed unique constraint

context PERSON inv UQ_PERSON_NAME:
def LEGAL_ENTITY_NAME: Boolean =
  NOT(LEGAL_ENTITY.allInstances() -> exists(1 | 1.NAME = self.NAME))
LEGAL_ENTITY_NAME

class 1e : LEGAL_ENTITY inv UQ_LEGAL_ENTITY_NAME:
def PERSON_NAME: Boolean =
  NOT(PERSON.allInstances() -> exists(p | p.NAME = self.NAME))
PERSON_NAME

representing the individual subclasses contain columns representing the attributes of the particular subclass and a column with the PK value restricted by the PK constraint to identify individual instances of the particular subclass. Then, instances of the subclasses are stored both in the superclass table and the appropriate subclass tables (according to the meta-properties of the generalization set – see below).

Additionally, as a record in a subclass table represents an instance of the subclass which always has also a record in the superclass table, there is a one-to-one reference between each of the subclass tables and the superclass table. As there can be at most one record in the subclass table referencing a single record in the superclass table (the whole pair represents a single instance), this reference is realized by the PK value in the subclass table and it is restricted by the FK constraint. Thanks to the PK constraint, this reference is always unique and mandatory for the particular subclass table, but optional for the superclass table.

Therefore, in our approach, we generate a table for each class of the generalization set with the following properties:

- For the superclass, a table is generated containing the PK column and constraint, the columns for the attributes of the superclass and the DISCRIMINATOR column. As the name of the table, the name of the superclass is used.

- For each of the subclasses, a table is generated with the name of the particular subclass. The table contains the PK column and constraint, the columns for the
attributes of the particular subclass and the FK constraint defined on the PK column referencing the superclass table.

The realization of the mandatory attributes of the superclass and the subclasses is simple in this variant. As each class is realized by a separate table, NOT NULL constraints can be easily defined for all columns representing the mandatory attributes of the respective classes in their respective tables. Also, the UNIQUE constraints for the individual attributes of the superclass and subclasses can be easily realized by UNIQUE constraints defined on the individual columns representing the attributes in the respective tables. However, the correct realization of the meta-properties isCovering and isDisjoint of the generalization set requires checking the existence of referencing records in the appropriate subclass tables. This check is based on the value of the discriminator in the superclass table, whose possible values are restricted according to the meta-properties of the actual generalization set.

In total, the following situations must be handled:

- \{disjoint, complete\} generalization set: The discriminator value of any record in the superclass table must match one of the subclasses.
- \{overlapping, complete\} generalization set: The discriminator value of any record in the superclass table must match one of the subclasses or their combination.
- \{disjoint, incomplete\} generalization set: The discriminator value of any record in the superclass table must match the superclass or one of the subclasses.
- \{overlapping, incomplete\} generalization set: For each record, the discriminator must match the superclass or a combination of the subclasses.
- In all cases, there must be a record in each of the tables representing the subclasses identified by the discriminator value of the record in the superclass table, referencing the particular record in the superclass table. Furthermore, there must be no records in the other subclass tables referencing the same record in the superclass table. In the case of the discriminator value matching the superclass, there must be no referencing record in any of the subclass tables.

As these situations require checking existence of records in multiple other tables, they cannot be defined as standard column-level constraints. Instead, a special generalization set constraint is defined as an OCL invariant in the following form:

- The constraint is defined in context of the superclass table.
- The name of the constraint is generated by concatenating prefix GS, the name of the superclass and the name of the generalization set.
6. Transformation of UML PIM into RDB PSM

**Constraint 6.3** Example of the OCL invariants for a \{complete, disjoint\} generalization set realized by related tables

```oclint
context SUBJECT inv GS_Subject_Type:
def Person_Instance: Boolean = self.DISCRIMINATOR = 'Person'
  AND NOT (LEGALENTITY.allInstances() ->exists (l | l.LEGALENTITY_ID = self.SUBJECT_ID))
def LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'LegalEntity'
  AND NOT (PERSON.allInstances() ->exists (p | p.PERSON_ID = self.SUBJECT_ID))
  AND LEGALENTITY.allInstances() ->exists (l | l.LEGALENTITY_ID = self.SUBJECT_ID)

Person_Instance OR LegalEntity_Instance
```

- In the body of the constraint, for each possible value of the discriminator, a Boolean variable is defined using the `def` statement with the name generated by concatenating the value of the discriminator and postfix \_Instance. The value of the variable is determined as logical conjunction (AND) of the following comparisons:
  - The value of the DISCRIMINATOR column is compared with the expected value representing the appropriate class using the equality operator (=).
  - For each of the classes identified by the discriminator value, instances of the type of the appropriate table are retrieved using the allInstances() operation and checked by the `exists` operation with the condition of the reference value matching the PK value of the contextual instance of the type of the superclass table.
  - For each of the other classes not identified by the discriminator value, instances of the type of the appropriate table are retrieved using the allInstances() operation and checked by the `exists` operation with the condition of the reference value matching the PK value of the contextual instance of the type of the superclass table. The whole expression is negated by the NOT operation.

- Finally, all the defined variables are combined together using logical disjunction (OR).

The example of application of this proposed transformation is shown in [Figure 6.7](#), where the RDB PSM for the generalization set of Person and LegalEntity classes specializing the Subject class shown in [Figure 5.5](#) is shown. The generalization set constraint realizing the meta-properties of the generalization set is shown in [Constraint 6.3](#). Examples of the constraint for the other values of the meta-properties can be found in [subsection C.2.2](#).

Additionally, as the generalization relation in UML is rigid, it is necessary to prevent changing the references between the records in the particular subclass tables and the record in the superclass table. In fact, the references are restricted by immutability on both sides, which should be realized in the same way as the immutability meta-properties of standard associations (see [subsubsection 6.1.7.1](#)).
6.1.4 Special Multiplicities

As discussed in subsubsection 2.2.1.2, for each association, multiplicities are define to restrict the number of related instances. Usually, the values 0 and 1 are used for the minimal multiplicity to define the optionality or mandatoriness of the related instance, respectively, and values 1 and * for the maximal multiplicity to restrict the relation to a single instance or a collection of instances. However, in general, the multiplicity values can be also different, restricting the possible number of related instances more specifically by (how we call them) special multiplicity values.

As discussed in subsection 6.1.2, each association is transformed into reference stored in a special column in the source table with the FOREIGN KEY constraint. Using the reference mechanism, the target maximal multiplicity is automatically restricted to 1. Furthermore, the target minimal multiplicity can be restricted to 1 by defining the reference column NOT NULL and the source maximal multiplicity can be restricted to 1 by defining the reference column UNIQUE. However, to restrict the other special source multiplicity values, a special multiplicity constraint must be defined as such values cannot be restricted by the standard means of FOREIGN KEY, NOT NULL and UNIQUE constraints. The same also applies to the source multiplicity value of 1, as there is no way to request a referencing record in a different table. Therefore, we also count such required source multiplicity among the special multiplicity values and restrict it using a special form of the special multiplicity constraint – a mandatory multiplicity constraint.

The situation of the mandatory minimal source multiplicity is common when transform-
ing OntoUML model. The transformation of Roles lead to strictly mandatory \textit{one-to-many} or \textit{one-to-one} associations between the class representing the Role and the truthmaker (see subsection 5.1.2 and Figure 5.2). The transformed Phases realized by a common abstract Phase class lead to a strictly mandatory \textit{one-to-one} association between the abstract Phase class and the class representing the \textit{identity bearer} of the Phase (see subsection 5.1.3 and Figure 5.4). Furthermore, the association representing the \textit{characterization} relation between the bearer and its Quality or Mode is a strictly mandatory association (see subsection 5.1.5). Also, many Part-Whole relations are transformed into associations with source table minimal multiplicity of 1 (see subsubsection 5.1.6.2): the \textit{containment} relation between a Quantity and the type of its container, the \textit{membership} relation between the Collective and the type of its members and the \textit{subCollectionOf} relation between Collectives.

Our approach to this constraint for the special multiplicity values of the source table was discussed in [A.1] and [A.2]. The idea of the constraint is to check the number of records in the source table related to the individual records in the target table. Therefore, the \textit{special multiplicity constraint} is defined as an OCL invariant in the following form:

- The constraint is defined in context of the target table.
- The name of the constraint is generated by concatenating prefix MUL, the name of the source table and the name of the reference column.
- In the body of the constraint, an Integer variable \textit{count} is defined with the value determined by the operation \textit{count} executed on the result of the operation \textit{allInstances}, which is executed on the type representing the source table, with the condition of the reference value matching the PK value of the contextual record in the target table.
- Finally, the actual multiplicity constraint is realized by comparing the variable \textit{count} with the multiplicity values using the operators \texttt{>=} and \texttt{<=} for the minimal and maximal values, respectively, connected by logical conjunction (AND). In the case of any of the multiplicities unrestricted (minimal multiplicity of 0 and maximal multiplicity of *), the appropriate comparison can be omitted.

An example of this applied transformation of special multiplicity values can be found in Figure 6.8 where the \textit{many-to-many} association between the classes Work and Series shown in Figure B.3 is transformed into its realization by the intermediating table WORK_TO_SERIES and its references. The special multiplicity of works for a single series is realized by the \textit{special multiplicity constraint} shown in Constraint 6.4.

In the case of only the source minimal multiplicity of 1, the constraint can be simplified, as only existence of a referencing record must be checked. We call such simplified multiplicity constraint \textit{mandatory multiplicity constraint} and define it in the following form:

- The constraint is defined in context of the target table.

\footnote{Mandatory relation at the both sides.}
Constraint 6.4 Example of the OCL invariant for a special multiplicity constraint in RDB PSM

context SERIES inv MUL_WORK_TO_SERIES_SERIES_ID:
def count: Integer =
WORK_TO_SERIES.allInstances().count(wts | wts.SERIES_ID = self.SERIES_ID)
number >= 2

Constraint 6.5 Example of the OCL invariant for a mandatory multiplicity constraint in RDB PSM

context WRITER inv MUL_AUTHORSHIP_WRITERID:
AUTHORSHIP.allInstances().exists(a | a.WRITER_ID = self.WRITER_ID)

○ The name of the constraint is generated by concatenating prefix MUL, the name of the source table and the name of the reference column.

○ In the body of the constraint, the actual constraint is realized by the operation exists executed on the result of the operation allInstances, which is executed on the type representing the source table, with the condition of the reference value matching the PK value of the contextual record in the target table.

An example of this mandatory multiplicity constraint can be found in Constraint 6.5 where the OCL invariant is shown for the reference between tables AUTHORSHIP and WRITER shown in Figure 6.9 (this model is the result of the transformation of the association between the classes Writer and Authorship shown in Figure B.3).
6. Transformation of UML PIM into RDB PSM

6.1.5 Exclusivity Constraints

As discussed in subsection 5.1.3, the Phases and their phase partitions can be transformed into a set of classes with associations to the class representing the identity bearer of the Phases. To preserve the disjointness of the phase partition, the OCL constraint such as shown in Constraint 5.1 must be defined to restrict the exclusivity of an instance of the Phase classes related to a single instance of the identity bearer.

When transforming such UML PIM into the RDB PSM, even such constraints must be preserved to ensure that only data valid according to the initial OntoUML PIM can be really handled in the database. Because the constraint is defined on the PIM level based on the classes and the associations between them, it must be transformed to restrict the data in the database tables representing the classes and the references representing the associations. This approach was introduced in [A.9], where the transformation of Roles and Phases from the OntoUML PIM into their realization in a relational database was discussed.

In the UML PIM, the associations between each class representing a Phase type from the phase partition and the class representing their identity bearer is a one-to-one association with the mandatory side of the identity bearer and optional side of the phase. Applying the transformation to such associations discussed in subsubsection 6.1.2.3, the associations are transformed into references located in the tables representing the individual Phase classes, referencing record in the table representing the class of the identity bearer.

When transforming the constraint, a new OCL invariant is created in context of the table representing the class in context of the original OCL invariant in the following form:

- The constraint is defined in context of the table representing the class in context of the original constraint.
- The name of the constraint is the same as the original constraint.
- In the body of the constraint, for each of the associations compared with OclVoid, a Boolean variable is defined using the def statement with the name generated from...
6.1. Transformation of UML PIM Constructs

Figure 6.10: RDB PSM with transformed exclusive Phases

the name of the associated class and postfix _Instance. The value of the variable is determined by the following expression:

\[ \text{<target>}.allInstances() \rightarrow \text{exists}(x|x.<\text{FK}>=\text{self.<PK>}) \]

where

- \( \text{<target>} \) is the name of the table representing the associated class,
- \( \text{<FK>} \) is the name of the reference column referencing the table representing the class in context of the original OCL invariant,
- and \( \text{<PK>} \) is the name of the PRIMARY KEY column of the table representing the class in context of the original OCL invariant.

- Finally, all the defined variables are combined together using the XOR operator.

This constraint defines that for each record in the table representing the original constrained class there is a referencing record only in one of the referencing tables representing the original Phase classes. An example of the transformed exclusivity constraint is shown in Constraint 6.6, where the original Constraint 5.1 is transformed for the RDB PSM shown in Figure 6.10, which represents the transformed UML PIM shown in Figure 5.3.

6.1.6 Enumeration Constraints

As discussed in section 5.2, certain patterns in the UML PIM can be optimized to reduce the number of classes, which are created as result of the transformation of the OntoUML PIM.
6. Transformation of UML PIM into RDB PSM

Constraint 6.6 OCL invariant for the transformed exclusivity constraint

context COPY inv EX_Copy_Condition:
def Undamaged_Instance: Boolean = UNDAMAGED.allInstances()->exists(u|u.COPY_ID = self.COPY_ID)
def Damaged_Instance: Boolean = DAMAGED.allInstances()->exists(d|d.COPY_ID = self.COPY_ID)
def Destroyed_Instance: Boolean = DESTROYED.allInstances()->exists(d|d.COPY_ID = self.COPY_ID)

Undamaged_Instance XOR Damaged_Instance XOR Destroyed_Instance

According to subsection 5.2.1 in certain situations, the subclasses forming a generalization set specializing another class (for instance classes representing Subkind types from the OntoUML PIM) may be reduced into values of a special discriminator attribute of the superclass. In other situations, according to subsection 5.2.2, a set of classes exclusively related to another class may be reduced into values of a special phase attribute of the class they are related to. However, in both cases, a special enumeration constraint is defined to restrict the possible values of such special attribute to preserve the meaning of the types from the OntoUML PIM the reduced classes in the UML PIM represented.

When transforming the UML PIM into the RDB PSM, these enumeration constraints restricting the possible values of the discriminator attribute need to be transformed into the RDB PSM as well. As during the standard transformation of the UML PIM the classes are transformed into database tables and the attributes of the class into columns of the table, it is just needed to transform the references to classes and their attributes into references to the respective tables and their columns in the constraints. Therefore, for each enumeration constraint in the UML PIM, a new OCL invariant in the following form:

○ The constraint is defined in context of the table representing the class in context of the original constraint.

○ The name of the constraint is the same as the original constraints.

○ In the body of the constraint, for each of the comparisons of the values of the discriminator attribute with the possible values in the original constraint, a comparison of the value in the column representing the discriminator attribute with the same value is defined. All the comparisons are then joined by the OR operator.

An example of the applied transformation of an enumeration constraint is shown in Constraint 6.7. This constraint represents the realization of the Constraint 5.2 defined for the RDB PSM shown in Figure 6.11, where the result of the transformation of the UML PIM shown in Figure 5.9 is shown.

6.1.7 Immutability

As discussed in section 3.7 UFO and OntoUML redefine the principles of Part-Whole relations in a conceptual model based on cognitive science and psychology. Beside the option-
6.1. Transformation of UML PIM Constructs

Figure 6.11: RDB PSM with transformed class Person with an immutable discriminator attribute

Constraint 6.7 OCL invariant for the transformed enumeration constraint

context PERSON inv EN_Person_Gender:
self.GENDER = 'Man' OR self.GENDER = 'Woman'

ality and mandatoriness, UFO and OntoUML also address essentiality and inseparability of the parts. As discussed in subsubsection 5.1.6.1, these properties of the Part-Whole relations are realized by the immutable meta-property of the appropriate end of the association, stating that the related instance or instances cannot change after initialization. An example of a transformed OntoUML PIM with an essential and inseparable part is shown in Figure C.1 for the original OntoUML PIM shown in Figure 3.11. Moreover, the same concept of immutability is also used in the UML PIM to restrict the immutable values of an attribute of a class. In context of our approach, this constraint is used in the optimization of rigid generalization sets such as discussed in subsection 5.2.1.

When transforming the UML PIM with such constraints of the associations or the attributes into the RDB PSM, even this meta-property should be transformed and realized. Without such constraint in the RDB PSM, it would be possible to change the related records representing the related instances or the values in the columns representing the immutable attributes, and therefore violate the domain constraints defined in the initial OntoUML conceptual model. However, as this meta-property does not restrict a static state of the model but rather the dynamic aspect of changes of the model in time, it is not possible to define such constraint by the standard means in the RDB PSM. Instead, a special constraint must be defined.

Again, as in the other similar situations, we use OCL for defining such additional constraints in the RDB PSM. In this case, we cannot use OCL invariants, as the constraint does not restrict the data in any particular time, but it needs to prevent changing the values. Therefore, we need to compare the values before and after executing the operations on the record in the constrained tables. Therefore, we use OCL precondition and postcondition constraints to define them. As the RDB PSM defines a model of a relational database, we suppose the standard SQL DML operations INSERT, UPDATE and DELETE [27].
6. Transformation of UML PIM into RDB PSM

6.1.7.1 Immutability of Associations

The transformation of the immutable meta-property of an association depends on the multiplicities of the relation and the direction of the reference realizing the association. As the immutable meta-property can be defined at each end of the association but the association is realized by an uni-directional reference, it is necessary to prevent changes of either the referenced records or the referencing records.

In the following paragraphs, the individual situations are discussed.

Immutability of Target Table. The first situation is when the immutable meta-property is defined for the association on the side of the class, which is realized by the target table of the reference. Example of such association is shown in Figure 5.7. Thanks to the multiplicities of the relation, when transforming this model into the RDB PSM using the approach discussed in subsection 6.1.2, the association is transformed into reference from the table `BOOK_EDITION` with the original immutable meta-property restricting the target table. The resulting RDB PSM is shown in Figure 6.12.

In such a situation, it is needed to prevent a record in the source table (table `BOOK_EDITION` in this case) to change the referenced record in the target table (table `BOOK`). This change can be only made by the UPDATE operation when changing the reference value. Such restriction can be realized by the OCL postcondition constraint comparing the reference value before and after the operation and checking that it was not changed. Therefore, an immutability constraint in the form of an OCL postcondition is defined in the following form:

- The constraint is defined in the context of the UPDATE operation of the source table.
- The name of the constraint is generated by concatenating prefix IM, the name of the source table, the name of the reference column and postfix UPD.
- In the body of the constraint, a comparison is defined comparing the value in the reference column with the value in the same column before the operation (using the @pre annotation) using the equality operator (=).
6.1. Transformation of UML PIM Constructs

**Constraint 6.8** OCL postcondition constraint for the *immutability constraint* for the target table `BOOK_EDITION`.

```
context BOOK_EDITION::UPDATE() post IM_BOOK_EDITION_BOOK_ID_UPD:
self.BOOK_ID = self.BOOK_ID@pre
```

Figure 6.13: RDB PSM with transformed immutable association restricting the source table.

**Constraint 6.9** OCL postcondition constraint for the *immutability constraint* for the target table `BOOK_EDITION`.

```
context CLIENT::UPDATE() post IM_CLIENT_POSTAL_ADDRESS_ID_UPD:
self.POSTAL_ADDRESS_ID = self.POSTAL_ADDRESS_ID@pre
```

An example of the *immutability constraint* for the target table is shown in Constraint 6.8, where the constraint restricts the reference between the tables `BOOK_EDITION` and `PERSON`.

**Immutability of Source Table.** When the *immutable* meta-property is defined on the side of the class, which is realized by the source table of the reference, it restricts changing the records in the source table referencing particular record in the target table. Example of such association is shown in Figure 5.6, which is transformed into the RDB PSM shown in Figure 6.13 with the reference from the table `CLIENT` to the table `POSTAL_ADDRESS`. In this case, also the source size of the reference is restricted by the *immutability*.

Similar to the case of immutable target table, this constraint can be violated by the `UPDATE` operation on the records in the source table: when changing the reference value, the instance represented by the old referenced record in the target table looses the instance it was related to, thus violating the *immutability* constraint defined by the *immutable* meta-property. In Constraint 6.9, the OCL postcondition constraint is shown for the `UPDATE` operation executed on the table `CLIENT`, checking the reference value in the column `POSTAL_ADDRESS_ID`.

Moreover, unlike the *immutability* of the target table, the constraint can be also violated by the `DELETE` operation. When a record is deleted from the source table, the instance represented by the old referenced record in the target table looses the related instance of
Constraint 6.10 OCL precondition constraint for the DELETE operation realizing the immutable meta-property of an association defined at the side of the source class

**context** CLIENT::DELETE() **post** IM_CLIENT_POSTAL_ADDRESS_ID_DEL:

self.POSTAL_ADDRESS_ID = OclVoid OR NOT (POSTAL_ADDRESS.allInstances() ->exists(pa | pa.POSTAL_ADDRESS_ID = self.POSTAL_ADDRESS_ID@pre))

the class realized by the source table, thus violating the constraint. Although it might seem to be restricted by the special multiplicity constraint as discussed in subsection 6.1.4 such constraints only restrict the number of related instances in any consistent time, not the identity of the related instances. Therefore, it is still possible to delete a related record and replace it by inserting a new record with the same reference value. Then, the multiplicity constraint would not be violated, but the immutability constraint is violated. Moreover, when the source maximal multiplicity is higher than 1, then the whole set of instances of the source class related to the instance of the target class is immutable and should not be changed.

Therefore, an additional constraint must be defined to prevent deleting records from the source table. However, there can be a situation, when the target minimal multiplicity is 0, thus allowing the instance of the source class not to be related to any instance of the target class. Moreover, it is needed to allow deleting the records with a mandatory reference value when the referenced record does not exist (for instance when completely deleting the related data). Therefore, the constraint must prevent the deleting of only such records which either do not have the reference value or do not reference an existing record in the target table. Therefore, an OCL postcondition is defined in the following form:

- The constraint is defined in context of the DELETE operation on the source table.
- The name of the constraint is generated by concatenating prefix IM, the name of the source table, the name of the reference column and postfix DEL.
- The body of the constraint consists of two parts. In the first, the value of the reference column is compared with OclVoid using the equality operator (=). In the second, operation exists with the condition of equality of the PRIMARY KEY value in the target table and the old reference value in the contextual table, executed on the result of operation allInstances performed on the target table. The result of this exists operation is negated. Finally, both parts are combined using the logical disjunction (OR).

An example of such OCL postcondition is shown in Constraint 6.10 for the DELETE operation executed on the table CLIENT.

Additionally, the immutable meta-property of the source table can be also violated by the INSERT operation. By inserting new records into the source table with references to a record in the target table, it adds new instances to the immutable set of its related instances. However, the immutable meta-property restricts changing this set after the
Constraint 6.11 OCL postcondition constraint for the \texttt{UPDATE} operation realizing the \textit{immutable} meta-property of an attribute

\begin{verbatim}
context PERSON::UPDATE() post IM_PERSON.GENDER_UPD:
    self.GENDER = self.GENDER@pre
\end{verbatim}

initialization of the instances. Therefore, during the initialization, it must be possible to add the new instances to the set – to actually create this \textit{immutable} set. But, in the relational databases and OCL, it is not possible to distinguish the initialization phase from the phase of altering already existing records. Therefore, this part of the realization of the \textit{immutable} meta-property cannot be realized in the relational database\textsuperscript{5}.

6.1.7.2 Immutability of Attributes

The transformation of the \textit{immutable} meta-property of an attribute is much simpler than in the case of associations. The \textit{immutability} restricts only changes of the value of the attribute. Therefore, when transforming the constraint into the RDB PSM, the changes of values in the column representing the attribute must be prevented.

As inserting a new record does not mean changing the value and deleting the record also does not mean changing the value for the represented instances, only the \texttt{UPDATE} operation needs to be checked. Therefore, the \textit{immutability constraint} is realized in the same form as in the case of \textit{immutability} of the target side of a reference, checking the value of the column representing the constrained attribute. An example of such a constraint is shown in Constraint 6.11, where the \textit{immutability} of the attribute \texttt{gender} of the class \texttt{Person} from the UML PIM shown in Figure 5.9 is defined for the resulting RDB PSM shown in Figure 6.11.

6.2 Discussion

In this section, we discuss various problems and limitations of our approach to the transformation of the UML PIM created by the transformation of an OntoUML into its realization in the RDB PSM. We also address the options introduced in section 6.1 and present conclusions and reasoning for the best selection of available method according to the particular situation. Finally, we also suggest certain optimizations of the transformed RDB PSM based on manual decisions of the analyst or designer and the knowledge of the domain.

6.2.1 Generalization

As discussed in subsection 6.1.3, when transforming the generalization sets from the UML PIM into its realization in the RDB PSM, there are several options – using a single table;

\textsuperscript{5}It is possible to activate and deactivate various constraints and checks, but, it is not a systematic solution.
using individual tables for each valid combination of classes of the instances; or using a table for the superclass and related tables for each of the subclasses. As discussed in the individual subsections of subsection 6.1.3, each of these variants has certain advantages and disadvantages.

**Single table.** The realization by a single table (subsubsection 6.1.3.1) results in less tables in the system and removes the need of joining multiple tables to retrieve complete data of instances of a subclass. However, the mandatoriness of the attributes of the subclasses are more complicated to realize, as the constraints must be qualified by the type of the represented instance. On the other hand, these constraints are combined with the realization of the meta-properties of the realized generalization set, which prevent storing instances of invalid combination of the subclasses by checking the discriminator value and the NULL values for all the columns representing the attributes of the other classes.

As the result, the more attributes of the subclasses there is, the more columns must be checked in the OCL constraint realizing the multiplicity constraints and the meta-properties of the generalization set. Also, the table becomes very sparse – it contains a lot of NULL values, but still taking the space in the memory to physically save the table. Moreover, when realizing the whole generalization set in a single table, the concept of separated entities is lost in the RDB PSM. This can be a problem when evolving the model and adding new columns for new attributes of the classes, as it complicates the multiplicity and generalization set constraints.

On the other hand, as discussed in subsection 7.3.1, the realization of the generalization set constraints is much simpler in the case of a single table, as data only of a single record in a single table must be checked.

