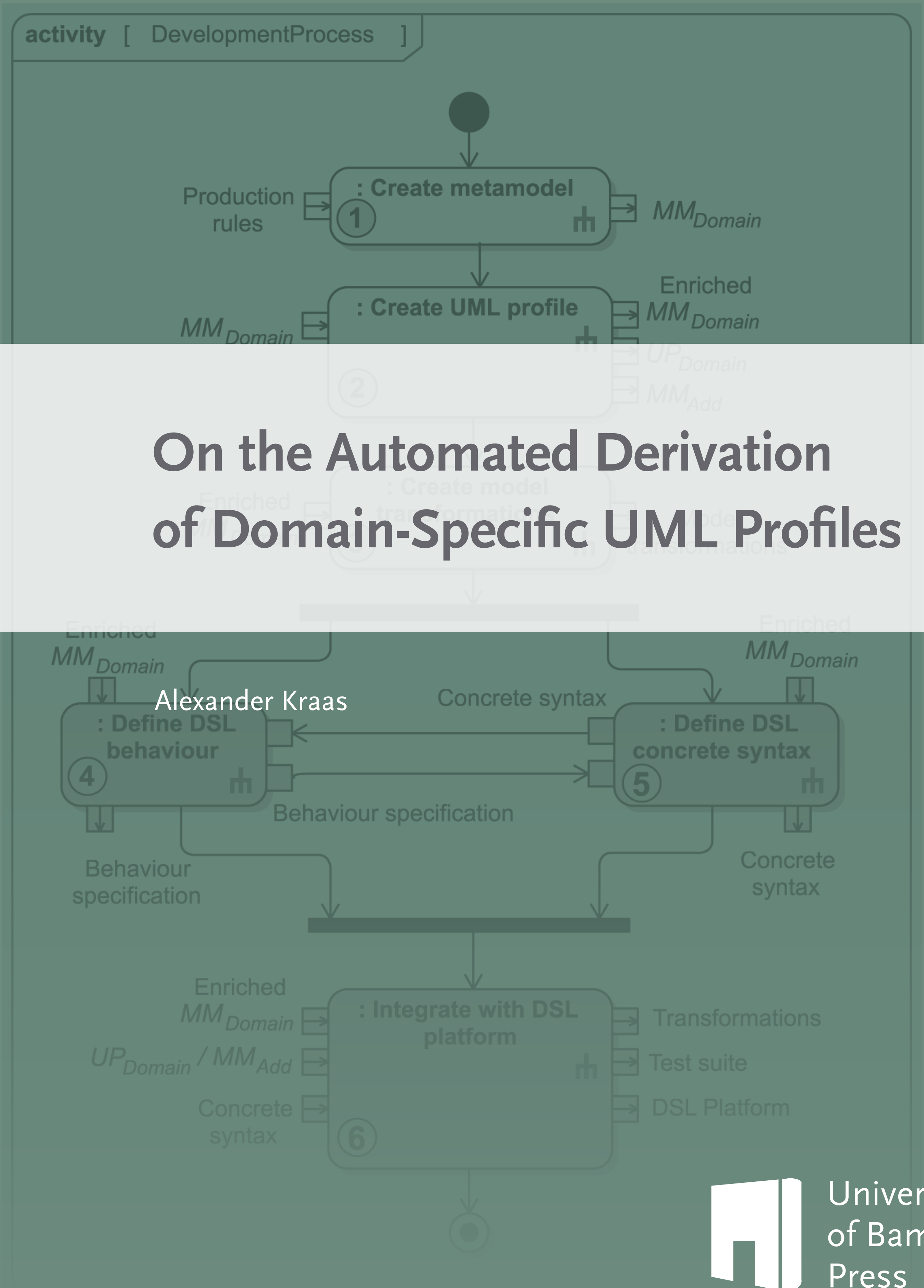
37
Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg



On the Automated Derivation of Domain-Specific UML Profiles

Alexander Kraas

University of Bamberg Press

37 Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Schriften aus der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg

Contributions of the Faculty Information Systems and Applied Computer Sciences of the Otto-Friedrich-University Bamberg

Band 37

University of Bamberg Press
2019

On the Automated Derivation of Domain-Specific UML Profiles

Alexander Kraas

University of Bamberg Press
2019

Bibliographische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie; detaillierte bibliographische Informationen sind im Internet über http://dnb.d-nb.de/ abrufbar.

Diese Arbeit hat der Fakultät Wirtschaftsinformatik und Angewandte Informatik der Otto-Friedrich-Universität Bamberg als Dissertation vorgelegen.
1. Gutachter: Prof. Dr. Gerald Lüttgen, Otto-Friedrich-University Bamberg, Germany
2. Gutachter: Prof. Dr. Richard Paige, McMaster University, Canada
Tag der mündlichen Prüfung: 13.05.2019

Herstellung und Druck: docupoint, Magdeburg
Umschlaggestaltung: University of Bamberg Press

ISSN: 1867-7401
ISBN: 978-3-86309-670-0 (Druckausgabe)
eISBN: 978-3-86309-671-7 (Online-Ausgabe)
URN: urn:nbn:de:bvb:473-opus4-548916
DOI: http://dx.doi.org/10.20378/irbo-54891

Abstract

Similar to general-purpose languages, *domain-specific languages (DSL)* can be developed based on grammar formalisms, the model-driven engineering (MDE) is also becoming more and more important for the development of DSLs. On the one hand, metamodels can be used to define the syntax and semantics of DSLs. On the other hand, a DSL can be realized by adapting the *Unified Modeling Language (UML)* via the profiling mechanism, i.e., by defining a UML profile. For example, metamodels for a DSL can be created with the language concepts provided by the *Meta Object Facility (MOF)*, distinguishing between the *Essential MOF (EMOF)* and the *Complete MOF (CMOF)*. The latter variant is based on the EMOF but provides additional language concepts. A higher degree of abstraction and reuse of existing metamodels can be achieved by employing the language concepts provided by CMOF, which can be advantageous for the creation of more complex DSLs.

Apart from DSLs proposed in the literature, a model-based specification of DSLs becomes increasingly important for standardization. Both metamodels and UML profiles are often provided for standardized DSLs. The mappings of metamodels to UML profiles are usually only specified in some abstract manner by such standards. Therefore, such mappings do not provide all the details required to create executable model transformations. Moreover, the static semantics of metamodels and/or UML profiles are usually specified only in natural language. Thus, on the one hand, ambiguities can occur, and on the other hand, the soundness of the static semantics cannot be verified and validated without a manual translation into a machine-processable language.

The main goal of this dissertation is to remedy the identified weaknesses in developing DSLs, for which a CMOF-compliant metamodel and a UML profile shall be created. To achieve this goal, we propose a holistic MDE-based approach for automatically deriving UML profiles and model transformations based on CMOF metamodels. This approach enables an automatic transfer of DSL's static semantics to UML profiles, so that the well-formedness of UML models with applied UML profile of a DSL can be verified automatically. In addition, the interoperability between UML and DSL models can already be validated in the development phase using the derived model transformations. Apart from new DSLs that are created from scratch, our approach also supports a migration of existing grammar-based DSLs towards CMOF-based metamodels, provided syntax rules exist.

To verify and evaluate the presented derivation approach, we have implemented a toolchain that we used to conduct two case studies on the *Specification and Description Language (SDL)* and the *Test Description Language (TDL)*.

Zusammenfassung

Ähnlich wie Allzwecksprachen können *domänenspezifische Sprachen (DSL)* mittels Grammatikformalismen entwickelt werden, aber auch das modellgetriebene Engineering (MDE) gewinnt für letztgenannte Sprachen immer mehr an Bedeutung. Einerseits können Metamodelle dazu verwendet werden, die Syntax und Semantik von DSLs zu definieren. Andererseits kann eine DSL auch durch eine Anpassung der *Unified Modeling Language (UML)* unter Verwendung des Profilierungsmechanismus, welcher auf UML-Profilen basiert, realisiert werden. So können beispielsweise Metamodelle für DSLs mit den Sprachkonzepten der *Meta Object Facility (MOF)* erstellt werden, wobei zwischen dem *Essential MOF (EMOF)* und dem *Complete MOF (CMOF)* unterschieden wird. Die letztgenannte Variante basiert auf dem EMOF, bietet aber zusätzliche Sprachkonzepte. Mit den Sprachkonzepten des CMOF kann ein höherer Grad an Abstraktion und Wiederverwendung bestehender Metamodelle erreicht werden, was für die Erstellung komplexer DSLs von Vorteil sein kann.

Neben den in der Literatur vorgestellten DSLs wird eine modellbasierte Spezifikation von DSLs auch in der Standardisierung immer wichtiger. Sowohl Metamodelle als auch UML-Profile werden oft für standardisierte DSLs angeboten. Die Abbildung von Metamodellen auf UML-Profile wird von solchen Standards meistens nur abstrakt spezifiziert. Daher liefern diese Abbildungen nicht alle Details, die für die Erstellung von ausführbaren Modelltransformationen erforderlich sind. Darüber hinaus wird die statische Semantik von Metamodellen und/oder UML-Profilen in der Regel nur in natürlicher Sprache angegeben. So können einerseits Unklarheiten auftreten, andererseits kann die Korrektheit der statischen Semantik nicht ohne eine manuelle Übersetzung in eine maschinenlesbare Sprache verifiziert und validiert werden.

Hauptziel der vorliegenden Dissertation ist es, die identifizierten Schwachstellen bei der Entwicklung von DSLs, für die ein CMOF-konformes Metamodell und ein UML-Profil erstellt werden sollen, zu beheben. Um dieses Ziel zu erreichen, wird ein ganzheitlicher MDE-basierter Ansatz zur automatischen Ableitung von UML-Profilen und Modelltransformationen auf Basis von CMOF-Metamodellen vorgeschlagen. Dieser Ansatz ermöglicht eine automatische Übertragung der statischen Semantik auf UML-Profile, so dass die Wohlgeformtheit von UML-Modellen, die ein UML-Profil einer DSL angewendet haben, automatisch überprüft werden kann. Des Weiteren kann die Interoperabilität zwischen UML- und DSL-Modellen bereits in der Entwicklungsphase mit den abgeleiteten Modelltransformationen validiert werden. Neben neu zu entwickelnden DSLs ermöglicht unser Ansatz auch eine Migration bestehender Grammatik-basierter DSLs zu CMOF-basierten Metamodellen, sofern Syntaxregeln existieren.

Um den vorgestellten Ableitungsansatz zu überprüfen und zu bewerten, wurde eine Werkzeugkette implementiert, mit der zwei Fallstudien für die *Specification and Description Language (SDL)* und die *Test Description Language (TDL)* durchgeführt wurden.

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Research Objective and Questions 6
1.3 Research Methodology 8
1.4 Obtained Results 9
1.5 Publications 10
1.6 Structure of the Thesis 13

2 Foundations of MDE-based Language Development **15**
2.1 Design and Development Process 15
2.2 Metamodelling Employing the MOF 17
2.2.1 The 4-Layered MOF Hierarchy 17
2.2.2 The Essential MOF 19
2.2.3 Package 19
2.2.4 The Complete MOF 23
2.3 Using OCL for Specifying the Static Semantics 28
2.4 Model Transformations 31
2.4.1 Model-to-Model Transformations 33
2.4.2 Model-to-Text Transformations 36
2.4.3 Notational Conventions for Model Transformations 39
2.5 The Unified Modeling Language and UML Profiles 40
2.5.1 The UML Diagrams 41
2.5.2 Extending the UML Using UML Profiles 44
2.6 Methods for Verification and Validation 47
2.6.1 Background 47
2.6.2 Metamodels and UML Profiles. 49
2.6.3 Verification of Model Transformations. 50
2.7 Summary 53

3 A New UML Profile for SDL – Lessons Learned **55**
3.1 Evolution of SDL and our Contributions 55
3.1.1 Historical Evolution of SDL and of its UML Profile 56
3.1.2 Our Contributions to a new UML Profile for SDL 57
3.1.3 The SDL-UML Modelling and Validation Framework 58
3.2 Foundations of SDL and of its UML Profile 59

3.2.1 Fundamental Parts of an SDL Specification 59
3.2.2 Formalization of SDL 63
3.2.3 The UML Profile for SDL 66
3.3 Identified Shortcomings of SDL-UML 2007 71
3.3.1 Drawbacks of the Data Type Model 72
3.3.2 Issues Concerning the Representation of SDL Expressions . 76
3.3.3 Shortcomings of SDL's Action Language 81
3.4 Introduced Enhancements 85
3.4.1 Specification of Data Types 86
3.4.2 Representation of SDL Expressions 92
3.4.3 Improving and Formalizing the Static Semantics 94
3.5 Validation and Verification of the UML Profile 103
3.5.1 Validation of Structural Aspects 103
3.5.2 Verification of the Static Semantics 104
3.5.3 Conclusions 106
3.6 Details of the SU-MoVal Framework 107
3.6.1 Editor for the Textual Notation 107
3.6.2 QVT-based Transformations 109
3.7 Related Work 112
3.8 Summary and Conclusions for an Automatic UML Profile Derivation 114

4 An Automatic Derivation Approach 117
4.1 Overview of the Approach 117
4.1.1 Our Overall Approach 118
4.1.2 The 'Abstract Concepts' 119
4.1.3 Our MDE-based Development Process 120
4.1.4 Implementation of the DSL Metamodelling and Derivation Toolchain 122
4.2 Metamodel Derivation 124
4.2.1 Process for Creating Metamodels 124
4.2.2 Design Decisions for the Metamodel Derivation 126
4.2.3 Production Rules and Annotations 130
4.2.4 Running Example 133
4.2.5 Transformation of Production Rules 134
4.3 Approach for Deriving UML Profiles 142
4.3.1 Process for Deriving UML Profiles and Additional Metaclasses 142
4.3.2 Design Decisions for the Profile Derivation 143
4.3.3 Assumption on and Enrichment of the Source Metamodel . 150
4.3.4 The UML Profile Derivation 154
4.3.5 OCL Expressions for Stereotype Attributes 170

4.3.6 Additional OCL Constraints 173
4.4 Approach for Deriving 'Additional' Metaclasses 179
4.4.1 Design Decisions for the Derivation of Metaclasses 180
4.4.2 Derivation of 'Additional' Metaclasses 181
4.4.3 Additional OCL Constraints 189
4.5 Update of Existing OCL Expressions 190
4.5.1 The OCL Metamodel 190
4.5.2 Design Decisions for Updating OCL Expressions 193
4.5.3 The OCL Update in Detail 195
4.5.4 Implementation Aspects 197
4.6 Derivation of M2M Transformations 197
4.6.1 Process for Deriving Model Transformations 198
4.6.2 Design Decisions for Deriving of Transformations 199
4.6.3 Target Language and Employed Technology 201
4.6.4 Common Concepts for Deriving the Transformations . . . 202
4.6.5 Derivation of the DM-to-UML Transformation 208
4.6.6 Deriving the UML-to-DM Transformation 222
4.7 Related work 228
4.8 Summary and Conclusions 231

5 Case Study-Based Evaluation 233
5.1 Evaluation Objectives and Requirements 233
5.2 Evaluation Design 234
5.3 Case Study A: Test Description Language 235
5.3.1 Background 235
5.3.2 Example Model used for the Evaluation 237
5.3.3 Preparation of the Metamodel 238
5.3.4 Evaluation of the Derived UML Profile 249
5.3.5 Evaluation of the Updated OCL Constraints 265
5.3.6 Evaluation of the Derived Model Transformations 268
5.3.7 Summary of Evaluation Results. 283
5.4 Case Study B: Specification and Description Language 284
5.4.1 Evaluation of the Derived Metamodel 284
5.4.2 Evaluation of the Derived UML Profile 297
5.4.3 Evaluation of Derived 'Additional' Metaclasses 313
5.4.4 Evaluation and Validation of the Approach's Applicability . 323
5.4.5 Summary of the Evaluation Results 326
5.5 Discussion of the Evaluation Results 327

6 Conclusions and Future Work 331
6.1 Summary 331
6.2 Results 333

6.3 Limitations 336
6.4 Conclusions 337
6.5 Directions for Future Research 338

Bibliography **341**

Index **357**

List of Tables

1.1 Standardized domain-specific languages having a metamodel and a UML profile defined 4
1.2 Chronological list of publications in relation to sections of this thesis 11

2.1 The possible collection types of a *Property* 22

3.1 Test cases for verifying OCL *Constraints* of the «SendSignalAction» stereotype 105
3.2 Different ambiguities of SDL's concrete syntax 108
3.3 Short-hand notations to be expanded by model transformations . . 112