Still, because of these complicated conditional constraints for the mandatoriness of the columns and the generalization set meta-properties, we consider this realization useful only in the case of generalization sets, where there is not many attributes in the subclasses and where the probability of evolving the model is low.

**Separate tables.** In the realization by separate tables (subsubsection 6.1.3.2), the mandatoriness of the attributes can be realized simply by defining the columns appropriate columns NOT NULL, as all records in each of the tables represent instances of the same class. The data of instances of different classes from the generalization set are saved into different tables, which is useful from the point of view of the query efficiency.

On the other hand, additional tables must be also defined for any possible combination of classes from the generalization set, which the instances can instantiate. This leads to an additional table for the instances of the superclass in the case of an incomplete generalization set and even more additional tables for various combinations of the subclasses in the case of an overlapping generalization set. The more subclasses there is, the more tables must be generated and the more complicated the database schema becomes.

Moreover, this solution also leads to duplicating the column representing a single attribute (attributes of the superclass, and in the case of an overlapping generalization
set also the attributes of the subclasses) in multiple tables. Because of that, data of an attribute are distributed in multiple tables, which complicates their querying as well as checking. This becomes especially problematic in the case a uniqueness of an attribute distributed in multiple tables, as the uniqueness must be preserved across all the tables containing the column representing the attribute.

Also, it complicates the realization of associations of the superclass. These associations cannot be realized by a reference targeting the record representing the instance of the superclass, as such record can be stored in any of the tables. Instead, the association must be realized by a reference in the other direction – i.e. from the individual tables realizing the generalization set to the table representing the related class. However, in certain situations, this leads even to more complicated realization, when the maximal multiplicity of the other related class is greater than 1, as in such case, the relation cannot be realized by a simple reference, but requires a special intermediate referencing table [A.1]. The same problems also apply for the associations of the subclasses in the case of an overlapping generalization set, as all combinations of the other subclasses with the particular subclass share the same association.

Because of these complications of this realization, we do not consider it really useful in general for the complete realization of the generalization sets transformed from the initial OntoUML PIM, and we propose this realization only as an optimization of the transformed RDB PSM by the analyst or designer after careful consideration of the benefits.

Referencing tables. In the realization by the referencing tables [subsubsection 6.1.3.3], the realization of the mandatoriness of the attributes is simple – by standard NOT NULL constraints on the columns representing the attributes in the tables representing the particular classes from the generalization set. Therefore, the number of attributes of the subclasses does not matter as it does in the case of single table realization. However, on the other hand, to retrieve complete data of the instances, data from multiple tables must be joined. In the case of a disjoint generalization set, it is only joining the table representing the particular subclass and the table representing the superclass, but in the case of overlapping generalization sets, it is also joining the other subclass tables according to the actual type of the stored instance indicated by the discriminator value in the superclass table.

Also the actual realization of the meta-properties of the generalization set requires searching in multiple tables to check the existence of the appropriate records in the subclass tables according to the discriminator value. As discussed later in [subsubsection 7.3.1.2] this is less efficient than searching in a single table and certain realizations cannot be really fully used. On the other hand, it is not necessary to check the values in all the columns of the tables, only the reference values are needed, which may be indexed to increase the search efficiency.

Still, we consider this realization the most useful. Although it is needed to join multiple tables to get the complete instance data and the meta-properties constraints require searching in multiple tables, the realization of the multiplicity and uniqueness constraints
of the attributes, as well as the associations with other classes can be realized easily. Therefore, when transforming the generalization sets from the UML PIM into the RDB PSM, we transform the generalization sets using the approach of referencing tables and leave the other possibilities only as suggested optimizations in certain situations after careful consideration of the analyst or designer of the database.

### 6.2.2 Mutually Dependent References for One-To-One Associations

As discussed in subsubsection 6.1.2.3, the realization of the one-to-one association in the RDB PSM depends on the minimal multiplicities of the association. The reason for this is the fact, that it is not possible to restrict the source minimal multiplicity, while the target minimal multiplicity can be restricted by the NOT NULL constraint defined on the reference column. However, in the case of one-to-one association with the minimal multiplicities of 1 at the both ends of the association, the direction is not important, as in any case, the reference will be restricted by the NOT NULL constraint and the source minimal multiplicity will be needed to be restricted by a special multiplicity constraint as discussed in subsection 6.1.4.

In [A.1], we discussed the possibilities of the transformation of binary relationships from a PIM into a PSM for relational databases. In the paper, we also suggested using a pair of reverse references to realize a one-to-one association with minimal multiplicities of 1 at both ends of the association. According to this approach, the one-to-one association in the PIM is transformed into references in both of the tables representing the related classes in the RDB PSM, each referencing a record in the other of the two tables. Both these references are restricted by a NOT NULL constraint and a UNIQUE constraint, making each record in one table referencing exactly one record in the other table, while each record in the other table references exactly one record in the first table. In Figure 6.14, an example of the reverse mutual references is shown for the transformed one-to-one association between the classes Client and PostalAddress shown in Figure B.3.

However, as we discussed in [A.1], the reverse references must be restricted by an additional mutuality constraint to make sure, that the record in one table referenced from the other table references the same record back. Otherwise, without such constraint, the two reverse references would rather realize two disjunct one-to-one associations as the record referenced by a record in one of the tables could reference a different record in that table instead of the referencing one. This constraint can be defined as OCL invariant in the RDB PSM. In Constraint 6.12, an example of the OCL constraint is shown for the mutual references shown in Figure 6.14. In the OCL constraint, from all records in the other related table, the records with the referenced PRIMARY KEY value referenced by the contextual record are selected, and for all of these records the reference value must refer the PRIMARY KEY value of the contextual record of the contextual table.

Although a valid possible solution for the realization of one-to-one associations, we do not use it in our approach. There are several reasons for not using this solution in comparison to the approach discussed in subsubsection 6.1.2.3.
6.2. Discussion

Constraint 6.12 OCL constraint for the mutually dependent references

context CLIENT inv MUT_CLIENT_POSTAL_ADDRESS:
POSTAL_ADDRESS.allInstances ()
->select (pa | pa.POSTAL_ADDRESS_ID = self.POSTAL_ADDRESS_ID)
->forall (pa | pa.CLIENT_ID = self.CLIENT_ID)

- It leads to increased data load, as for each such association, two reference columns must be created (one in each of the related tables) instead of a single one, and thus more reference data must be saved for the related records.

- There are more constraints which must be checked whenever the data in the related tables are manipulated. For a single reference, only a single FOREIGN KEY constraint, a single NOT NULL constraint, a single UNIQUE constraint and the special multiplicity constraint for the source minimal multiplicity is used. For the mutual references, two FOREIGN KEY constraints, two NOT NULL and UNIQUE constraints and the special mutuality constraint must be used to ensure the data consistency. It means more computing power needed for all the DML operations.

Therefore, we do not use mutual references for the realization of one-to-one associations, even when the minimal multiplicities at the both sides of the association are set to mandatory (value 1).

6.2.3 Single Table Realization of One-to-One Associations

As discussed in subsubsection 6.1.2.3 one-to-one associations in the UML PIM are transformed into a single reference between the two tables in the RDB PSM representing the two related classes. This reference is always restricted by a UNIQUE constraint to limit the maximal multiplicity of records in the source table referencing the same record in the target table to 1. An example of such realization is shown in Figure 6.4.
6. Transformation of UML PIM into RDB PSM

As discussed in [A.1], the one-to-one association between two classes can also be realized by a single table representing the whole association. In this realization, both related classes are realized by a single table containing the columns representing the attributes of both classes. In such a case, there is no need for a reference realizing the association, saving the space needed for the reference column as well as the FOREIGN KEY constraint. Also, only a single PRIMARY KEY column is needed, identifying a single record representing both related instances at the same time. On the other hand, also all the other relations of any of the two related classes must be transformed into a relation to the combined table and realized appropriately by a reference from or to the combined table.

In the case of a fully mandatory association (mandatory at both ends such as the association between the classes Client and PostalAddress shown in Figure B.3, the mandatoriness of the attributes defined in the UML PIM can be realized as usual by the NOT NULL constraints, as each record in the table represents related instances of the both related classes. An example of the table realizing the mandatory one-to-one association shown in Figure B.3 is shown in Figure 6.15.

In the case of a one-side optional one-to-one association (association with the minimal multiplicity of the source class 0 such as shown in the altered model shown in Figure 6.16), only the mandatoriness of attributes of the mandatory class can be realized by the standard NOT NULL constraints, as the record always represents an instance of that class. But, the instance of the other optional class may not be contained in the record. Therefore, the mandatoriness of the attributes of this optional class must be qualified by the fact that the record actually represents related instances of both classes, similarly as in the case of the generalization set realization by a single table as discussed in subsubsection 6.1.3.1. For this purpose, a special column must be created to hold this flag, containing the NULL value when the record represents only an instance of the mandatory class and any non-NULL value when it represents both related instances. An example of such realization is shown in Figure 6.17 where the column IS_POSTAL_ADDRESS is defined to signal that the record represents both related instances. The mandatoriness constraints of the optional class’s columns is shown in Constraint 6.13.

Finally, when the one-to-one association is optional at the both sides, then the record
6.2. Discussion

Client
- phone: String
- e-mail: String

PostalAddress
- name: String
- street: String
- streetNumber: String
- city: String
- zipCode: String
- country: String

Figure 6.16: UML PIM with a one-side optional one-to-one association

Client

PostalAddress

Figure 6.17: RDB PSM with a single table realizing a one-side optional one-to-one association

Constraint 6.13 OCL constraint for the mandatoriness of attributes of the optional class in the single table realization of a one-to-one association

```
context CLIENT_POSTAL_ADDRESS inv OPT_CLIENT_POSTAL_ADDRESS:
  (self.IS_POSTAL_ADDRESS = OclVoid
   AND self.NAME = OclVoid AND self.STREET = OclVoid
   AND self.STREET_NUMBER = OclVoid AND self.CITY = OclVoid
   AND self.ZIP_CODE = OclVoid AND self.COUNTRY = OclVoid)
OR
  (self.IS_POSTAL_ADDRESS <> OclVoid
   AND self.NAME <> OclVoid AND self.STREET <> OclVoid
   AND self.STREET_NUMBER <> OclVoid AND self.CITY <> OclVoid
   AND self.ZIP_CODE <> OclVoid AND self.COUNTRY <> OclVoid)
```
can represent instance of either of the classes. Therefore, a flag column must be defined for both of the classes to signal, that the particular record represents an instance of that particular class (or both). Then, the mandatoriness of the attributes of both the classes must be defined by a special OCL constraint and qualified by the value of the flag.

In context of the transformation of an OntoUML PIM, this variant of the realization of a one-to-one association can be used in several cases discussed in the following sections.

### 6.2.3.1 Optimization of Abstract Phase Classes

As discussed in subsubsection 5.1.3.2, the phase partitions of Phases from the OntoUML PIM can be transformed into the UML PIM as a generalization set of Phase classes specializing a special abstract phase class. This special abstract phase class is related by fully mandatory one-to-one association to the class representing the identity bearer of the Phases. Then, when transforming this model into the RDB PSM, the one-to-one association is transformed into a mandatory unique reference restricted by a special multiplicity constraint and the generalization set is transformed either into columns of the abstract phase table (subsubsection 6.1.3.1), or into exclusive referencing tables (subsubsection 6.1.3.3).

According to the optimization approach discussed above, when transforming this generalization set into the RDB PSM, this one-to-one association can be realized by realizing the related classes by a single table. As the associations is fully mandatory and the abstract phase class does not contain any attributes, no special constraints are needed. Then, depending on the realization of the generalization set, this combined table is a) either also combined with all the columns realizing the attributes of the individual Phase classes and restricted by the generalization set constraint as discussed in subsubsection 6.1.3.1 or b) referenced by the tables realizing the individual Phase classes and restricted by the exclusivity constraint as discussed in subsubsection 6.1.3.3.

In Figure 6.18, the RDB PSM of the transformed UML PIM of the phase partition realized by abstract phase class shown in Figure 5.4 is shown. Applying the proposed optimization, the tables COPY and CONDITION are combined into a single table COPY and the references from the individual tables representing the Phase classes reference the records in the table COPY. The optimized RDB PSM is shown in Figure 6.19 and the generalization set OCL constraint is shown in Constraint 6.14.

If you compare this model with the RDB PSM realizing the phase partition realized by the exclusive Phase classes in the UML PIM such as shown in Figure 6.10, you can find out, that the models are almost the same. The only differences are the references, which are realized by a separate column in the case of exclusive phases realization because of a standard association, in contrast to the case of abstract phases where the references can be combined with the PRIMARY KEY columns thanks to the generalization. Also, the OCL constraint realizing the meta-properties of the generalization set, although qualified by the discriminator value, equals in the meaning to the exclusivity constraint, as the generalization set is always \(\{\text{disjoint, complete}\}\), and thus the references from the tables realizing the subclasses are always exclusive.
6.2. Discussion

Figure 6.18: RDB PSM with the *phase partition* realized by *abstract phase* class and referencing tables

Figure 6.19: RDB PSM with the optimized *phase partition* realized by *abstract phase* class and referencing tables
6. Transformation of UML PIM into RDB PSM

### Constraint 6.14
OCL constraint for the optimized phase partition realized by abstract phase class and referencing tables

```oclude
class COPY inv GS_Copy_Condition : 
    def Undamaged_Instance : Boolean = self.DISCRIMINATOR = 'Undamaged'
    AND UNDAMAGED.allInstances() ->exists (u | u.UNDAMAGED_ID = self.COPY_ID)
    AND NOT (DAMAGED.allInstances() ->exists (d | d.DAMAGED_ID = self.COPY_ID))
    AND NOT (DESTROYED.allInstances() ->exists (d | d.DESTROYED_ID = self.COPY_ID))

def Damaged_Instance : Boolean = self.DISCRIMINATOR = 'Damaged'
    AND NOT (UNDAMAGED.allInstances() ->exists (u | u.UNDAMAGED_ID = self.COPY_ID))
    AND DAMAGED.allInstances() ->exists (d | d.DAMAGED_ID = self.COPY_ID)
    AND NOT (DESTROYED.allInstances() ->exists (d | d.DESTROYED_ID = self.COPY_ID))

def Destroyed_Instance : Boolean = self.DISCRIMINATOR = 'Destroyed'
    AND NOT (UNDAMAGED.allInstances() ->exists (u | u.UNDAMAGED_ID = self.COPY_ID))
    AND NOT (DAMAGED.allInstances() ->exists (d | d.DAMAGED_ID = self.COPY_ID))
    AND DESTROYED.allInstances() ->exists (d | d.DESTROYED_ID = self.COPY_ID)
```

Therefore, in this case, it doesn’t matter if the phase partition is realized in the UML PIM by the abstract phase class or the exclusive associations. In both cases, the transformation results into the same final RDB PSM. As this proposed optimization of the generalization set representing the phase partition removes the realization of the abstract phase table containing no useful data from the domain and the required UNIQUE and FOREIGN KEY constraint without creating any additional constraints or issues, we suggest using this optimization approach for all such situations. However, this optimization cannot be implemented automatically, as the UML PIM containing this pattern does not necessarily need to be the realization of a phase partition at all. Therefore, the knowledge of the domain and the initial OntoUML PIM is needed, thus applying this optimization manually after appropriate consideration by the analyst or application designer.

### 6.2.3.2 Optimization of Qualities

Another common example of a fully mandatory one-to-one association in the OntoUML PIM is the relation between a Quality and its bearer (subsection 3.6.1). When this relation is transformed into the UML PIM, it is realized by a standard one-to-one association between the classes representing the Quality universal and its bearer universal.

When transforming this one-to-one association from the UML PIM into the RDB PSM, it leads to a single reference between the tables representing the Quality class and the class of its bearer restricted by the UNIQUE constraint and the special multiplicity constraint for the minimal multiplicity of the source table as discussed in subsection 6.1.2.3. As discussed in subsection 6.2.3, this mandatory unique reference between the tables can be optimized into a single table containing columns of both the Quality table and the bearer table without any complicated constraints.

However, we do not recommend using this optimization in the case of Qualities. The reason for this is that by this optimization, the concept of the Quality defined separately from its bearer is lost. As any property can be defined as a simple attribute of the universal
in the OntoUML PIM, defining them as a separate Quality universal puts special emphasis on the meaning of the properties and their structure. Therefore, it make sense preserving this distinction also in the other derived models.

Using the optimization causes loosing the separate concept of the Quality, which may cause problems when evolving the model (for instance when a new property is identified as part of the Quality). Also, the efficiency of the data querying can be an issue in certain situations when the Quality defines properties of big data (e.g., images, files, etc.). For instance, some frameworks require reading whole records from the database, which can be very expensive for binary data attached to some simple data of the same record. When separated to separate tables, the binary data can be loaded from the related record only when needed, and thus increase the efficiency of the queries for only the basic data.

In conclusion, although it is possible to realize the one-to-one associations by a single table, it is beneficial in only in certain situations. In the case of mandatory one-to-one associations, it removes the duplicity of the reference value, however, the separated concepts of separate classes are lost, which can cause problems when evolving the model or efficiency of the data layer in the application. In the case of optional one-to-one associations, it also leads to more complicated multiplicity constraints for the columns representing the attributes of the optional classes and special columns needed to signal the fact, that the record represents an instance of that class.

Therefore, in our approach, we do not use this approach to the realization of the one-to-one associations automatically. Instead, we use the standard realization by a single reference between the tables representing the related classes as discussed in subsubsection 6.1.2.3. However, we suggest this optimization of the association for the consideration of the analyst or designer of the model in situations, where there is not many mandatory attributes in the optional classes, in the case of mandatory one-to-one associations and in the situation when the lost of separated concepts is not important.
In this chapter, we discuss the details of the transformation of RDB PSM into SQL ISM. This transformation is the third step of our approach to the complete transformation of an OntoUML conceptual model into its realization in a relational database.

As discussed in section 2.1, ISM is the lowest level of abstraction of the system, represented by the actual source codes of the application. In context with the relational databases, this model consists of the SQL scripts containing the statements to create the individual database tables with their columns, constraints and indices, as well as other SQL scripts for various views, triggers, functions and procedures. Therefore, this transformation consists mainly in the generation of the SQL statements for the constructs defined in the RDB PSM model from the previous step.

The input of the transformation is the RDB PSM model containing the definition of the database tables, their columns of various datatypes, NOT NULL constraints, UNIQUE constraints, PRIMARY KEY and FOREIGN KEY constraints and the OCL constraints defining additional constraints for the relational database, which cannot be expressed directly in the model using the UML Data Modelling profile discussed in subsection 2.2.2. The output of the transformation are the SQL scripts for creating the individual database tables, views, constraints and other constructs.

The chapter is structured as follows:

- in section 7.1 the basic principles of the transformation are discussed;
- in section 7.2 the basic options for the realization of the additional constraints are introduced;
- in section 7.3 the details of the possible realizations of the individual types of the additional OCL constraints defined in RDB PSM are discussed;
- in section 7.4 the discussion to our approach to this step of the transformation is provided.
Our approach to the transformation of RDB PSM and the OCL constraints is based on SQL:1999 standard [39]. To demonstrate our approach, we provide examples of the SQL statements based on the Oracle Database 12c DBMS [85], which implements this standard almost completely; thus it allows to use most of the standard constructs. Therefore, all the examples are using the Oracle dialect of the SQL for the expressions and statements [88]. In subsection 7.4.1 the capabilities of other popular RDBMSs are discussed in context with the possible usage of the proposed realizations.

The complete SQL ISM model created as the result of the proposed transformation of the RDB PSM model from the previous step presented in Figure B.4 is shown in chapter 4 of the attached Running Example (see Appendix B).

7.1 Basic Transformation Principles

In this section, the basic principles of transforming the RDB PSM into the database scripts composing the SQL ISM. As generating SQL scripts for creating database objects from a database model is well-known and implemented by many database modelling tools (e.g. Enterprise Architect [32]), this section provides only overview of the standard principles of such generation [52].

7.1.1 Tables and Column-Based Constraints

When transforming the RDB PSM into the SQL ISM, for each database table defined in the RDB PSM, a CREATE TABLE statement is generated. This statement defines the name of the table to be created and the list of its columns [88]. For each of the columns, the name and datatype is defined according to the values defined in the RDB PSM. Moreover, the definition of a column in the statement can also contain the NOT NULL clause to apply NOT NULL constraint, if defined in the RDB PSM.

Furthermore, ALTER TABLE ADD CONSTRAINT statements are generated for the PRIMARY KEY constraint (if defined in the RDB PSM) and each of the unique columns. Each of these statements define the name of the altered table, the name of the constraint, its type (PRIMARY KEY or UNIQUE) and the list of columns composing the constraint as defined in the RDB PSM [88]. Both of these constraints can also be defined inside of the CREATE TABLE statement, as they are restricting only a single table, which is just being created by the statement, and its columns, and therefore there is no problem of non-existence of the referred table and columns as it is in the case of FOREIGN KEY constraints (see subsection 7.1.2) [27, 52].

In SQL 7.1 the CREATE statements for the SUBJECT table and its column-based constraints are shown as generated by Enterprise Architect from the RDB PSM shown in Figure 6.7.
7.1. Basic Transformation Principles

### SQL 7.1 SQL ISM with the CREATE statements for a table

```sql
CREATE TABLE "SUBJECT" (
    "SUBJECT_ID" NUMBER(8) NOT NULL,
    "DISCRIMINATOR" VARCHAR2(50) NOT NULL,
    "CITY" VARCHAR2(50),
    "COUNTRY" VARCHAR2(50) NOT NULL,
    "NAME" VARCHAR2(50) NOT NULL);

ALTER TABLE "SUBJECT" ADD CONSTRAINT "PK_SUBJECT" PRIMARY KEY ("SUBJECT_ID");
ALTER TABLE "SUBJECT" ADD CONSTRAINT "UQ_NAME" UNIQUE ("NAME");
```

### SQL 7.2 SQL ISM with the ADD CONSTRAINT statement for a FOREIGN KEY constraint

```sql
ALTER TABLE "PERSON" ADD CONSTRAINT "FK_PERSON_ID"
    FOREIGN KEY ("PERSON_ID") REFERENCES "SUBJECT" ("SUBJECT_ID");
```

#### 7.1.2 References

As discussed in subsection 6.1.2, relation between records is realized by references – the referencing (source) table contains a special column, in which the individual records hold the identifier of the record in the other (target) table it is related to. To preserve the consistency of such references, the FOREIGN KEY constraint is defined. Such constraint ensures that for each record in the source table, the referenced record in the target table really exists.

When generating the SQL ISM from the RDB PSM, for each FOREIGN KEY constraint, the ALTER TABLE ADD CONSTRAINT statement is generated to create it in the database. The statement defines the name of the constraint, the table containing the constraint and the columns composing the FOREIGN KEY. Although the creation of this constraint can be combined with the CREATE TABLE statement of the source table, it requires the referenced table to exist in the time of creation of the table. As it is complicated to determine the order of creating the tables in a complex RDB PSM containing a lot of references, it is easier to create the FOREIGN KEY constraints separately from the tables, after all of the tables are created [27, 52]. An example of such statement is shown in SQL 7.2, altering the table PERSON and defining the FOREIGN KEY constraint for the reference between the tables PERSON and SUBJECT shown in Figure 6.7.

With such FOREIGN KEY constraint definition, the constraint is checked automatically and immediately for each record affected by any INSERT, UPDATE or DELETE operation, and the operation is immediately rolled back, if the updated data violate the constraint. However, this immediate checking might be inefficient for big data operations or even impossible for complex data operation changing data in multiple tables at the same time. In such situations, it is more efficient to define the FOREIGN KEY constraint as

\[\text{In general, the reference can target also the same table, realizing a cyclic relation between record in the same table.}\]
DEFERRABLE \[39\]. Such constraints are checked at the end of the transaction for all records affected by the operations in that transaction at once. This allows inserting data with invalid reference values first, and then fixing this inconsistency by updating the data in the referenced table. This is especially important when there are multiple constraints checking the same reference from different directions, which is frequent when realizing the constraints discussed in section 7.3. In SQL 7.3, the DEFERRABLE version of the FOREIGN KEY constraint shown in SQL 7.2 is shown.

### 7.2 Basic Realization of Constraints

Beside standard database constraints like PRIMARY KEY, FOREIGN KEY and UNIQUE, various additional constraints derived from the original OntoUML PIM need to be realized in the final SQL ISM. There are several distinct options, how these constraints can be realized. In our approach, we use three of these possible realizations: database views, CHECK constraints and triggers.

In the following subsections, the basic principles of each of these possible realizations of the constraints are introduced. The details of using these approaches for the particular types of constraints are discussed in section 7.3. The other viable approaches not used in our approach are discussed in section 7.4 together with the discussion to the efficiency and suitability of the proposed realizations.

#### 7.2.1 Views

Database views can be used to store various queries on the data in the database. Such queries can contain complex statements including complicated WHERE conditions and JOIN statements. Therefore, they can also be used to realize various complex OCL constraints by querying only the valid data meeting the constraints and hiding all data violating the constraints. This approach has been inspired by Dresden OCL Toolkit \[15\], which utilizes database views for querying the data violating the constraints. In our approach, we prefer using the views querying the correct data. We have already discussed the utilizion of this approach for source multiplicity and special multiplicity constraints in \[A.4\] and \[A.5\].

An example of a view is shown in SQL 7.4. This view queries records from the SUBJECT table, for which there is a related record in either the PERSON table or the LEGAL_ENTITY table, but not in both of them (this is one of possible realizations of generalization set constraints discussed in detail in subsubsection 7.3.1.2).
7.2. Basic Realization of Constraints

SQL 7.4 SQL ISM with the CREATE VIEW statement for the view definition

```sql
CREATE VIEW SUBJECT_VIEW AS SELECT * FROM SUBJECT WHERE
  (DISCRIMINATOR = 'Person'
   AND EXISTS (SELECT 1 FROM PERSON WHERE PERSON_ID = SUBJECT_ID)
   AND NOT EXISTS (SELECT 1 FROM LEGAL_ENTITY WHERE LEGAL_ENTITY_ID = SUBJECT_ID))
OR
  (DISCRIMINATOR = 'LegalEntity'
   AND NOT EXISTS (SELECT 1 FROM PERSON WHERE PERSON_ID = SUBJECT_ID)
   AND EXISTS (SELECT 1 FROM LEGAL_ENTITY WHERE LEGAL_ENTITY_ID = SUBJECT_ID))
```

Such realization of the constraints does not slow down the DML operations when inserting, updating or deleting data, as the view is used only for querying the data. However, they do not actually prevent creation of data violating the constraints. It is still possible to create invalid data in the database by modifying the data in the underlying tables and query them using standard SELECT expressions on the underlying tables. Therefore, it is necessary to ensure the correct usage of the views on the application level for valid data querying.

To overcome the problem of having invalid data in the database just hidden by the WHERE clause of the view, the view should be also used to modify the data in the underlying tables. To be able to execute DML operations on a view, the view must be updatable. A view is updatable, if:

- it does not use a `DISTINCT` quantifier, a `GROUP-BY` or a `HAVING` clause,
- all derived columns appear only once in the `SELECT` list,
- each column of the view is derived from exactly one table,
- and the table is used in the query expression in such a way that its PRIMARY KEY or other candidate key relationships are preserved [39].

If the view is updatable, then DML operations like INSERT, UPDATE and DELETE can be executed on the view. In fact, the operations are translated to the corresponding underlying table or tables, and executed on the data directly in these tables. Therefore, it is possible not only to manipulate with the data which are not accessible by the view, but it is also possible to violate the constraints realized by the view. To prevent such operations that affect the data which are not selected by the view, the view must be defined with the `WITH CHECK OPTION` clause [39]. This clause prevents insertion of records not accessible by the view and update operations which make accessible records inaccessible. Thanks to this check, the view can be used to modify the data in the underlying table by INSERT or UPDATE operations, while checking that the resulting data will still comply to the defined condition of the view.

An example of an updatable view definition is shown in SQL 7.5. In SQL 7.6, an example of the error, thrown by the Oracle database when an inserting data violating the WHERE clause of the view, is shown.
7. Transformation of RDB PSM into ISM

**SQL 7.5** SQL ISM with the CREATE VIEW statement for the updatable view definition

```
CREATE VIEW SUBJECT_VIEW AS SELECT * FROM SUBJECT WHERE
    (DISCRIMINATOR = 'Person'
    AND EXISTS (SELECT 1 FROM PERSON WHERE PERSON_ID = SUBJECT_ID)
    AND NOT EXISTS (SELECT 1 FROM LEGAL_ENTITY WHERE LEGAL_ENTITY_ID = SUBJECT_ID))
OR
    (DISCRIMINATOR = 'LegalEntity'
    AND NOT EXISTS (SELECT 1 FROM PERSON WHERE PERSON_ID = SUBJECT_ID)
    AND EXISTS (SELECT 1 FROM LEGAL_ENTITY WHERE LEGAL_ENTITY_ID = SUBJECT_ID))
WITH CHECK OPTION;
```

**SQL 7.6** Error thrown by the Oracle database when inserting data violating the view WHERE clause

```
Error starting at line: 1 in command –
INSERT INTO SUBJECT_VIEW (SUBJECT_ID, DISCRIMINATOR, CITY, COUNTRY, NAME)
VALUES (5, 'Person', 'rybolzde@fit.cvut.cz', 'Prague', 'Czech Republic', 'Z. Rybola')
Error report –
SQL Error: ORA-01402: view WITH CHECK OPTION where-clause violation
01402. 00000 – "view WITH CHECK OPTION where-clause violation"
*Cause: *
*Action: *
```

In context of the transformation of the RDB PSM discussed in chapter 6 into the SQL ISM, using the updatable views with the CHECK OPTION constraint to realize the OCL constraints gives the application the possibility to use the view to modify the data in the underlying table by INSERT and UPDATE operations executed on the view and prevent violation of the realized constraint. However, when the WHERE clause of the view includes checking data in other tables as well, the constraint can be violated by DML operations executed on the other tables. The realization of such checks is complex and depends on the form and contents of the particular OCL constraint. Therefore, we discuss the details in the corresponding sections of section 7.3.

### 7.2.2 CHECK constraints

CHECK constraints are table-level constraints defining certain restrictions for values of the columns in the table, which must be met by all the records in the table \[39\]. The constraint is defined as a predicate expression, which must result into a boolean value (TRUE, FALSE or UNKNOWN). The predicate expression may restrict values in a single column or in multiple columns of the table. The constraint is checked whenever a value is inserted or updated in the column, and the operation is rolled back when the constraint is violated.

Typical usage of CHECK constraints is to restrict a range for the numeric values or provide a list of valid values that can be stored in a column of the table. An example of such CHECK constraint is shown in **SQL 7.7**, where the restriction that the DISCRIMINATOR column of the SUBJECT table can contain only values Person or LegalEntity is shown.
### 7.2. Basic Realization of Constraints

**SQL 7.7** SQL ISM with the `ADD CONSTRAINT` statement to define a CHECK constraint

```
ALTER TABLE "SUBJECT" ADD CONSTRAINT "EN_SUBJECT_TYPE" CHECK
  (DISCRIMINATOR = 'Person' OR DISCRIMINATOR = 'LegalEntity');
```

**SQL 7.8** Error thrown by the Oracle database when inserting data violating the CHECK constraint

```
Error starting at line : 1 in command —
INSERT INTO SUBJECT_VIEW (SUBJECT_ID, DISCRIMINATOR, CITY, COUNTRY, NAME)
VALUES (5, 'Per', 'rybolzde@fit.cvut.cz', 'Prague', 'Czech Republic', 'Zdenek Rybola')
Error report —
SQL Error: ORA-02290: check constraint RYBOLZDE.EN_SUBJECT_TYPE) violated
02290. 00000 — "check_constraint.%s.%s) violated"
*Cause: The values being inserted do not satisfy the named check
*Action: do not insert values that violate the constraint.
```

**SQL 7.9** Error thrown by the Oracle database when creating a CHECK constraint with subqueries

```
Error starting at line : 1 in command —
ALTER TABLE SUBJECT ADD CONSTRAINT GS_SUBJECT_TYPE CHECK
  (DISCRIMINATOR = 'Person'
   AND NOT EXISTS (SELECT 1 FROM PERSON WHERE PERSON_ID = SUBJECT_ID)
   OR (DISCRIMINATOR = 'LegalEntity'
    AND NOT EXISTS (SELECT 1 FROM LEGAL_ENTITY WHERE LEGAL_ENTITY_ID = SUBJECT_ID))
   OR
   AND EXISTS (SELECT 1 FROM LEGAL_ENTITY WHERE LEGAL_ENTITY_ID = SUBJECT_ID)))
Error report —
SQL Error: ORA-02251: subquery not allowed here
02251. 00000 — "subquery not allowed here"
*Cause: Subquery is not allowed here in the statement.
*Action: Remove the subquery from the statement.
```

Then, when inserting a new record into the table with a different `DISCRIMINATOR` value, the error shown in **SQL 7.8**.

In context of the transformation of the RDB PSM discussed in [chapter 6](#) into the SQL ISM, the CHECK constraints can be used to realize the OCL constraints defined in the RDB PSM. However, although valid according to the SQL:1999 specification [39], the common contemporary database engines (including Oracle Database 12c) do not support subqueries in the definition of the CHECK constraints (see [subsection 7.4.1](#) for the comparison of the most common database engines and their capabilities). In **SQL 7.9** the error thrown by Oracle database when creating a CHECK constraint with subqueries is shown. Therefore, the possibility of using the CHECK constraints for the individual types of constraints derived from the OntoUML PIM is discussed in the individual subsections of section 7.3.
Triggers

Triggers are special procedures available in many relational databases \[39\] connected to some special events on a table. According to \[89\] and SQL:1999 specification, the triggers consist of the following parts:

- **name**, uniquely identifying the trigger,
- **triggering event** – a DML operation (INSERT, UPDATE, DELETE) initiating execution of the trigger,
- **activation time**, defining when the trigger should be executed – either BEFORE or AFTER the actual DML operation triggering the trigger,
- **granularity**, defining the scope of the trigger – if it should be executed for each affected row (FOR EACH ROW) or for the whole statement (FOR EACH STATEMENT),
- **condition**, defining the condition when the trigger is executed,
- and **action** – the body of the trigger, which is a sequence of statements executed, when the trigger is triggered.

The body (i.e. the action part) of such a trigger can contain complex logic, consisting of many statements including data querying and manipulation using DML operations. In the body of the trigger, both original record data and the new record data can be accessed by special keywords. Therefore, in context of the realization of the OCL constraints in the RDB PSM, triggers can be used for complex data validations according to the defined constraints before or after any DML operation on the associated table. In \[89\], the authors suggested an approach for deriving triggers from the OCL constraints they realize.

Being able to define complex queries in the trigger body, the triggers are capable to deal with almost every possible constraint by checking the constraint violation and throwing an exception, when the constraint is violated by the DML operation. Thanks to such exception, the whole transaction is rolled back and the data consistency is preserved. Although the checks in the triggers slow down each constrained DML operation, according to \[A.5\], the time increase seems not to be substantial. On the other hand, it is possible to automatically and entirely prevent creating invalid data in the database and save a lot of checking implementation on the application level.

In \[A.4\] and \[A.5\], the idea of using the triggers for checking the source multiplicity and special multiplicity constraints was discussed. However, the same approach can be also used for the other types of OCL constraints defined in the RDB PSM and derived from the OntoUML PIM, as discussed in the individual subsections of section 7.3.

An example of a trigger is shown in SQL 7.10. The trigger is executed before inserting any row into the table CORRESPONDENCE_SUBJECT, detecting if there is a valid combination of the discriminator value and existing record in the corresponding table. If such valid record is not detected, the value inserted into the \_count variable is 0. When such result is found, an application error is raised, signalling the violation of the constraint.
7.3 Realization of Specific OCL Constraints

As discussed in section 7.1, the individual tables defined in the RDB PSM are transformed into CREATE TABLE statements to create the individual tables in the database and their columns, including the NOT NULL constraints for mandatory columns, UNIQUE constraints for columns with unique values, PRIMARY KEY constraints for unique identification of the records and FOREIGN KEY constraints for the references. However, the RDB PSM derived by the transformations of the original OntoUML PIM also contains various OCL constraints defining more complex restrictions for the data. Such constraints must be transformed into more complex structures.

Our approach to the realization of these special OCL constraints in the relational database was introduced for the first time in [A.1], where we discussed transformation of binary relationships from a PIM into a relational database and the realization of various multiplicity constraints using database views. In [A.2], this approach was discussed in context of complete transformation of a PIM into the PSM of a relational database. In [A.3], the idea of realizing the additional multiplicity constraints using CHECK constraints and database triggers was introduced. In [A.4], the possible realizations of special multiplicity constraints are discussed in more details, accompanied by basic selection and insertion experiments. This comparison of possible realizations was later extended by additional possible realizations and additional experiments in [A.5].

The same approach to the realization of the special multiplicity constraints can be also used for the realization of the other additional OCL constraints derived during the transformation of the OntoUML PIM into the RDB PSM. In this approach, we use database views, CHECK constraints and triggers to realize the OCL constraints and enforce the data consistency according to the constraints and restrictions defined by the individual
OntoUML types used in the initial OntoUML PIM. In [A.8], the transformation of Rigid Sortal universal types is discussed with the realization of OCL constraints defining the meta-properties of rigid generalization sets. In [A.10], more examples of the possible realizations of the generalization set constraints were presented and discussed. In [A.9], the transformation of Anti-rigid Sortal types is discussed with the realization of OCL constraints defining the exclusivity of Phases.

In the remaining of this section, the individual types of constraints derived during the transformation of the OntoUML PIM into the RDB PSM and their possible realizations in the SQL ISM are discussed one by one.

7.3.1 Generalization Set Constraints

As discussed in subsection 6.1.3, the generalization sets defined in the UML PIM can be transformed using various approaches: by a single table realizing the whole generalization set, by individual tables for all valid combinations of subclasses having instances, and by individual tables for each subclass with references to the table realizing the superclass. In the case of single table or related tables, OCL constraints are defined to preserve the meta-properties isDisjoint and isCovering of the generalization set are defined. In the following subsubsections, the transformation of these OCL constraints is discussed for each of the variants.

7.3.1.1 Single Table

When the generalization set is transformed into a single table containing the columns representing the attributes of all the subclasses and the superclass as discussed in subsection 6.1.3.1, the generalization set constraint is defined restricting the valid combinations of the discriminator value and empty and non-empty values in the individual columns of the table. An example of such constraint is shown in Constraint 6.1 for the disjoint,complete generalization set shown in Figure 6.5.

In the following paragraphs, the individual possible realizations of the constraint are discussed.

View. As discussed in subsection 7.2.1, a database view can be defined to limit the access only to the valid data meeting the restrictions defined by the OCL constraint. When using such a view, only records with the valid combination of the discriminator value and the empty and non-empty values in the individual columns of the table as defined in the OCL constraint are retrieved from the database, hiding all other records with invalid values.

Therefore, when transforming the generalization set constraint from the RDB PSM into its realization in the SQL ISM, a CREATE VIEW statement can be generated, defining a view with the following properties:

- The name of the constraint is used as the name of the view.
7.3. Realization of Specific OCL Constraints

**SQL 7.11** SQL ISM with the CREATE VIEW statement for the checked updatable database view definition realizing a *generalization set constraint*

```sql
CREATE VIEW GS_SUBJECT_TYPE AS
SELECT * FROM SUBJECT s WHERE
  (s.DISCRIMINATOR = 'Person'
   AND s.PERSON_LAST_NAME IS NOT NULL AND s.PERSON_GENDER IS NOT NULL
   AND s.LEGAL_ENTITY_TITLE IS NULL AND s.LEGAL_ENTITY_VAT IS NULL)
OR (s.DISCRIMINATOR = 'LegalEntity'
   AND s.PERSON_LAST_NAME IS NULL AND s.PERSON_GENDER IS NULL
   AND s.LEGAL_ENTITY_TITLE IS NOT NULL AND s.LEGAL_ENTITY_VAT IS NOT NULL)
WITH CHECK OPTION;
```

- In the SELECT clause of the SELECT statement, all columns of the table in context of the original OCL constraint are selected.
- In the FROM clause of the SELECT statement, records from the table in context of the original OCL constraint are queried.
- In the WHERE clause of the SELECT statement, for each Boolean variable in the original OCL constraint, a Boolean expression is generated comparing the individual columns with the expected values according to the comparison in the original constraint, transforming the comparison with `OclVoid` into IS NULL or IS NOT NULL expressions. Then, these expressions are connected by the OR operator just as defined in the original constraint.

Such realization of the OCL constraint provides the possibility to access only valid data in the database. Still, such realization does not prevent creating invalid data in the database as the DML operations are still executed directly on the actual table without any checks executed. However, as such database view meets the criteria for an *updatable* view, it is defined with the WITH CHECK OPTION clause. Then, the DML operations can be executed on the view. Such operations are then translated to the actual underlying table and the query of the view is checked to prevent inserting records not accessible by the view or updating a record to make it inaccessible.

An example of such a view is shown in **SQL 7.11** where the CREATE VIEW statement for the view realizing the *generalization set constraint* defined in **Constraint 6.1** is shown. The transformation of the other variants of the *generalization set constraint* for the other combinations of the meta-properties are transformed in the very same way, the WHERE-clause just consists of more Boolean expressions.

As the constraint restricts only data of a single record in a single table, it can be violated only by inserting a record with invalid values or updating a record and setting invalid values for it. Using the *updatable* view, these operations are checked, and therefore the data consistency is ensured. Still, the DML operations can be executed directly on the original underlying table and thus cause violation of the constraint. Therefore, it is the responsibility of the application to correctly use the views for querying and manipulating the data in the table to prevent the violation of the constraint.
7. Transformation of RDB PSM into ISM

SQL 7.12 SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the
CHECK constraint definition realizing a generalization set constraint

```sql
ALTER TABLE SUBJECT ADD CONSTRAINT GS_SUBJECT_TYPE CHECK (    (DISCRIMINATOR = 'Person'
    AND PERSON_LastNAME IS NOT NULL AND PERSON_GENDER IS NOT NULL
    AND LEGAL_ENTITY_TITLE IS NULL AND LEGAL_ENTITY_VAT IS NULL)
OR (DISCRIMINATOR = 'LegalEntity'
    AND PERSON_LastNAME IS NULL AND PERSON_GENDER IS NULL
    AND LEGAL_ENTITY_TITLE IS NOT NULL AND LEGAL_ENTITY_VAT IS NOT NULL));
```

CHECK constraint. As discussed in subsection 7.2.2, CHECK constraints can be used
to realize table-level constraints defining certain restrictions for the values in the individual
columns of that table. As the generalization set constraints in the case of the single table
realization only restrict the valid combinations of values in the columns of a single table,
the CHECK constraint can be used to check the values.

Therefore, when transforming the generalization set constraint from the RDB PSM into
its realization in the SQL ISM, an ALTER TABLE ADD CONSTRAINT statement can
be generated, defining a CHECK constraint with the following properties:

- The ALTER TABLE statement alters the table in context of the original OCL con-
  straint.

- The ADD CONSTRAINT statement defines a CHECK constraint.

- The name of the OCL constraint is used as the name of the CHECK constraint.

- In the body of the CHECK constraint definition, for each Boolean variable in the ori-
  ginal constraint, a Boolean expression is generated comparing the individual columns
  with the expected values according to the comparison in the original OCL constraint,
  transforming the comparison with OclVoid into IS NULL or IS NOT NULL expres-
  sions. Then, these expressions are connected by the OR operator just as defined in
  the original constraint.

An example of such a view is shown in SQL 7.12 where the CHECK constraint defi-
nition realizing the generalization set constraint defined in Constraint 6.1 is shown. The
transformation of the other variants of the generalization set constraint for the other com-
binations of the meta-properties are transformed in the very same way, the definition just
consists of more Boolean expressions.

As the CHECK constraint is executed for each DML operation causing creation of data
in the table, it is entirely ensured that no data violating the realized constraint can be
inserted into the constrained table, and therefore, also no invalid data can be retrieved
from the table.
Trigger. As discussed in subsection 7.2.3, triggers are special procedures executed when certain DML operations are executed on a table, that can contain complex logic and data operations. Therefore, they can be used to realize complex constraints, including the generalization set constraints. As the constraint can be violated only by inserting a record with invalid combination of values or setting an invalid combination of values by updating an existing record. Therefore, the trigger needs to check the INSERT and UPDATE operations on the constrained table and raise an application error, if there is an invalid combination of the new values of the record. Furthermore, as the constraint restrict only values of a single record, it can be executed for each affected record individually.

Therefore, when transforming the constraint into its realization in the SQL ISM, the constraint is transformed into a CREATE TRIGGER statement with the following properties:

- The name of the original OCL constraint is used for the name of the trigger.
- The trigger is defined to be executed FOR EACH ROW BEFORE the INSERT and UPDATE operations on the table in context of the original OCL constraint.
- In the DECLARE section of the trigger, a Boolean variable is defined for each Boolean variable in the original OCL constraint, using the same name with the prefix l_.
- In the body of the trigger:
  - For each variable in the original OCL constraint, the value of the appropriate variable from the DECLARE section is set as the result of an expression comparing individual columns with the expected values according to the comparison in the original OCL constraint, transforming the comparison with OclVoid into IS NULL or IS NOT NULL expressions.
  - Then, an IF statement is defined with the condition of negated logical disjunction of the individual variables. In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

An example of such trigger is shown in SQL 7.13 where the CREATE TRIGGER statement realizing the generalization set constraint shown in Constraint 6.1 is shown. The transformation of the other variants of the generalization set constraint for the other combinations of the meta-properties are transformed in the very same way, the definition of the trigger just consists of more Boolean variables, their initialization and their comparison in the IF statement.

Using this realization of the constraint, all possible operations causing the creation of invalid data in the table realizing the whole generalization set are automatically checked. Therefore, it is not possible to create invalid data and the generalization set constraint is entirely realized on the database level, with no need for additional checking on the application level.
7. Transformation of RDB PSM into ISM

SQL 7.13 SQL ISM with the CREATE TRIGGER statement for the trigger definition for a generalization set constraint

```
CREATE TRIGGER GS_SUBJECT_TYPE
BEFORE INSERT OR UPDATE ON SUBJECT
FOR EACH ROW
DECLARE
    l_person_instance BOOLEAN;
    l_legal_entity_instance BOOLEAN;
BEGIN
    l_person_instance := (:new.DISCRIMINATOR = 'Person'
        AND :new.PERSON_LAST_NAME IS NOT NULL AND :new.PERSON_GENDER IS NOT NULL
        AND :new.LEGAL_ENTITY_TITLE IS NULL AND :new.LEGAL_ENTITY_VAT IS NULL);
    l_legal_entity_instance := (:new.DISCRIMINATOR = 'LegalEntity'
        AND :new.PERSON_LAST_NAME IS NULL AND :new.PERSON_GENDER IS NULL
        AND :new.LEGAL_ENTITY_TITLE IS NOT NULL AND :new.LEGAL_ENTITY_VAT IS NOT NULL);

    IF NOT (l_person_instance OR l_legal_entity_instance) THEN
        raise_application_error (-20101, 'OCL constraint GS Subject Type violated!');
    END IF;
END;
```

7.3.1.2 Referencing Tables

When the generalization set is transformed into referencing tables as discussed in subsection 6.1.3.3, the generalization set constraint is defined restricting the valid combinations of the discriminator value and referencing records in the subclass tables. An example of such constraint is shown in Constraint 6.3 for the \{disjoint, complete\} generalization set shown in Figure 6.7.

In the following paragraphs, the individual possible realizations of the constraint are discussed.

Views. As discussed in subsection 7.2.1, a database view can be defined to limit the access only to the valid data meeting the restrictions defined by the OCL constraint. In the case of the generalization set constraint and the realization by referencing tables, a view can be defined to query only such records from the table in context of the original OCL constraint (the superclass table), which have a valid combination of the discriminator value and existing referencing records in the appropriate referencing tables (the subclass tables) as defined in the generalization set constraint.

Although the data of the instances are distributed in multiple tables (the superclass table and the appropriate subclass tables), the complete data of the valid instances can be retrieved by joining the view with the queries from the subclass tables. Thanks to joining the view, all invalid records in the subclass tables are automatically filtered out.

Therefore, when transforming the generalization set constraint from the RDB PSM into its realization in the SQL ISM, a CREATE VIEW statement can be generated, defining a view with the following properties:

- The name of the OCL constraint is used as the name of the view.
7.3. Realization of Specific OCL Constraints

- In the SELECT clause of the SELECT statement, all columns of the table in context of the original OCL constraint are selected.

- In the FROM clause of the SELECT statement, records from the table in context of the original OCL constraint are queried.

- In the WHERE clause of the SELECT statement, for each of the Boolean variables in the original OCL constraint, a Boolean expression is generated realizing its definition. In the expressions, each \texttt{exists} operation on \texttt{allInstances} in a table from the OCL constraint is transformed into an \texttt{EXISTS} expression with a subquery selecting records from the particular table with the \texttt{WHERE} clause realizing the condition from the original OCL constraint. Finally, the generated Boolean expressions are combined using the OR operator as defined in the original OCL constraint.

Such realization of the OCL constraint provides the possibility to access only valid data in the database. Still, such realization does not prevent creating invalid data in the database as the DML operations are still executed directly on the actual table without any checks executed. However, as such database view meets the criteria for an \textit{updatable} view, it is defined with the \texttt{WITH CHECK OPTION} clause. Then, the DML operations can be executed on the view. Such operations are then translated to the actual underlying table and the query of the view is checked to prevent inserting records not accessible by the view or updating a record to make it inaccessible.

However, when using such \textit{updatable} view for inserting a new record or updating an existing record in the table, a referencing record in the appropriate subclass tables must exist. On the other hand, to insert a record into the subclass tables, the referenced record in the superclass table must exist, which is checked by the \texttt{FOREIGN KEY} constraint. To allow the insertion with such mutually dependent constraints, the \texttt{FOREIGN KEY} must be defined \texttt{DEFERRABLE} as discussed in \texttt{subsection 7.1.2}. With such deferred reference checking, it is possible to insert the data in a single transaction by inserting records into the appropriate subclass tables first, without checking the reference validity, and then inserting the record in the superclass table, checking the existence of the referencing records in the subclass tables. The references are checked at the end of the transaction before committing the changes to the database, and when the references are not valid (i.e. the appropriate superclass table record was not inserted), the whole transaction is rolled back, undoing all the data changes.

An example of such view is shown in \texttt{SQL 7.14}, where the \texttt{CREATE VIEW} statement realizing the \textit{generalization set constraint} shown in \texttt{Constraint 6.3} is shown. The transformation of the other variants of the \textit{generalization set constraint} for the other combinations of the meta-properties are transformed in the very same way, the definition of the view just consists of more Boolean expressions.

This checked updatable view checks the DML operations executed on the superclass table through the view, preventing creation of a record in the superclass table without appropriate referencing record by an \texttt{INSERT} or \texttt{UPDATE} operations. Beside that, the \texttt{DEFERRED FOREIGN KEY} constraint ensures that at the end of the transaction, each
SQL 7.14 SQL ISM with the CREATE VIEW statement for the database view definition realizing a generalization set constraint

```
CREATE VIEW GS_SUBJECT_TYPE AS
SELECT * FROM SUBJECT s
WHERE
  (s.DISCRIMINATOR = 'Person'
   AND EXISTS (SELECT 1 FROM PERSON p WHERE p.PERSON_ID = s.SUBJECT_ID)
   AND NOT EXISTS (SELECT 1 FROM LEGALENTITY le WHERE le.LEGAL_ENTITY_ID = s.SUBJECT_ID))
OR (s.DISCRIMINATOR = 'LegalEntity'
   AND NOT EXISTS (SELECT 1 FROM PERSON p WHERE p.PERSON_ID = s.SUBJECT_ID)
   AND EXISTS (SELECT 1 FROM LEGALENTITY le WHERE le.LEGAL_ENTITY_ID = s.SUBJECT_ID))
WITH CHECK OPTION;
```

record in the particular subclass table references an existing record in the superclass table. Also, the PRIMARY KEY constraints on the reference columns in the subclass table prevent having multiple records in the same table referencing the same record in the superclass table. However, there are still other situations, which may cause violation of the constraint. The overview of such additional situations causing the constraint violation are itemized in the following list, together with the possible checks realized by database views:

- When inserting a record into a subclass table, it should be checked that the referenced record does not exist. Otherwise, thanks to the condition of the updatable view, such record would already have appropriate referencing records and the new record would be violating the constraint. Therefore, an updatable view on the subclass table should be used for the INSERT operation to check this condition. An example of such a view is shown in SQL 7.15.