4.1 Properties that redefine and/or subset other properties of UML metaclasses 149

5.1 Stereotypes of 'MM2Profile' applied to metaclasses and attributes of MM_{TDL} 244
5.2 Comparison between model elements of MM_{std} and MM_{TDL} 248
5.3 Comparison of metamodel MM_{TDL} and the UML profile UP_{TDL} . . 251
5.4 Comparison of the UML profiles UP_{std} and UP_{TDL} 261
5.5 Source and updated OCL *Constraints* of TDL data type definitions . 266
5.6 Comparison of MM_{TDL} and transformation $T_{TDL\text{-}to\text{-}UML}$ 269
5.7 Comparison of MM_{TDL} and the transformation $T_{UML\text{-}to\text{-}TDL}$ 277
5.8 SDL's production rules compared with the derived metamodel MM_{SDL} 287
5.9 Model elements of the reworked metamodel MM_{SDL} 296
5.10 Comparison of metamodel MM_{SDL} and UML profile UP_{SDL} 298
5.11 Comparison of the standardized UP_{std} and the derived UP_{SDL} . . . 305
5.12 Comparison of the metamodels MM_{SDL} and MM_{VS} 314
5.13 The standardized metaclasses that represent SDL expressions vs. the automatically derived ones in MM_{VS} 318

List of Figures

2.1 MDE-based development process for DSLs 16
2.2 The 4-layered MOF hierarchy 18
2.3 The most important part of the abstract syntax of the EMOF 20
2.4 The most important part of the abstract syntax of the CMOF 25
2.5 The metaclasses defining the type system of OCL 29
2.6 QVTo transformation example 34
2.7 Taxonomy of UML's structural and behavioural diagrams 41
2.8 UML state machine and activity diagram examples representing the behaviour of a telephone 43
2.9 A *Stereotype* specification and its application 45
2.10 A *Stereotype* that is defined as required 46

3.1 The historical evolution of SDL and of its UML profile 55
3.2 The components of the SU-MoVal framework 58
3.3 SDL example for the Session Initiation Protocol (SIP) 62
3.4 Formalization of the dynamic semantics of SDL 66
3.5 SDL-UML class and composite diagrams 68
3.6 SDL-UML state machine and activity diagram examples. 71
3.7 Abstract syntax of SDL-UML 2007 for specifying data types 72
3.8 SDL data type definitions specified in SDL's textual notation 73
3.9 Abstract syntax of SDL-2000 and SDL-UML 2007 for expressions . 77
3.10 Excerpt of the syntax and semantics of the «Expression» stereotype 78
3.11 Abstract syntax of exemplary stereotypes representing SDL statements 82
3.12 Syntax and semantics of SDL-UML 2007 stereotypes «SendSignalAction» and «SetAction» 84
3.13 Data type definitions and their abstract syntax in SDL-UML 2013 . 87
3.14 Abstract syntax of SDL-UML 2013 for representing expressions . . 92
3.15 Syntax and semantics of the metaclasses SdlExpression and LiteralValue in SDL-UML 2013 95
3.16 OCL constraints of the «SendSignalAction» stereotype of SDL-UML 2013 97
3.17 Excerpt of the static semantics of the SDL output node 99
3.18 The OCL-defined query staticSort() of the metaclass SdlExpression 101
3.19 SU-MoVal Editor for the textual notation 107

3.20 Transformation chain that creates SDL-UML elements 111

4.1 Transformation steps and their derived artefacts 118
4.2 Our MDE-based development process for DSLs 121
4.3 The Eclipse-based toolchain and its components 122
4.4 Activity for manually creating or deriving a metamodel 125
4.5 Extract of the concrete syntax of the adapted production rule notation 131
4.6 Syntax of a state machine DSL used for the running example . . . 134
4.7 Metamodel that is derived from production rules 136
4.8 Activity for deriving UML profiles and addtional metaclasses 143
4.9 Default mapping of a metamodel to a corresponding UML profile . 144
4.10 Multiple inheritance and the *type* of a stereotype attribute 147
4.11 The 'MM2Profile' UML Profile that is used to enrich a metamodel 151
4.12 A UML profile that is derived from a metamodel 156
4.13 Mapping of MC_{St} metaclasses to *Stereotypes* 158
4.14 Mapping to derived attributes. 163
4.15 Variants for recomputing the *type* property of stereotype attributes . 166
4.16 Subsetting MC_{St} attribute and the corresponding 'derived' and 'read-only' stereotype attribute 170
4.17 A modified version of metamodel MM_{Domain} and the 'additional' metaclasses that are derived from it 182
4.18 Basic structure of the abstract syntax of OCL 191
4.19 Abstract syntax of OCL's FeatureCallExp 192
4.20 The OCL ExpressionInOcl metaclass 193
4.21 Activity for deriving model transformations 198
4.22 Mapping operation signatures that are used for generated transformations 204
4.23 Helper operation for determining an *EnumerationLiteral* 208
4.24 Derived mapping operations that map DSL model elements to UML elements with applied stereotypes 215
4.25 Disjuncting mapping applyStereotype() that is generated based on the running example 217
4.26 Example of a generated helper operation for *Enumeration* data types 220
4.27 Generated mapping operation that maps a UML element with applied stereotype 227

5.1 TDL data and test configuration definition example 236
5.2 TDL test behaviour example. 238
5.3 MC_{AC} metaclasses defined for creating the metamodel MM_{TDL} . . 240
5.4 Metaclasses that define TDL's data type concept 242

5.5 TDL's predefined queries and their implementation for MM_{TDL} . . 246
5.6 Metaclasses of MM_{TDL} and the *Stereotypes* derived from them for UP_{TDL}, which define TDL's data type concept 256
5.7 Metaclasses of MM_{TDL} that define the syntax of TDL test descriptions and the *Stereotypes* derived from them for UP_{TDL} 258
5.8 *Stereotypes* in UP_{std} and UP_{TDL} used to define the syntax of TDL test descriptions 263
5.9 Mapping operations of transformation $T_{TDL\text{-}to\text{-}UML}$ derived from metaclasses in MM_{TDL} 271
5.10 Helper function for determining an *EnumerationLiteral* in UP_{TDL} . 276
5.11 Mapping operations of transformation $T_{UML\text{-}to\text{-}TDL}$ 279
5.12 Reworked mapping operation of transformation $T_{TDL\text{-}to\text{-}UML}$ 281
5.13 Transformed TDL example model 282
5.14 The 'Abstract Concepts' for a state machine in SDL 285
5.15 Production rule of SDL Procedure Definition and the metaclass derived from it 290
5.16 Production rule of SDL Expression and the metaclass of MM_{SDL} derived from it 294
5.17 The «ProcedureDefinition» stereotype and the MC_{St} metaclass from which it is derived 302
5.18 The «StateMachine» stereotype and its OCL *Constraints* as standardized for SDL-UML 2013 306
5.19 OCL *Constraints* and OCL-defined attributes of the «ProcedureDefinition» stereotype of the derived UP_{SDL} 309
5.20 The derived metaclass MC_{VS} SdlExpression 316
5.21 Derived metaclasses of MM_{VS} that represent SDL expressions . . . 319
5.22 The TimerRemainingDuration metaclass and its *Constraints* in MM_{VSstd} 320
5.23 The TimerRemainingDuration metaclass and its *Constraints* in MM_{VS} 322
5.24 Example of an SDL Procedure Definition represented in terms of UP_{SDL} and MM_{SDL} models 324

List of Pseudocode Listings

1 Mapping operations toAbstractClass(), toNonAbstractClass() and toEnumeration() . . . 135
2 Mapping operations that map Expression elements . . . 138
3 The queries mapsToStereotype(), isAbstractConcept() and mapsToAdditionalMetaclass() . . . 157
4 The toStereotype() mapping operation and associated queries . . . 159
5 The isDerivedProperty() query . . . 162
6 The toDerivedProperty() mapping operation . . . 164
7 The toNonDerivedProperty() mapping operation . . . 165
8 The getNewType() query . . . 168
9 The mapping operations toOperation() and toParameter() for processing MC_{St} operations . . . 169
10 The queries generateOclSpec() and oclContext() . . . 172
11 The queries oclSelectExp() and oclTypeCast() . . . 174
12 Queries for creating OCL constraints on *Stereotypes* . . . 177
13 The queries constraintSpec(), oclNavigationExp() and oclTypeVerificationExp() . . . 178
14 Mapping operation toAddMetaclass() and the addMcSuperClasses() query . . . 184
15 Mapping operation toProperty() and the getUmlProperty() query . 186
16 The mapping operations toOperation() and toParameter() for processing $\mathrm{MC}_{\mathrm{AMC}}$ operations . . . 188
17 The genOclSpecForTypeConstraints() query . . . 189
18 Top-level template genDmToUml() of transformation $T_{GenDM\text{-}to\text{-}UML}$ 209
19 The mcAcMapping() templates and associated queries of transformation $T_{GenDM\text{-}to\text{-}UML}$ . . . 211
20 The mcStMapping() templates of transformation $T_{GenDM\text{-}to\text{-}UML}$ . . 213
21 The stereotypeAppl() and stApplMapping() templates and the mappableStAtt() query of transformation $T_{GenDM\text{-}to\text{-}UML}$ . . . 216
22 The mcAddMapping() templates of transformation $T_{GenDM\text{-}to\text{-}UML}$. 218
23 The acDtHelperOp() template of transformation $T_{GenDM\text{-}to\text{-}UML}$ . . . 219
24 The various attMapping() templates and the targetName() query of transformation $T_{GenDM\text{-}to\text{-}UML}$ . . . 221
25 Top-level template genUmlToDm() of transformation $T_{GenUML\text{-}to\text{-}DM}$ 223

26 The $\mathrm{mcStMapping}()$ templates and associated queries of transformation $T_{\mathit{GenUML\text{-}to\text{-}DM}}$ 225

1 Introduction

The use and development of *domain-specific languages (DSL)* has become increasingly important in recent years. A DSL is a computer language designed to solve problems in a particular technical or application-specific domain. To tackle this challenge, a DSL provides appropriate language constructs and/or notations. By contrast, *general-purpose languages (GPL)* such as Java or C++ can be applied across multiple domains to solve a variety of different problems. However, Mernik et al. [106] point out that the notation of GPLs often cannot be adapted to meet domain-specific requirements. Furthermore, they argue that domain-specific aspects cannot always be expressed concisely by means of language concepts provided by a GPL.

A wide range of definitions for a DSL is given in the literature. One of the most frequently cited papers on this subject was published by van Deursen et al. [32], where the authors define a DSL as follows:

> *"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."*

As in the case of GPLs, the concrete syntax of a DSL can be represented in different kinds of notations. The most common are textual or graphical representations [77, 106, 145], or a combination of both variants, e.g., all DSLs listed in Table 1.1.

In recent years, various approaches for developing DSLs have been proposed, which can be divided into two categories. The first category comprises approaches based on the formalisms already used for GPLs, such as context-free grammars. The advantage is that existing language development tools such as parser-generators can also be used for DSLs. Klint et al. [82] propose the generic term *grammarware* to summarize all approaches based on grammar-formalisms and related development tools.

The second category encompasses all approaches that apply *model-driven engineering (MDE)* [140]. In the literature, the terms model-driven language engineering [26, 59, 145] or development [106, 149] are also employed to refer to such approaches. In some cases, the DSLs developed by applying an MDE approach are called *domain-specific modelling languages (DSML)*. In the following, we do not distinguish between the two terms DSL and DSML; instead, we consistently use the term DSL.

The most important activity [26, 59, 145] of an MDE-based development of a DSL is the design and creation of a so-called *metamodel*. This activity is usually referred to as *metamodeling*. Typically, a metamodel defines the abstract syntax and static semantics of the DSL to be implemented. Depending on the applied approach [26, 135], a metamodel can also capture the static semantics of a DSL. Various definitions for a metamodel are given in the literature (e.g., [26, 93]). A definition that captures exactly our point of view concerning a metamodel is given by da Silva, who defines a metamodel *"as a model that defines the structure of a modeling language"* [140].

In addition to a metamodeling approach, a DSL can also be implemented by customizing the *Unified Modeling Language (UML)* [115]. Either the UML metamodel can be modified or extended to meet the requirements of a DSL, or so-called *UML profiles* [46] can be used. A metamodel-based customization of the UML is referred to as heavyweight extension, while UML profiles are a build-in profiling mechanism of the UML. Because UML profiles do not modify the UML metamodel, they are considered as a lightweight extension to the UML [34, 83]. When a DSL is implemented by employing a UML profile, a language engineer has to specify a set of UML stereotypes, which are a particular kind of UML element. A UML stereotype can extend a UML metaclass by introducing additional attributes, operations and constraints. Different manual or generative approaches for implementing a DSL based on UML profiles can be found in the literature, e.g., in [52, 97, 138].

Apart from metamodels, model transformations are another important component of MDE [93, 136, 140]. The following three types of model transformations can be distinguished: *text-to-model (T2M)*, *model-to-model (M2M)* and *model-to-text (M2T)*. For example, T2M transformations are employed to generate a model based on the textual notation of a DSL. In contrast, M2T transformations are used to generate text, e.g., source code, based on a model. Models as instances of a source metamodel can be transformed into models of a target metamodel using M2M transformations. Different transformation languages and technologies are available for each of the three transformation types. For instance, the *Query/View/Transformation (QVT) standard [113]* can be used for M2M transformations, whereas M2T transformations can be defined with the *MOF Transformation Language (MTL) [111]*.

1.1 Motivation

A large number of DSLs is proposed in the literature, which are defined either by means of a metamodel or a UML profile. DSLs for which both a metamodel and an associated UML profile are available are the exception. Both artefact kinds are usually only available for DSLs that are standardized, e.g., by the *Object Manage-*

ment Group (OMG)[1], the *International Telecommunications Union (ITU)*[2] or the *European Telecommunications Standards Institute (ETSI)*[3]. Examples of DSLs for which both a metamodel and a UML profile are available are listed in Table 1.1.

The present dissertation has been inspired by our experience of creating a new version of a UML profile [69] for the *Specification and Description Language (SDL)* [70]. SDL is a DSL that has been used for more than three decades [129] in the area of telecommunications for the modelling of communication protocols and distributed systems. All standardization activities concerning SDL are sponsored by the ITU. SDL specifications can be modelled using textual and graphical concrete syntax. The static and dynamic semantics of SDL are standardized and employ grammar formalisms [72, 73].

Although the semantics of SDL is formally specified, this is not the case for the first version of the UML profile [65] for SDL, because the static and dynamic semantics of this UML profile are given in natural language only. Moreover, only a small subset of SDL's language constructs is supported. Even though we were able to remedy these drawbacks by a manual revision, the effort for this was significant and also error-prone. We used the gained experience to analyse other standardized DSLs (listed in Table 1.1) and identified similar shortcomings, which motivated us to research on the automated derivation of UML profiles.

Based on our analysis, we identified the four following areas for improvements in the MDE-based development of standardized DSLs:

Metamodelling and metamodel derivation. Metamodels can be created using language concepts provided by a meta-metamodel, where the most popular one is the *Meta Object Facility (MOF)* [121], which is standardized by the OMG. For example, the UML metamodel [115] is such an MOF-based metamodel. A metamodel can be implemented not only with the MOF, but also with other meta-metamodels such as the Ecore Meta-Metamodel of the *Eclipse Modeling Framework (EMF)* [142]. The MOF specification defines two different meta-metamodels: the *Essential MOF (EMOF)* and the *Complete MOF (CMOF)*. According to [121], both meta-metamodels have different objectives:

> *"A primary goal of EMOF is to allow simple metamodels to be defined using simple concepts while supporting extensions for more sophisticated metamodelling using CMOF."*

Because EMOF was inspired by Ecore, both meta-metamodels are almost identical with regards to their supported language concepts (see [47, 142]). Even though CMOF is based on EMOF, it offers additional language concepts. For example, the subsetting and redefinition of metaclass attributes are such language concepts. As

[1] http://www.omg.org/
[2] http://www.itu.int/
[3] http://www.etsi.org/

Table 1.1: Standardized domain-specific languages having a metamodel and a UML profile defined

Language	Metamodel	Constraints defined	OCL defined attributes	OCL defined operations	Notation	Organi-sation
Modeling and Analysis of Real-Time Embedded Systems (MARTE) [117]	CMOF-based	natural language	no	no	textual & graphical	OMG
Service oriented architecture Modeling Language (SoaML) [118]	CMOF-based	natural language	no	no	graphical	OMG
Software & Systems Process Engineering Metamodel (SPEM) [112]	CMOF-based	natural language	no	no	graphical	OMG
Specification and Description Language (SDL) [70]	CMOF-based [128, 135]	natural language	no	no	textual & graphical	ITU
Business Process Model and Notation (BPMN) [119]	EMOF-based	natural language	defined for the UML profile	no	graphical	OMG
Test Description Language (TDL) [39]	EMOF-based	OCL constraints	no	no	textual & graphical	ETSI

pointed out by Alanen and Porres [8], the language concepts of CMOF can be particularly useful for creating new metamodels by extending an existing one. The authors mention the UML [115] metamodel as example, where the 'Kernel' package is refined by other packages that provide more concrete language concepts. In addition, Scheidgen [134] concludes that a higher degree of modularity and abstraction can be obtained, and the reuse of existing language concepts can become possible by employing the language concepts of CMOF. Due to these advantages, a CMOF-based metamodel for SDL is proposed in [128, 135]. However, this metamodel is not part of the SDL standard. Further examples of standardized DSLs with CMOF-based metamodels are given in Table 1.1.