- When changing the reference value of a record in a subclass table, it should be checked that neither the new referenced record nor the new referencing record does not exist. Otherwise, the new referenced record would already have appropriate referencing records thanks to the condition of the updatable view, and the old referenced would loose a required referencing record. Therefore, a special updatable view on the subclass table should be used for the UPDATE operation, which can access only records meeting the condition and cannot create records violating it thanks to the WITH CHECK OPTION clause. As the condition is the same as in the case of inserting into the subclass table, the same view can be used. An example of such view is shown in SQL 7.15. However, such view can be used only for updating the reference value, as all current valid records (i.e. records with valid reference) are filtered out, because such records cannot change the reference value.

- When deleting a record from a subclass table, its referenced record in the superclass table should not exist. Otherwise, after deleting the referencing record, it would become invalid. Therefore, an updatable view on the subclass table should be used for the DELETE operation to check this condition. As the condition is the same
### 7.3. Realization of Specific OCL Constraints

**SQL 7.15** SQL ISM with the CREATE VIEW statement for the database view definition for a subclass table realizing a *generalization set constraint*

```sql
CREATE VIEW GS_SUBJECT_TYPE_PERSON AS
SELECT * FROM PERSON p WHERE
NOT EXISTS (SELECT 1 FROM SUBJECT s WHERE s.SUBJECT_ID = p.PERSON_ID)
WITH CHECK OPTION;
```

As in the case of the INSERT operation, the save view can be used, just as shown in [SQL 7.15](#).

As discussed above, the view can be easily used to hide all invalid records from querying. However, although *updatable*, it is not able to prevent all possible situations which may cause violation of the constraint when executing DML operation. Although other *updatable* views can be defined, they still cannot cover all situations: when updating a record in a subclass table, different views would be needed to change different data of the record. Moreover, the actual tables are still accessible by the application. Therefore, it is still the application responsibility to use the views with the implemented constraint checks correctly.

**CHECK constraints.** As discussed in subsection 7.2.2, CHECK constraints can be used to realize table-level constraints which must be satisfied by all records in the table at any time. As the *generalization set constraint de facto* restricts the possible values in the discriminator column and the values of the PRIMARY KEY column, which must be referenced from the particular subclass tables, it seems logical that the constraint can be realized by a CHECK constraint.

Therefore, when transforming the *generalization set constraint* from the RDB PSM into its realization in the SQL ISM, it can be transformed into the ALTER TABLE ADD CONSTRAINT statement defining the CHECK constraint with the following properties:

- The ALTER TABLE statement alters the table in context of the original OCL constraint.
- The ADD CONSTRAINTS statement defined a CHECK constraint with the name of the original OCL constraint used as the name of the CHECK constraint.
- In the body of the CHECK constraint definition, for each Boolean variable in the original constraint, a Boolean expression is generated realizing its definition. In the expressions, each `exists` operation on `allInstance` in a table from the OCL constraint is transformed into an EXISTS expression with a subquery selecting records from the particular table with the WHERE clause realizing the condition from the original OCL constraint. Finally, the generated Boolean expressions are combined using the OR operator as defined in the original OCL constraint.
An example of such a CHECK constraint is shown in SQL 7.16, where the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint realizing the generalization set constraint shown in Constraint 6.3 is shown. Similar statement would be generated also for the other types of generalization set constraints for the other combinations of the meta-properties.

However, as discussed in subsection 7.2.2, the contemporary common database engines (including Oracle Database 12c) do not support subqueries in the definition of the CHECK constraints. Therefore, this realization of the OCL constraint is not applicable. Moreover, this CHECK constraint would only check DML operations on the superclass table, while the constraint can be violated also by operation on the subclass tables. Additional CHECK constraints would be needed for the subclass tables, but even those would contain subqueries. Therefore, the CHECK constraints, although valid according to the specification, are not applicable for the realization of the generalization set constraints on a generalization set realized by referencing table, until the database engines support using of subqueries in the CHECK constraints.

**Triggers.** As discussed in subsection 7.2.3, triggers are special procedures executed when certain DML operations are executed on a table, that can contain complex logic and data operations. Therefore, they can be used to realize complex constraints, including the generalization set constraint restricting the valid combination of the discriminator values in the constraint table (the superclass table) and existing referencing records in the appropriate referencing tables (subclass tables).

As the data of instances are distributed in multiple tables (the superclass table and the subclass tables), DML operations on any of them can cause violation of the constraint. In the following list, an overview of the possible situations causing violation of the generalization set constraint is shown with the needed checks and triggers:

1. inserting a record in the superclass table without having the appropriate records in the subclass tables;
2. inserting a record in a subclass table, referencing an existing record in the superclass table of a different type (when such record exists, thanks to ad 1), all required referencing records must already exist, and therefore adding a new one leads to an unexpected referencing record;
3. updating a record in the superclass table and changing its discriminator value without referencing records in the appropriate subclass tables according to the new discriminator value;

4. updating a record in the superclass table and changing its PRIMARY KEY value, which is not referenced by records in the appropriate subclass tables according to the discriminator value of the updated record;

5. updating a record in the subclass table and changing its reference value to a value referencing a record in the superclass table of a different type (such record does not expect a referencing record in the updated subclass table), or from a value referencing an existing record in the superclass table (such record would loose the required referencing record);

6. deleting a record from a subclass table referencing an existing record in the superclass table (such record would loose the required referencing record).

Although there are the FOREIGN KEY constraints defined in the subclass tables, checking the existence of the referenced record in the superclass table, and the references are combined with the PRIMARY KEY values in the subclass tables, preventing saving multiple records with the same reference value, they are not capable of preventing the situations listed above. To prevent such situations, the following types of triggers should be defined:

**T1 – BEFORE INSERT OR UPDATE on the superclass table:** The trigger checks that when inserting a new record into the superclass table, the records in the appropriate subclass tables referencing the inserting record exist according to the discriminator value. The same trigger is also used for the checking when updating a record in the superclass table (especially when changing the PRIMARY KEY value or the discriminator value) – the new version of the superclass table record must have appropriate records in the subclass tables.

**T2 – BEFORE INSERT on each of the subclass tables:** The trigger checks that when inserting a new record into the particular subclass table, the referenced record in the superclass table does not yet exist. This check is needed because when inserting into the superclass table, the records in the appropriate subclass tables must already exist. Any record inserted after the insertion of the record into the superclass table must necessarily be invalid.

**T3 – BEFORE UPDATE on each of the subclass tables:** The trigger checks that when updating a record in the subclass table and changing its reference value, there should be no record in the superclass table referenced by neither the old reference value nor the new reference value. Existence of the record referenced by the old reference value means leaving an invalid record in the superclass table. Existence of the record referenced by the new reference value means it is the type not expecting
referencing record in that subclass table (the record in the superclass table can be created only when the appropriate records in the subclass tables exist, creating a new record in another subclass table referencing the same record means violation of the constraint).

**T4 – BEFORE DELETE OR UPDATE on each of the subclass tables:** The trigger checks that the record in the superclass table referenced by the deleted record does not exist. Otherwise, deleting the record in the subclass table would invalidate the record in the superclass table.

Therefore, when transforming the *generalization set constraint* from the RDB PSM into its realization in the SQL ISM, a CREATE TRIGGER statement is generated for the trigger **T1** on the table in context of the original OCL constraint (the superclass table) and other CREATE TRIGGER statements are generated for triggers **T2**, **T3** and **T4** on each of the referencing tables defined in the original OCL constraint (the subclass tables).

The CREATE TRIGGER statement for the trigger **T1** on the superclass table is defined in the following form (an example of the trigger for the *generalization set constraint* defined in **Constraint 6.3** is shown in **SQL 7.17**):

- The name of the original OCL constraint is used for the name of the trigger.
- The trigger is defined to be executed FOR EACH ROW BEFORE the INSERT and UPDATE operations on the table in context of the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable **l_count** is defined to hold the number of found records.
- In the body of the trigger:
  - A SELECT statement is defined, selecting **count** of records from the special table **DUAL** with the WHERE clause realizing the OCL constraint. The result is stored into the defined variable **l_count**. In the WHERE clause, for each of the Boolean variables in the original OCL constraint, a Boolean expression is generated realizing its definition. In these expressions, each **exists** operation on **allInstances** in a table from the OCL constraint is transformed into an EXISTS expression with a subquery selecting records from the particular table with the WHERE clause realizing the condition from the original OCL constraint. In all the comparisons, the new values of the record are used using the **:new** keyword. Finally, the generated Boolean expressions are combined using the OR operator as defined in the original OCL constraint.
  - Then, an IF statement is defined with the condition of the **l_count** value being equal to 0 (meaning that the expected referencing records were not found). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.
7.3. Realization of Specific OCL Constraints

**SQL 7.17** SQL ISM with the CREATE TRIGGER statement for the trigger definition on the superclass table realizing a *generalization set constraint*

```sql
CREATE OR TRIGGER GS_SUBJECT_TYPE
BEFORE INSERT OR UPDATE ON SUBJECT
FOR EACH ROW
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT COUNT(*) INTO l_count FROM DUAL WHERE
  (:new.DISCRIMINATOR = 'Person'
   AND EXISTS (SELECT 1 FROM PERSON p WHERE p.PERSON_ID = :new.SUBJECT_ID)
   AND NOT EXISTS (SELECT 1 FROM LEGAL_ENTITY le WHERE le.LEGAL_ENTITY_ID = :new.SUBJECT_ID))
  OR (:new.DISCRIMINATOR = 'LegalEntity'
    AND NOT EXISTS (SELECT 1 FROM PERSON p WHERE p.PERSON_ID = :new.SUBJECT_ID)
    AND EXISTS (SELECT 1 FROM LEGAL_ENTITY le WHERE le.LEGAL_ENTITY_ID = :new.SUBJECT_ID)));
  IF l_count = 0 THEN
    raise_application_error (-20101, 'OCL constraint GS_Subject_Type violated!');
  END IF;
END;
```

The CREATE TRIGGER statement for the trigger \textit{T2} on a subclass table is defined in the following form (an example of the trigger for the *generalization set constraint* defined in \textit{Constraint 6.3} and the PERSON table is shown in **SQL 7.18**):

- The name of the trigger is generated by concatenating the name of the original OCL constraint, the name of the table and postfix \textit{INS}.
- The trigger is defined to be executed \textit{FOR EACH ROW BEFORE} the INSERT operation on the particular table.
- In the DECLARE section of the trigger, a Number variable \textit{l_count} is defined to hold the number of found records.
- In the body of the trigger:
  - A SELECT statement is defined, selecting \textit{count} of records from the special table DUAL into the defined variable \textit{l_count}. In the \textit{WHERE} clause of this statement, an EXISTS expression is defined with a subquery selecting from the table in context of the original OCL constraint (the superclass table) with the \textit{WHERE} condition of the PRIMARY KEY value equal to the reference value of the new record (using the keyword \textit{:new}).
  - Then, an IF statement is defined with the condition of the \textit{l_count} value being higher than 0 (meaning that the unexpected referenced record was found). In the \textit{THEN} clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

135
SQL 7.18 SQL ISM with the CREATE TRIGGER statement for the trigger definition for the INSERT operation on a subclass realizing a \textit{generalization set constraint}

\begin{verbatim}
CREATE TRIGGER GS_SUBJECT_TYPE_PERSON_INS
BEFORE INSERT ON PERSON
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT COUNT(*) INTO l_count FROM DUAL WHERE EXISTS (SELECT 1 FROM SUBJECT s WHERE s.SUBJECT_ID = :new.PERSON_ID);
  IF l_count > 0 THEN
    raise_application_error (-20101, 'OCL constraint GS_Subject_Type_violated!');
  END IF;
END;
\end{verbatim}

The CREATE TRIGGER statement for the trigger \textbf{T3} on a subclass table is defined in the following form (an example of the trigger for the \textit{generalization set constraint} defined in \textbf{Constraint 6.3} and the \textbf{PERSON} table is shown in \textbf{SQL 7.19}):

\begin{itemize}
  \item The name of the trigger is generated by concatenating the name of the original OCL constraint, the name of the table and postfix \texttt{UPD}.
  \item The trigger is defined to be executed \texttt{FOR EACH ROW BEFORE} the \texttt{UPDATE} operation on the particular table.
  \item In the DECLARE section of the trigger, a Number variable \texttt{l_count} is defined to hold the number of found records.
  \item In the body of the trigger:
    \begin{itemize}
      \item An IF statement is defined with the condition of distinct old and new reference values of the updated record using the \texttt{:old} and \texttt{:new} keywords. In the THEN clause of this IF statement, a SELECT statement is defined, selecting \texttt{count} of records from the special table \texttt{DUAL} into the defined variable \texttt{l_count}. In the WHERE clause of this statement, an EXISTS expression is defined with a subquery selecting from the table in context of the original OCL constraint (the superclass table) with the WHERE condition of the PRIMARY KEY value equal to the old reference value (using the keyword \texttt{:old}) or the new reference value (using the keyword \texttt{:new}).
      \item Then, an IF statement is defined with the condition of the \texttt{l_count} value being higher than 0 (meaning that the unexpected referenced record was found for the old or the new value). In the THEN clause of the statement, an application error is raised with the code \texttt{-20101} and the message, that the OCL constraint is violated.
    \end{itemize}
\end{itemize}
SQL 7.19 SQL ISM with the CREATE TRIGGER statement for the trigger definition for the UPDATE operation on a subclass realizing a generalization set constraint

```
CREATE TRIGGER GS_SUBJECT_TYPE_PERSON_UPD
BEFORE UPDATE ON PERSON
FOR EACH ROW
DECLARE
    l_count NUMBER := 0;
BEGIN
    IF :old.PERSON_ID <> :new.PERSON_ID THEN
        SELECT COUNT(*) INTO l_count FROM DUAL WHERE EXISTS (SELECT 1 FROM SUBJECT s
        WHERE s.SUBJECT_ID = :old.PERSON_ID OR s.SUBJECT_ID = :new.PERSON_ID);
    END IF;
    IF l_count > 0 THEN
        raise_application_error (-20101, 'OCL_constraint_GS_Subject_Type_violated!');
    END IF;
END;
```

The CREATE TRIGGER statement for the trigger **T4** on a subclass table is defined in the following form (an example of the trigger for the generalization set constraint defined in Constraint 6.3 and the **PERSON** table is shown in SQL 7.20):

- The name of the trigger is generated by concatenating the name of the original OCL constraint, the name of the table and postfix DEL.
- The trigger is defined to be executed FOR EACH ROW BEFORE the DELETE operation on the particular table.
- In the DECLARE section of the trigger, a Number variable **l_count** is defined to hold the number of found records.
- In the body of the trigger:
  - A SELECT statement is defined, selecting **count** of records from the special table **DUAL** into the defined variable **l_count**. In the WHERE clause of this statement, an EXISTS expression is defined with a subquery selecting from the table in context of the original OCL constraint (the superclass table) with the WHERE condition of the PRIMARY KEY value equal to the reference value of the deleted record (using the keyword :old).
  - Then, an IF statement is defined with the condition of the **l_count** value being higher than 0 (meaning that the unexpected referenced record was found). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

When using the triggers for realizing the generalization set constraints, the the trigger **T1** checks the existence of referencing records before creating the referenced record. However, this conflicts with the FOREIGN KEY constraints defined for the references.
SQL 7.20 SQL ISM with the CREATE TRIGGER statement for the trigger definition for the DELETE operation on a subclass realizing a generalization set constraint

```
CREATE TRIGGER GS_SUBJECT_TYPE_PERSON_DEL
BEFORE DELETE ON PERSON
FOR EACH ROW
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT COUNT(*) INTO l_count FROM DUAL WHERE EXISTS (  
    SELECT 1 FROM SUBJECT s WHERE s.SUBJECT_ID = :old.PERSON_ID);  
  IF l_count > 0 THEN
    RAISE_APPLICATION_ERROR (-20101, 'OCL_constraint_GS_Subject_Type_violated!');
  END IF;
END;
```

Therefore, the FOREIGN KEY constraints must be defined DEFERRABLE, so they are checked at the end of the transaction, after the referenced record is truly inserted.

In total, when using the triggers, the consistency of the data in the database is ensured entirely. It is because the triggers are triggered for all operations which can cause violation of the generalization set constraint and all operations really causing the violation are rolled back. On the other hand, additional queries are executed when manipulating data in the tables, checking the data already in the tables. Therefore, the consistency is achieved in exchange for query and DML operation efficiency. See subsection 7.4.6 for the discussion about the efficiency of the constraint realizations.

7.3.2 Distributed Unique Constraint

As discussed in subsubsection 6.1.3.2, the generalization sets defined in the UML PIM can be also transformed into a set of separate tables realizing the individual types of the instances according to the meta-properties of the generalization set. Such tables contain always all data of the instances, thus the tables are completely independent of each other (unlike in the case of the realization by referencing tables discussed in subsection 6.1.3.3).

However, when an attribute shared by multiple types of instances (superclass attribute, or even a subclass attribute in the case of an overlapping generalization set) is required to be unique, special distributed unique constraints must be defined, as the data of such attribute are stored in multiple tables. These constraints are defined for all of the tables realizing the generalization set and containing the column realizing the constrained attribute, checking that the value in the constrained column in the particular table does not exist in the other tables. Examples of such constraint are shown in Constraint 6.2.

In the following paragraphs, the individual possible realizations of the constraint are discussed.
7.3. Realization of Specific OCL Constraints

SQL 7.21 SQL ISM with the CREATE VIEW statement for the database view definition realizing a distributed unique constraint

```sql
CREATE VIEW UQ_PERSON_NAME AS
SELECT * FROM PERSON p WHERE (NOT EXISTS
(SELECT 1 FROM LEGAL_ENTITY le WHERE le.NAME = p.NAME))
WITH CHECK OPTION;
```

Views. As discussed in subsection 7.2.1, a database view can be defined to limit the access only to the valid data meeting the restrictions defined by the OCL constraint. When using such a view for the distributed unique constraint, only records with the unique values in the constrained column not existing in the other tables representing the other combinations of classes from the generalization set can be queried. Therefore, when transforming the distributed unique constraint from the RDB PSM into its realization in the SQL ISM, a CREATE VIEW statement can be generated, defining a view with the following properties:

- The name of the OCL constraint is used as the name of the view.
- In the SELECT clause of the SELECT statement, all columns of the table in context of the original OCL constraint are selected.
- In the FROM clause of the SELECT statement, records from the table in context of the original OCL constraint are queried.
- In the WHERE clause of the SELECT statement, for each of the Boolean variables in the original OCL constraint, a Boolean expression is generated realizing its definition. In the expressions, each negated `exists` operation on `allInstances` in a table from the OCL constraint is transformed into an NOT EXISTS expression with a subquery selecting records from the particular table with the WHERE clause realizing the condition from the original OCL constraint. Finally, the generated Boolean expressions are combined using the AND operator as defined in the original OCL constraint.

As such view meets the requirements for being updatable, it can be defined with the WITH CHECK OPTION. Then, such view can be used beside querying also for the DML operations. When executing such DML operations, the constraint is also checked and its violation is prevented. An example of such view is shown in SQL 7.21, where the CREATE VIEW statement realizing the constraint `UQ_PERSON_NAME` shown in Constraint 6.2 is shown.

As the constraint can be violated only by inserting or updating records in the individual tables and such updatable views are defined for all of these tables, the constraint is enforced entirely. However, the original underlying tables can be still accessed directly, and therefore this solution is not completely reliable. It is still the responsibility of the application to use the views correctly for the queries and DML operations.
7. Transformation of RDB PSM into ISM

SQL 7.22 SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing a distributed unique constraint

```sql
ALTER TABLE PERSON ADD CONSTRAINT UQ_PERSON_NAME CHECK ( NOT EXISTS (SELECT 1 FROM LEGAL_ENTITY le WHERE le.NAME = NAME) ) ;
```

CHECK constraints. As discussed in subsection 7.2.2, CHECK constraints can be used to realize table-level constraints defining certain restrictions for the values in the individual columns of that table. As the distributed unique constraints restrict the values in the constrained column to values not existing in the other tables, it seems viable to realize such constraints using the CHECK constraints.

Therefore, when transformation a distributed unique constraint from the RDB PSM into its realization in the SQL ISM, an ALTER TABLE ADD CONSTRAINT statement can be generated to define the CHECK constraint in the following form:

- The ALTER TABLE statement alters the table in context of the original OCL constraint.
- The ADD CONSTRAINT statement defines a CHECK constraint with the name of the original OCL constraint used as the name of the CHECK constraint.
- In the body of the CHECK constraint definition, for each of the Boolean variables in the original OCL constraint, a Boolean expression is generated realizing its definition. In the expressions, each negated exists operation on allInstances in a table from the OCL constraint is transformed into an NOT EXISTS expression with a subquery selecting records from the particular table with the WHERE clause realizing the condition from the original OCL constraint. Finally, the generated Boolean expressions are combined using the AND operator as defined in the original OCL constraint.

An example of such CHECK constraint is shown in SQL 7.22 where the ALTER TABLE ADD CONSTRAINT statement for the constraint UQ_PERSON_NAME shown in Constraint 6.2 is shown. Although such CHECK constraint is able to completely prevent violation of the constraint by DML operations on the particular table, it is unfortunately not applicable, as it contains subqueries which are not supported by the common contemporary database engines. Therefore, this realization is not applicable until the database engines support such subqueries in the CHECK constraints.

Triggers. As discussed in subsection 7.2.3, triggers are special procedures which are executed when certain DML operations are executed on a table and which can contain complex logic and data operations. Therefore, they can be used to realize complex constraints. In the case of the distributed unique constraints, a trigger can check, that the value of the manipulated record in the constrained table does not exist in the other tables and throw an application error if it exists.
As the distributed unique constraint can be only violated by inserting a record with a value already existing in the other tables or by updating a record and setting its value to a value already existing in the other tables, only a single trigger would be needed for the realization, being executed for each affected record individually before the INSERT and UPDATE operations. Therefore, when transforming the distributed unique constraint from the RDB PSM into its realization in the SQL ISM, a CREATE TRIGGER statement can be generated, defining a trigger in the following form:

- The name of the original OCL constraint is used for the name of the trigger.
- The trigger is defined to be executed FOR EACH ROW BEFORE the INSERT and UPDATE operations on the table in context of the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable `l_count` is defined to hold the number of found records.
- In the body of the trigger:
  - A SELECT statement is defined, selecting `count` of records from the special table DUAL into the defined variable `l_count`. In the WHERE clause of this statement, for each of the Boolean variables in the original OCL constraint, a Boolean expression is generated realizing its definition. In these expressions, each `exists` operation on `allInstances` in a table from the OCL constraint is transformed into an EXISTS expression with a subquery selecting records from the particular table with the WHERE clause realizing the condition from the original OCL constraint. In all the comparisons, the new values of the record are used using the `:new` keyword. Finally, the generated Boolean expressions are combined using the AND operator as defined in the original OCL constraint.
  - Then, an IF statement is defined with the condition of the `l_count` value being higher than 0 (meaning that the unexpected records with the same value were found in the other tables). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

An example of such a trigger is shown in SQL 7.23 where the CREATE TRIGGER statement is shown for the constraint `UQ_PERSON_NAME` shown in Constraint 6.2.

Using this realization of the constraint, all possible operations causing the creation of invalid data in the tables realizing the generalization set are checked. Therefore, it is not possible to create invalid data and the distributed unique constraint is entirely realized on the database level.

### 7.3.3 Special and Mandatory Multiplicity Constraints

As discussed in subsection 6.1.4 in certain situations, the multiplicity values of the associations defined in the UML PIM cannot be realized in the RDB PSM by the standard
**SQL 7.23** SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing a distributed uniqueness constraint

```sql
CREATE TRIGGER UQ_PERSON_NAME
BEFORE INSERT OR UPDATE ON PERSON
FOR EACH ROW
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT count(1) INTO l_count FROM DUAL WHERE (
    EXISTS (SELECT 1 FROM LEGAL_ENTITY le WHERE le.NAME = :new.NAME));
  IF l_count > 0 THEN
    RAISE_APPLICATION_ERROR (-20010, 'OCL constraint UQ_PERSON_NAME violated!');
  END IF;
END;
```

**View.** As discussed in subsection 7.2.1, a database view can be defined to limit the access only to the valid data meeting the restrictions defined by the OCL constraint. In context of the special multiplicity constraint, the view can be used to access only such records in the table which are related to the appropriate number of records in the other table and hide the rest.

Therefore, when transforming the special multiplicity constraint from the RDB PSM into the SQL ISM, a CREATE VIEW statement can be generated, defining a view with the following properties:

- The name of the OCL constraint is used as the name of the view.
- In the SELECT clause of the SELECT statement, all columns of the table in context of the original OCL constraint are selected.
7.3. Realization of Specific OCL Constraints

SQL 7.24 SQL ISM with the CREATE VIEW statement for the view realizing a *special multiplicity constraint*

\[
\text{CREATE VIEW MUL_WORK_TO_SERIES_SERIES_ID AS}
\]

\[
\text{SELECT * FROM SERIES s WHERE}
\]

\[
\text{(SELECT COUNT(1) FROM WORK_TO_SERIES wts WHERE wts\.SERIES_ID = s\.SERIES_ID) \text{ \textgreater= 2}}
\]

\[
\text{WITH CHECK OPTION;}
\]

- In the FROM clause of the SELECT statement, records from the table in context of the original OCL constraint are queried.

- The WHERE clause of the SELECT statement realizes the body of the original OCL constraint, depending on the type of the *special multiplicity constraint*:

  - In the case of the standard *special multiplicity constraint*, the WHERE clause consists of a SELECT subquery for each of the count comparisons in the original OCL constraint (lower and upper limit). In this SELECT subquery, the count of records in the referencing table (checked by the allInstances and count operations in the original OCL constraint) with the WHERE clause realizing the comparison in the count operation of the original OCL constraint is selected. Then, the result of the subquery is compared with the appropriate limit by the \( \text{\textgreater= } \text{or } \text{\textless=} \) operators. Finally, if both limits are defined, the comparison results are combined using the logical conjunction (AND).

  - In the case of the *mandatory multiplicity constraint*, the WHERE clause consists of an EXISTS expression with a subquery selecting records from the referencing table (the table checked by the allInstances and exists operations in the original OCL constraint) with the with the WHERE clause realizing the condition from the original OCL constraint.

As such view meets the requirements for being *updatable*, it can be defined with the WITH CHECK OPTION. Then, such view can be used beside querying also for the DML operations. When executing such DML operations, the constraint is also checked and its violation is prevented. However, in such situation, the condition of the view requires having the referencing records before creating the referenced record, which is in conflict with the FOREIGN KEY constraint requiring the existence of the referenced record. Therefore, the FOREIGN KEY constraint must be defined DEFERRABLE to allow the checking at the end of the transaction, when also the referenced record is already inserted.

An example of the view definition for the *special multiplicity constraint* is shown in SQL 7.24, where the CREATE VIEW statement realizing the constraint shown in Constraint 6.4 is shown. An example of the view definition for a *mandatory multiplicity constraint* is shown in SQL 7.25, where the CREATE VIEW statement realizing the constraint shown in Constraint 6.5 is shown.

However, such *updatable* view is only capable of checking the DML operations for the table in context of the original OCL constraint. Still, the *special multiplicity constraint* can...
be violated by DML operations on the referencing table, creating additional referencing records and exceeding the upper limit or removing some of the referencing records and getting below the lower limit. Unfortunately, these situations cannot be completely realized and checked by database views. It is because it is not possible to effectively restrict the condition in the DELETE operation, allowing deleting many records at the same time. Also, any of the records in the related table can be removed unless the total number of related records gets below the limit, and therefore it is not possible to define a view to filter only deletable records, as it was in the case of generalization set realized by referencing tables in [subsection 7.3.1.2]. Similarly, it is not possible to define a view checking the condition that the updated record changing the reference value does not exceed the upper limit for the new referenced record nor causes getting below the lower limit for the old referenced record. For checking of such situation, a different type of realization is necessary (see the other paragraphs).

**CHECK constraint.** As discussed in [subsection 7.2.2] CHECK constraints can be used to realize table-level constraints which must be satisfied by all records in the table at any time. As the special multiplicity constraint de facto restricts the PRIMARY KEY value, which must be referenced from the appropriate number of records in the referencing table, it seems reasonable to check the value and existence of the referencing records by a CHECK constraint.

Therefore, when transforming the special multiplicity constraint from the RDB PSM into its realization in the SQL ISM, an ALTER TABLE ADD CONSTRAINT statement can be generated to define the CHECK constraint in the following form:

- The ALTER TABLE statement alters the table in context of the original OCL constraint.
- The ADD CONSTRAINT statement defines a CHECK constraint with the name of the original OCL constraint used as the name of the CHECK constraint.
- In the body of the CHECK constraint definition, the realization of the actual OCL constraint is located, depending on the type of the special multiplicity constraint:
  - In the case of the standard special multiplicity constraint, the body of the constraint consists of an expression checking that the value of the PRIMARY KEY column is in the set of all valid values using the IN operator. This set of valid

---

**SQL 7.25 SQL ISM with the CREATE VIEW statement for the view realizing a mandatory multiplicity constraint**

```sql
CREATE VIEW MUL_AUTHORSHIP_WRITER_ID AS
  SELECT * FROM WRITER w WHERE EXISTS
  (SELECT 1 FROM AUTHORSHIP a WHERE a.WRITER_ID = w.WRITER_ID)
WITH CHECK OPTION;
```
7.3. Realization of Specific OCL Constraints

SQL 7.26 SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the
CHECK constraint definition realizing special multiplicity constraint

```
ALTER TABLE SERIES ADD CONSTRAINT MUL_WORK_TO_SERIES_SERIES_ID CHECK (SERIES_ID IN (SELECT wts.SERIES_ID FROM WORK_TO_SERIES wts GROUP BY wts.SERIES_ID HAVING COUNT(wts.SERIES_ID) >= 2));
```

SQL 7.27 SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the
CHECK constraint definition realizing simplified special multiplicity constraint

```
ALTER TABLE WRITER ADD CONSTRAINT MUL_AUTHORSHIP_WRITER_ID CHECK (WRITER_ID IN (SELECT WRITER_ID FROM AUTHORSHIP));
```

values is created by selecting the reference value from all records in the related
table checked by the `count` operation in the original OCL constraint, grouped
by the reference value and filtered by the limits using the `HAVING` clause with
the `COUNT` operation on the reference value compared with the limits defined
in the original OCL constraint.

- In the case of the mandatory multiplicity constraint, the body of the constraint
  consists of an expression checking that the value of the PRIMARY KEY column
  is in the set of all valid values using the `IN` operator. The set of valid values
  is created by selecting the reference value from all the records in the related table
  checked by the `count` operation in the original OCL constraint.

An example of the CHECK constraint definition for a special multiplicity constraint
is shown in [SQL 7.26] where the ALTER TABLE ADD CONSTRAINT statement realizing
the constraint shown in [Constraint 6.4] is shown. An example of the CHECK constraint
definition for a mandatory multiplicity constraint is shown in [SQL 7.27] where the ALTER
TABLE ADD CONSTRAINT statement realizing the constraint shown in [Constraint 6.5]
is shown.

Both variants of the CHECK constraint check every operation on the constrained table
and prevent creating a record in that table without appropriate number of related records
in the related table. However, these CHECK constraints do not check the operations
executed on the other related table, where deleting or updating a record may violate the
OCL constraint by exceeding the upper limit or getting below the lower limit. Therefore,
additional CHECK constraint would be needed also for the referencing table. Such CHECK
constraint would be able to prevent inserting and updating records exceeding the upper
limit, but it is not capable of prevent deleting and updating records getting below the lower
limit. Moreover, the checking would require the FOREIGN KEY constraint restricting the
reference defined as DEFERRABLE.

Unfortunately, as the CHECK constraints realizing the special multiplicity constraints
(including the mandatory multiplicity constraints) are based on subqueries selecting the
referencing records, they cannot be implemented in the common contemporary database
engines, as they do not support subqueries in the CHECK constraint definitions. Therefore, this realization is not applicable until the database engines start to support it.

Trigger. As discussed in subsection 7.2.3, triggers are special procedures executed when certain DML operations are executed on certain database table. As they can contain complex logic and data operations, they can be also used for checking complex constraints, including the special multiplicity constraints defined in the RDB PSM.

As the special multiplicity constraint restricts the number of related records in the related table referencing the same record in the table in context of the constraint, it can be violated by various DML operations executed both on the constrained table and the related table checked by the OCL constraint. In total, the following situations can cause violation of the constraint:

1. inserting a new record into the constrained table with inappropriate number of referencing records in the related table;
2. inserting a new record into the referencing table, exceeding the upper limit of referencing records for the same referenced record;
3. updating a record in the constrained table and changing its PRIMARY KEY value, thus effectively changing the number of referencing records in the related table;
4. updating a record in the related table and changing its reference value to reference a different record in the constrained table, thus exceeding the upper limit of referencing records for the new referenced record or getting below the lower limit of referencing records for the old referenced record;
5. deleting a record from the related table, getting below the lower limit of records referencing the same record as the deleted record.

Although there is the FOREIGN KEY constraint defined in the referencing table, checking the existence of the referenced record, the situations above must be checked by triggers. In total, the following triggers are needed:

T1 – BEFORE INSERT OR UPDATE ON the constrained table: The trigger checks the number of records in the related table referencing the new value of the PRIMARY KEY of the inserted or updated record in the constrained table. The number of records must be between the lower and upper limit defined in the OCL constraint. The records in the related table are not affected by the checked DML operation, therefore the check can be executed BEFORE the operation. Moreover, only the records referencing the actually inserted/updated record need to be searched, therefore, the check can be executed FOR EACH ROW, using the variable new to access the new value of the PRIMARY KEY of the record.
SQL 7.28 SQL ISM with the CREATE TRIGGER statement for the trigger definition on the constrained table realizing a *special multiplicity constraint*

```sql
CREATE TRIGGER MUL_WORK_TO_SERIES_SERIES_ID
BEFORE INSERT OR UPDATE ON SERIES
FOR EACH ROW
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT COUNT(*) INTO l_count
  FROM WORK_TO_SERIES wts WHERE wts.SERIES_ID = :new.SERIES_ID;
  IF NOT(l_count > 2) THEN raise_application_error
    (-20101, 'OCL constraint MUL_WORK_TO_SERIES_SERIES_ID violated!');
  END IF;
END;
END;
```

**T2 – AFTER INSERT OR UPDATE OR DELETE ON the related table:** The trigger checks the number of records in the related table referencing the same record in the constrained table. As this check searches for records in the affected table, it cannot be executed FOR EACH ROW, but must run for the whole operation. Because of that, it is not possible to check just the affected record, but whole table must be checked and the number of referencing records for any record in the table in context of the original OCL constraint must be in the defined limits. On the other hand, in the case of the *mandatory multiplicity constraint*, the INSERT operation does not have to be checked, as inserting a new record cannot violate the lower limit restriction. Similarly, when there is no upper limit in the *special multiplicity constraint*, the INSERT operation does not have to be checked, and when there is no lower limit, the DELETE operation does not have to be checked.

Therefore, when transforming a *special multiplicity constraint* or a *mandatory multiplicity constraint* from the RDB PSM into its realization in the SQL ISM, a CREATE TRIGGER statement is generated for the trigger **T1** on the table in context of the original OCL constraint. Also, another CREATE TRIGGER statement is generated for the trigger **T2** on the other related table checked by the original OCL constraint.

The CREATE TRIGGER statement for the trigger **T1** is generated in the following form:

- The name of the original OCL constraint is used for the name of the trigger.
- The trigger is defined to be executed FOR EACH ROW BEFORE the INSERT and UPDATE operations on the table in context of the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable `l_count` is defined to hold the number of found records.
- In the body of the trigger:
7. Transformation of RDB PSM into ISM

SQL 7.29 SQL ISM with the CREATE TRIGGER statement for the trigger definition on the constrained table realizing a mandatory multiplicity constraint

```
CREATE TRIGGER MUL_AUTHORSHIP_WRITER_ID
BEFORE INSERT OR UPDATE ON WRITER
FOR EACH ROW
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT COUNT(1) INTO l_count FROM AUTHORSHIP a WHERE a.WRITER_ID = :new.WRITER_ID;
  IF l_count = 0 THEN raise_application_error (-20101, 'OCL_constraint_MUL_AUTHORSHIP_WRITER_ID_violated!');
END IF;
END;
```

- A SELECT statement is defined, selecting the `count` of records from the related table checked by the operations `allInstances` and the `count` (in the case of `special multiplicity constraint`) or the `exists` (in the case of `mandatory multiplicity constraint`) in the original OCL constraint with the WHERE clause realizing the condition of the `count` operation in the original OCL constraint. The result is stored into the defined variable `l_count`.

- Then, an IF statement is defined with the negated condition of the `l_count` value being in the specified limits defined in the original OCL constraint. In the case of the `mandatory multiplicity constraint`, this condition of the IF statement is comparison of the `l_count` variable with the value 0 (signalling that the required referencing record was not found). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

An example of the statement for the `special multiplicity constraint` shown in [Constraint 6.4](#) is shown in [SQL 7.28](#). The example of the statement for the `mandatory multiplicity constraint` is shown in [SQL 7.29](#) for the constraint shown in [Constraint 6.5](#).

Furthermore, for the trigger `T2` realizing the `special multiplicity constraint`, the CREATE TRIGGER statement is generated in the following form:

- The name of the trigger is generated by combining the name of the original OCL constraint and postfix `REL`.
- The trigger is defined to be executed AFTER the INSERT (unless no upper limit defined), UPDATE and DELETE (unless no lower limit defined) operations on the related table checked by the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable `l_count` is defined to hold the number of found records.
- In the body of the trigger:
7.3. Realization of Specific OCL Constraints

- A SELECT statement is defined, selecting the count of records from the table constrained by the original OCL constraint into the variable l_count without appropriate number of referencing records in the related table. Therefore, in the WHERE clause of this SELECT statement, the negated expression is defined, being composed of subqueries selecting the count of records in the related table having the reference value equal to the PRIMARY KEY value of the record in the outer table and comparing the count with the appropriate limits defined in the original OCL constraint.

- Then, an IF statement is defined with the condition of the l_count value being higher than the value 0 (meaning that for some record in the constrained table, an inappropriate number of referencing records was found). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

In the case of the mandatory multiplicity constraint, the CREATE TRIGGER statement for the trigger T2 is generated in the following form:

- The name of the trigger is generated by combining the name of the original OCL constraint and postfix _REL.
- The trigger is defined to be executed AFTER the UPDATE and DELETE operations on the related table checked by the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable l_count is defined to hold the number of found records.
- In the body of the trigger:

  - A SELECT statement is defined, selecting the count of records from the table constrained by the original OCL constraint into the variable l_count without an existing referencing record in the related table. Therefore, in the WHERE clause of this SELECT statement, the NOT EXISTS expression is defined with a subquery selecting records in the related table having the reference value equal to the PRIMARY KEY value of the record in the outer table.

  - Then, an IF statement is defined with the condition of the l_count value being higher than the value 0 (meaning that for some record in the constrained table, the referencing record was found). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

An example of the statement for the special multiplicity constraint shown in Constraint 6.4 is shown in SQL 7.30. The example of the statement for the mandatory multiplicity constraint is shown in SQL 7.31 for the constraint shown in Constraint 6.5.
7. Transformation of RDB PSM into ISM

SQL 7.30 SQL ISM with the CREATE TRIGGER statement for the trigger definition on the related table realizing the *special multiplicity constraint*

```sql
CREATE TRIGGER MUL_WORK_TO_SERIES_SERIES_ID_REL
AFTER UPDATE OR DELETE ON WORK_TO_SERIES
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT COUNT(1) INTO l_count FROM SERIES s WHERE NOT (  
    (SELECT COUNT(*) FROM WORK_TO_SERIES wts WHERE wts.SERIES_ID = s.SERIES_ID) >= 2);  
  IF l_count > 0 THEN raise_application_error (-20101, 'OCL constraint MUL_WORK_TO_SERIES_SERIES_ID_violated!');
  END IF;
END;
END;
```

SQL 7.31 SQL ISM with the CREATE TRIGGER statement for the trigger definition on the related table realizing the *mandatory multiplicity constraint*

```sql
CREATE TRIGGER MUL_AUTHORSHIP_WRITER_ID_REL
AFTER UPDATE OR DELETE ON BOOK_EDITION
DECLARE
  l_count NUMBER := 0;
BEGIN
  SELECT COUNT(1) INTO l_count FROM WRITER w WHERE  
    NOT EXISTS (SELECT 1 FROM AUTHORSHIP a WHERE a.WRITER_ID = w.WRITER_ID);  
  IF l_count > 0 THEN raise_application_error (-20101, 'OCL constraint MUL_AUTHORSHIP_WRITER_ID_violated!');
  END IF;
END;
END;
```

With this realization by database triggers, the consistency of the data in the database is ensured entirely. It is because the triggers are triggered for all operations which can cause violation of the *special multiplicity constraints* (including the *mandatory multiplicity constraints*) and all operations really causing the violation are rolled back.

### 7.3.4 Exclusivity Constraints

As discussed in [subsubsection 5.1.3.1](#) the transformation of *Phases* from the OntoUML PIM may lead into their realization by individual *Phase* classes in the UML PIM with a special *exclusivity constraint* defining the exclusivity of the relation of the *identity bearer* to only one of the *Phase* classes. Then, when transforming such UML PIM into the RDB PSM, the classes are transformed into database tables, the associations into references between the tables and the *exclusivity constraint* defined on the *identity bearer* class into an *exclusivity constraint* defined in context of the table representing the *identity bearer*, restricting the references between the tables. An example of such *exclusivity constraint* is shown in [Constraint 6.6](#).

When transforming the RDB PSM into its realization in an SQL ISM, the exclusivity...
7.3. Realization of Specific OCL Constraints

Constraints must also be transformed. In the following paragraphs, the individual possible realizations of the exclusivity constraints in the SQL ISM are discussed.

Views. As discussed in subsection 7.2.1, database views can be defined and used to access only valid data according to the constraints. Following this approach, also the exclusivity constraint can be realized by a database views. Such view queries only such records from the table in context of the OCL constraint (the table of the identity bearer), which is referenced only from one of the other checked tables (the Phase tables).

Therefore, when transforming the exclusivity constraint from the RDB PSM into the SQL ISM, a CREATE VIEW statement can be generated, defining a view with the following properties:

- The name of the original OCL constraint is used as the name of the view.
- In the SELECT clause of the SELECT statement, all columns from the table in context of the original OCL constraint are selected.
- In the FROM clause of the SELECT statement, records from the table in context of the original OCL constraint are queried.
- The WHERE clause realizes the actual OCL constraint. For each of the Boolean variables defined in the original OCL constraint by the def statements, a Boolean expression is generated. In each of these Boolean expressions, an EXISTS expression is defined with a subquery selecting referencing records from the table checked in the particular def section of the original OCL constraint, combined by logical conjunction (AND operator) with NOT EXISTS expressions with subqueries selecting records with the same referencing value from the other tables checked by the other def sections of the original OCL constraint. Finally, all these Boolean expressions are combined by logical disjunction (OR operator).

As the view definition meets the criteria for an updatable view, it can be also used for the DML operations. By using the view for the DML operations, it is not possible to insert a record into the constrained table without referencing records exactly in one of the related tables checked by the original OCL constraint. Also, the checked updatable view prevents updating a record in the table and changing its PRIMARY KEY value to a value, which is not referenced by exclusive records in the related tables. However, to allow inserting the record in one of the related tables before inserting the record it references, the FOREIGN KEY constraints in all these related tables must be defined as DEFERRABLE, otherwise there would be a conflict of the view condition and the FOREIGN KEY constraint.

An example of such view is shown in SQL 7.32, there the CREATE VIEW statement is defined for the exclusivity constraint defined in Constraint 6.6.

Even with such checked updatable view used for the DML operations on the constrained table, it is not entirely prevented to create invalid records in the constrained table, as it is possible to modify the data in the related tables after inserting the record in the constrained
SQL 7.32 SQL ISM with the CREATE VIEW statement for the view realizing an exclusivity constraint

```
CREATE VIEW EX_COPY_CONDITION AS
 SELECT * FROM COPY c WHERE
  (EXISTS (SELECT 1 FROM UNDAMAGED u WHERE c.COPY_ID = u.COPY_ID)
   AND NOT EXISTS (SELECT 1 FROM DAMAGED d WHERE c.COPY_ID = d.COPY_ID)
   AND NOT EXISTS (SELECT 1 FROM DESTROYED d WHERE c.COPY_ID = d.COPY_ID))
 OR (NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE c.COPY_ID = u.COPY_ID)
    AND EXISTS (SELECT 1 FROM DAMAGED d WHERE c.COPY_ID = d.COPY_ID)
    AND EXISTS (SELECT 1 FROM DESTROYED d WHERE c.COPY_ID = d.COPY_ID))
 WITH CHECK OPTION;
```

table. The INSERT operation can be checked by a special checked updatable view, selecting only records with the reference value not existing in the other related tables. The same view might be also used for the UPDATE operations to check, that when changing the reference value, the new reference value does not exist in the other related tables. However, the updated record must not be the last record 2 with the old reference value, otherwise the referenced record in the constrained table would be invalidated by losing the only exclusive referencing record. Unfortunately, this situation cannot be checked by a view, as it is not able to check the old values. The same also applies for the DELETE operation, where it is not possible to prevent removing the last referencing record.

Therefore, it is not possible to completely prevent violation of the constraints by database views. Moreover, it is still possible to access the data directly in the tables and create invalid data. Therefore, it is the responsibility of the application using the actual database to use the database views to access only the valid data, as well as manipulate with the records in the constrained table using the checked updatable view.

CHECK constraints. As discussed in subsection 7.2.2, CHECK constraints are another way to check the validity of the data in a table. Using a CHECK constraint, it is possible to define a table-level constraint which must be satisfied by all records in the table. As the exclusivity constraint de facto restricts the values of the PRIMARY KEY, which must be referenced only from one of the exclusive related tables, it seems reasonable to use the CHECK constraints to prevent values referenced from multiple tables and violating the constraint.

Therefore, when transforming the exclusivity constraint from the RDB PSM into the SQL ISM, an ALTER TABLE ADD CONSTRAINT statement can be generated to define the CHECK constraint in the following form:

- The ALTER TABLE statement alters the table in context of the original OCL constraint.

2The exclusivity constraint does not restrict the maximal number of referencing records from the exclusive related table, only the exclusive existence of records only in one of the tables.
7.3. Realization of Specific OCL Constraints

SQL 7.33 SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing an exclusivity constraint

```
ALTER TABLE COPY ADD CONSTRAINT EX_COPY_CONDITION CHECK ( COPY_ID IN ( SELECT COPY_ID FROM UNDAMAGED u WHERE NOT EXISTS (SELECT 1 FROM DAMAGED d WHERE d.COPY_ID = u.COPY_ID) AND NOT EXISTS (SELECT 1 FROM DESTROYED d WHERE d.COPY_ID = u.COPY_ID) UNION SELECT COPY_ID FROM DAMAGED d WHERE NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = d.COPY_ID) AND NOT EXISTS (SELECT 1 FROM DESTROYED e WHERE e.COPY_ID = d.COPY_ID) UNION SELECT COPY_ID FROM DESTROYED d WHERE NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = d.COPY_ID) AND NOT EXISTS (SELECT 1 FROM DAMAGED d WHERE d.COPY_ID = u.COPY_ID) ) )
```

◦ The ADD CONSTRAINT statement defines a CHECK constraint with the name of the original OCL constraint used as the name of the CHECK constraint.

◦ In the body of the CHECK constraint definition, the value of the PRIMARY KEY column is checked to be in the set of all valid reference values used in the related tables by the IN operator. This set is constructed by the UNION of selections from the individual related tables. Each of these selections selects only such records from the particular related table, whose reference value is not existing in any of the other related tables, defined by logical conjunction of NOT EXISTS expressions with subqueries selecting from the individual related tables, except the particular one from the outer SELECT, with the WHERE clause checking the equality of the reference values.

This CHECK constraint is able to check the valid PRIMARY KEY value for any record in the constrained table, when any DML operation is executed on the table. However, the check is based on existence of records in some of the related tables, which conflicts with the FOREIGN KEY constraints, which require the referenced record to exist. Therefore, the FOREIGN KEY constraint must be defined DEFERRABLE.

Although the check prevents violation of the constraint by DML operations on the constrained table, it can be also violated by DML operations on the related tables checked by the original exclusivity constraint by creating a record in another of the exclusively related tables by an INSERT or UPDATE operation or by removing the last record referencing an existing record in the constraint table by a DELETE or UPDATE operation. A special CHECK constraint could be defined to check the INSERT and UPDATE operations to prevent creating a conflicting referencing record, but it is not possible to prevent removing by the CHECK constraints.

Moreover, even though allowed by the SQL:1999 specification [39], the current database engines (including Oracle Database 12c) do not support subqueries in the CHECK constraints. Therefore, this solution, although valid according to the SQL specification, cannot be used until the database engines support the CHECK constraints with subqueries.
Triggers. As discussed in subsection 7.2.3, triggers are special procedures executed when certain DML operations are executed on a table. These procedures can contain complex logic and data operations. Therefore, they can be also used to realize complex constraints, including the exclusivity constraints.

As the exclusivity constraint restricts the existence of records in the related tables referencing a record in the table in context of the constraint, it can be violated by various DML operations executed on the constrained table as well as the individual related tables checked by the exclusivity constraint. In total, the following situations can cause violation of the constraint:

- inserting a new record into the constrained table without referencing records in exactly one of the exclusively related tables;
- updating a record in the constrained table and changing its PRIMARY KEY value to a value, which is not referenced from exactly one of the exclusively related tables;
- inserting a new record into some of the exclusively related tables, referencing a record referenced from another of the exclusively related tables, and thus making it referenced by records in multiple tables;
- updating a record in one of the related tables and changing its reference value to a value referencing a different record in the constrained table, which is referenced from another of the exclusively related tables (thus making it referenced from multiple related tables), or from a value referencing an existing record in the constraint table, which is not restricted by any other record in the particular related table (thus removing the last required referencing record);
- deleting the last record from the related table, which is referencing an existing record in the constraint table, and thus making it referenced by no related record.

Although there is the FOREIGN KEY constraint defined in the related tables, checking the existence of the referenced record in the constrained table, the situations mentioned above must be checked by triggers. However, as these triggers are executed immediately when executing the particular DML operations and they require having the referencing records first, which is conflicting with the FOREIGN KEY constraints, the FOREIGN KEY constraints must be defined DEFERRABLE. Then, in total, the following triggers are needed:

T1 – BEFORE INSERT OR UPDATE ON the constrained table: The trigger checks the existence of records in the related tables referencing the new value of the PRIMARY KEY of the inserted or updated record in the constrained table. A record can exist only in one of the exclusively related tables. The records in the related table are not affected by the checked DML operation, therefore the check can be executed BEFORE the operation. Moreover, only the records referencing the actually inserted/updated record need to be searched, therefore, the check can be
executed FOR EACH ROW, using the variable new to access the new value of the PRIMARY KEY.

**T2 – BEFORE INSERT ON each of the related tables:** The trigger checks that the record referenced by the new record in the related table is not referenced from any of the other related tables. Because it checks only records in the other tables using the value of the new record, it can be executed FOR EACH ROW. Moreover, it can be also executed before the actual INSERT operation is executed, as it does not affect the search.

**T3 – AFTER UPDATE OR DELETE ON the related table:** The trigger checks that the record referenced by the old reference value of the updated record in the related table does not exist or there are other records in the same table referencing it, as otherwise it would not be referenced from any of the exclusively related tables. Similarly, the same trigger can also check the DELETE operation able to cause the same problem of removing last referencing record. Furthermore, the trigger also checks, that the new reference value references a record in the constrained table not referenced from any of the other exclusively related tables. As the check needs to search the table affected by the operation, it cannot be executed FOR EACH ROW but for the whole statement at once. Because of that, it is not possible to check just the affected record, but the whole table must be checked AFTER the operation to detect any possible violation created by the operation.