In recent years, various approaches (e.g., [6, 44, 94, 153]) and tools (e.g., [155, 158, 159]) for generating metamodels based on syntax rules have been proposed. Thus, a metamodel for an existing DSL can be generated, provided that syntax rules are available. However, Fischer et al. [44] and Scheidgen [135] remark that the obtained metamodels can only be considered as "initial" or "primitive". Often, additional effort is required to finalize such a metamodel. For instance, this is the case if language concepts of an existing metamodel (e.g., the UML metamodel) shall be reused in order to achieve a higher degree of abstraction, as recommended by Clark et al. [26] and Fischer et al. [44]. In addition, the approaches and tools for generating metamodels cited above does not support the generation of CMOF-based metamodels.

Because of the discussed advantages of CMOF, the generation of CMOF-based metamodels for DSLs with a high number of language concepts could be promising, but an automatic derivation of such metamodels is currently not supported.

UML profile creation. Provided that an EMOF-based metamodel is available, a UML profile can be created based on that metamodel, e.g., by applying one of the manual approaches discussed in [97, 138, 146]. Furthermore, an automatic derivation as proposed by Giachetti et al. [50] or Wimmer et al. [152] is applicable to derive a UML profile directly or indirectly based on a metamodel. However, a common disadvantage of these approaches is that they only support language concepts of the EMOF, so they are not applicable to CMOF-based metamodels.

As previously stated, we consider the use of CMOF-based metamodels as an advantage for complex DSLs. Therefore, an approach for the automatic derivation of UML profiles should also be able to process such metamodels, taking into account the CMOF language concepts.

Static semantics. Typically, metamodels define not only the abstract syntax but also the static semantics of a DSL. The OMG has standardized the *Object Constraint Language (OCL)* [120] to enable the specification of the static semantics of metamodels in a computable notation. OCL constraints contained in metamodels must be fulfilled by a model as an instance of a metamodel at any time; they are also referred to as *well-formedness rules*. OCL is comparable to a first-order predicate logic as utilized for computer languages or DSLs that are based on grammarware (e.g., SDL [72]). A translation of OCL into a first-order predicate logic is also possible, as demonstrated by Beckert et al. [15].

Although OCL constraints play an important role in metamodeling, they are usually defined in natural language for standardized DSLs, as shown in Table 1.1. Due to this fact, OCL constraints must be created manually for each implementation, which requires effort and may cause errors. Even if OCL constraints are present in a metamodel, this is not necessarily the case for a corresponding UML profile, see, e.g., the *Test Description Language (TDL)* listed in Table 1.1. To remedy this situation, one could consider to copy OCL constraints from a metamodel to a UML profile, but this is infeasible because UML stereotypes and UML metaclasses exist as separate instances in a UML model. However, without having proper OCL constraints for a UML profile, the well-formedness of a UML model having applied this UML profile cannot be verified. Therefore, the OCL constraints of a metamodel must be adapted before they can be transferred to a derived UML profile.

The approaches discussed above for the derivation of UML profiles do not support an automatic transfer of the static semantics, but this is an important prerequisite for the creation of valid UML models.

Model interoperability. If not only a metamodel but also a UML profile is available for a DSL, a user can either create domain models (as an instance of the metamodel) or UML models with the UML profile applied. Typically, a tool manufacturer only implements either the metamodel-based or the UML-based variant of a DSL. Models of all MOF-based modelling languages can be exchanged between different tools via the standard *XML Metadata Interchange (XMI)* [122] format, but only interoperability between tools supporting the same metamodel is ensured. The interoperability between a UML-based and a metamodel-based implementation of a particular DSL cannot be achieved, as argued in [2, 51, 80, 101]. This is because UML models and DSL models are instances of different metamodels.

In order to obtain model interoperability, e.g., between a DSL-specific editor and a UML modelling tool, the aforementioned works propose to employ a particular kind of model transformation, which may be generated (semi-)automatically depending on the approach. This kind is also referred to as model bridge [141, 24]. A limitation is that only EMOF-based metamodels are supported. Thus, the processing of CMOF-based metamodels is an open question. Moreover, the approaches above do not treat the challenges of potential semantic and syntactic gaps between two modelling languages, as discussed in [58, 124, 127]. For example, a semantic gap exists if a language concept of a source language has no corresponding concept in the target language. Moreover, a syntactic gap may exist if an model element of a source model has to be located at another position in the target model. In our case, these gaps may exist for a mapping of DSL models to UML models or vice versa. Thus, the question arises how appropriate mapping rules can be defined and generated.

1.2 Research Objective and Questions

Research objective. Based on the open issues and drawbacks identified above, we summarize the main objective of this dissertation as follows:

> *"Establishing a holistic MDE-based approach for the automatic derivation of UML profiles from CMOF-based metamodels of existing or new DSLs, while considering static semantics and model interoperability."*

By using the term 'existing' we wish to express that our approach is applicable to existing DSLs, which are based on grammar formalisms. In contrast, we employ the term 'new' to address those DSLs that are designed and developed from scratch. In this case, the metamodel of a DSL must be created manually instead of deriving it automatically from syntax rules.

Research questions. Building on our research objective, we now identify the research questions that need to be addressed by our work. In particular, we use these questions to assess whether our research objective has been met or not.

Different tools for generating EMOF-compliant metamodels based on production rules are available, but this is not the case for CMOF-based metamodels, where automatically derived metamodels have to be revised manually. In particular, this concerns the reuse of generic language concepts (i.e., the metaclasses) of existing metamodels and the employment of the language concepts of *redefinition* and *subsetting* provided by the CMOF. Therefore, our first research question is as follows:

Research Question 1 *Is it possible to support CMOF language concepts when generating metamodels from grammar production rules?*

As argued above in the case of DSLs for which a CMOF-based metamodel and a UML profile shall be created, there are open questions with regards to the automatic derivation of a UML profile, the transfer of the static semantics, and the generation of model transformations. Moreover, the existing approaches only support derivations that are not directly based on the metamodel of a DSL. To keep the effort as low as possible, we consider a derivation directly based on a metamodel as more appropriate. If CMOF-compliant metamodels are used as source for deriving UML profiles, the question arises how to map CMOF-specific concepts to corresponding constructs of a UML profile. To solve these problems, we postulate the following research question:

Research Question 2 *Is the automatic derivation of a UML profile from a CMOF-based metamodel possible when taking into account the language concepts provided by the CMOF?*

The static semantics of metamodels of standardized DSLs are often available in natural language only, but not in terms of OCL constraints. The same situation applies to UML profiles defined for such DSLs. Due to missing OCL-based constraints, the well-formedness of a UML model with applied UML profile of a DSL cannot be verified and, thus, the specification of invalid models is possible. To improve this situation, we must provide an appropriate answer to the following research question:

Research Question 3 *Can the static semantics of a DSL-specific metamodel be transferred automatically to a derived UML profile?*

An important aspect for DSLs that can be implemented either based on a metamodel or a UML profile is the interoperability of models between these two technical domains. This kind of interoperability can usually be obtained by employing a particular type of model transformation, which can be (semi-)automatically generated using existing approaches. Current open issues are the handling of semantic and syntactic gaps and the support of CMOF-based metamodels.

We consider the generation of executable model transformations as an advantage for an MDE-based implementation of a DSL for which both a CMOF-based metamodel and a corresponding UML profile shall be created. This would enable the verification of whether the defined requirements for a DSL are met not only by the metamodel but also by the UML profile, at an early stage in the development and/or standardization process.

Because our objective is to derive UML profiles based on CMOF-compliant metamodels, the question arises whether model transformations can be generated directly from such metamodels in the light of potentially semantic and syntactic gaps. Hence, we have to answer the following research question:

Research Question 4 *To what extent is it possible to automatically derive model transformations from a single CMOF-based metamodel to obtain model interoperability?*

Topics not covered in this thesis. The aspects addressed by us mainly relate to the abstract syntax and static semantics of a DSL implemented by means of an MDE-based approach. Consequently, we do not treat the implementation of textual or graphical editors for the concrete syntax of a DSL.

In case of an implementation based on a UML profile, existing UML modelling tools, such as Eclipse Papyrus[4], can be used to graphically specify a model. The advantage is that no implementation or only a tool-specific configuration is required. In contrast, a metamodel-based DSL realization requires the creation of a graphical or textual editor. Although many parts of such editors can be generated by existing tools, some parts of the editors must be implemented manually. For example, the *Graphical Modeling Framework (GMF)* [36] can be utilized to generate graphical editors for DSLs based on Eclipse. In the case of a textual notation, tools such as xText [159], TEF [158] or EMFText [155] are available for generating textual editors.

Moreover, we do not consider the specification of the dynamic semantics of a DSL, because different approaches are available. According to Clark [26], the dynamic semantics of an MDE-based DSL can be defined by translational, operational, extensional or denotational techniques. For example, Scheidgen [135] proposes the use of an action language that employs UML activities and OCL to specify dynamic semantics. A translation approach is applied for the *Action Language for Foundational UML (ALF)*, where language constructs of ALF are mapped to the *Foundational UML (fUML)* – an executable subset of UML – using transformations. A translational approach can also be utilized for the dynamic semantics of UML profiles, as discussed in [131].

1.3 Research Methodology

In the following, we briefly overview the methodology applied for our research. In order to meet our previously outlined research objective, we conduct three consecutive steps:

[4] https://www.eclipse.org/papyrus/

1. SDL [70] is used to identify general weaknesses in the creation of UML profiles for a DSL. We consider SDL and its associated UML profile as a good research subject, because SDL has a history of more than thirty years and the language has continued to evolve. Numerous publications on various topics concerning SDL can be found in the literature (e.g., see [19, 37, 43, 128, 129, 137]). This includes proposals for the specification of UML profiles [38, 151, 150] and metamodels [44, 135] for SDL.

2. Based on the results of the first step, we intend to solve the identified problems of hand-crafted UML profiles by introducing a holistic derivation approach. First, we define the rules that are required to derive the required artefacts. Then, we implement a toolchain that realises our derivation approach. In addition, we specify an MDE-based development process that takes the creation and quality assurance of the artefacts created by our toolchain into account.

3. Finally, we conduct two case studies for existing DSLs to verify and evaluate the applicability of our derivation approach and the toolchain. Based on the obtained results we also evaluate whether our approach meets the research objective and the associated research questions.

1.4 Obtained Results

As previously highlighted, the shortcomings of a manual UML profile creation that we identified during our work on a new edition of the UML Profile for SDL [69] inspired us to develop an automated solution for this task. To overcome this problem, we have defined the above research question for this dissertation. We address this question by developing a derivation approach for the fully automated generation of UML profiles from CMOF-based metamodels. While generating UML profiles, we also transfer the OCL-defined static semantics from a metamodel to the created UML profile. The presented derivation approach is in particular intended for DSLs, for which a metamodel and a corresponding UML profile are to be created, as is the case for the discussed DSLs in Table 1.1. To facilitate an efficient MDE-based development of DSLs, our holistic derivation approach also offers the semi-automated generation of CMOF-based metamodels and model transformations for model interoperability.

The first contribution of our work is the discussed derivation approach itself, which we have defined by means of several model transformations. Except of the transfer of the OCL-defined static semantics, we specified one dedicated transformation for each artefact type to be generated. In detail, our holistic derivation approach considers the following aspects:

- Semi-automatic generation of CMOF-based metamodels from grammar production rules of existing DSLs;
- Fully-automatic generation of UML profiles from CMOF-based metamodels, whereby we map subsetting or redefining metaclass attributes to OCL-defined stereotype attributes, whose values are automatically computed at runtime;
- Fully-automatic transfer of the static semantics specified via OCL to a derived UML profile;
- Semi-automatic generation of model transformations to achieve model interoperability between DSL models and UML models with applied UML profile.

As a second contribution of our work, we have implemented the *DSL Metamodeling and Derivation Toolchain (DSL-MeDeTo)* that implements all mentioned aspects of our holistic derivation approach. To verify our approach and evaluate our toolchain, we conducted two case studies for the DSLs *Specification and Description Language (SDL)* and *Test Description Language (TDL)*. Our toolchain and artefacts that we generated for the two case studies are available as open source software and can be obtained via our homepage [156].

Because our derivation approach was inspired by our work on a new edition of the UML profile for SDL, we also count our contributions in this area to the results we achieved. In particular, we fundamentally contributed to a new edition of the UML profile for SDL, which was published as ITU-T Rec. Z.109 [69] in 2013. To evaluate this UML profile, we have implemented the *SDL-UML Modeling and Validation (SU-MoVal)* toolchain, which is also available as open source via our homepage [156].

1.5 Publications

As our research objective is inspired by our work on a new edition of the UML profile for SDL [69], we divide our publications into the following subject areas: new version of the UML profile for SDL, and holistic derivation approach for DSLs. To provide a comprehensive overview, we summarize our various publications chronologically in Table 1.2. We distinguish contributions for conferences, workshops and standardization. In addition, each contribution is related to the corresponding section of this dissertation. Compared to the publications listed in Table 1.2, the relevant sections of this dissertation discuss the specific subjects in much more detail.

Table 1.2: Chronological list of publications in relation to sections of this thesis

Number	Title	Type	Section
1	*Results in using the new version of the SDL-UML profile* [92]	workshop paper	3.3
2	*Open issues of the SDL-UML profile* [85]	technical report	3.3, 3.4
3	*The SDL-UML profile revisited* [87]	workshop paper	3.4
4	*A model-based formalization of the textual notation for SDL-UML* [86]	conference paper	3.6
5	*ITU-T Recommendation Z.109: Specification and Description Language – Unified Modeling Language profile for SDL-2010* [69]	standard	3.4
6	*Realizing model simplifications with QVT operational mappings* [88]	workshop paper	3.6, 4.6
7	*Towards an extensible modeling and validation framework for SDL-UML* [89]	conference paper	3.6
8	*On the automated derivation of domain-specific UML profiles* [91]	conference paper	4.1 – 4.6, 5.4
9	*Automated tooling for the evolving SDL standard: From metamodels to UML profiles* [89]	conference paper	4.1 – 4.5, 5.3

New version of the UML profile for SDL. Before we started our research on the automatic derivation of UML profiles from metamodels, we contributed significantly to a new edition of the UML profile for SDL. Based on the shortcomings that we identified in the old edition of this profile from 2007 [66], we submitted several proposals for improvement to the responsible ITU-T working group. Based on these contributions, we compiled the latest edition of the related standard that was published in 2013 as ITU-T Rec. Z.109 [69].

1. Comparative case study [92] between native SDL and the UML profile for SDL [66] of 2007. We identified missing OCL constraints and a time-consuming effort needed for specifying SDL statements and expressions using the graphical notation as the major drawbacks of the UML profile for SDL of 2007.

2. *Open issues of the SDL-UML profile* [85]: A technical report about further shortcomings and errors of the UML profile for SDL, published in 2007 [85]. The report was submitted to the working group of the ITU responsible for the standardization of SDL.

3. *The SDL-UML profile revisited* [87]: Analysis of and improvement on shortcomings of the language concepts for data types and for the specification of expressions and values, provided by the UML profile for SDL of 2007 [66].

4. *A model-based formalization of the textual notation for SDL-UML* [86]: A proposal for a model-based formalization of a textual notation for the action language of the UML profile of SDL, whose notation embraces a subset of the concrete syntax of SDL [70].

5. *ITU-T Recommendation Z.109* [69]: In response to our contributions (1), (2) and (3), the author of this thesis was appointed to create a new edition of the UML profile for SDL, published as ITU-T Rec. Z.109 [69] in 2013.

6. *Model simplifications via QVT operational mappings* [88]: An approach for the metamodel-based generation of model transformations, which can be employed to realize transformation patterns for model simplifications. The metamodel for the textual notation of SDL, as proposed in [86], is used to demonstrate the applicability of the proposed transformation patterns.

7. *Towards an extensible modeling and validation framework for SDL-UML* [89]: An overview of our *SDL-UML Modelling and Validation (SU-MoVal)* framework that supports the specification and validation of UML models with applied UML profile for SDL of 2013 [69]. All constraints of this UML profile, which are specified in native language, were implemented using OCL, so an automatic verification of the static semantics of UML models with applied UML profile is made possible. In addition, the framework provides a textual notation editor for the action language, as proposed in [86].