Therefore, when transforming the *exclusivity constraint* from the RDB PSM into its realization in the SQL ISM, three separate CREATE TRIGGER statements are generated to define the individual triggers discussed above. The CREATE TRIGGER statement for the trigger T1 is generated in the following form (an example of such trigger for the *exclusivity constraint* defined in Constraint 6.6 is shown in SQL 7.34):

- The name of the original OCL constraint is used for the name of the trigger.
- The trigger is defined to be executed FOR EACH ROW BEFORE the INSERT and UPDATE operations on the table in context of the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable l.count is defined to hold the number of found records.
- In the body of the trigger:
  - A SELECT statement is defined, selecting count of records from the special table DUAL with the WHERE clause realizing the OCL constraint. The result is stored into the defined variable l.count. In the WHERE clause, for each of the Boolean variables defined in the original OCL constraint by the def statements, a Boolean expression is generated. In each of these Boolean expressions, an EXISTS expression is defined with a subquery selecting referencing records from
SQL 7.34 SQL ISM with the CREATE TRIGGER statement for the trigger definition on the constrained table realizing an *exclusivity constraint*

```sql
CREATE TRIGGER EX_COPY_CONDITION
BEFORE INSERT OR UPDATE ON COPY
DECLARE
    l_count NUMBER;
BEGIN
    SELECT COUNT(1) INTO l_count FROM DUAL WHERE (
        EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = :new.COPY_ID)
        AND NOT EXISTS (SELECT 1 FROM DAMAGED d WHERE d.COPY_ID = :new.COPY_ID)
        OR (NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = :new.COPY_ID)
            AND EXISTS (SELECT 1 FROM DAMAGED d WHERE d.COPY_ID = :new.COPY_ID))
    ) OR (
        NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = :new.COPY_ID)
        AND NOT EXISTS (SELECT 1 FROM DESTROYED d WHERE d.COPY_ID = :new.COPY_ID)
    ) OR (
        NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = :new.COPY_ID)
        AND NOT EXISTS (SELECT 1 FROM DAMAGED d WHERE d.COPY_ID = :new.COPY_ID)
        AND NOT EXISTS (SELECT 1 FROM DESTROYED d WHERE d.COPY_ID = :new.COPY_ID)
    );
    IF l_count = 0 THEN
        raise_application_error (-20101, 'OCL constraint EX_Copy_Context violated!');
    END IF;
END;
END;
```

The CREATE TRIGGER statements for the trigger T2 are generated for all tables checked by the *exists* operations in the original OCL constraint. Each of these statements is generated in the following form (an example of the trigger for the table DAMAGED realizing the exclusivity constraint defined in Constraint 6.6 is shown in SQL 7.35):

- The name of the original OCL constraint is used for the name of the trigger, combined with the name of the checked table and postfix _INS._
- The trigger is defined to be executed FOR EACH ROW BEFORE the INSERT operation on the related table checked in the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable l_count is defined to hold the number of found records.
7.3. Realization of Specific OCL Constraints

SQL 7.35 SQL ISM with the CREATE TRIGGER statement for the trigger definition for the INSERT operation on a related table realizing an exclusivity constraint

```sql
CREATE TRIGGER EX_COPY_CONDITION_DAMAGED_INS
BEFORE INSERT ON DAMAGED
FOR EACH ROW
DECLARE
  l_count NUMBER;
BEGIN
  SELECT COUNT(1) INTO l_count FROM DUAL WHERE (
    EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = :new.COPY_ID)
    OR EXISTS (SELECT 1 FROM DESTROYED d WHERE d.COPY_ID = :new.COPY_ID));
  IF l_count > 0 THEN
    RAISE_APPLICATION_ERROR (-20101, 'OCL_constraint.EX_Copy.Condition_violated!');
  END IF;
END;
```

- In the body of the trigger:
  - A SELECT statement is defined, selecting `count` of records from the special table `DUAL` with the WHERE clause selecting records with the same reference value from the other related tables. In this WHERE clause, for each of the Boolean variables defined in the original OCL constraint by the `def` statements not checking the particular related table on which the trigger is created, a Boolean expression is generated. In each of these Boolean expressions, an `EXISTS` expression is defined with a subquery selecting records from the table checked by the particular `def` section of the original OCL constraint with the same reference value as the new reference value of the inserted record. The results of these Boolean expressions for each of the `def` sections are combined by logical disjunction (`OR` operator). Finally, the result of this selection is stored into the defined variable `l_count`.
  - Then, an IF statement is defined with the condition of the `l_count` value being higher than 0 (meaning that an unexpected record was found in some of the other tables). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

Furthermore, for all tables checked by the `exists` operations in the original OCL constraint, the CREATE TRIGGER statements for the trigger \textit{T3} are generated in the following form (an example of the trigger for the table `DAMAGED` realizing the exclusivity constraint defined in [Constraint 6.6] is shown in SQL 7.36):

- The name of the original OCL constraint is used for the name of the trigger, combined with the name of the checked table and postfix `_UPD_DEL`.
- The trigger is defined to be executed AFTER the UPDATE operation on the related table checked in the original OCL constraint.
7. Transformation of RDB PSM into ISM

SQL 7.36 SQL ISM with the CREATE TRIGGER statement for the trigger definition for the UPDATE and DELETE operations on a related table realizing an *exclusivity constraint*

```sql
CREATE TRIGGER EX_COPY_CONDITION_DAMAGED_UPD_DEL
AFTER UPDATE OR DELETE ON DAMAGED
DECLARE
  l_count NUMBER;
BEGIN
  SELECT COUNT(1) INTO l_count FROM COPY c WHERE (  
    NOT ( (EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = c.COPY_ID) AND NOT EXISTS (SELECT 1 FROM DAMAGED d WHERE d.COPY_ID = c.COPY_ID))  
    OR (NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = c.COPY_ID) AND EXISTS (SELECT 1 FROM DAMAGED d WHERE d.COPY_ID = c.COPY_ID))  
    OR (NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = c.COPY_ID) AND NOT EXISTS (SELECT 1 FROM DESTROYED d WHERE d.COPY_ID = c.COPY_ID))  
    OR (NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = c.COPY_ID) AND NOT EXISTS (SELECT 1 FROM DESTROYED d WHERE d.COPY_ID = c.COPY_ID))  
    OR (NOT EXISTS (SELECT 1 FROM UNDAMAGED u WHERE u.COPY_ID = c.COPY_ID) AND EXISTS (SELECT 1 FROM DESTROYED d WHERE d.COPY_ID = c.COPY_ID))  
    ) ) ) ;
  IF l_count > 0 THEN
    raise_application_error (-20101, 'OCL_constraint_EX_COPY_CONDITION_violated!');
  END IF;
END;
END;
```

- In the DECLARE section of the trigger, a Number variable `l_count` is defined to hold the number of found records.

- In the body of the trigger:
  - A SELECT statement is defined, selecting `count` of records from the table in context of the original OCL constraint which does not have appropriate referencing records in exactly one of the exclusively related tables. In the WHERE clause of this SELECT statement, negated (by operator NOT) logical disjunction (OR operator) of Boolean expressions created for each of the Boolean variables defined in the original OCL constraint by the `def` statements is generated. In each of these Boolean expressions, an EXISTS expression with a subquery selecting records from the table checked by the particular `def` section of the original OCL constraint referencing the checked record from the outer selection is combined by logical conjunction (AND operator) with NOT EXISTS expressions with subqueries selecting records from the other related tables checked in the other `def` sections referencing the same record from the outer selection. Finally, the result of this selection is stored into the defined variable `l_count`.
  - Then, an IF statement is defined with the condition of the `l_count` value being higher than 0 (meaning that an unexpected record was found in the constrained table without exclusive referencing from exactly one of the related tables). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.
With this realization of the *exclusivity constraints* by database triggers, the consistency of the database is entirely ensured. Each operation able to cause the violation of the constraint is automatically checked and all operations causing the violation are rolled back. Therefore, it is not possible to create data violating the constraint by any of the DML operations.

**7.3.5 Enumeration Constraints**

As discussed in subsection 5.2.1 and subsection 5.2.2, certain patterns in the UML PIM such as rigid generalization sets and exclusive relations can be in certain situations optimized into a single attribute with a restricted set of valid values. Such set of valid values is expressed by an *enumeration constraint* such as shown in Constraint 5.2 for the optimized UML PIM shown in Figure 5.9. Then, when transforming such UML PIM into the RDB PSM, the class is transformed into a table and the *enumeration constraint* defined in context of the class is transformed into an *enumeration constraint* defined in context of the table realizing the class to restrict the values in the particular column of that table, such as shown in Constraint 6.7 for the transformed RDB PSM shown in Figure 6.11.

When transforming the RDB PSM into the SQL ISM, also these *enumeration constraints* must be transformed into their realization in the SQL ISM. In the following paragraphs, the details of the possible realizations of such constraints are discussed.

**View.** As discussed in subsection 7.2.1, database views can be used to limit the access only to the records meeting the condition of the transformed OCL constraint. In the case of the *enumeration constraint*, a view can be defined to select only such records from the table constrained by the constraint, which have a valid value in the constrained column as defined in the original OCL constraint, hiding all the records with the invalid values.

Therefore, when transforming such *enumeration constraint* from the RDB PSM into its realization in the SQL ISM, a CREATE VIEW statement can be generated to define a view with the following properties:

- The name of the original OCL constraint is used as the name of the view.
- In the SELECT clause of the SELECT statement, all columns from the table in context of the original OCL constraint are selected.
- In the FROM clause of the SELECT statement, records from the table in context of the original OCL constraint are queried.
- In the WHERE clause of the SELECT statement, the actual body of the original OCL constraint is realized, comparing the value in the constrained column with the values defined in the original OCL constraint using the equality operator (=) and logical disjunction (OR operator).
SQL 7.37 SQL ISM with the CREATE VIEW statement for the view definition realizing an *enumeration constraint*

```sql
CREATE VIEW EN_Person_Gender AS
SELECT * FROM PERSON p WHERE p.GENDER = 'Man' OR p.GENDER = 'Woman'
WITH CHECK OPTION;
```

As such view meets the condition of an *updatable view* (see subsection 7.2.1 for the notion of updatable views), it can be defined with the WITH CHECK CLAUSE and used also for DML operations on the data in the constrained table. Then, the DML operations performed on the view are translated to the actual underlying table and checked against the query of the view, rejecting any operations causing inaccessibility of the new record.

An example of such a view definition is shown in [SQL 7.37](#) where the CREATE VIEW statement is shown for the *enumeration constraint* defined in [Constraint 6.7](#).

As the *enumeration constraint* restricts only the values in a single column of a single table, only the INSERT and UPDATE operations can really cause violation of the constraint. Therefore, using the *checked updatable* view completely prevents creating invalid data in the table. However, it is still possible to use the original table directly, and thus cause violation of the *enumeration constraint*. Therefore, it is still the responsibility of the application to use only the *checked updatable* view to manipulate the data in the table, as well as access only the valid data in the table.

**CHECK constraint.** As discussed in subsection 7.2.2, CHECK constraints can be used to define table-level constraints for the data in the individual columns of the table, which must be satisfied by all records in the table at any time. This also applies for the *enumeration constraint*, which defines the valid values of the *discriminator* column for all records in the table.

Therefore, when transforming the *enumeration constraint* from the RDB PSM into its realization in the SQL ISM, an ALTER TABLE ADD CONSTRAINT statement can be generated with the following properties:

- The ALTER TABLE statement alters the table in context of the original OCL constraint.
- The ADD CONSTRAINT statement defines a CHECK constraint with the name of the original OCL constraint used as the name of the CHECK constraint.
- In the body of the CHECK constraint, the actual body of the original OCL constraint is realized, comparing the value in the constrained column with the values defined in the original OCL constraint using the equality operator (\(=\)) and logical disjunction (OR operator).

An example of such a CHECK constraint is shown in [SQL 7.38](#) where the ALTER TABLE ADD CONSTRAINT statement is shown for the *enumeration constraint* defined in [Constraint 6.7](#).
7.3. Realization of Specific OCL Constraints

SQL 7.38 SQL ISM with the ALTER TABLE ADD CONSTRAINT statement for the CHECK constraint definition realizing an *enumeration constraint*

```
ALTER TABLE PERSON ADD CONSTRAINT EN_Person_Gender CHECK (  
    GENDER = 'Man' OR GENDER = 'Woman'
);  
```

As this CHECK constraint is evaluated for each record affected by any INSERT and UPDATE operation performed on the constrained table, it prevents creating a record with an inappropriate value in the *discriminator* column. As the *enumeration constraint* restricts only values in this single column of a single table, no other operations can create data violating the constraint. Therefore, the realization of the enumeration constraint by the CHECK constraint ensures the data consistency according to this constraint entirely.

**Triggers.** As discussed in subsection 7.2.3, triggers are special database procedures, which are executed when certain DML operations are performed on certain table, depending on the definition of the trigger. These procedures can contain complex logic and data operations. Therefore, the triggers can be also used to check that the data of the affected records are meeting the conditions defined by various constraints.

In context of the *enumeration constraint*, a trigger can be used to check the data of the inserted or updated record in a table constrained by an enumeration constraint and prevent such operation, which violates the constraint. As the *enumeration constraint* restricts only a single column of a single table, it can be violated only by inserting a record with inappropriate value in the *discriminator* column or by updating a record to an inappropriate value. Therefore, a trigger is needed only for the INSERT and UPDATE operations. As in both cases, it is needed to check the new value in the *discriminator* column, both operations can be checked by the same trigger, resulting into only a single trigger needed to realize the *enumeration constraint*. Moreover, this trigger can be executed BEFORE the actual operation, as it is possible to detect the violation before actually inserting the new data into the table, and it can be executed FOR EACH ROW, allowing to check only the affected record instead of searching the whole table.

Therefore, when transforming the *enumeration constraint* from the RDB PSM into its realization in the SQL ISM, a CREATE TRIGGER statement can be generated defining a trigger with the following properties:

- The name of the original OCL constraint is used as the name of the trigger.

- The trigger is defined to be executed BEFORE the INSERT and UPDATE operations on the table in context of the original OCL constraint.

- In the DECLARE section of the trigger, a Boolean variable `l_valid` is defined to hold the result of the checking.

- In the body of the trigger:
SQL 7.39 SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing an "enumeration constraint"

```sql
CREATE TRIGGER EN_Person_Gender
BEFORE INSERT OR UPDATE ON PERSON
FOR EACH ROW
DECLARE
  l_valid BOOLEAN;
BEGIN
  l_valid := :new.GENDER = 'Man' OR :new.GENDER = 'Woman';
  IF NOT(l_valid) THEN
    raise_application_error (-20101, 'OCL constraint EN_Person_Gender violated!');
  END IF;
END;
```

- The value of the variable `l_valid` is set as logical disjunction (OR operator) of comparisons of the discriminator value checked by the original OCL constraint with the values defined in the constraint.
- Then, an IF statement is defined with the condition of negated value of variable `l_valid` (meaning that an invalid value of the discriminator column was detected). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the OCL constraint is violated.

An example of such a trigger definition is shown in SQL 7.39, where the ALTER TABLE ADD CONSTRAINT statement is shown for the "enumeration constraint" defined in Constraint 6.7. This trigger is executed for each record affected by any INSERT and UPDATE operation performed on the constrained table, throwing the application error whenever inappropriate values are set and causing the operation to roll back. As no other operations can cause violation of the enumeration constraint, this trigger ensures the data consistency according to this constraint entirely.

7.3.6 Immutability Constraints

As discussed in subsubsection 5.1.6.1 when transforming the OntoUML PIM into the UML PIM, the essentiality and inseparability of Part-Whole relations is transformed into immutable meta-properties of the particular end of the association realizing the Part-Whole relation in the UML PIM. The immutable meta-property is also used in the case of optimization of rigid generalization sets as discussed in subsection 5.2.1, as well as other situation, where the values of an attribute should not change. Then, as discussed in subsection 6.1.7 when transforming the UML PIM into the RDB PSM, these immutable meta-properties are transformed into immutability constraints – OCL postconditions. In total, two types of immutability constraints are used: one checking the UPDATE operation to prevent changes of the constrained column (being it a standard attribute or a reference), and one checking the DELETE operation to prevent removing a referencing record from an immutable set.
7.3. Realization of Specific OCL Constraints

of referencing records. An example of the *immutability constraint* checking the UPDATE operation is shown in Constraint 6.8, an example of the *immutability constraint* checking the DELETE operation is shown in Constraint 6.10.

When transforming the RDB PSM into the SQL ISM, even these special *immutability constraints* should be transformed. Unlike the other types of constraints discussed above, the *immutability constraints* do not define required structure of valid data of the records saved in the tables. Instead, these constraints restrict the possible operations executed on these tables and their effects on the affected records. Because of that, it is not possible to use the views and CHECK constraints for the realization of these constraints. The only way of realizing the *immutability constraints* is using database triggers.

Therefore, the individual OCL postconditions defined in the *immutability constraints* in the RDB PSM are transformed into CREATE TRIGGER statements, defining the triggers for the appropriate DML operation and checking the constrained values. As these triggers need to check the values of the actually affected record, it needs to be executed FOR EACH ROW separately. Moreover, although the OCL constraint is defined as postcondition to access the new and the old value, the trigger can be executed BEFORE the actual DML operation.

In the case of the *immutability constraint* checking the UPDATE operation, the CREATE TRIGGER statement is generated with the following properties:

- The name of the original OCL constraint is used as the name of the trigger.
- The trigger is defined to be executed FOR EACH ROW individually BEFORE the UPDATE operation on the table in context of the original OCL constraint.
- In the body of the trigger, an IF statement is defined with the condition of having the old and new values of the column checked by the original OCL constraint different using the :old and :new keywords and the inequality operator (<>). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the constraint is violated.

An example of such trigger definition is shown in SQL 7.40, where the CREATE TRIGGER statement is shown for the trigger realizing the *immutability constraint* defined in Constraint 6.8.

In the case of the *immutability constraint* checking the DELETE operation, the CREATE TRIGGER statement is generated with the following properties:

- The name of the original OCL constraint is used as the name of the trigger.
- The trigger is defined to be executed FOR EACH ROW individually BEFORE the DELETE operation on the table in context of the original OCL constraint.
- In the DECLARE section of the trigger, a Number variable l_count is defined to hold the number of found records.
SQL 7.40 SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing an *immutability constraint* checking the UPDATE operation

```sql
CREATE TRIGGER IM_BOOK_EDITION_BOOK_ID_UPD
BEFORE UPDATE ON BOOK_EDITION
FOR EACH ROW
BEGIN
  IF :old.BOOK_ID <> :new.BOOK_ID THEN raise_application_error
    (-20101, 'OCL_constraint.IM_BOOK_EDITION_BOOK_ID_UPD_violated!');
END IF;
END;
```

SQL 7.41 SQL ISM with the CREATE TRIGGER statement for the trigger definition realizing an *immutability constraint* checking the DELETE operation

```sql
CREATE TRIGGER IM_CLIENT_POSTAL_ADDRESS_ID_DEL
BEFORE DELETE ON CLIENT
FOR EACH ROW
DECLARE
  l_count NUMBER;
BEGIN
  SELECT COUNT(1) INTO l_count FROM DUAL WHERE (EXISTS
    (SELECT 1 FROM POSTAL_ADDRESS pa WHERE pa.POSTAL_ADDRESS_ID = :old.POSTAL_ADDRESS_ID));

  IF :old.POSTAL_ADDRESS_ID IS NOT NULL AND l_count > 0 THEN raise_application_error
    (-20101, 'OCL_constraint.IM_CLIENT_POSTAL_ADDRESS_ID_DEL_violated!');
END IF;
END;
```

- In the body of the trigger:
  - A SELECT statement is defined, selecting `count` of records from the special table `DUAL`. In the WHERE clause of this select statement, an EXISTS expression is defined with a subquery selecting records from the table checked by the `exists` operation in the original OCL constraint with the PRIMARY KEY value equal to the reference value of the actually deleted record.
  - Then, an IF statement is defined with the condition of not having the old reference value of the actually deleted record empty (checked by NOT NULL operator) and having the value of variable `l_count` higher than 0 (meaning that the referenced record exists and would lose a referencing record by the DELETE operation). In the THEN clause of the statement, an application error is raised with the code -20101 and the message, that the constraint is violated.

An example of such trigger definition is shown in [SQL 7.41](#), where the CREATE TRIGGER statement is shown for the trigger realizing the *immutability constraint* defined in [Constraint 6.10](#).

With this realization, it is not possible to violate the *immutability* constraints defined by the transformed OCL postcondition constraints. All the constrained operations are checked by a trigger, the data are validated and any operation causing the violation of
the respective constraint is rolled back by an application error thrown by the trigger. However, as discussed in subsection 6.1.7, the OCL postcondition constraints defined in the RDB PSM as the result of the transformation of the immutable meta-properties of the associations cannot completely restrict all the operations able to violate the immutability of the references. Therefore, even these triggers cannot guarantee the complete immutability of the references.

7.4 Discussion

In this section, we provide discussion to the approaches discussed in the previous sections of this chapter. We provide basic overview of the support of the constraints by various database engines and we discuss two other options of the realization of the constraints in the RDB PSM – by stored procedures and INSTEAD OF triggers. We also discuss the optimal selection from the proposed realizations for the individual types of constraints defined in the transformed RDB PSM based on the advantages and limitations of each option. We also address the problem of combining multiple constraints defined for a single database table, including the combination of the used realization options. Finally, we discuss the efficiency of the constraint implementations.

7.4.1 Database Systems

Although new types and architectures of database systems emerge to react on current requirements of practice, including NoSQL, NewSQL and Handoo database systems [6], the DB-Engines Ranking [7] shows that relational databases are still the most popular and common database systems used in practice. Therefore, in our approach, we focus on the transformation of the OntoUML conceptual models into the realization in relational databases. According to [7], the top 4 most popular database engines are Oracle (latest release Oracle database 12c [85]), MySQL (latest release MySQL 5.7 [90]), Microsoft SQL Server (latest release SQL Server 2016 [91]) and PostgreSQL (latest release PostgreSQL 9.6.2 [92]).

All of these database engines provide the basic functionality of a relational database management system, including tables, columns, various data types and constraints. As discussed in [93], the enforcement of integrity constraints is one of the most needed functionalities of advanced database systems, as these constraints specify the admissible data, as well as portions of the domain knowledge. However, different database engines provide different support for the needed integrity constraints. In Table 7.1, the overview of the support of the most important types of constraints in the most common database engines is shown.

All of the compared engines provide full support for NOT NULL constraints, as well as PRIMARY KEY and FOREIGN KEY constraints according to the SQL:1999 specification. The implementation of the UNIQUE constraints is different in MS SQL Server, where even
the NULL values are included in the *uniqueness*, therefore, only a single record can have a NULL value in the constrained column.

Regarding the CHECK constraints, all of the compared engines, with the exception of MySQL, support only the row level, checking only the values of the affected row – no subqueries and aggregation functions are allowed, although defined in the SQL:1999 specification. In MySQL, the CHECK constraints are correctly parsed, but ignored by all storage engines and not applied to the DML operations.

Regarding the execution time of the constraints, MySQL and MS SQL Server do not support deferred constraints’ checking. This is crucial limitation regarding the possible realizations of the constraints – especially the *special multiplicity constraints* and *exclusivity constraints*, which are based on checking the existence of referencing records in the other related tables before allowing to insert a record into the constrained table. Therefore, in the case of these database engines, only the views can be used to limit the access only to the valid data but it is not possible to prevent creation of invalid data directly in the database. Similarly in the case of generalization, it is more beneficial to use another solution than referencing tables and generalization set constraints checked by the triggers.

Regarding the views, all of the compared database engines support *updatable* views, including the WITH CHECK OPTION clause used to enforce the checking of the view condition when inserting or updating records using the view.

Also, all of the compared database engines support triggers, however, as shown in [Table 7.2](#), their capabilities differ. MySQL supports only triggers for individual trigger events, thus the same trigger used for an INSERT and UPDATE operation must be defined as two separate triggers. Regarding the activation time, MS SQL Server supports only triggers executed after the actual DML operation, which, fortunately, is not limiting in context of the constraints discussed in this thesis. Regarding the granularity of the trigger, MySQL supports only row-level triggers executed for each row, which causes *mutating table* errors when realizing the *special multiplicity constraints*. MS SQL Server, on the other hand, supports only statement-level triggers executed without reference to the

### Table 7.1: Overview of the support of constraints in the most popular RDBMSs

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Oracle</th>
<th>MySQL</th>
<th>SQL Server</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT NULL</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>y (N-)</td>
<td>y (N-)</td>
<td>y (N+)</td>
<td>y (N-)</td>
</tr>
<tr>
<td>PRIMARY KEY</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>CHECK</td>
<td>y (Row)</td>
<td>N</td>
<td>y (Row)</td>
<td>y (Row)</td>
</tr>
<tr>
<td>DEFERRABLE</td>
<td>y</td>
<td>N</td>
<td>N</td>
<td>y</td>
</tr>
<tr>
<td>Views</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Updatable views</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>WITH CHECK OPTION</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

...
7.4. Discussion

Table 7.2: Overview of the support of triggers in the most popular RDBMSs

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Oracle</th>
<th>MySQL</th>
<th>SQL Server</th>
<th>PostgreSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of events</td>
<td>n</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Activation time</td>
<td>B/A</td>
<td>B/A</td>
<td>A</td>
<td>B/A</td>
</tr>
<tr>
<td>Granularity</td>
<td>R/S</td>
<td>R</td>
<td>S</td>
<td>R/S</td>
</tr>
</tbody>
</table>

actual affected record, but there is a special table INSERTED containing the new records, which can be used to process the individual records one by one.

7.4.2 Stored Procedures and INSTEAD OF Triggers

As discussed in section 7.1, there are multiple options to realize various constraints defined in the RDB PSM. Simple column-based constraints can be realized by simple NOT NULL and UNIQUE constraints defined directly on the constrained column. More complicated constraints can be realized by database views (subsection 7.2.1), CHECK constraints (subsection 7.2.2) or triggers (subsection 7.2.3).

Beside these possible realizations, the constraints can be also realized by special stored procedures and INSTEAD OF triggers [52]. Similar to the triggers, the stored procedures can be defined to check the affected data before actually performing the DML operations causing the data changes. As such, the procedure can prevent causing violation of the constraints by throwing an application error and rolling the transaction back. However, unlike triggers which are executed automatically before or after the associated DML operation in the same transaction, the stored procedures must be called explicitly. Furthermore, the triggers are attached to the DML operation which is still executed before or after the trigger, while the stored procedure needs to actually perform the DML operation itself for instance, inserting the actual data into the table or updating the records) in the case that the data are valid, or the operation needs to be executed after the checking stored procedure, but in the same transaction. Additionally, while the row-level triggers can directly access the affected data using the :old and :new variables [3], the stored procedures are isolated and to execute the actual DML operation, all the data must be provided as parameters of the procedure.

When comparing the stored procedures with the CHECK constraints, the stored procedures are much more complicated. Similar to the triggers, the CHECK constraints are automatically evaluated for all DML operations affecting the data in the constrained table, with direct access to the affected data. Even comparing the stored procedures with the database views, they seem to be less efficient. Although the views often cannot entirely prevent creating invalid data in the underlying tables, even when updatable and defined with the WITH CHECK OPTION clause, they are capable of hiding all the invalid data and provide access only to the data meeting the defined constraints. It is true that the views must be used explicitly and any change in the structure of the underlying table
requires the update of the view definition, like the stored procedures do. But, the views
can be used in the very same way as the standard tables in the DML operations, while the
procedures often require a different way of invoking from the actual application.