Holistic derivation approach. The second group of publications comprises the outcomes of our research on a holistic derivation approach for DSLs, for which a CMOF-compliant metamodel and a corresponding UML profile shall be created. Our contributions to conferences are as follows:

8. *On the automated derivation of domain-specific UML profiles* [91]: An approach for the automatic derivation of UML profiles based on CMOF-compliant metamodels for DSLs, which automatically updates the static semantics and transfers it to the derived UML profiles. The applicability of the approach is demonstrated based on case study for SDL.

9. *Automated tooling for the evolving SDL standard: From metamodels to UML profiles* [90]: Discussion of the results that can be obtained by applying our holistic derivation approach to SDL. In particular, the following derived artefacts are treated: a CMOF-based metamodel, a UML profile, and model transformations enabling model interoperability.

1.6 Structure of the Thesis

The remainder of this dissertation is structured as follows. In Chap. 2, we discuss the foundations of the model-based engineering of DSLs, which include the topics of metamodelling, model transformations, and the fundamentals of UML and its profiling mechanism.

As pointed out in the motivation section, the shortcomings of the withdrawn version of the UML profile for SDL and the experience gained during our work on a new version of this UML profile are the main motivation for developing a holistic derivation approach. Therefore, the identified shortcomings and our improvements are treated in the first part of Chap. 3. In the second part, we use this information to define requirements for an automated derivation approach of UML profiles, which is presented in Chap. 4. There, we first provide a brief overview of all aspects of our derivation approach and of the toolchain that implements this approach. In addition, we introduce an appropriate MDE-based development process. In the remaining sections of Chap. 4, we discuss the details for deriving CMOF-compliant metamodels, UML profiles, and model transformations. In addition, we also treat the automatic transfer of the static semantics from metamodels to UML profiles.

The evaluation of our holistic derivation approach and of the generated artefacts is discussed in Chap. 5 based on two different DSLs. Finally, we use the obtained evaluation results in Chap. 6 to answer our research questions, so we can assess whether our research objective could be met. Moreover, we also discuss known limitations of our derivation approach and address further research directions.

2 Foundations of MDE-based Language Development

To enable a detailed discussion of our research in the following chapters, the present chapter gives a brief introduction to the foundations of model-based engineering of DSLs. However, before we explain the basic principles of *Model-Driven Engineering (MDE)* and related technologies, we introduce the general activities in an MDE-based process for developing DSLs. Then, we continue with the technologies that are important to the MDE: metamodelling using MOF, OCL-based specification of the static semantics of metamodels, and model transformations. Then, we cover the aspects of UML that are important to our work, including UML profiles as a standardized extension mechanism. Finally, we discuss the verification and validation of metamodels and model transformations.

2.1 Design and Development Process

In the motivation section of our work, we argued that it is insufficient just to define a derivation approach for an MDE-based development of DSLs. Due to the various artefacts to be created and their dependence among each other, we must also define an appropriate development process. In the past, a variety of design and development processes have been proposed for a grammar- (e.g., [82, 106]) or metamodel-based implementation (e.g., [26, 59, 145]) of DSLs. This also embraces approaches (e.g., [97, 138, 146]) for implementing DSLs using UML profiles. Typical artefacts realized in such a development process are:

- a metamodel that defines the abstract syntax of the DSL;
- a set of OCL constraints specifying static semantics;
- behavioural specifications that define dynamic semantics;
- a set of production rules that capture the concrete syntax of textual DSLs.

The activities of a metamodel-based development process for DSLs presented in [26, 59, 145] are similar. However, the details of these activities are discussed in detail only by Strembeck and Zdun [145]. Furthermore, the authors illustrate the relationships between the activities to be conducted and the artefacts to be created by using activity diagrams. Because of these advantages, in the following chapters we want to apply the process proposed by Strembeck and Zdun in a modified way

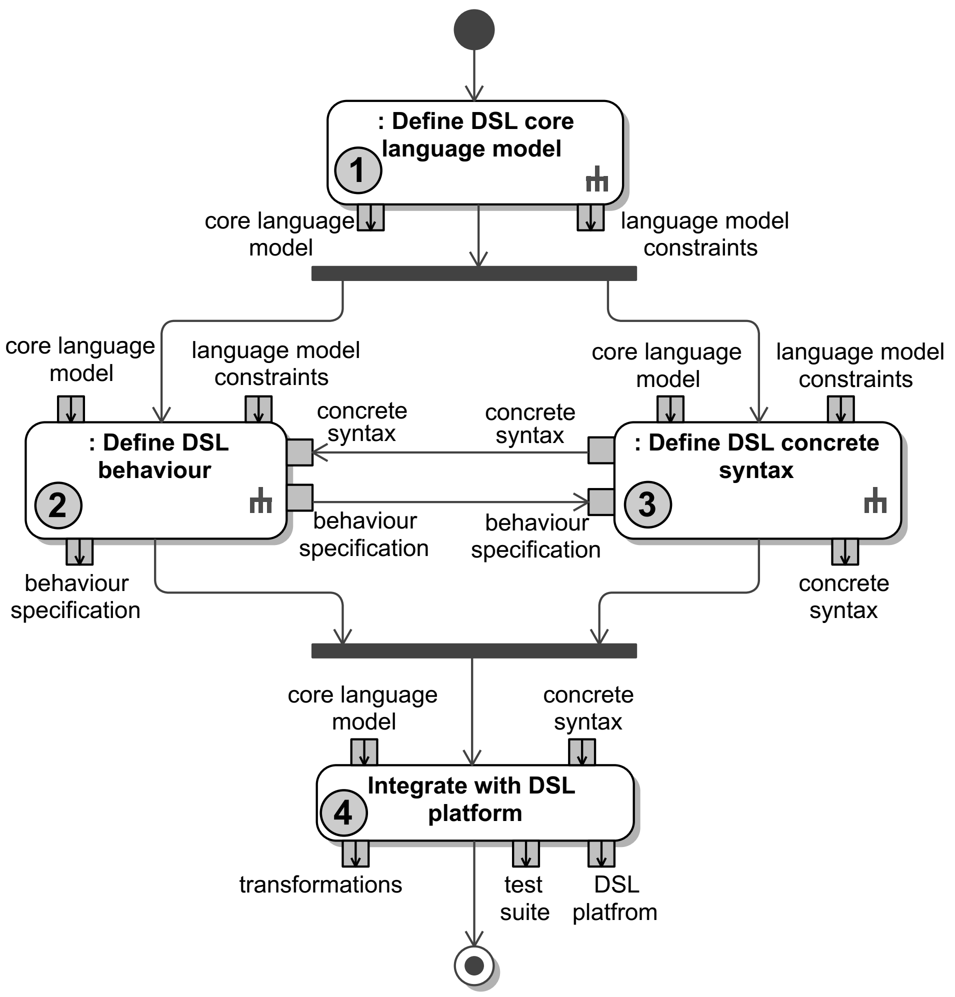


Figure 2.1: MDE-based development process for DSLs adapted from [145]

to our approach. To better explain the modifications introduced by us, we present the original process below.

The process proposed by Strembeck and Zdun [145] consists of one main activity and several sub-activities. In the following we only discuss the main activity of the process shown in Fig. 2.1 and briefly explain the purposes and outcomes of each sub-activity. The main activity consists of four actions, each of which invokes a sub-activity associated to it.

The main process starts with Action (1), which creates the metamodel for the DSL to be implemented. Strembeck and Zdun call this metamodel 'DSL core language model'. In addition to the abstract syntax, which is defined by the classes and their attributes in the metamodel, constraints that define the static semantics of a DSL are specified as part of Action (1). Thereafter, it has to be checked whether the constraints specified for the metamodel are valid and complete. If this is not the case, then the constraints must be revised or supplemented. Furthermore, it has to be analysed whether the language concepts defined by the metamodel are suitable and appropriate for the target domain of the DSL. If these objectives are not met, the language concepts defined by the metamodel must be modified.

Because of the described tasks, we can consider the creation of a metamodel and its constraints as an iterative activity.

Based on the Actions (2) and (3) in Fig. 2.1, the concrete syntax and the behaviour of a DSL has to be developed in parallel. This is in contrast to the development processes proposed in [26, 59], where these aspects of a DSL are developed one after the other. Strembeck and Zdun [145] argue that it is often useful to develop the concrete syntax and the behaviour in parallel, as this allows an exchange of intermediate results. Thus, it shall be ensured that the concrete syntax and corresponding behavioural definitions are consistent, complete and well integrated with each other. Since the research on the concrete syntax and behaviour of DSLs are not a subject of our work, we will not go into further details on both issues and refer to related work, e.g., [26, 59].

Action (4) is the final step of the development process, where the developed artefacts are integrated into a DSL platform, such as an *Integrated Development Environment (IDE)*. For instance, the integration can be realized by means of a mapping of language concepts of the DSL to appropriate features of the platform. For this purpose, transformations must be created that implement this mapping. In the case of Eclipse, the mapping can be implemented using code generators provided by Eclipse-EMF. As argued in [26], the integration can also be realized by means of a mapping to another language. Finally, integration tests have to be conducted to verify the successful integration of the DSL.

2.2 Metamodelling Employing the MOF

A meta-metamodel defines the language concepts applicable to create a metamodel, which then specifies the syntax and static semantics of a computer language or DSL [79, 134, 136]. According to this view, a meta-metamodel in turn can be considered as a particular variant of a metamodel, which defines a modelling language for the creation of metamodels. The elements of a metamodel represent instances of the language concepts provided by the meta-metamodel used. Therefore, we can also consider a metamodel as an instance of a meta-metamodel.

In contrast to metamodels, a meta-metamodel is not an instance of another model; instead, it is usually self-descriptive, which means that it is described using the language concepts it provides. For instance, this approach is used to specify the MOF [121].

2.2.1 The 4-Layered MOF Hierarchy

The language concepts provided by the MOF are based on the concepts known from object-oriented programming [134, 7]. The MOF is described using a class model consisting of several class diagrams. In order to enable the self-description of the MOF, a reflection mechanism is defined based on an object model as an in-

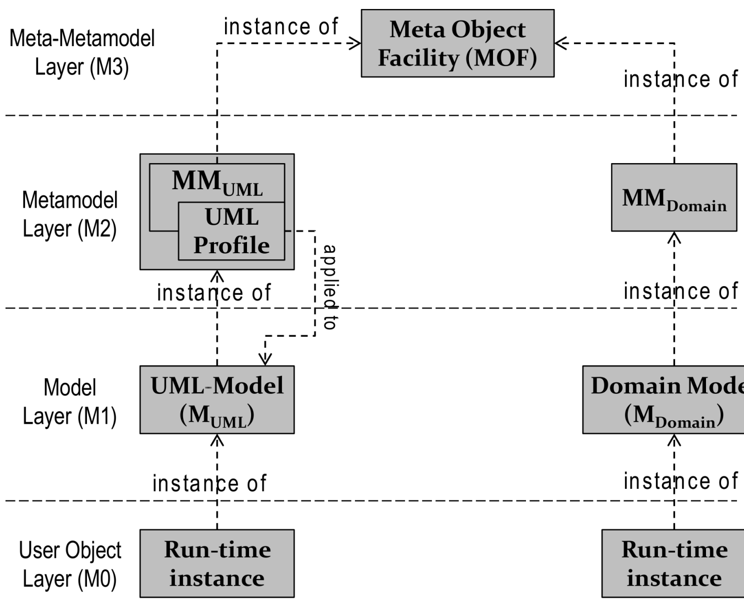


Figure 2.2: The 4-layered MOF hierarchy

stantiation of the above class model. This mechanism also provides a concept for navigating from an object to the class instantiated by it. This can also be employed to realize any number of modelling layers, also known as meta-layers. Thus, a multi-layered hierarchy of abstractions can be used to describe modelling languages. The MOF does not define a rigid number of meta-layers. However, at least two layers are required.

The four-layered metamodel architecture used for UML can be considered as one of the best known. As shown in Fig. 2.2, the MOF is located at the top layer (M3) of the hierarchy, and the metamodels as instances of it are located at the next lower layer (M2). For example, the UML metamodel is such a metamodel. Analogously, metamodels of DSLs can reside on the same layer, while models as instances of a particular metamodel are on model layer (M1). For example, such a model may correspond to the source code of a program implemented by means of a DSL. The user object layer (M0) is the lowest layer of the hierarchy, and runtime objects are located there as instances of the models of a language.

The MOF specification [121] defines not just one but two meta-metamodels; the *Essential MOF (EMOF)* and the *Complete MOF (CMOF)* based on it. Mainly, metaclasses of the UML metamodel were reused to specify the language concepts of both meta-metamodels. Thus, not only the MOF but also metamodels based on it can be modelled employing UML class diagrams. However, the EMOF and CMOF define a set of constraints for UML class diagrams, ensuring the creation of MOF-compliant metamodels.

Although only CMOF-based metamodels are the focus of our research, we first briefly describe the language concepts of the EMOF to better explain the difference to the CMOF. In addition, we require this information to discuss our derivation approach in the following chapters.

2.2.2 The Essential MOF

The language concepts provided by the EMOF are mainly used to create simple metamodels with a small number of metaclasses. Therefore, the EMOF does neither support the reuse of existing metamodels nor concepts for abstraction. In the following, we introduce the most important language concepts of the EMOF. For ease of understanding, we discuss these concepts based on the class diagram shown in Fig. 2.3.

2.2.3 Package

A package is the container for all model elements of a metamodel; thus, one can conclude that a *Package* represents the specification of a metamodel. The elements directly contained in a *Package* are referred to as packaged elements, which can be accessed via the *packagedElement* property of a *Package*. In addition, further packages may be contained in a *Package*, so a more meaningful structuring of the language concepts a metamodel provides becomes feasible. All *Packages* contained in a *Package* are represented as items of the *nestedPackage* property.

Classifier. Due to the constraints defined for EMOF, a *Package* may contain only certain types of *Classifier*, such as *Class* and *DataType*. A *Classifier* represents a *Type* definition, and it can have a generalization relationship to other *Classifiers*; Thus, a generalization hierarchy between concrete and more general *Classifiers* can be established. A generalization relationship between two *Classifiers* is specified by means of a *Generalization* (see item (1) in Fig. 2.3).

Class. The most important language concept for creating metamodels is *Class*, because this concept represents a metaclass. Since *Class* is a concrete subtype of *Classifier*, a class can specialize other classes that are referenced using the *superClass* property (see Fig. 2.3). The features of a *Class* are divided into structural and behavioural features. An attribute represents a *StructuralFeature* of a *Class*, and it is specified by means of a *Property*. The set of all attributes of a *Class* is represented by the *ownedAttribute* property. By contrast, an *Operation* defines a *BehavioralFeature* of a *Class*. The set of all operations of a *Class* can be obtained via the *ownedOperation* property.

DataType. Just as *Class*, the language concept *DataType* is a specialization of *Classifier*. A *DataType* introduces a type whose instances are identified only by their value. To enable the specification of structured data types, a *DataType* can have attributes defined that are represented by means of *Properties*. In addition, a *DataType* can also have operations.

As a specialization of *DataType*, an *Enumeration* can be employed to introduce enumeration types. The literals of an *Enumeration* are defined employing *EnuermationLiteral*.

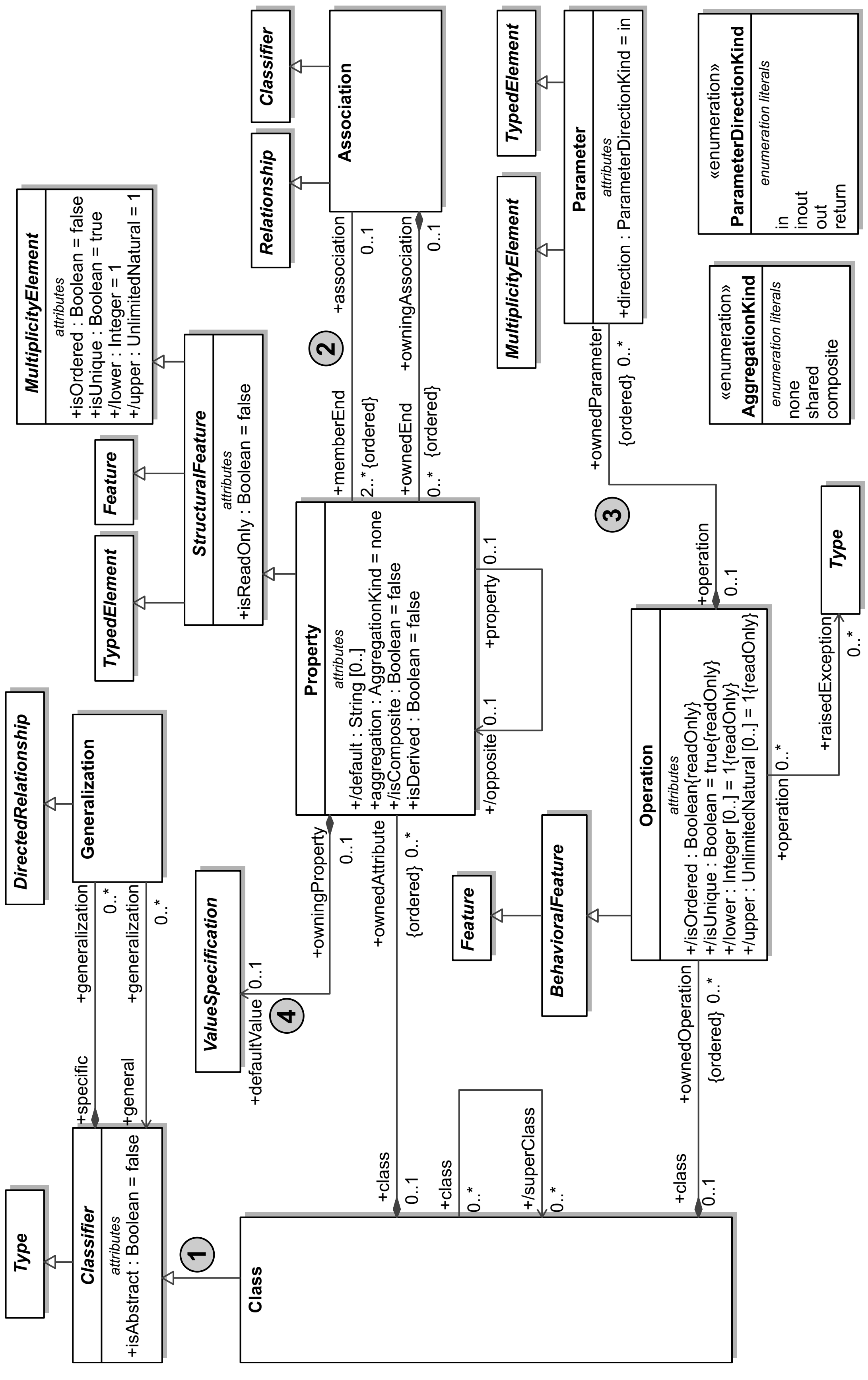


Figure 2.3: The most important part of the abstract syntax of the EMOF [121]

The *PrimitiveType* is another specialization of *DataType*, and it is used to specify predefined data types without regarding details of the internal structure. For example, the primitive types String, Boolean, Integer, Real, UnlimitedNatural are the predefined data types for UML and MOF.

Property. A *Property* represents a structural feature of a *Class* or *DataType*. In this context, a *Property* is referred to as attribute. In addition to the explicit creation as an attribute, a *Property* can also be implicitly introduced by an *Association*, as explained below.

As shown in Fig. 2.3, *Property* is not just an indirect specialization of *Feature*, but it also inherits from *TypedElement* and *MultiplicityElement*. The *type* property is inherited from *TypedElement*, and thus, it represents a reference to a type instance. The properties inherited from *MultiplicityElement* enable a *Property* to represent not only single values or objects (single-valued) but also four different collection types (multi-valued). The cardinality of a *Property* is determined by the properties *lower* and *upper*. Because the default value for both properties is 1, a *Property* is single-valued by default. To define collection types, a value greater than 1 has to be assigned to the *upper* property. In a class diagram, the cardinality of a property is displayed textually according to the following syntax:

```
"[" <lower> ".." <upper> "]"
```

As various variants for displaying the property cardinality are defined (see [116]), the syntax above is only a simplification. Both values of the cardinality have to be specified as numbers, whereby for the upper cardinality an asterisk-sign denotes a collection of infinite size.

Four different types of collections are available for a *Property*. The concrete type is determined by the two properties *isOrdered* and *isUnique*. Table 2.1 shows the relationship between both properties and the collection types. If the *isOrdered* property of a *Property* has assigned the value ‘true’, this is indicated in a class diagram with the keyword ordered. If the *isUnique* property has assigned the value ‘false’, this is displayed using the keyword nonunique. Because of the default values of *isOrdered* and *isUnique*, a multi-valued property always represents a Set by default.

Instead of an explicit value assignment, the *isDerived* property can be used to specify that the value of a *Property* is to be calculated at runtime. If the *isDerived* property has assigned the value ‘true’, the *name* of a *Property* is preceded with a slash in a class diagram, as this is the case, e.g., for all the attributes of the Operation class shown in Fig. 2.3. The information for computing the value of a *Property* can be specified by the *defaultValue* property, which also defines the initial value of a *Property*. In addition, the *isReadOnly* property defines that a read-only access to a *Property* is permitted. In a class diagram, read-only access is represented using the keyword readOnly.

Table 2.1: The possible collection types of a *Property*

Collection type	*isOrdered*	*isUnique*
Set	false	true
OrderedSet	true	true
Bag	false	false
Sequence	true	false

The *aggregation* property determines whether the items of a *Property* are part of a *Classifier* instance (*aggregation* == aggregation) or references (*aggregation* == none) to such instances.

Operation. An *Operation* defines a behavioural feature of a *Class* or *DataType*, and it owns an ordered set of *Parameters* that can be accessed via the *ownedParameter* property. Just as *Property*, *Parameter* is a specialization of *MultiplicityElement* and *TypedElement*. The properties inherited from both language concepts have the same purpose as for *Property*. By contrast, *Operation* does not inherit from both mentioned concepts; instead, it has corresponding derived properties. The values of these properties are determined based on the result *Parameter* of an *Operation*.

Association. An *Association* defines a semantic relationship between two metaclasses, which is denoted as binary association. The instance of an *Association* is referred to as link, which is used to connect objects as concrete instances of *Class*. As pointed out above, the specification of an *Association* between *Classes* implicitly defines *Properties* at each end of the *Association*. These *Properties* can be owned by either an *Association* itself or a *Class* at an end of the *Association*. All *Properties* of an *Association* are referenced by the *memberEnd* property, regardless of their owner. In addition, the *Properties* directly owned by an *Association* are represented as items of the *ownedEnd* property.

In a class diagram, a line drawn between two *Classes* is the simplest *Association* variant, e.g., see the association marked with (2) in Fig. 2.3. The implicitly created *Properties* of this *Association* represent references, i.e. *aggregation* = none. In the case of metamodels, it is assumed that both *Properties* of such an *Association* are owned by the *Classes* at the association ends. The *Association* is considered to be navigable in both directions, i.e. a *Class* at one end can be accessed via the *Property* at the *opposite* end of the *Association*, and vice versa.

In addition to the *name* of a *Property*, all the other textual information described above are displayed at an association end. It has to be noted that from the perspective of a *Class*, its *Property* introduced by an *Association* is always displayed at the opposite end of the *Association*. For example, the *Association* marked with (3) in

Fig. 2.3 has to be interpreted in such a way so that ownedParameter is associated with the Operation class, and operation is related to the Class parameter.

An *Association* can also define a *Property* as a composite aggregation, which is graphically represented as a filled diamond located at the *Class* that owns the *Property*. In other words, the composite aggregation symbol is displayed directly at the association end connected with the owning *Class*. Only one *Property* of an *Association* can be defined as composite aggregation. For example, the composite aggregation symbol of the ownedParameter of the *Association* marked with (3) in Fig. 2.3 is shown at the association end connected to the Operation class.

As shown by item (4) in Fig. 2.3, an *Association* may also have a so-called navigation arrow. This indicates that only navigation in the direction to the *Property* marked with the arrow is possible, but not in the opposite direction. By default, the *Property* displayed at the arrow is owned by a *Class*, and the *Property* at the opposite end is owned by the *Association*. In the case of our example above, defaultValue is owned by Property class, whereas owningProperty is owned by the association.

2.2.4 The Complete MOF

In addition to the language concepts provided by EMOF, the CMOF introduces additional language concepts for the creation of metamodels. As in the case of EMOF, UML class diagrams can be used to create CMOF-based metamodels. However, CMOF-specific constrains apply, so that the following features can be used in addition to those provided by EMOF:

- import and merge of packages;
- redefinition of operations and metaclass attributes (redefinition);
- metaclass attributes can be specified as subsets of other attributes (subsetting);
- ability to define metaclass attributes as derived unions so their values are computed based on other attributes;
- well-formedness rules can be defined in the context of metaclasses;
- use of operations as model queries that extract information from a model through read-only access; and
- derived attributes can have a specification, so their values can be calculated at runtime.

In [7, 134] it is emphasized that especially the CMOF features subsetting and redefinition of attributes enable the use of abstraction in metamodelling. Furthermore, it is argued that package merge and import enable the reuse of language concepts and a modularization of modelling languages.

A brief introduction to the language concepts introduced by CMOF or the changes made to those provided by EMOF is given below. Fig. 2.4 illustrate the abstract syntax of the main language concepts already provided by EMOF (see Fig. 2.3) including the features added by CMOF. The shown diagram can also be considered as a good example of a CMOF-compliant metamodel, because it illustrates the application of some language concepts of the CMOF. This is possible due to the self-description of CMOF.

ElementImport. The import of elements from an existing metamodel can be realized for CMOF-based metamodels using *ElementImport*. The elements imported in this way do not become part of the importing metamodel because the import mechanism relies on references. Thus, a modification of an imported element is not possible. However, imported *Classifiers* can be referenced as *type* of *TypedElements*, such as *Property*. In addition, a further specialization of imported *Classifiers* by means of inheritance is possible. An imported element becomes part of the namespace of the importing metamodel.

PackageImport. If not only single elements but all elements of a *Package* have to be imported, this can be realized with *PackageImport*. The same rules apply as above for *ElementImport*.

PackageMerge. The package merge is another mechanism that can be employed to reuse parts of existing metamodels. In contrast to importing single elements or packages, *PackageMerge* combines the elements contained in two *Packages* into one new *Package*. First, the content of the first *Package* is copied to the new one. Then, each element of the second *Package* is copied, as far as no element with the same *name* exists in the *Package* to be created. If this is the case, the features of the element to be copied are added to the already existing element.

For example, until the release of UML version 2.5 [116], *PackageMerge* was used extensively to create the UML metamodel, as its specification in previous UML versions was split into different 'language units' (e.g, see [115]). Each unit is represented as a *Package* and contains interconnected language concepts. Based on the individual language units, four different levels of compliance are defined. A UML metamodel which complies to a distinct level can be created by merging the language units required for that level. In addition, *PackageMerge* is used for abstracting the specification of UML language concepts. Starting from the 'kernel' package, which defines basic language concepts such as *Classifier*, additional features are added to the metaclasses contained in other language units.

Property. CMOF supplements the language concept *Property* with additional features that allow a *Property* to represent a derived union, to be defined as a subset of other properties, or to redefine other properties. All mentioned features are applicable not only individually, but also in combination to a *Property*.

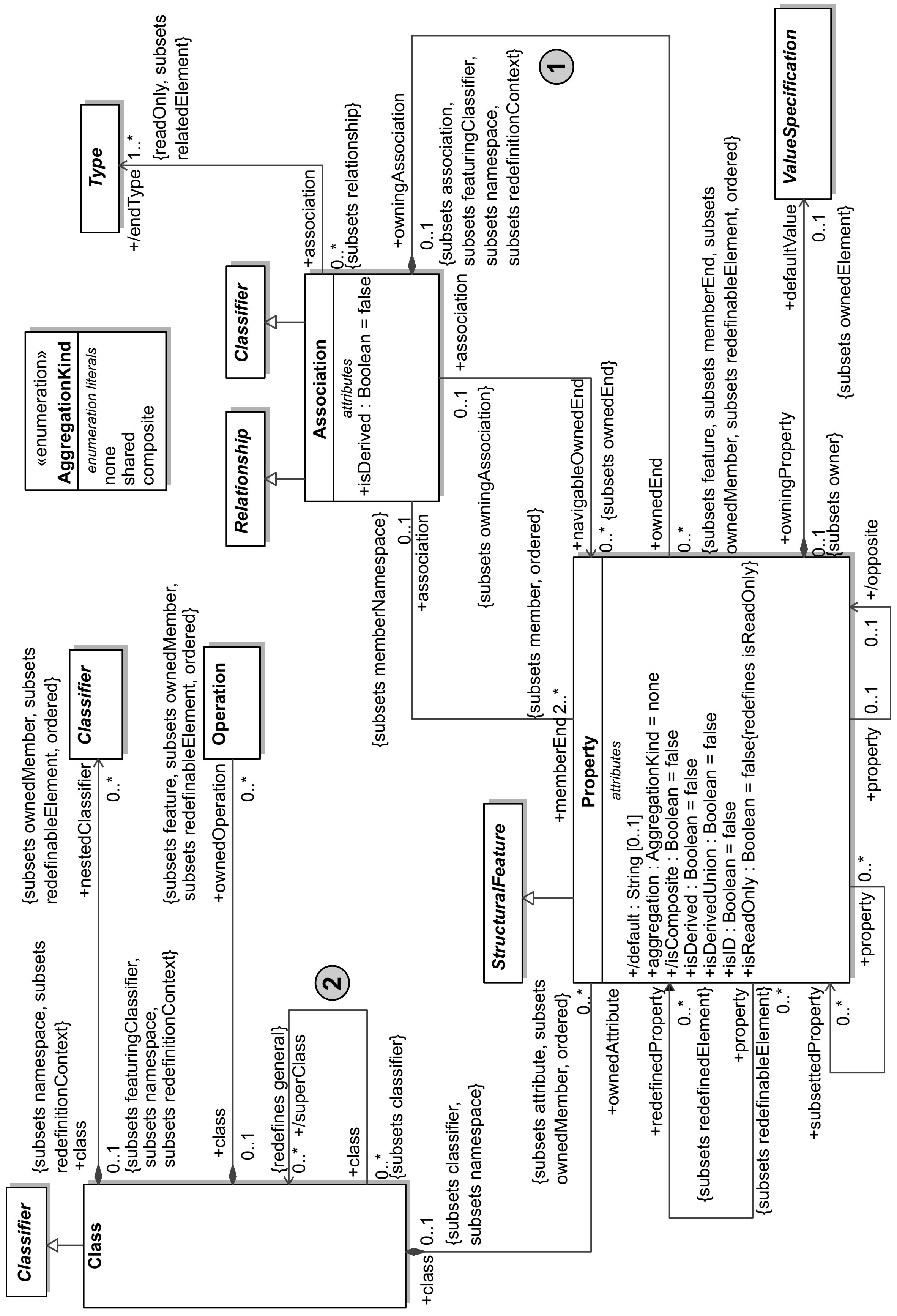

Class
Classifier
Operation
Classifier
StructuralFeature
Property
attributes
+/default : String [0..1]
+aggregation : AggregationKind = none
+/isComposite : Boolean = false
+isDerived : Boolean = false
+isDerivedUnion : Boolean = false
+isID : Boolean = false
+isReadOnly : Boolean = false{redefines isReadOnly}
Relationship
Classifier
Association
attributes
+isDerived : Boolean = false
«enumeration»
AggregationKind
enumeration literals
none
shared
composite
Type
ValueSpecification
+/endType 1..*
{readOnly, subsets relatedElement}
+association 0..*
{subsets relationship}
+owningAssociation 0..1
{subsets association, subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+association 0..1
{subsets memberNamespace}
+memberEnd 2..*
{subsets member, ordered}
+association 0..1
{subsets owningAssociation}
+navigableOwnedEnd 0..*
{subsets ownedEnd}
+ownedEnd 0..*
{subsets feature, subsets memberEnd, subsets ownedMember, subsets redefinableElement, ordered}
+owningProperty 0..1
{subsets owner}
+defaultValue 0..1
{subsets ownedElement}
+/opposite 0..1
+property 0..1
+property 0..*
+subsettedProperty 0..*
+redefinedProperty 0..*
{subsets redefinedElement}
+property 0..*
{subsets redefinableElement}
+ownedAttribute 0..*
{subsets attribute, subsets ownedMember, ordered}
+class 0..1
{subsets classifier, subsets namespace}
+nestedClassifier 0..*
{subsets ownedMember, subsets redefinableElement, ordered}
+class 0..1
{subsets namespace, subsets redefinitionContext}
{subsets featuringClassifier, subsets namespace, subsets redefinitionContext}
+ownedOperation 0..*
{subsets feature, subsets ownedMember, subsets redefinableElement, ordered}
+class 0..1
+/superClass 0..*
{redefines general}
+class 0..*
{subsets classifier}
1
2

Figure 2.4: The most important part of the abstract syntax of the CMOF [121]

Subsetting: Based on the foundations of the set theory, the language concept *Property* is extended in CMOF so that a *Property* pSub can be declared as a subset of another *Property* pSuper. In the context of UML – and thus also in the MOF – pSub is called *subsetting property* and pSuper is referred to as *subsetted property*. The *subsettedProperty* property is used to declare the *Properties* that shall be subsetted. Since *subsettedProperty* is multi-valued, a not only a single but also multiple *Properties* can be subsetted.