Another option is using INSTEAD OF triggers. This kind of triggers is executed instead
of the attached DML operation. Therefore, similar to the procedures, such triggers can
check the data before the actual DML operation. However, also the actual DML operation
must be realized in the trigger. Therefore, it is more complicated than the standard triggers
used for the realization of the individual constraints discussed in section 7.3.

Moreover, both the procedures and INSTEAD OF triggers are sensitive on changes of
the columns of the individual tables, as they need to reflect the data to be inserted or
updated. Therefore, by adding a new column, the procedures and INSTEAD OF triggers
must be updated. In the case of the realizations proposed in section 7.2, they are defined
in general form, using only the PRIMARY KEY and reference values, and therefore they
are independent of almost any structural changes of the underlying tables.

For these reasons, we do not consider using stored procedures nor INSTEAD OF triggers
suitable for realizing the constraints defined in the RDB PSM and we suggest using the
views, CHECK constraints or triggers instead.

7.4.3 Optimal Realization for Individual Constraint Types

Various constraints defined in the RDB PSM can be realized by different ways in the
SQL ISM. In our approach, the constraints can be realized by database views, CHECK
constraints or triggers. In section 7.3, we discussed the principles of the individual possible
realizations of each of the constraints. However, each of the proposed realizations has
certain advantages and limitations in comparison to the other realization.

In Table 7.3, the overview of all the realizations discussed in section 7.3 is shown for
the individual constraints in the RDB PSM, presenting the real possibility of using such
realization and highlighting the optimal and recommended realization. In the following
sections, the individual constraints are discussed and compared, proposing the optimal
realization based on the advantages and limitations of the realizations.

7.4.3.1 Generalization Set Constraints

As discussed in subsection 6.1.3, generalization sets in the UML PIM can be transformed
into three different realizations in the RDB PSM – a single table, individual tables or
referencing tables. In the cases of the single table realization and the referencing tables
realization, special OCL generalization set constraints are defined to preserve the meta-
properties isDisjoint and isComplete of the generalization set. In subsection 7.3.1, the
possible realizations of these generalization set constraints are presented.

In general, database views can be used to limit the access only to the data in the
underlying tables satisfying the generalization constraints, CHECK constraints can be
used to check the validity of the data in the tables and triggers can be used to check the
data of the affected records for certain DML operations. However, for different realizations
Table 7.3: Overview of possible realizations of various constraints in a RDB PSM model

<table>
<thead>
<tr>
<th>Constraint</th>
<th>Views</th>
<th>CHECK</th>
<th>Triggers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalization set - Single table</td>
<td>P</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Generalization set - Referencing tables</td>
<td>P</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Distributed unique</td>
<td>P</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Special multiplicity</td>
<td>P</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Exclusivity</td>
<td>P</td>
<td>X</td>
<td>C</td>
</tr>
<tr>
<td>Enumeration</td>
<td>P</td>
<td>C*</td>
<td>C</td>
</tr>
<tr>
<td>Immutability</td>
<td>X</td>
<td>X</td>
<td>P</td>
</tr>
</tbody>
</table>

Legend:
X – cannot be used at all
P – can be used to partially ensure the constraint satisfaction
C – can be used to completely ensure the constraint satisfaction
* – applicable only in some RDBMSs
**bold** – recommended realization

of the generalization sets, these possible realizations have certain limitations and not all of them are suitable.

In the following paragraphs, the individual realizations of the generalization constraints are discussed in context of the realization of the generalization sets.

**Single Table.** In the case of the realization of the generalization set by a single table (subsubsection 6.1.3.1), the attributes of all the classes in the generalization set (including the superclass) are realized by columns in the same table. The generalization set constraint, then, restricts the valid combination of values in the columns representing the attributes of the individual classes depending on the value of the discriminator, which identifies the type of the represented instance – the classes it is instance of.

Thanks to this presence of all the columns in the same table, the database view realizing the generalization set constraint simply selects records with appropriate combination of values in the columns from the table. Moreover, this view is updatable. Therefore, it can be defined with the WITH CHECK OPTION clause and used for executing the DML operations, checking all the affected records against the condition of the view. As the constraint can be violated only by setting invalid combination of values in the individual columns of this single table, using the view is able to prevent violation of the constraint. However, this is not enforced entirely, as the data in the table can be altered directly and thus the constraint can be violated.

In contrast to that, the realization of the generalization set constraint by the CHECK constraint prevents creating invalid data in the table entirely. It is because the CHECK constraint defines the condition which must be satisfied by all records in the table, and thus it is checked for any affected record by any DML operation automatically. As in this
case all the data are stored in the same table and the constraint only checks values in the individual columns of the table, there is no problem with the realization of such CHECK constraint in the common contemporary database engines.

Finally, the realization of the generalization set constraint by triggers is also able to entirely prevent creating invalid data in the table. However, unlike the CHECK constraint which is automatically associated to all DML operations affecting the data in the constrained table, the triggers must be explicitly associated to the INSERT and UPDATE operations on the constrained table. Also, the actual definition of the trigger is much more complicated than the definition of the CHECK constraint: in the CHECK constraint, only the boolean expression is defined, while in the trigger, the full script of checking the data must be defined, including raising the actual application error.

Therefore, in conclusion, in the case of the single table realization of the generalization set, we recommend using the realization of the generalization set constraint by the CHECK constraint.

Referencing Tables. In the case of the generalization set realized by referencing tables (subsubsection 6.1.3.3), a separate table is used for each of the classes from the generalization set with references from the tables representing the subclasses to the table representing the superclass. Instances are then stored as records in multiple tables, depending on the actual combination of classes the instance instantiates – data of a single instance are distributed as records in multiple tables. In this case, the generalization set constraint restricts valid combination of records in the individual tables depending on the value of the discriminator column in the table representing the superclass, which identifies the actual type of the instance.

The realization of the generalization set constraints by database view is based on selecting records from the superclass table, which have appropriate records in the other tables referencing it according to the value of the discriminator. By joining this view, valid records in the subclass tables can also be found, hiding the invalid records away. Also this view is updatable and can be defined with the WITH CHECK OPTION clause and used for the DML operations on the superclass table. However, as discussed in subsubsection 7.3.1.2, although it is able to prevent some of the operations capable of violating the constraint, other DML operations cannot be checked by such view and may cause the violation. Therefore, the views cannot be used to entirely enforce the validity of the data, but only to query the valid data.

According to SQL:1999, the constraint might be also realized by a CHECK constraint restricting the possible values of the PRIMARY KEY in the table representing the superclass. However, the current database engines do not support subqueries in the CHECK constraints. Therefore, the CHECK constraints are not applicable.

Finally, realization by the triggers is also possible. Special triggers must be attached to the INSERT and UPDATE operations on the superclass table, as well as on all DML operations on each of the subclass tables, checking the affected records as well as the related records in the other tables realizing the generalization set. Although more complicated than
the views, the triggers are capable to entirely prevent the violation of the constraint, as they are associated to all possible operations causing the violation of the constraint.

Therefore, in conclusion, we recommend using the triggers for the realization of the generalization set constraints in the case of using referencing tables for the realization of the generalization sets, as it is the only viable option able to entirely ensure the data validity.

7.4.3.2 Distributed Unique Constraints

In the case of the generalization set realized by individual tables as discussed in section 6.1.3.2, a separate table is used for each of the combinations of classes from the generalization set, which can have instances, depending on the meta-properties of the generalization set. In each of the tables, only records representing instances of the same classes are stored. Thanks to that, there is no need for the special generalization set constraints. However, on the other hand, a special distributed unique constraint must be realized to ensure unique values of the attributes restricted by a uniqueness constraint, as the data of that attribute are distributed in all the tables representing the combinations of classes with the class of that attribute.

When realizing this distributed unique constraint by the database views, a view is created for each of the tables realizing the generalization set, selecting only records with values in the constrained column not existing in the other tables. These views are updatable, and therefore they can be defined with the WITH CHECK OPTION clause and used for the DML operations, checking the condition of the view for each affected record. As the constraint can be violated only by inserting a duplicate value by an INSERT or UPDATE operation, these views are able to prevent creation of such invalid data. However, it is not enforced entirely, as the tables can still be accessed directly and invalid data can be created in them.

The realization of the distributed unique constraints by CHECK constraints is possible according to the SQL:1999 specification, however, as discussed in subsection 7.2.2, the common contemporary database engines do not support subqueries in the CHECK constraints. As the distributed unique constraint in its principle is about checking values in other tables, the realization by the CHECK constraints is not applicable.

Still, the realization of the distributed unique constraints by the triggers is possible. These triggers must be attached to the INSERT and UPDATE operations on each of the tables realizing the generalization set, checking the affected records and the existence of the value of the constrained column in the other tables. Although more complicated than the definition of the views, the triggers are executed automatically for all the associated DML operations, and thus can automatically prevent creating invalid data in the underlying tables.

Therefore, in conclusion, we recommend using the realization by triggers for the realization of the distributed unique constraints in the case of the realization of the generalization set by individual tables.
7. Transformation of RDB PSM into ISM

7.4.3.3 Special Multiplicity Constraints

As discussed in subsection 6.1.4, in certain situations the multiplicities of associations defined in the UML PIM cannot be realized by the standard constraints in the RDB PSM. In such situations, special multiplicity constraints must be defined in the RDB PSM to check the appropriate number of referencing records in the tables according to the multiplicities of their relation.

As discussed in subsection 7.3.3, there are multiple options available to realize these special multiplicity constraints (including the mandatory multiplicity constraints) in the SQL ISM created by transforming the RDB PSM – by database views, CHECK constraints and triggers. However, each of them has its own advantages and limitations.

The realization of the special multiplicity constraints by database views is based on selecting only records from the target table with appropriate number of referencing records in the source table. Using this view, it is possible to retrieve only records not violating the special multiplicity constraint. To retrieve the valid records in the source table, the SELECT statement can be joined with the view to get only records referencing a record in the target table with appropriate number of referencing records. As this view is updatable, it can be defined with the WITH CHECK OPTION clause and used to execute DML operations on the target table. When executing such operations, it is possible to verify the appropriate number of referencing records for each affected record, otherwise the operation is rolled back. However, as discussed in subsection 7.3.3, the constraint can also be violated by DML operations on the source table. Although a special view can be defined on this table as well, still it cannot prevent all possible operations causing the violation of the constraint.

Furthermore, as discussed in subsection 7.3.3, the special multiplicity constraints can also be realized by CHECK constraints. However, although valid according to the SQL:1999 specification, the common contemporary database engines do not allow subqueries in the CHECK constraints. As the special multiplicity constraint is based on the references, the related data in multiple tables must be checked. Therefore, it is not possible to use this realization until the database engines start supporting subqueries in the CHECK constraints.

Finally, the special multiplicity constraints can also be realized by triggers. Special triggers are defined to check the appropriate number of referencing records in the source table for various DML operations on both the source table and the target table of the constrained reference. Although more complicated than the views, the triggers are able to check the data of any of the operations capable of causing violation of the special multiplicity constraint and rolling back the transaction in the case of violation of the constraint. Therefore, they are capable of enforcing the constraint satisfaction entirely.

In conclusion, the realization of the special multiplicity constraints by the triggers is the only possible option fully enforcing the satisfaction of the constraints. Therefore, we recommend using the triggers for the realization.
7.4.3.4 Exclusivity Constraints

As discussed in subsubsection 5.1.3.1, the Phases and their phase partition can be transformed from the OntoUML PIM into the UML PIM as a set of classes related to the class of the identity bearer with an exclusivity constraint. When transforming the exclusivity constraint into the RDB PSM, it is transformed to restrict the references in the tables representing the Phase classes. Then, as discussed in subsection 7.3.4, such exclusivity constraint can be realized by database views, CHECK constraints or triggers in the SQL ISM. Each of these possible realizations, however, have their advantages and limitations which are compared in the following paragraphs, recommending the optimal realization.

When realizing the exclusivity constraint by database views, a view is used to limit the access only to valid records in the constrained table, in which the records should be referenced only from one of the related tables. To access all valid record in the referencing tables, this view can be also used to join the selection from the referencing tables, hiding non-exclusive records. Moreover, as the view is updatable, it can be defined with the WITH CHECK OPTION clause and used for DML operations on the constrained table, preventing creation of records with non-exclusive referencing records in the other related tables. However, as discussed in subsection 7.3.4, even such view is not able to entirely prevent violation of the constraint, as the records in the other related tables can still be altered to create non-exclusive records or delete the exclusive referencing record. Moreover, the constrained tables can still be used directly, and therefore it requires explicit use of the view.

In the case of the realization by CHECK constraints, a CHECK constraint on the constrained table is able to prevent creation of an invalid record completely, by checking the PRIMARY KEY value referenced by an exclusive record in the other related tables. Other CHECK constraints are able to prevent creating of non-exclusive referencing records in the other tables. However, these CHECK constraints are unable to prevent deleting of the exclusive referencing record, which may cause violation of the constraint. Moreover, as the constraint is based on checking existence of records in other tables, a subquery is needed in the CHECK constraint definition, which is not supported in the contemporary common database engines. Therefore, this realization is actually not applicable.

In contrast to the previous options, the realization by triggers is able to completely ensure the satisfaction of the exclusivity constraint. Triggers are created to check every DML operation on all the tables (the constrained table, as well as the other related tables checked by the exclusivity constraint) and roll back any operation violating the constraint. Although the definition is more complicated than the views (selection from the tables combined with local variable and explicit application error raising), these checks are executed automatically with the associated DML operations on the associated tables, without any need of explicit use. On the other hand, the realization by the triggers slows down the DML operations, while not slowing down the queries, as all data in the tables are valid.

Thanks to the fact that only the triggers are capable of entirely ensure the satisfaction of the exclusivity constraint, we recommend using the triggers for its realization.
7. Transformation of RDB PSM into ISM

7.4.3.5 Enumeration Constraints

As discussed in subsection 6.1.6, *enumeration constraints* are used to restrict the value in a column of a table to a set of possible valid values. Therefore, when transforming such *enumeration constraint* into its realization in the SQL ISM, the values in the constrained column must be compared with the possible values. As discussed in subsection 7.3.5, this check can be realized by database views, CHECK constraints or triggers, each of them having certain advantages and limitations.

When realizing the *enumeration constraint* by database views, a view is used to access only records in the constrained table satisfying the constraint – thus having a valid value in the constrained column. As the view is *updatable*, it can be defined with the WITH CHECK OPTION clause and used for the DML operations on the table. As the constraint limits only values in a simple column of a single table without any need to check other tables, the constraint can only be violated by inserting an invalid value in the INSERT and UPDATE operations. Using the checked updatable view, it is possible to entirely prevent creation of such invalid records. However, the view must be used explicitly, as the original table can still be accessed by DML operations which may cause violation of the constraint.

When using the CHECK constraints for the realization of the *enumeration constraints*, a CHECK constraint is used for checking the value in the constrained column, comparing it with the possible valid values. This CHECK constraint is executed automatically for any INSERT or UPDATE operation executed on the constrained table, checking any affected record. As the CHECK constraints checks only values in the single table, there is no problem with its realization, and it is capable of entirely prevent violation of the constraint.

A trigger can also be used to check the affected record and prevent violation of the constraint by executing the trigger for any INSERT or UPDATE operation on the table. However, in contrast to the CHECK constraint, the definition of the trigger is more complicated and the application error must be explicitly raised to signal the violation of the constraint. Also, the trigger must be explicitly associated to the appropriate DML operations, while the CHECK constraint is associated automatically by defining it in context of the table.

In conclusion, all realizations are able to entirely ensure the satisfaction of the constraint. Since the views must be used explicitly and the triggers require explicit raising of the application error when it is violated, we recommend using the realization of the *enumeration constraints* by the CHECK constraints. In the case of unsupported CHECK constraints (such as in MySQL database as discussed in subsection 7.4.1), the realization by the checked updatable view is recommended.

7.4.3.6 Immutability Constraints

As discussed in subsection 6.1.7, the *immutability constraints* aim on preventing changes of relations between related instances or values of attributes after the initialization of the instances. In the RDB PSM, these constraints are defined as OCL postconditions, checking
the value of the constrained column or the records related by the reference before executing
the appropriate DML operation and after its execution.

As discussed in subsection 7.3.6 when transforming these immutability constraint into
their realization in the SQL ISM, only the realization by the triggers is possible. It is
because the constraints do not define validity of a static state of the database in certain
time, but rather the dynamic aspect of the data changes in the database. Therefore, only
constructs able to access both the old data and the new data are capable of doing such
checking.

However, even the triggers are not capable of preventing all possible operations caus-
ing the violation of the constraint. In the case of immutability constraint of the source
table of a reference, the constraint can be violated by inserting additional records into the
immutable set of referencing records. However, the trigger is not able to prevent it as it
cannot distinguish the initialization phase from the post-initialization phase, when the set
is already frozen.

In conclusion, the only way of realizing the immutability constraints is by using trig-
gers, although it cannot ensure the satisfaction of the constraints entirely. Therefore, we
recommend using them to prevent at least some of the operations causing the violation of
the constraints.

7.4.4 Combining Multiple Constraints

In section 7.3, the realization all the types of constraints in the RDB PSM created as
result of the transformation of an OntoUML PIM is discussed one by one. The proposed
realizations are presented isolated from any other constraints. However, in a real domain,
several constraints are often defined for the same object type, thus restricting data in a
single table by multiple constraints. For instance, in the running example shown in Fig-
ure B.4, the records in the COPY table are constrained by an enumeration constraint shown
in Constraint 3.26 in the attached Running Example (see Appendix B), restricting the
values of the CONDITION column to valid copy conditions defined in the OntoUML PIM as
Phases, and an exclusivity constraint shown in Constraint 3.42 in the attached Running
Example, restricting the referencing records in the related tables BORROWED, DISCARDED
and AVAILABLE to be exclusive.

When all constraints defined for a single table are realized by triggers, each constraint is
checked independently of the others. However, as all the triggers are automatically executed
before or after the actual constrained DML operation, but still in the same transaction, all
of them are checked and violation of any of them is capable of blocking the DML operation
and rolling the transaction back. Moreover, the order of the execution of the triggers is
irrelevant, as the triggers do not modify the data but only read and validate them.

Similar situation happens, when all the constraints defined for a single table are real-
ized by CHECK constraints. When executing the DML operations on the table, all the
constraints defined on the table are checked, including all CHECK constraints. Violating
any of them causes the operation to fail. Therefore, the order of checking of the CHECK
constraints does not matter.
Because of the automatic execution of the CHECK constraints and the triggers, also the situation when some of the constraints are realized by CHECK constraints and others are realized by individual triggers does not pose a problem. Simply, they can be executed in any order and violation of any of them blocks the whole DML operation.

When one of the constraints is realized by a simple database view, the constraint is not checked when the DML operations are executed, but all the other constraints realized by CHECK constraints or triggers are. Therefore, data violating only the single constraint realized by the database view can be present in the table. Then, the database view should be used for querying data from the table to filter out all the records violating that constraint.

When this single constraint is realized by a checked updatable view (an updatable view with the WITH CHECK OPTION clause), it should be used for the DML operations instead of executing the operations on the original table. When using it this way, the condition realizing the constraint is checked and the operation is blocked, when the condition is violated. Moreover, as the DML operation is automatically translated to the operations on the underlying tables, also all the CHECK constraints and triggers defined for the original table are executed. Therefore, all the constraints are checked and violation of any of them can be detected and prevented.

When more of the constraints defined for a single table are realized by database views using the approach discussed in subsection 7.2.1 and the individual subsections of section 7.3, each of the views checks only the condition of a single constraint. By querying data from one of the views, only data valid according to the constraint realized by the view are retrieved, but these data can contain records violating the other constraints realized by separate views. The same also applies for executing the DML operations on the view, as only the condition of the single realized constraint is checked. For the model shown in Figure B.4, the CREATE VIEW statement realizing the enumeration constraint is shown in SQL 7.42, and the CREATE VIEW statement realizing the exclusivity constraint is shown in SQL 7.43.

In such situations, an additional database view must be defined to combine all the views realizing the individual constraints. Such a view is constructed as a JOIN of records in all the views realizing the constraints defined on the same table. As all the views select records from the same table, the JOIN is NATURAL based on equality of the PRIMARY KEY values. The view combining the enumeration and exclusivity constraints on the table COPY is shown in SQL 7.44. This view can be used to retrieve only records meeting the conditions of both the constraints realized by the individual views.
SQL 7.43 SQL ISM with the CREATE VIEW statement for the exclusivity constraint on the table COPY

```
CREATE VIEW EX_Copy_Availability AS
SELECT * FROM COPY c
WHERE
  (EXISTS (SELECT 1 FROM AVAILABLE a WHERE a.COPY_ID = c.COPY_ID)
   AND NOT EXISTS (SELECT 1 FROM BORROWED b WHERE b.COPY_ID = c.COPY_ID)
   AND NOT EXISTS (SELECT 1 FROM DISCARDED d WHERE d.COPY_ID = c.COPY_ID))
OR (NOT EXISTS (SELECT 1 FROM AVAILABLE a WHERE a.COPY_ID = c.COPY_ID)
    AND EXISTS (SELECT 1 FROM BORROWED b WHERE b.COPY_ID = c.COPY_ID)
    AND NOT EXISTS (SELECT 1 FROM DISCARDED d WHERE d.COPY_ID = c.COPY_ID))
AND NOT EXISTS (SELECT 1 FROM DISCARDED d WHERE d.COPY_ID = c.COPY_ID)
WITH CHECK OPTION;
```

SQL 7.44 SQL ISM with the CREATE VIEW statement for the exclusivity constraint on the table COPY

```
CREATE VIEW COPY_Valid_view AS
SELECT * FROM GS_Copy_Condition NATURAL JOIN EX_Copy_Availability;
```

SQL 7.45 Error thrown by the Oracle database when inserting into the combined view

```
Error starting at line : 30 in command -
INSERT INTO COPY_Valid_view (COPY_ID, EDITION_ID, CONDITION)
  VALUES (10,10,'Damaged')
Error at Command Line : 30 Column : 30
Error report -
SQL Error: ORA-01733: virtual column not allowed here
01733. 00000 - "virtual column not allowed here"
*Cause: *
*Action: *
```

However, as this view selects from other views, although updatable, it cannot be defined with the WITH CHECK OPTION clause and used for the DML operations. When trying to do so, the Oracle database throws an exception that a virtual column (i.e. a column of the underlying view) cannot be updated (see SQL 7.45). Because of this, such combined view can be used only to retrieve the valid data from the database meeting the conditions of all the constraints, but the data consistency in the underlying tables cannot be guaranteed in such case. Therefore, we recommend to prefer other possible realizations of the constraints (i.e. by CHECK constraints and triggers), when the consistency of the data in the database is important and multiple constraints are defined for a single table.

### 7.4.5 Conflicting Constraints

As discussed in section 7.3, the satisfaction of the realized OCL constraints is checked whenever a DML operation capable of violating the constraint is performed, regardless the way it is realized – if it is a trigger, a CHECK constraint or a checked updatable view.
However, in certain cases, this check is mutually dependent on another check performed on the objects the constraint is dependent on. For instance, the special multiplicity constraint requires the records in the source table to exist before inserting the referenced record into the target table. However, there is the FOREIGN KEY constraint defined for the reference from the source table to the target table, checking that the referenced record exists in the target table.

As discussed in [subsection 7.1.2], this problem can be solved by defining the FOREIGN KEY constraint DEFERRABLE. Such deferred constraint is checked at the end of the transaction, and therefore data violating the FOREIGN KEY constraint can be inserted into the source table referencing a non-existent record, which is inserted later. When inserting the record into the target table, the referencing records in the source table are successfully found. Finally, when the transaction is finished, the FOREIGN KEY constraint is checked, rolling back the transaction if violated.

However, there can also be situations, when multiple OCL constraints are mutually dependent. An example of such situation is a circle in mandatory references such as shown in [Figure 7.1]. As discussed in [subsection 7.1.2], each of the references is restricted by a FOREIGN KEY. Additionally, as discussed in [subsection 6.1.4], each of the references is also constrained by a special multiplicity constraint, as the source table multiplicity is 1..*.

When realizing this special multiplicity constraint as discussed in [subsection 7.3.3], a view or a trigger is defined to check the existence of appropriate number of records in the source table referencing the affected record in the target table. As the realized special multiplicity constraint conflicts with the FOREIGN KEY constraint restricting that reference, the FOREIGN KEY constraint is defined DEFERRABLE to be checked at the end of the transaction. But, there is also conflict of the special multiplicity constraints: a record in the table TABLE_A requires a referencing record in the table TABLE_B, which in turn requires a referencing record in the table TABLE_C, which requires a referencing record in the table TABLE_A. As the realization of the special multiplicity constraints is executed immediately at the time of the DML operation, no record is possible to be inserted into any of the tables because of these mutually dependent constraints. Unfortunately, none of the possible realizations of the special multiplicity constraints can be deferred like the FOREIGN KEY constraints – neither the views, nor the triggers – with an exception of the CHECK constraint, which, unfortunately, cannot be really used because of the subquery.

Similar situations may also occur for other types of constraints, such as exclusivity constraints, generalization set constraints or combinations of these constraints (including the special multiplicity constraint).

Therefore, in such situations, either one of the constraints must be removed, or the whole model must be manually updated to remove the conflict between the constraints. This can be achieved for instance by choosing a different realization of the associations or a different realization of the constraints. However, this optimization of the model is dependent on the actual model and domain and must be performed manually by the analyst or designer after appropriate consideration.

One possible solution of the problem of mutually dependent special multiplicity constraints is shown in [Figure 7.2]. The reference between the tables TABLE_C and TABLE_A is
Figure 7.1: RDB PSM with the mutually dependent mandatory references

Figure 7.2: RDB PSM with updated model solving the problem of mutually dependent mandatory references

exchanged with an intermediating table TABLE_C_TO_TABLE_A with references to the both tables. Although both these references are restricted by the special multiplicity constraints, they are dependent only on the existence of a record in this intermediating table, which can be inserted without problems if the FOREIGN KEY constraints are deferred. Then, the records can be inserted in the following order: TABLE_C_TO_TABLE_A, TABLE_A, TABLE_B, TABLE_C.