For a subsetting *Property* pSub and for the subsetted *Property* pSuper the following constraints apply:

- the *names* of pSub and pSuper must not be equal;
- pSub must be contained either in the same *Classifier* C as pSuper or in a subtype of C;
- The *type* of pSub must conform;[1] to that of pSuper;
- the *upper* cardinality of a pSub must be less than or equal to that of pSuper.

The implementation and formalization of new language concepts provided by CMOF were analysed and demonstrated by some works, e.g., [5, 8, 9, 134]. It was criticized that the semantics for property subsetting in UML, and thus also for MOF, was only specified very vaguely. In particular, this concerns the processing of the four collection types which a *Property* can represent and the rules for computing the values of a subsetted *Property*. For this reason, the semantics was revised for the current UML version 2.5 [116]. Now it is assumed that, independent of the collection type of a subsetting property pSub and its subsetted property pSuper, a set of values has to be determined for both properties. Both sets are obtained by removing all duplicates, and all elements of the value set of pSub must be contained in the value set of pSuper.

In a class diagram, the keyword subsets followed by the name of a subsetted *Property* pSuper is displayed close to a subsetting *Property*. Each subsetted *Property* is displayed separately. If a subsetting *Property* is an explicitly defined metaclass attribute, the information is shown subsequent to the attribute specification. If the *Property* represents an association end, the information is displayed close to that end, e.g., see the association marked with (1) in Fig. 2.4.

Derived union: A *Property* represents a strict union of all *Properties* subsetting it, if its *isDerivedProperty* property has assigned the value 'true'. The value of a *Property* that is a strict union is computed based on its subsetting *Properties*. For this reason, a manual value assignment to such a *Property* shall not possible. Therefore, the properties *isDerived* and *isReadOnly* of the *Property* must have assigned

[1] According to UML [116], *"... one Type conforms to another, if any instance of the first Type may be used as the value of a TypedElement whose type is declared to be the second Type."*

the value 'true'. In a class diagram, a *Property* defined to be a derived union is annotated with the keyword union.

Redefinition: A *Property* of a *Classifier* can redefine a single or multiple other inherited *Properties*, which have to be referenced by the *redefinedProperty* property. The name and visibility of a *Property* have not match with the redefined *Properties*. Redefinition can be used to define or modify the *defaultValue* for a *Property*. A redefining *Property* can be declared as derived, regardless of the redefined *Properties*. Furthermore, the *type* of a *Property* can also be modified by redefinition. However, the *Classifier* referenced by the *type* property must always conform to that of the redefined *Properties*. Moreover, a redefining *Property* can restrict the cardinality.

In a class diagram, a redefinition is displayed textually using the keyword 'redefined' followed by the name of the redefined *Property*. Each property listed in the *redefinedProperty* property is displayed separately. For instance, a redefining association is shown by item (2) in Fig. 2.4.

Operation. In contrast to EMOF, the language concept *Operation* can have defined pre-, post- and body conditions. Each of these conditions is specified by means of a *Constraint*, which may include an OCL expression. If the *isQuery* property of an *Operation* has assigned the value 'true', this specifies a model query that must not change the state of a metamodel during the invocation. Furthermore, an *Operation* of a *Classifier* can redefine an inherited operation.

Conditions: The purposes of the three condition types which can be defined for an *Operation* are as follows:

- *Body condition:* The *bodyCondition* property of an *Operation* is used to specify a machine-processable specification for computing the result of a model query.
- *Pre-conditions:* An *Operation* can only be invoked if all *Constraints* defined by the *precondition* property are successfully evaluated.
- *Post-conditions:* The result of an *Operation* computed by the body condition must satisfy all *Constraints* specified by the *postCondition* property.

Redefinition: An *Operation* can redefine a single or multiple inherited *Operations*, which have to be referenced by the *redefinedOperation* property. According to the UML specification [116, p. 155], the *bodyCondition* of an *Operation* can be modified by redefinition, whereas only additional *Constraints* can be added to the pre- and post-conditions. Furthermore, the *type* of a *Parameter* of a redefining *Operation* can be changed; but it must conform to that of the redefined *Operation*.

It is required that the number of *Parameters* of a redefining *Operation* must be the same as that of the redefined *Operation*. Furthermore, the direction and the collection type of the *Parameters* of the redefining *Operation* have to match with those of the redefined *Operation*.

Constraint. A *Constraint* is used to specify conditions or constraints in natural language or a computer language, such as OCL. In CMOF-based metamodels *Constraints* are employed to define well-formedness rules for metaclasses. In addition, the pre-, post-, and body conditions of an *Operation* are specified as *Constraints*. The *specification* of a *Constraint* consists of an *OpaqueExpression*, which contains the concrete expression or statement expressed in terms of a String.

2.3 Using OCL for Specifying the Static Semantics

Well-formedness rules as part of the static semantics are specified by *Constraints* in a metamodel, but the language for defining these rules remains open in the MOF specification [121]. Due to this reason, for example, we could use natural language to specify the rules. However, inaccuracies and the lack of computerized processing have to be taken into account as possible disadvantages of such an approach. These drawbacks can be avoided by using the *Object Constraint Language (OCL)*.

According to the OCL specification [120], only a subset of the language constructs provided by OCL can be employed for metamodelling. Because this topic is a major objective of our research, below we discuss only OCL language constructs that are applicable to metamodels.

OCL expressions are used in class diagrams to define invariants, queries on objects of a model, pre- and post-conditions for operations, and the initial or runtime values of properties. Since the MOF, and the metamodels based on it, are specified by means of UML class diagrams, OCL can be used not only for UML models but also for all MOF-based metamodels. Apart from class diagrams, OCL expressions are also applicable for other UML diagram types, e.g., state machine diagrams. Because of the objective of our research, we only consider the application of OCL in class diagrams.

OCL type system. OCL is a typed language, and therefore, each OCL expression is associated with a particular type. The evaluation of an OCL expression always returns a value that must conform to the type of expression. The type system of OCL shown in Fig. 2.5 is similar to that of the MOF. Each *Classifier* instance contained in a model is represented in OCL as an instance of the corresponding OCL type. Attributes and operations of a *Classifier* can be accessed in OCL expression using the '.' navigation operator.

```
(A) clsInstance.attA
(B) clsInstance.operationA()
```

The type system of OCL is closely linked to a set of predefined data types, such as Boolean or Integer, which are part of the *OCL Standard Library* discussed below. In contrast to the MOF, multi-valued attributes of a *Classifier* are represented as explicitly defined collection types in OCL. For this purpose, the OCL type system provides the four collection types Set, OrderedSet, Sequence and Bag. In contrast

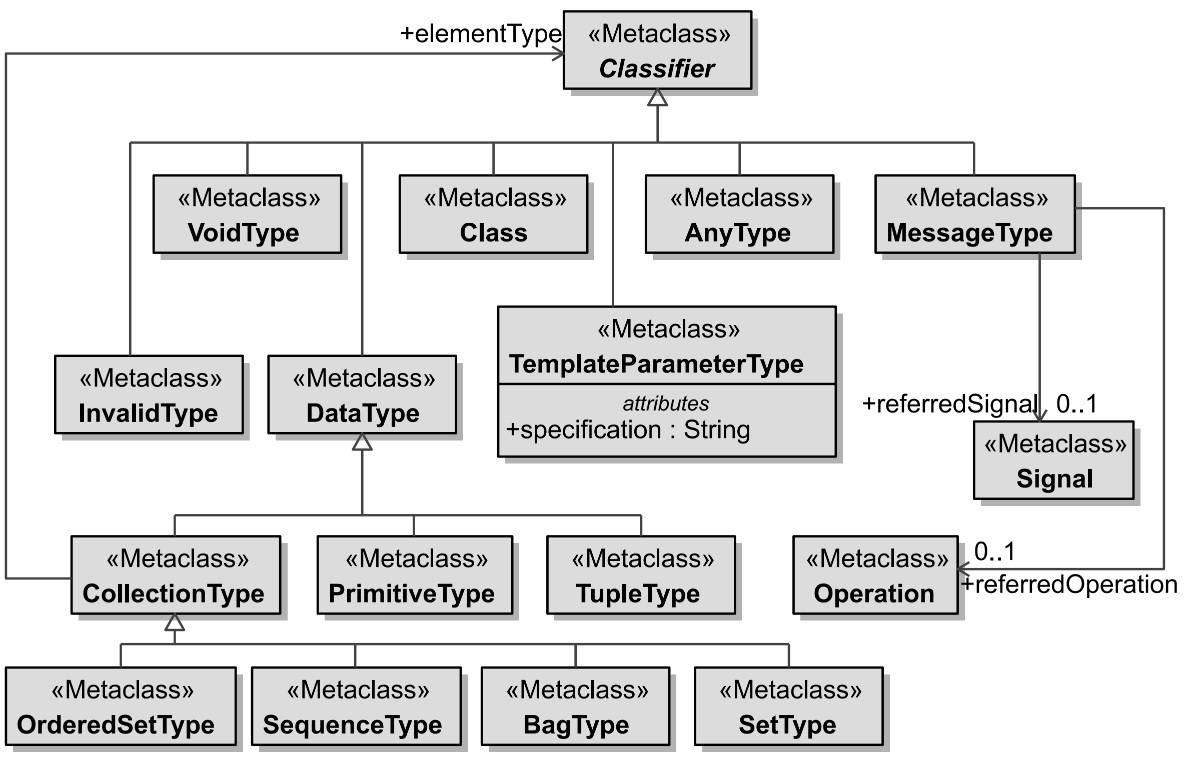


Figure 2.5: The metaclasses defining the type system of OCL adapted from [120]

to the MOF, the OCL provides a tuple type that enables the definition of lists of arbitrary size, where each list item has a specific type and an optional name. An item of such a tuple type can be accessed in the same way as an attribute of a *Class* or *DataType* employing the '.' navigation operator.

The OCL standard library. The predefined types of OCL are contained in the *OCL Standard Library*, which also defines the operations applicable to these types, such as the operations oclIsTypeOf(T) and oclIsKindOf(T). The first operation returns the value 'true' if the type of an OCL expression exactly matches the type specified by T, while the second returns this value for subtypes of T as well.

```
clsInstance.oclIsTypeOf(AClass)
```

Most of the operations defined for the primitive types of OCL represent infix operators, e.g., logical or arithmetical operators, used as infix notation in OCL expressions. The following example shows the application of the Boolean and operator and of the comparison operator '=':

a **and** b = **true**

Furthermore, the OCL Standard Library also provides operations that are applicable only on collection types. Many of these operations correspond to the operators known from the set theory. For example, the union operation returns the union of two collections as result. Operations calls on collections are mapped to iterator expressions in the abstract syntax of OCL. In contrast to attributes of a

Classifier, the navigation operator '->' is employed to invoke a collection operator as shown in the following example:

```
collA−>union(collB)
```

Context. An OCL expression must always be defined in the context of a specific *Classifier* or one of its *Features*. If OCL expressions are not part of a model but are defined in a standalone OCL document, the OCL expression must always be preceded by a context declaration, which starts with the **context** keyword.

In the case of invariant constraints, the context declaration consists only of the name of the relevant *Classifier*, whereas OCL expressions for a *Property* or a query *Operation* require further information. If an OCL expressions is embedded in a model, its context is derived from its position in the model and thus does not have to be explicitly specified. Since the CMOF enables the embedding of OCL expressions into metamodels, and we only cover CMOF-based metamodels in our work, we only consider this use case of OCL.

The **self** keyword can be used in OCL expressions to access an implicit variable that holds the particular instance of the context classifier. In OCL expressions, it is not mandatory to use of the **self** keyword for accessing the attributes and operations of the current context classifier. The following example represents an attribute navigation starting from the context classifier to its attribute attA, once with and once without a preceding **self** keyword.

```
self.attA ⇔ attA
```

If an OCL expression is specified in the context of an operation, then, in addition to the self variable, further implicitly defined variables for the parameters and the result of the operation are available. The access to such a variable takes place via the name of the corresponding parameter.

Invariant constraints. Invariants are *Constraints* that must be met by all instances of a *Classifier* throughout the runtime. The *ownedRule* property of a *Classifer* contains all the invariant *Constraints* defined for this *Classifier*.

An invariant is specified by means of an OCL expression with *Boolean* result type. Typically, invariants are used to restrict the set of valid values for an attribute according to defined rules. For example, the following invariant specifies that value of the name attribute of MyClass must be at least 10 characters long.

```
context MyClass
inv: self.name.size() >= 10
```

Initial and derived value expressions. OCL expressions can define the initial or runtime value of a *Property*. In both cases, the textual notation of such an OCL expression is represented as an *OpaqueExpression* that is owned by the *Property*. The result type of the OCL expression must be type-compliant with the *type* of the

Property. Below, we give two examples for specifying the initial and derived value using OCL:

(A) **context** MyClass::name : **String**
  init: 'InitName'

(B) **context** MyClass::name : **String**
  derived: 'StaticName'

Query operations. An *Operation* that is marked as query can be specified using an OCL expression, and its result is determined by evaluating this OCL expression. As we pointed out for the context of expressions, the values passed via the parameters of an *Operation* can be accessed in OCL expressions via implicitly defined variables. The evaluation result of the query is implicitly assigned to the result parameter. The following example shows a query that returns the current value of the name attribute of the MyClass class.

context MyClass::getClassName() : **String**
body: **self**.name

Pre- and post-conditions. As we mentioned for the MOF, an *Operation* may have defined pre- and post-conditions. *Constraints* that define these conditions are specified in same manner as invariants, so they consist of an OCL expression with Boolean result. In contrast to the *bodyCondition* of an *Operation*, OCL expressions that define a pre- or post-condition can access not only the self variable but all operation parameters, including the result parameter.

2.4 Model Transformations

In addition to metamodelling, model transformations are a further key technology for the MDE-based development of DSLs [93, 136, 140]. In the past, various approaches to realize model transformations for different application domains have been proposed. A general overview and classification of these approaches based on various criteria can be found in the literature, e.g., in [30, 31, 104]. To classify the model transformations which we introduce in the following chapters, we briefly discuss the criteria required for our work.

One of the main categories by which transformation approaches are classified in the literature is the kind of artefacts that serve as input to transformations or are generated as output. In general, these artefacts are models or text that usually represents program code. Therefore, we can distinguish the following transform types: *text-to-model (T2M)*, *model-to-model (M2M)*, and *model-to-text (M2T)*. Because we wish to research on the automated derivation of UML profiles from metamodels, M2M transformation approaches are mainly relevant to us. Since another objective of our work is to derive M2M transformations that enable model interoperability, we also consider M2T transformation approaches which can be employed to generate program code for such M2M transformations.

A further classification criterion discussed in the literature depends on the metamodels used to instantiate the input and output models of M2M transformations. If the input and output models are instances of the same metamodel, an M2M transformation is referred to as *endogenous transformation*, whereas an M2M transformation whose input and output models are instances of different metamodels is called *exogenous transformation*.

In-place transforms are a special variant of endogenous transformations because they do not create new models but modify the input models. Thus, an output model of such a transformation is the modified input model. Consequently, both models are instances of the same metamodel, and this is the characteristic of an endogenous transformation. In contrast to in-place transformations, *out-place transformations* create new output models based on the input models. Since the models involved can be instances of the same or different metamodels, an out-place transformation can be either an endogenous or an exogenous transformation.

Another classification criterion for model transformations is the level of abstraction on which their input and output artefacts (i.e. models or text) reside. Model transformations whose input and output artefacts are located at the same level of abstraction are referred to as *horizontal transformations*. For example, a DSL model and a corresponding UML model with applied UML profile reside on the same level of abstraction. Therefore, a M2M transformation that transforms the former into the latter model or vice versa can be considered to be a horizontal transformation. In contrast, model transformations whose input and output artefacts are located on different levels of abstraction are *vertical transformations*. For instance, an M2T transformation that generates program code based on a model can be considered as vertical transformation.

In addition to the transformation approaches discussed in the literature, various transformation languages are proposed for the implementation of model transformations (e.g., in [42, 75, 76, 95]). Depending on the application scenario, transformation languages support different programming paradigms such as imperative, relational or template-based. As we already highlighted in the motivation for our work, we wish to implement our derivation approach employing standardized technologies. Therefore, we use one of the three transformation languages standardized in the QVT specification [113] to implement M2M transformations, while the *MOF Model for Text Transformation Language (MTL)* [111] is employed by us to realize M2T transformations. In the following, we give a brief introduction to both standards.

2.4.1 Model-to-Model Transformations

The QVT specification [113] embraces the three interrelated transformation languages *'Core'*, *'Relations' (QVTr)* and *'Operational Mappings' (QVTo)*. These languages either reuse or supplement language constructs of OCL.

According to Kurtev [95], the Core language and QVTr are declarative transformation languages at different levels of abstraction. The Core language is used to define the semantics of QVTr, and it provides only a fewer number of language concepts compared to the latter. By contrast, QVTr was developed for the specification of model transformations by end users.

Transformations defined by QVTr specify a set of relations between metaclasses. Provided the relations of such a transformation do not specify a particular transformation direction, bidirectional mappings can be realized. As pointed out in [31, 33], just as all other relational transform approaches, QVTr is free from side effects. Elements for an output model can only be implicitly generated because QVTr does not provide language constructs to explicitly create them. Thus, model transformations specified by means of QVTr are not applicable to solve all kinds of transformation problems.

To prevent the limitations of QVTr, the relations in QVTr transformations can alternatively be implemented using imperative language concepts of QVTo. In the literature [31, 33], this kind of model transformations is referred to as *hybrid transformations*, because more than one transformation language is used to implement them. Furthermore, model transformations can also be specified using pure QVTo. According to the QVT specification [113], such a transformation is denoted as *operational transformation*. However, it should be noted that such transformations are only unidirectionally applicable, whereas pure QVTr transformations are bidirectional.

From a theoretical point of view, in addition to the Core language, we have three options for realizing our approach to automatically derive UML profiles employing QVT. We could implement the required M2M transformations in pure QVTo or QVTr. Alternatively, we could also choose a hybrid transformation approach. Because of the aspects we discuss in Chap. 4, we opted for a QVTo-based solution, and therefore, only the details of QVTo are introduced below.

The 'Operational Mappings' language. QVTo, in contrast to QVTr, offers not only the ability to explicitly create model elements, but also language constructs for directing the control flow of a transformation, such as loop constructs. The imperative expressions of QVT extend those of OCL which have become part of QVTo. In the following, we explain the basic structure of QVTo transformations using the example of the Book2Publication transformation shown in Fig. 2.6, which transforms a Book element into a Publication element. Since the input and output models are instances of different metamodels, our example is an exogenous out-place transformation.

```
1  // Optional metamodel specifications
2  metamodel BOOK {
3    class Book {title: String; composes chapters: Chapter [*];}
4    class Chapter {title : String; nbPages : Integer;}
5  }
6  metamodel PUB {
7    class Publication {title : String; nbPages : Integer;}
8  }
9
10 // Transformation signature
11 transformation Book2Publication(
12 in bookModel:BOOK, out pubModel:PUB);
13
14 // Transformation main() operation
15 main() {
16   bookModel−>objectsOfType(Book)−>map book_to_publication();
17 }
18
19 // Mapping operation
20 mapping Book::book_to_publication () : Publication
21 when { self.nbPages > 0 } // Guard condition
22 {
23   title := self.title;
24   nbPages := self.chapters−>nbPages−>sum();
25 }
```

Figure 2.6: QVTo transformation example (adapted from [113])

General structure: In the first part of our example two metamodels BOOK and PUB are defined, including their metaclasses. The specification of metamodels is optional and is only required if the metamodels associated with a transformation are not implemented by the platform on which a transformations is conducted.

The second part of a transformation consists of the mandatory operation signature, which embraces the name and the input and output parameters of the transformation. A parameter of the signature always consists of a parameter direction (**in**, **out** or **inout**), a name and a type, which has to refer to a metamodel. For instance, the bookModel parameter in Line 12 of our example is an input parameter, with the Book metamodel defined as parameter type. All parameters of a transformation signature have global visibility, and thus, they can be accessed in any operation of a transformation.

The main() operation specified in Line 15 of our example is used to initialize the transformation and to invoke the topmost mapping operation, which usually processes the root element of the input model. After invoking a transformation, the main() operation is called implicitly before all others. The body of this operation may include an ordered list of imperative expressions. Typically, these expressions are used to select the model elements to be transformed. Afterwards, these elements are passed to mapping operations which transform them into corresponding elements of the output model.

The main() operation of our example consists of a single imperative expression that selects and transforms all Book elements. To transform these elements, the mapping operation book_to_publication() is invoked. As shown in Line 16, the invocation of a mapping operation is specified using the **map** keyword.

Mapping Operations: The most important part of QVTo transformations are mapping operations, which are specified below the main() operation. A mapping operation usually consists of a signature, an optional guard condition, and an operation body containing a list of imperative expressions. Our transformation example given in Fig. 2.6 contains exactly one mapping operation, which is specified in Line 20. This mapping operation is an out-place mapping of a Book element to a Publication element in the output model. In contrast, an in-place mapping operation can modify attributes of an input element while not being able to create a new element in an output model.

Among other parts, the signature of a mapping operation embraces an optional context parameter, an operation name, optional operation parameters and optional result parameters. A signature definition always starts with the keyword **mapping**. Subsequently, the type of the context parameter (i.e., a metaclass or *DataType*) can be defined. If specified, this parameter can be accessed using the keyword **self** in the operation body. For instance, the signature of the book_to_publication() mapping operation of our example has defined a context parameter that is of type Book.

Following the context parameter type, separated by two colons, the name of the mapping operation is specified. Then, a list of operation parameters can be defined within opening and closing brackets. Finally, the optional result parameters are specified at the end of a signature. In the operation body, these parameters can be referenced using the keyword **result**. For example, the signature of the book_to_publication() mapping operation introduces one result parameter of type Publication but no operation parameters.

All parameters of a mapping operation have a parameter direction, which can optionally be specified using the keywords **in**, **inout** and **out**. By default, the direction for context and operation parameters is defined as **in**, whereas the direction of output parameters must always be **out**. In the signature of our book_to_publication() example, we have not explicitly defined the direction of the context parameter, and thus, its direction is **in** by default.

The parameters defined in a signature and their directions also determine whether the associated mapping operation is an in-place or out-place mapping. Since the mapping operation of our example has both a context parameter with a direction **in** and a result parameter, this is an out-place mapping. However, if a context parameter with a direction **inout** and no result parameter is specified for an operation signature, the associated mapping operation is an in-place mapping.

An optional guard condition may be specified below the signature of a mapping operation. A guard condition starts with the keyword **when** and has an OCL expression with a Boolean result type. A mapping operation is only processed if the evaluation of its guard condition, if any, returns the value 'true'. For instance, the mapping operation of our example transformation has defined a guard condition in Line 21.

Typically, the operation body of a mapping operation consists of a list of imperative expressions which assign values to the attributes of an output element. Except of values for primitive types, such as String, values of source element attributes cannot be simply assigned to attributes of the output element. This is because the input and output elements of a mapping operation are usually instances of metaclasses that belong to different metamodels. For this reason, the values of a source element must be transformed by invoking appropriate mapping operations before being assigned to attributes of the output element.

For instance, the mapping operation in Line 20 of Fig. 2.6 includes two assignment expressions which directly assign values to attributes of the output element. Since both attributes have a primitive data type defined as type, no mapping operations have to be invoked before the assignments. However, the invocation of a mapping operation employing the **map** keyword is shown in Line 16 of our example.

In addition to assignment expressions, QVTo provides imperative expressions for object resolution, which is an important language concept for mapping operations. Using this kind of expressions, it is possible to determine model elements already created by mapping operations. For instance, based on a created element in the output model, the associated source element in the input model can be determined, and vice versa. Object resolution expressions always start with the keyword **resolve**.

Other kinds of operations: Apart from mapping operations, QVTo provides query and helper operations. QVTo queries are similar to those of OCL, but allow the use of imperative expressions in addition to OCL expressions. Just as OCL queries, QVTo queries must be free of side effects and cannot modify model elements. By contrast, QVTo helper operations can be used to modify elements passed by parameters. For example, the attribute values of an element passed using parameters may be modified by a helper operation.

2.4.2 Model-to-Text Transformations

As argued above, we consider to employ the *MOF Model for Text Transformation Language (MTL)* [111] to implement M2T transformations. MTL is based on a combination of parametrized text templates and OCL expressions. Model elements are passed as parameters to the templates, and OCL expressions are used to control the program flow of the transformation and to define queries. Below, we briefly introduce the fundamental language constructs of MTL, while the details of how we apply MTL are discussed in the next chapter.

Template: A 'template' defines a particular portion of text which can contain placeholders. Furthermore, a list of parameters can be defined for a 'template' so model elements can be passed when a 'template' is invoked. The first item of the parameter list is of particular interest, because it defines the context element of a 'template'.

The placeholders of a 'template' consist of 'template expressions', which are evaluated and replaced by the obtained result. However, only those model elements which are passed as parameter arguments can be accessed by 'placeholder expressions'. Moreover, a 'template' may have defined a Boolean guard condition so that it is invoked only if the guard evaluation returns the value 'true'.

Multiple templates can have the same name as long as different guard conditions are defined for them. Based on a set of templates having the same name, the template is invoked whose guard condition evaluates to a value of 'true'.

The textual notation of a 'template' is shown below. The parameter list enclosed by parentheses is always defined directly after the template name. Then, the optional guard condition which starts with a question mark is defined. The template body which consists of any kind of text and 'template expressions' is specified after the header of the template. The end of the template body is denoted by a completion mark.

```
[template templateName(pt : PT) ? ( optGuardExp )]
... template text ...
[/template]
```

Template expression: A 'template expression' is employed to determine the text to be used instead of placeholders. Hence, the result of a template expression must always be a String value. The textual notation of a 'template expressions' is shown in the first code line below.

A 'template invocation' as shown in the second line is a particular variant of a 'template expression'. Based on an expression whose type must match the context parameter, a template with the specified name is invoked. Arguments for further template parameters can be defined within enclosing brackets after the template name.

```
1.) [ exp /]
2.) [ elem.templateName( argExp ) /]
```

For block: A 'for block' is a statement which generates a particular text segment multiple times based on a collection of model elements. The processing of the 'for block' continues until all elements of the collection have been processed iteratively.

As shown below, the header of a ‘for block’ statement consists of a loop variable and an initialization expression which defines the collection of model elements to be processed. The loop variable holds the current element for a loop iteration, and the loop body specifies the text to be generated at each loop iteration.

```
[for (loopVar : T | initExp)]
... loop body text ...
[/for]
```

If block: If a text segment shall be processed only under a certain condition, this can be specified by an ‘if block’ statement. As shown below, such a statement consists of a Boolean expression and two different text segments. If the Boolean expression evaluates to ‘true’, the ‘consequence’ body is processed and otherwise the ‘else’ body, if present.

```
[if ( booleanExp )]
... consequence body text ...
[else]
... optional else body text ...
[/if]
```

Query definition: MTL queries have the same purpose as OCL, and they also consist of a signature and an OCL expression that defines the body of the query. However, the signature of an MTL query differs from that of an OCL query.

The signature of an MTL query does not start with the type of context parameter. Instead, this parameter must be explicitly defined as the first element in the signature’s parameter list. In our example below, the context parameter is named pt. The mandatory result type must be specified after the parameter list, and then the end of the signature is indicated by an equal sign. The body of the query consists of an OCL expression with a result type that must be type-compatible with that defined in the query signature.

```
[query queryName(pt: PT) : RT =
... body expression ...
/]
```

Escaping and comments: Since MTL statements are enclosed in square brackets, an escaping must occur if square brackets are specified as part of the text to be generated by a ‘template’. The required escaping sequence is shown in the first code line below, and the second line shows a comment which shall not be part of the generated text.

```
1) [ '[' /]
2) [* Comment text */]
```

2.4.3 Notational Conventions for Model Transformations

In the following chapters of our work, we describe the metamodel-based derivation of UML profiles and model transformations using pseudocode. Since these derivations are implemented by M2M and M2T transformations, we employ two different pseudocode notations.

Notation employed for M2M transformations. Our notation for specifying the pseudocode of the M2M transformation is based on a subset of QVTo. In derogation from the previously introduced language constructs of QVTo, we use the following concrete syntax for our notation:

$\mathrm{ST} \Rightarrow opName()$:RT introduces an out-place mapping operation that defines the mapping of an element (of type ST) in the source model to an element (of type RT) of the target model. The operation header is followed by an optional guard condition ("when { guard exp }") and an operation body.

$\mathrm{T} \rightleftharpoons opName()$ introduces an in-place mapping operation that is used to modify an existing element (of type T). This type of mapping operation also has an operation body and an optional guard condition.

query T::opName() : RT defines a query operation with result type RT for a model element of type T.

helper opName() : RT specifies a helper operation with result type RT without having a context type T.

self refers to the implicit context variable of type T that holds the current source element of a mapping operation, helper operation or query.

result refers to the implicit result variable of an 'out-place' mapping operation.

var varname : T := optValue introduces a variable varname of type T and initializes it with the result of optValue. Provided optValue is present, the variable type T can be omitted, because it is derived from the result type of optValue.

srcExp.att / srcExp.opName() is used to access an attribute att or invoke an operation opName().

srcExp->opName() invokes a predefined operation opName() of a collection type.

srcExp.map opName() / srcExp->map opName() is used to invoke a mapping operation opName() for a single-valued attribute or for a collection of elements.

srcExp.resolve(T) returns the element of type T that has been created during the mapping of that element returned by expression srcExp; or, for a mapped element its corresponding source element of type T is returned.

srcExp.asType(T) is used to type-cast an expression to a type T.

srcExp.isType(T) / srcExp.isKind(T) are used to determine whether the result of srcExp exactly matches type T, or to determine whether the result type is a subtype of T.

obj := exp / obj += exp are used to assign the value of exp to an element or a variable. In case of collection values, the first operator := overwrites already present values, whereas the second operator += appends the new values.

$\wedge$, $\vee$, ! represent the well-known Boolean operators. We use them in expressions instead of the corresponding OVTo keywords such as **and**.

Notation employed for M2T transformations. To explain M2T transformations that implement certain parts of our derivation approach, in the following chapters we use pseudocode that is based on the concrete syntax of MTL. In addition to the language concepts of MTL introduced in the previous section, we use the constructs discussed below.

Query definition: Instead of using MTL's notation for model queries, we employ the notational conventions as defined above for queries in M2M transformation.

Global variables: Even if MTL does not support to definition of global variables, we use them in our pseudocode as a shorthand notation to access attributes of stereotypes, which are applied to model elements used as input for our M2T transformations. Hence, we presume that all attributes of stereotypes are implicitly available as global variables having the same names as the corresponding stereotype attributes. For instance, a stereotype attribute st_att is considered to be accessible via the global variable ST_ATT in a M2T transformation.

2.5 The Unified Modeling Language and UML Profiles

The origin of the UML dates back to the second half of the 90s. Beginning with UML version 2.0, syntax and static semantics are defined using a CMOF-based metamodel. The latest version of UML is released as UML 2.5 [116].

Various modelling methods are summarized under the umbrella of UML, which are applicable by means of different UML diagram types. Each of these diagram types has a distinct objective and enables a specific view on a model. The UML provides 14 diagram types which can be divided into two groups, as shown in Fig. 2.7. The group of *structure diagrams* enables the specification of static aspects of a model, while the *behaviour diagrams* define the dynamic aspects of that model.

In addition to the class diagrams we already covered in the context of the MOF introduction, below we briefly explain the diagram types relevant to our thesis at hand. Since UML profiles are highly relevant to our work, we treat them separately

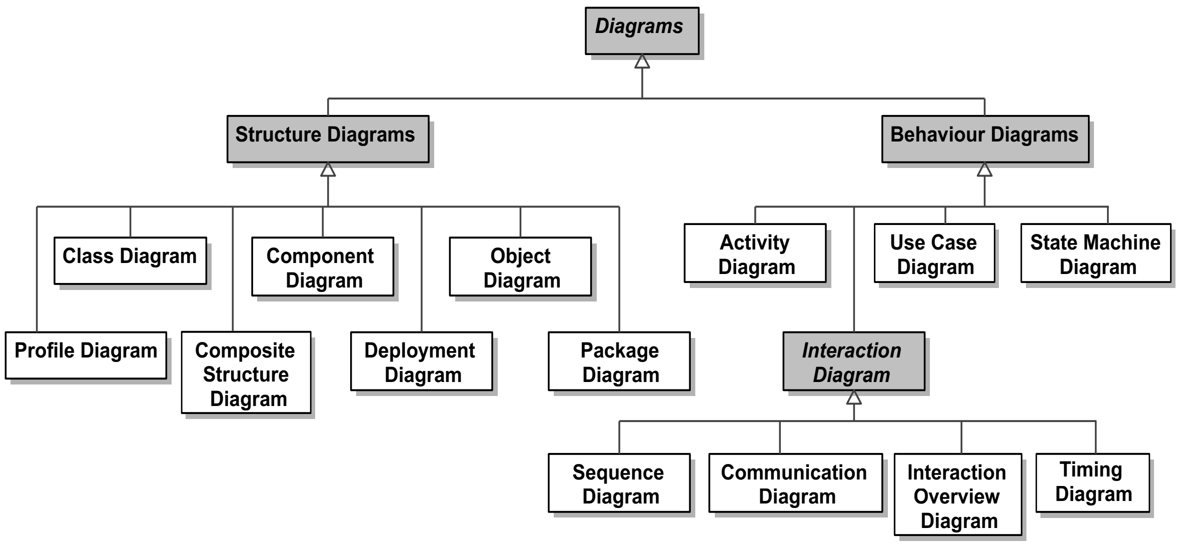


Figure 2.7: Taxonomy of UML's structural and behavioural diagrams according to [116]

in the second part of this section. To refer to UML metaclasses and their properties, we employ the same notation below as in the case of the MOF introduction.