In conclusion, when the model is created by the transformation of the RDB PSM into the SQL ISM and all the constraints are realized by the individual views, CHECK
7. Transformation of RDB PSM into ISM

constraints and triggers, the whole model needs to be verified and manually updated, when some conflicting constraints are identified.

7.4.6 Efficiency of the Constraints Realizations

As discussed in section 7.3, the individual OCL constraints derived during the transformation of the OntoUML PIM into the RDB PSM can be realized by several possible ways – by database views, CHECK constraints or triggers. Each of these variants has certain limitations discussed in subsection 7.4.3 and they affect the efficiency of the database operations.

As discussed in subsection 7.2.1, the views can be used to access only valid values in the database, filtering out all records violating the defined constraints. Therefore, such realization slows down the query operations by applying these filters. Moreover, when the view is defined with the WITH CHECK OPTION, it can be also used for the DML operations. In such a case, even the DML operations are slowed down as the condition of the view is checked for each affected row. This slowdown is especially substantial, when the constraint checks data in other table, as such table must be searched completely. On the other hand, this slowdown can be mitigated by defining indexes on the reference values to increase the efficiency of searching the referencing records. Also, the DML operations are checked by the updatable views and prevent inserting invalid data into the database, reducing the amount of data which must be searched and filtered in the query operations. However, as the views are not able to completely realize the individual constraints, they are really mostly used just for querying the data. Therefore, in context of the efficiency, using the views is beneficial in situations where a lot of DML operations is performed (which are not checked and slowed down) and the efficiency of the query operations is not crucial (as they are slowed down).

When using the realization by CHECK constraints as discussed in subsection 7.2.2, the condition of the constraint is checked for all DML operations affecting the records in the table, slowing them down. However, such CHECK constraints can completely prevent inserting any invalid data into the tables and therefore the query operations do not need to be checked and slowed down. Moreover, as the CHECK constraints in the common contemporary database engines do not support subqueries, only constraints checking data of a single record in a single table can be realized by CHECK constraints. Therefore, when using such CHECK constraints, the evaluation of the CHECK condition is not complicated and the slowdown is minimal.

Similarly, when using the triggers as discussed in subsection 7.2.3, all the DML operations able to cause the violation of the constraints are checked. These checks slow down the operations by querying data from the tables and checking their values. In the case of FOR EACH ROW triggers, these queries check only the affected rows in the particular table, eventually searching for referencing records in the related tables. However, in the case of statement-level triggers, the whole contents of the particular table checked by the trigger must be queried, as it is not possible to refer to the values of only the affected records, slowing down the operation even more substantially. On the other hand, prevent
creating invalid data. Therefore, the query operation do not need to be checked against the constraints and they are not slowed down.

As both the CHECK constraints and triggers affect the efficiency of the DML operations while not affecting the query operations, they are beneficial in the cases when the efficiency of the query operations is crucial in comparison to the frequency and speed requirements for the DML operations.

In [A5], we presented experiments comparing the efficiency of the proposed realizations of the special multiplicity constraints. It was proved, that the realizations of the constraints using the views and triggers slow down the database operations, but when using indexes, this slow down is not as marginal. The slowdown depends very much on the number of records in the tables, as the queries often need to query all records in certain tables, searching for the correct or incorrect referencing records. For small databases with hundreds of records, the queries are not expensive and the slowdown is low. However, the slowdown becomes substantial for databases with tens of thousands of records.

As the realization of the other OCL constraints derived from the transformed OntoUML PIM is based on the same approach, we suppose the results of similar experiments for the other types of constraints would bring similar results. However, the actual experiments are out of scope of this thesis and are subject of the future research.
Conclusions

In this thesis, we discussed the potential incorporation of OntoUML as a language for conceptual data modelling into the MDD approach to software engineering. This idea is motivated by the importance of high-quality and expressive conceptual modelling languages in the process of MDD and by the foundations of OntoUML, which is based on the Unified Foundational Ontology – an ontology based on psychology, cognitive science and mathematical theories of modal logic.

In chapter 1, the introduction to the topic was provided, explaining our motivation, defining our research questions and goals, providing the context of the most important related work and previous results, and specifying the contributions of the thesis.

In chapter 2, the background of our research was provided. Basic concepts of the MDD approach were introduced together with the most important concepts of the UML language related to data modelling. Then, basic concepts of the OCL language were introduced, since OCL is used to define additional constraints for the models in our approach. Also, a short overview of the tools for modelling in UML and OCL was provided, discussing their support for the model transformations and database script generation. Also, the overview of other related approaches was provided, comparing them to our approach.

In chapter 3, the OntoUML language was explained in detail together with its basic principles and concepts. The features of the language were discussed in context with the data modelling and illustrated on examples.

In chapter 4, our approach to the transformation of the OntoUML PIM is introduced. Also, the OntoUML PIM model of the running example used for demonstration of the approach was described in this chapter.

In chapter 5, the transformation of OntoUML PIM into UML PIM was discussed. In the individual sections of the chapter, individual OntoUML concepts were discussed and their transformation into an equivalent UML model was proposed.

In chapter 6, the transformation of the resulting UML PIM into RDB PSM was discussed. In the individual sections of the chapter, all the concepts and elements used in the model and created by the transformation of the initial OntoUML PIM were discussed and their transformation into an equivalent RDB PSM was proposed.
8. Conclusions

In chapter 7, the transformation of the resulting RDB PSM from the previous step into SQL ISM was discussed. We thoroughly considered the transformation of all the concepts and elements of the RDB PSM model created during the transformation of the UML PIM model from the previous step. For each of the concepts, the possible realizations preserving the semantics and constraints of the original model were proposed.

Finally, in this chapter, the conclusions of the thesis are provided, together with the proposition of future work.

8.1 Summary

In section 1.2, three research questions were identified in order to confirm the suitability of OntoUML for the conceptual data modelling in context of the MDD approach:

Q1 Is it possible to use OntoUML for conceptual data modelling?

Q2 Is it possible to transform an OntoUML model into a relational database model and generate SQL scripts from it?

Q3 Is it possible to realize all the implicit constraints defined by the types of universals and relations used in the OntoUML model in the relational database?

In order to answer question Q1, we provided a complex overview of the concepts of OntoUML and an example of a complex OntoUML PIM modelling the application data of a library information system. Based on that, we consider it proved, that OntoUML is suitable for creating conceptual data models in context with the MDD approach to software development.

In order to answer questions Q2 and Q3, we proposed a method for the transformation of OntoUML PIM into its realization in RDB, divided into three steps through intermediate UML PIM and RDB PSM. This division of the transformation allows reusing the existing know-how for the transformation of UML into RDB, reusing the first transformation step for transformations for other platforms and optional refactoring and optimizations of the models between the individual steps.

In the individual steps of the transformation, we considered all the concepts used in OntoUML, describing their gradual transformation into their proper realization in the database. We proved, that all these concepts can be transformed into SQL ISM (Q2). We also proved, that most of the constraints defined by the types of OntoUML universals and relations defined in OntoUML PIM can be realized in the database to preserve the constraints and prevent creation of invalid data in the database (Q3). We suggested three distinct possible realizations based on views, CHECK constraints and triggers, discussing their advantages and limitations and recommending the most suitable realization for each type of constraint.

Besides the basic way of transformation of the individual elements in each of the steps, various manual optimizations of the resulting model for its further simplification and higher
efficiency are also proposed. Since these optimizations are based on the knowledge of the domain and the targeted application, they cannot be done automatically and must be performed manually after proper consideration.

The proposed method is able to completely transform an OntoUML PIM model into its realization in a relational database, preserving all the semantics defined in the original OntoUML model, with the exception of the *immutability of parts and wholes* used in OntoUML and *immutability constraints* of associations used in UML.

Using the proposed optimizations, the model can be simplified to decrease the number of classes and tables realizing the initial OntoUML PIM model. On the other hand, the proper realization of all the implicit constraints determined by the kinds of universals represented by the type in the initial OntoUML PIM leads to a large number of additional database constructs checking the data, when performing the DML operations. As shown in the running example (see Appendix B), for the total count of 39 types and 45 relations defined in the initial OntoUML PIM model (see Table B.1), 68 special OCL constraints should be realized in the relational database (see Table B.3), which results in 119 triggers realizing the constraints (see Table B.4).

Based on these results, we suggest using the proposed transformation in the cases, where the database consistency is crucial. In the other cases, we suggest careful consideration of which constraints to realize and which are not necessary. Specifically, we suggest not realizing the *immutability constraints* of associations, since they cannot completely ensure the *immutability* of the associations.

### 8.2 Contributions of the Thesis

The contributions of the thesis can be summarized in the following list:

1. Complex overview of OntoUML and UFO-A concepts was provided – including the distinction of universals and individuals, together with the classification of the object types, moments and part-whole relations – thoroughly explained and illustrated on simple examples.

2. Example of a complex OntoUML conceptual model from the library domain was provided. This model contains most of the OntoUML and UFO-A concepts to demonstrate their gradual transformation.

3. Method of the transformation of OntoUML PIM into UML PIM was proposed, preserving the semantics defined by the OntoUML universal and relation types. The proposed transformation is illustrated on the running example of the Library OntoUML PIM.

4. Method of the transformation of UML PIM into RDB PSM was proposed, preserving all the semantics defined in the derived UML PIM model – focusing on the special multiplicity constraints and the constraints derived from the initial OntoUML model.
8. Conclusions

5. Method of the transformation of RDB PSM into ISM consisting of the SQL creation scripts was proposed, preserving the semantics derived from the OntoUML universal and relation types used in the initial OntoUML PIM, as well as the special multiplicity constraints. An overview of the available realizations of these constraints is provided and the advantages and disadvantages of each approach are discussed.

8.3 Future Work

Since the thesis is focused only on the explanation of the principles of OntoUML in context with conceptual data modelling used in the MDD approach to software engineering and the proposition of a method for the transformation of OntoUML PIM into its realization in the relational database, we suggest to investigate the following in the subsequent research:

- An experimental evaluation and comparison of the proposed realizations of the individual constraint types is necessary to assess the benefits of the solutions and usability of the proposed implementations in practice.

- Application of the proposed method of transformation to real-world systems would be highly beneficial to evaluate the suitability of OntoUML for conceptual data modelling, especially the comparison with the standard UML models in context of the size of the model, the size of the resulting database schema and the constraints checking.

- The implementation of a tool, implementing our proposed transformations, would be very useful to support the practical usage and evaluation on industrial case studies.

- Also, further investigation of other possible realizations or their combinations might be interesting, which may lead to improvements of our proposed method. For instance, defining a view querying the violating records and checking it by a CHECK constraint or a trigger.

- Since all the constraints and SQL scripts can be defined in many distinct but equivalent forms, we also consider interesting to evaluate these distinct forms of the OCL constraints and SQL scripts realizing the same constraints, searching for a less complex and more efficient form.
Bibliography


Reviewed Publications of the Author Relevant to the Thesis


The paper has been cited in: 195


The paper has been cited in:


The paper has been cited in:


Remaining Publications of the Author


The dissertation thesis is accompanied by a CD containing electronic version of the dissertation thesis, the Running Example, the dissertation thesis statement and all sources (figures, \LaTeX files, SQL files and the EA project. Below, the structure of the attached CD is shown.

```
readme.txt...........................................brief overview of the CD contents
   src
     phdthesis............sources of the dissertation thesis and the running example
       PhDThesis.tex...............main \LaTeX file of the dissertation thesis
       RunningExample.tex..........main \LaTeX file of the running example
       PhDThesisStatement.tex..main \LaTeX file of the dissertation thesis statement
       PhDThesisStatement.eap..main \LaTeX file of the dissertation thesis statement
     model
       model.eap.............................EA project with the model
     sql..................................SQL scripts of the running example
     text
       PhDThesis.pdf.....................text of the dissertation thesis
       PhDThesisStatement.pdf...........text of the dissertation thesis statement
       RunningExample.pdf...............text of the running example
```

Figure A.1: Structure of the attached CD
In this appendix, the complete models of the running example of the library information systems are shown with the applied transformations as proposed in the thesis.

In section 4.1 the OntoUML PIM model of the running example used for the illustration of our approach presented in the dissertation thesis is introduced. The complete running example, including all the consecutive models, can be found on the attached CD (see Appendix A). Below, the structure of the Running Example document is shown.

1 Library OntoUML PIM .................................................. 1
2 Library UML PIM ......................................................... 11
3 Library RDB PSM ......................................................... 21
4 Library SQL ISM ........................................................ 47

Figure B.1: Structure of the attached Running Example document
B. Running Example

B.1 Library OntoUML PIM

In Figure B.2, the complete OntoUML PIM of the running example is shown. Detail description of the model is described in section 4.1.

In Table B.1, the overview of the counts of various kinds of types and relations defined in the OntoUML PIM model is shown.

Table B.1: Library OntoUML PIM - counts of the elements of the model

<table>
<thead>
<tr>
<th>Type</th>
<th>Count</th>
<th>Relation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>4</td>
<td>generalization</td>
<td>31</td>
</tr>
<tr>
<td>Subkind</td>
<td>4</td>
<td>— generalization sets</td>
<td>10</td>
</tr>
<tr>
<td>Role</td>
<td>9</td>
<td>mediation</td>
<td>8</td>
</tr>
<tr>
<td>Phase</td>
<td>8</td>
<td>formal</td>
<td>1</td>
</tr>
<tr>
<td>Category</td>
<td>3</td>
<td>characterization</td>
<td>4</td>
</tr>
<tr>
<td>RoleMixin</td>
<td>1</td>
<td>memberOf</td>
<td>1</td>
</tr>
<tr>
<td>Mixin</td>
<td>1</td>
<td>subcollectiveOf</td>
<td>0</td>
</tr>
<tr>
<td>Mode</td>
<td>3</td>
<td>containment</td>
<td>0</td>
</tr>
<tr>
<td>Quality</td>
<td>1</td>
<td>subquantityOf</td>
<td>0</td>
</tr>
<tr>
<td>Relator</td>
<td>4</td>
<td>componentOf</td>
<td>0</td>
</tr>
<tr>
<td>Collective</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quantity</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>39</td>
<td>Total</td>
<td>45</td>
</tr>
</tbody>
</table>
Figure B.2: OntoUML PIM of the Library
B.2 Library UML PIM

In Figure B.3, the complete UML PIM of the running example, created by applying the proposed transformations of the initial OntoUML PIM, is shown. The detailed description of this model can be found in Chapter 2 of the attached Running Example.

In Table B.2, the overview of the numbers of various elements used in the Library UML PIM model is shown.

<table>
<thead>
<tr>
<th>Classes</th>
<th>Count</th>
<th>Relations</th>
<th>Count</th>
<th>OCL constraints</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>classes</td>
<td>30</td>
<td>generalization sets</td>
<td>7</td>
<td>exclusivity</td>
<td>1</td>
</tr>
<tr>
<td>im. attribute</td>
<td>1</td>
<td>generalization</td>
<td>14</td>
<td>enumeration</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>association</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>im. association</td>
<td>20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure B.3: UML PIM of the Library
B. Running Example

B.3 Library RDB PSM

In Figure B.4, the complete RDB PSM of the running example, created by applying the proposed transformations of the UML PIM shown in Figure B.3, is shown. The detailed description of this model can be found in chapter 3 of the attached Running Example.

In Table B.3, the overview of the numbers of various elements used in the Library RDB PSM model is shown.

Table B.3: Library RDB PSM - counts of elements of the model

<table>
<thead>
<tr>
<th>Element</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>31</td>
</tr>
<tr>
<td>reference</td>
<td>37</td>
</tr>
<tr>
<td>constraints</td>
<td>68</td>
</tr>
<tr>
<td>- generalization set</td>
<td>7</td>
</tr>
<tr>
<td>- special multiplicity</td>
<td>1</td>
</tr>
<tr>
<td>- mandatory multiplicity</td>
<td>11</td>
</tr>
<tr>
<td>- enumeration</td>
<td>2</td>
</tr>
<tr>
<td>- exclusivity</td>
<td>1</td>
</tr>
<tr>
<td>- immutable - update</td>
<td>32</td>
</tr>
<tr>
<td>- immutable - delete</td>
<td>14</td>
</tr>
</tbody>
</table>
Figure B.4: RDB PSM of the Library
B. Running Example

B.4 Library RDB PSM

The SQL ISM consists of the SQL scripts used for the creation of the database schema and the individual database objects – tables, constraints, views and triggers. The individual SQL scripts can be found in the electronic attachments. In Table B.4, the overview of the numbers of various elements used in the Library SQL ISM model is shown.

Table B.4: Library SQL ISM - counts of elements of the model

<table>
<thead>
<tr>
<th>Database construct</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE</td>
<td>31</td>
</tr>
<tr>
<td>FOREIGN KEY</td>
<td>37</td>
</tr>
<tr>
<td>VIEW</td>
<td>0</td>
</tr>
<tr>
<td>CHECK</td>
<td>3</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>119</td>
</tr>
<tr>
<td>- BEFORE</td>
<td>104</td>
</tr>
<tr>
<td>- AFTER</td>
<td>15</td>
</tr>
</tbody>
</table>
Additional Examples

In this appendix, additional examples for the illustration of our approach are provided.

C.1 Transformation of Part-Whole Relations from OntoUML PIM into UML PIM

In this section, additional examples of the UML PIM models created by the transformation of OntoUML PIM with various types of aggregates are shown as discussed in subsection 5.1.6.

In Figure C.1, the UML PIM model is shown with the transformed functional complex Car and its components Engine, Airbags and Chassis from the OntoUML PIM model shown in Figure 3.11.

In Figure C.2, the UML PIM model is shown with the transformed Collective InternationalBody, its subcollection SportClub, its subcollection DefensePlayers and its members Person. The original OntoUML PIM model of the Collectives is shown in Figure 3.13.

![Diagram of UML PIM model]

Figure C.1: UML PIM with transformed functional whole and its components
In Figure C.3 the UML PIM model is shown with the transformed Quantity Beer of certain Brand contained in Bottle and its subquantity Alcohol. The original OntoUML PIM model of the Quantities is shown in Figure 3.14.
C.2 Transformation of Generalization from UML PIM into RDB PSM

C.2. Transformation of Generalization from UML PIM into RDB PSM

Constraint C.1 Example of the OCL invariants for a \{overlapping, complete\} generalization set realized by a single table

context SUBJECT inv GS_Subject_Type:

def Person_Instance: Boolean = self.DISCRIMINATOR = 'Person'
AND self.PERSON_LAST_NAME <> OclVoid AND self.PERSON_GENDER <> OclVoid
AND self.LEGAL_ENTITY_TITLE = OclVoid AND self.LEGAL_ENTITY_VAT = OclVoid

def LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'LegalEntity'
AND self.PERSON_LAST_NAME = OclVoid AND self.PERSON_GENDER = OclVoid
AND self.LEGAL_ENTITY_TITLE <> OclVoid AND self.LEGAL_ENTITY_VAT <> OclVoid

def Person_LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'Person_LegalEntity'
AND self.PERSON_LAST_NAME <> OclVoid AND self.PERSON_GENDER <> OclVoid
AND self.LEGAL_ENTITY_TITLE <> OclVoid AND self.LEGAL_ENTITY_VAT <> OclVoid

Person_Instance OR LegalEntity_Instance OR Person_LegalEntity_Instance

Constraint C.2 Example of the OCL invariants for a \{disjoint, incomplete\} generalization set realized by a single table

context SUBJECT inv GS_Subject_Type:

def Subject_Instance: Boolean = self.DISCRIMINATOR = 'Subject'
AND self.PERSON_LAST_NAME = OclVoid
AND self.PERSON_GENDER = OclVoid
AND self.LEGAL_ENTITY_TITLE = OclVoid
AND self.LEGAL_ENTITY_VAT = OclVoid

def Person_Instance: Boolean = self.DISCRIMINATOR = 'Person'
AND self.PERSON_LAST_NAME <> OclVoid
AND self.PERSON_GENDER <> OclVoid
AND self.LEGAL_ENTITY_TITLE = OclVoid
AND self.LEGAL_ENTITY_VAT = OclVoid

def LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'LegalEntity'
AND self.PERSON_LAST_NAME = OclVoid
AND self.PERSON_GENDER = OclVoid
AND self.LEGAL_ENTITY_TITLE <> OclVoid
AND self.LEGAL_ENTITY_VAT <> OclVoid

Subject_Instance OR Person_Instance OR LegalEntity_Instance

C.2 Transformation of Generalization from UML PIM into RDB PSM

C.2.1 Single Table Realization

In this section, additional variants of the generalization set constraint are shown for the other combinations of the generalization set meta-properties. In Constraint C.1, the constraint for the \{overlapping, complete\} generalization set is shown. In Constraint C.2, the constraint for the \{disjoint, incomplete\} generalization set is shown. In Constraint C.3, the constraint for the \{overlapping, incomplete\} generalization set is shown.
C. Additional Examples

**Constraint C.3** Example of the OCL invariants for a \{overlapping,incomplete\} generalization set realized by a single table

```
context SUBJECT inv GS_Subject_Type:

def CorrespondenceSubject_Instance: Boolean = self.DISCRIMINATOR = 'Subject'
  AND self.PERSON_LAST_NAME = OclVoid AND self.PERSON_GENDER = OclVoid
  AND self.LEGAL_ENTITY_TITLE = OclVoid AND self.LEGAL_ENTITY_VAT = OclVoid

def Person_Instance: Boolean = self.DISCRIMINATOR = 'Person'
  AND self.PERSON_LAST_NAME <> OclVoid AND self.PERSON_GENDER <> OclVoid
  AND self.LEGAL_ENTITY_TITLE = OclVoid AND self.LEGAL_ENTITY_VAT = OclVoid

def LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'LegalEntity'
  AND self.PERSON_LAST_NAME = OclVoid AND self.PERSON_GENDER = OclVoid
  AND self.LEGAL_ENTITY_TITLE <> OclVoid AND self.LEGAL_ENTITY_VAT <> Ocl Void

def Person_LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'Person_LegalEntity'
  AND self.PERSON_LAST_NAME <> OclVoid AND self.PERSON_GENDER <> OclVoid
  AND self.LEGAL_ENTITY_TITLE <> OclVoid AND self.LEGAL_ENTITY_VAT <> OclVoid

Subject_Instance OR Person_Instance OR LegalEntity_Instance
OR Person_LegalEntity_Instance
```
C.2. Transformation of Generalization from UML PIM into RDB PSM

Constraint C.4 Example of the OCL invariants for a \{complete,overlapping\} generalization set realized by related tables

context SUBJECT inv GS_Subject_Type:

def Person_Instance : Boolean = self.DISCRIMINATOR = 'Person'
AND PERSON.allInstances()->exists(p|p.PERSON_ID = self.SUBJECT_ID)
AND NOT (LEGALENTITY.allInstances()->exists(le|le.LEGALENTITY_ID = self.SUBJECT_ID))
def LegalEntity_Instance : Boolean = self.DISCRIMINATOR = 'LegalEntity'
AND PERSON.allInstances()->exists(p|p.PERSON_ID = self.SUBJECT_ID)
AND LEGALENTITY.allInstances()->exists(le|le.LEGALENTITY_ID = self.SUBJECT_ID)
def Person_LegalEntity_Instance : Boolean = self.DISCRIMINATOR = 'Person_LegalEntity'
AND PERSON.allInstances()->exists(p|p.PERSON_ID = self.SUBJECT_ID)
AND LEGALENTITY.allInstances()->exists(le|le.LEGALENTITY_ID = self.SUBJECT_ID)
Subject_Instance OR Person_Instance OR LegalEntity_Instance

Constraint C.5 Example of the OCL invariants for a \{incomplete,disjoint\} generalization set realized by related tables

context SUBJECT inv GS_Subject_Type:

def Person_Instance : Boolean = self.DISCRIMINATOR = 'Person'
AND PERSON.allInstances()->exists(p|p.PERSON_ID = self.SUBJECT_ID)
AND NOT (LEGALENTITY.allInstances()->exists(le|le.LEGALENTITY_ID = self.SUBJECT_ID))
def LegalEntity_Instance : Boolean = self.DISCRIMINATOR = 'LegalEntity'
AND PERSON.allInstances()->exists(p|p.PERSON_ID = self.SUBJECT_ID)
AND LEGALENTITY.allInstances()->exists(le|le.LEGALENTITY_ID = self.SUBJECT_ID)
def Person_LegalEntity_Instance : Boolean = self.DISCRIMINATOR = 'Person_LegalEntity'
AND PERSON.allInstances()->exists(p|p.PERSON_ID = self.SUBJECT_ID)
AND LEGALENTITY.allInstances()->exists(le|le.LEGALENTITY_ID = self.SUBJECT_ID)
Subject_Instance OR Person_Instance OR LegalEntity_Instance

C.2.2 Referencing Tables Realization

In this section, additional variants of the generalization set constraint are shown for the other combinations of the generalization set meta-properties. In Constraint C.4, the constraint for the \{overlapping,complete\} generalization set is shown. In Constraint C.5, the constraint for the \{disjoint,incomplete\} generalization set is shown. In Constraint C.6, the constraint for the \{overlapping,incomplete\} generalization set is shown.
Constraint C.6 Example of the OCL invariants for a \{incomplete,overlapping\} generalization set realized by related tables

context SUBJECT inv GS_Subject_Type:

def Subject_Instance: Boolean = self.DISCRIMINATOR = 'Subject'
  AND NOT (PERSON.allInstances() -> exists(p | p.PERSON_ID = self.SUBJECT_ID))
  AND NOT (LEGAL_ENTITY.allInstances() -> exists(le | le.LEGAL_ENTITY_ID = self.SUBJECT_ID))
def Person_Instance: Boolean = self.DISCRIMINATOR = 'Person'
  AND PERSON.allInstances() -> exists(p | p.PERSON_ID = self.SUBJECT_ID)
  AND NOT (LEGAL_ENTITY.allInstances() -> exists(le | le.LEGAL_ENTITY_ID = self.SUBJECT_ID))
def LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'LegalEntity'
  AND LEGAL_ENTITY.allInstances() -> exists(le | le.LEGAL_ENTITY_ID = self.SUBJECT_ID)
def Person_LegalEntity_Instance: Boolean = self.DISCRIMINATOR = 'Person_LegalEntity'
  AND PERSON.allInstances() -> exists(p | p.PERSON_ID = self.SUBJECT_ID)
  AND LEGAL_ENTITY.allInstances() -> exists(le | le.LEGAL_ENTITY_ID = self.SUBJECT_ID)

Subject_Instance OR Person_Instance OR LegalEntity_Instance
OR Person_LegalEntity_Instance