2.5.1 The UML Diagrams

Apart from class and profile diagrams, only UML state machine and activity diagrams are relevant to our work. These two diagram typs are required in the context of UML models that have applied the UML profile for SDL. Therefore, we give a brief introduction to the essential model elements of both diagram types below.

State machine diagrams. This type of diagram enables behavioural modelling using the well-known formalism of the *finite state machines (FSM)*. A state machine diagram represents an instance of the *StateMachine* metaclass, and is thus a specialized form of *Behavior*. A *StateMachine* serves as a container for all elements that specify its details.

The behaviour of a UML state machine is defined by a set of different *Vertex* types and a set of *Transitions*. The vertices and transitions are always associated with a particular *Region* of a *StateMachine*. At least one *Region* must be present for a *StateMachine*. It is assumed that each *Region* has its own control thread. Thus, the presence of multiple *Region* represents the specification of concurrent behaviour. The vertices of a state machine are interconnected by means of *incoming* and *outgoing Transitions*.

State, *FinalState*, and *PseudoState* represent the most important specializations of *Vertex* in a state machine diagram. A *StateMachine* remains in a *State* until an *outgoing Transition* is triggered by the arrival of a defined event. Thus, this *Vertex* type can be considered as a stable state of a state machine. By contrast, a state machine cannot remain in a *PseudoState*.

A *State* represents either a *simple state* or *composite state*. The latter can be defined in two different variants. An *orthogonal composite state* consists of at least one *Region*, whereas a *submachine composite state* invokes another *StateMachine*. Conceptually, both composite state variants can be used to specify hierarchical state machines.

A *Transition* always starts at a *source* vertex and terminates at a *target* vertex, and its optional *trigger* property defines a set of events that can cause a *Transition* to be fired. If provided, the *guard* condition of a *Transition* must also be satisfied so that a transition can be fired. The *effect* property can define an optional behaviour to be processed by a fired transition. In a diagram, the three mentioned properties of a *Transition* are displayed textually according to the following syntax:

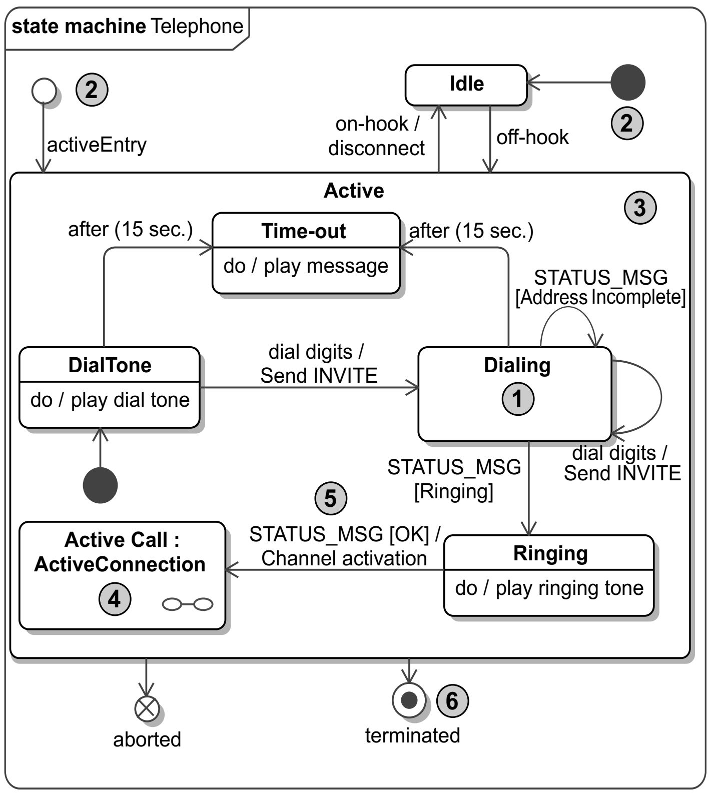
```
[ <trigger> ["," <trigger>]*
["[" <guard>"]"] ["/" <effect>] ]
```

If a *PseudoState* is the target of a fired *Transition*, it will be left immediately after the transition arrives, which can occur, e.g., via one of the defined outgoing transitions. A *PseudoState* can be used for different purposes, which depends on the value of its *kind* property. For example, a ‘choice’ *PseudoState* can be used to define a set of outgoing transitions, where only one of them is selected by evaluating the *guard* properties of the outgoing transitions.

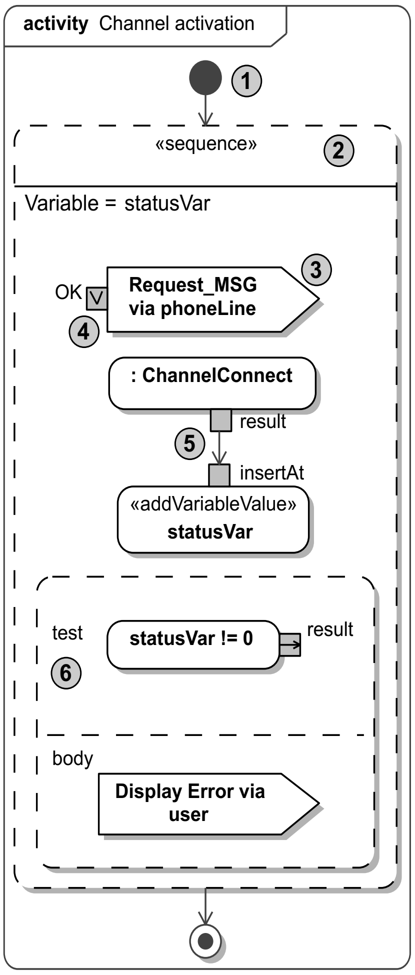
An exemplary state machine diagram specifying the behaviour of a telephone is shown in Fig. 2.8a. The *State* marked with (1) represents a simple state. By contrast, the *State* labelled with (3) defines an orthogonal composite state, and the element marked with (4) is a submachine composite state. The both symbols marked with (2) represent two different variants of *PseudoState*. Furthermore, the *Transition* labelled with (5) has defined a *SignalEvent* as a trigger, a guard condition, and an effect, which consists of the *Activity* ‘channel activation’. Item (6) represents a *FinalState*, which terminates the execution of the *Region* containing it.

Activity diagrams. This diagram type is used to specify an *Activity*, which is a particular kind of *Behavior*. An *Activity* is defined by a graph of nodes interconnected by edges. Different kinds of nodes are available for the specification of an *Activity*, where *Action*, *ControlNode*, *StructuredActivityNode* and *ObjectNode* are the main types. Numerous specialized variants of these nodes exist, so we can only treat the most important ones. The edges of an *Activity* can be specified either as *ControlFlow* or *ObjectFlow*.

An *Action* defines a fundamental unit of executable behaviour of an *Activity*, and it can have defined a set of inputs that may be converted to a set of outputs during the execution. Depending on the concrete subtype of an *Action*, both the input set and the output set can be empty. For example, *SendSignalAction* (see item (3) in Fig. 2.8b) is a particular variant of *Action* that specifies the sending of a *Signal*.



(a) UML state machine diagram



(b) UML activity diagram

Figure 2.8: UML state machine and activity diagram examples representing the behaviour of a telephone

In contrast to an *Action*, a *ControlNode* does not define a behaviour unit, but it coordinates the control or object flow of an *Activity*. Different types of *ControlNodes* exist, where some of them have a similar purpose as the different kinds of *PseudoState*. For example, *InitialNode* (see item (1) in Fig. 2.8b) and *FinalNode* can be employed to define the start and end of a control or object flow.

Various variants of *StructuredActivityNodes* can be employed to structure the nodes of an *Activity* for distinct purposes. A set of locally visible *Variables* can be defined that only the nodes enclosed by a *StructuredActivityNode* can access. The local variables can be used as an alternative to object flows. For example, the *SequenceNode* (see item (2) in Fig. 2.8b) is such a variant of *StructuredActivityNode*. All nodes contained in a *SequenceNode* are processed in the defined order. Thus, it is not required to interconnect the contained nodes using *ControlFlows*. *ConditionalNode* (see item (6) in Fig. 2.8b) and *LoopNode* are further variants of *StructuredActivityNode*, and they are used in a similar way as corresponding constructs of programming languages.

Pin is a particular form of *ObjectNode* which enables defining the inputs and outputs of *Actions* and *StructuredActivityNodes*. An *OutputPin* can serve as a starting point, and an *InputPin* as a target, for an *ObjectFlow* (see item (5) in Fig. 2.8b). A *ValuePin* is a specialization of *InputPin*, but it cannot be used as a target for an *ObjectFlow*; instead, it has a *ValueSpecification* (see item (4) in Fig. 2.8b) which defines the input. *ActionPin* is another variant of *InputPin*, where the input is specified using nested actions.

Since *Activity* is a particular form of *Behavior*, activities can also be used to specify the effect of a *Transition*. For example, the *Activity* shown in Fig. 2.8b defines the transition labelled with (5) in Fig. 2.8a. If an *Activity* is used to define the detailed behavior of a *Transition*, the set of employed *Actions* which can be contained in such an activity is sometimes denoted as *action language*.

2.5.2 Extending the UML Using UML Profiles

Just as any MOF-based metamodel, that of the UML can also be modified or extended at metamodel level [22, 34, 83]. The customization is based on a copy or a newly created version of the UML metamodel, which is obtained using packet merge. No restrictions concerning the applicable language concepts of the MOF exists, so existing metaclasses can be modified or new ones can be added. This type of customization is usually referred to as *heavyweight extension*.

Another way to extend a metamodel is to use a *middleweight extension* approach (e.g., see [22]), where a newly created metamodel imports the existing one. The metaclasses of the newly created metamodel can extend imported metaclasses by means of inheritance, and thus they can introduce new features, but existing metaclasses of the imported metamodel can neither be modified nor removed. Although the two techniques mentioned above can be applied for customizing the UML, they are not standardized. Therefore, a UML modelling tool must be customized to support extensions that are realized using one of these mechanisms.

By contrast, UML profiles are a standardized mechanism for adapting the UML to the specific requirements of a particular domain or application area. The UML specification [116, p. 276] defines a *Profile* as follows:

> *"A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain."*

In the above definition, the term 'reference metamodel' indicates that UML profiles are applicable not only to the UML metamodel but also to any kind of CMOF-based metamodel. As stated in the UML specification, modelling tools that are not UML-based require some kind of mechanism for applying profiles to models which are instances of particular metamodels other than that of the UML. Thus, we assumed that such a scenario is rarely used in practice.

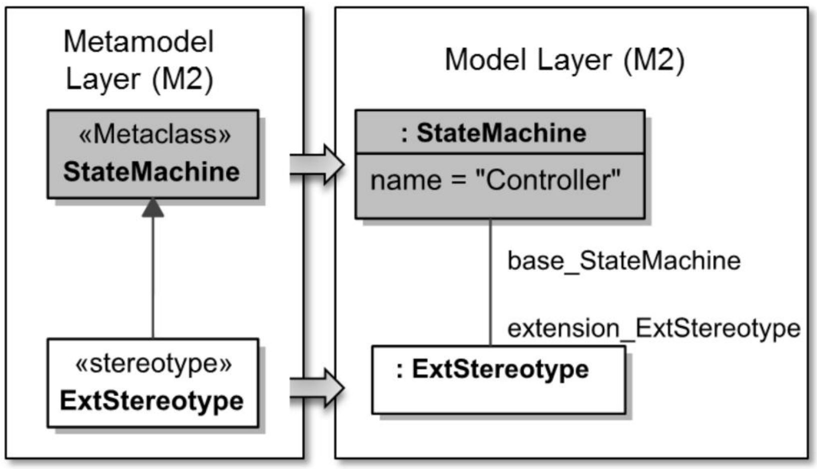


Figure 2.9: A *Stereotype* specification and its application

Moreover, the UML specification [116, p. 250] justifies that UML profiles, unlike the MOF-based extensions above, are not a first-class extension mechanism. Instead, UML profiles are referred to as a *lightweight extension* mechanism to the UML, which can be employed to define constraints, queries, properties, and the graphical representation for UML elements. In addition, the application of a *Profile* must not alter the UML model to which it is applied.

UML profiles in detail. A UML *Profile* is a particular subtype of *Packet*, and it can consist of a set of *Stereotypes*, *Associations*, *Classes*, and *DataTypes* (incl. *PrimitiveTypes* and *Enumerations*). However, *Stereotypes* as a specialization of *Class* are the primary constructs of a *Profile* for extending UML metaclasses.

A *Profile* is specified employing a *profile diagram*, which can be considered as a particular variant of a class diagram for which *Profile*-specific constraints apply. Apart from *Stereotypes*, all other kinds of elements contained in a *Profile* must conform to the constraints that are defined for class diagrams used to specify CMOF-based metamodels [116, p. 252]. Therefore, these kinds of elements cannot be employed as instances in UML models; that is, they can only be used within a *Profile* to specify the *type* of stereotype attributes. To avoid redundancies, we do not treat further details of the constraints here; instead, we analyse and discuss their consequences in the context of our derivation approach.

Because *Stereotype* is a subtype of *Class*, it can own *Properties*, *Operations*, and *Constraints*, but compared to a *Class* some restrictions apply for a *Stereotype*. As in the case of a *Class*, a *Stereotype* attribute can be defined by an *Association* instead of explicitly specifying it. The most significant difference between a *Stereotype* and a metaclass is that a *Stereotype* cannot extend a UML metaclass using inheritance, because *Generalization* relationships are only permitted between *Stereotypes* [116, p. 278].

The extension of a UML metaclass by a *Stereotype* is defined by means of an *Extension*, as shown on the left-hand side of Fig. 2.9. If an *Extension* is defined

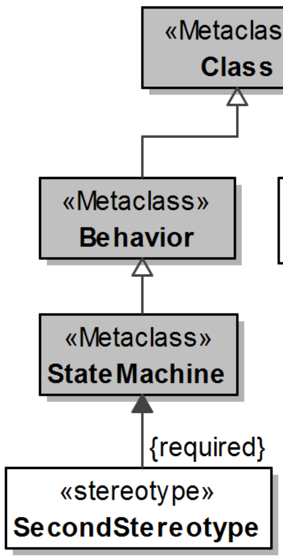


Figure 2.10: A *Stereotype* that is defined as required

as required (e.g., see Fig. 2.10), its associated *Stereotype* is automatically applied to all instances of the UML metaclass referenced by the *Extension*. By contrast, the *Stereotype* of an *Extension* without such a label must be manually applied to the desired UML elements.

A *Stereotype* defined as required is not only applied automatically to instances of the UML metaclass it extends, but also to instances of metaclasses inheriting from that metaclass. For instance, the «FirstStereotype» stereotype in Fig. 2.10 is automatically applied to instances of the metaclasses Class, Behavior and StateMachine. However, the automatic application of *Stereotypes* to instances of inheriting metaclasses can be problematic. For example, from a syntactic point of view, it may not be desired that a particular *Stereotype* is automatically applied to instances of inheriting metaclasses; but we cannot prevent this in the case of *Stereotypes* defined as required.

Since an *Extension* is a particular variant of an *Association*, it also represents a relationship between two *Properties*. If an *Extension* is defined between a *Stereotype* and a UML metaclass, the two properties *base_<MC>* and *extension_<ST>* are implicitly introduced. The abbreviation *<MC>* is a placeholder for the metaclass name and *<ST>* is a placeholder for the stereotype name. The *base_<MC>* property is a reference to the extended metaclass, whereas the *extension_<ST>* property introduces a reference to the extending *Stereotype*. Furthermore, the *base_<MC>* property is directly owned by the extending *Stereotype*, whereas the *extension_<ST>* property is part of the *Extension*. This is because a *Profile* is not permitted to modify UML metaclasses.

When a *Profile* is applied to a UML model, metaclasses and stereotypes are instantiated separately. This fact is shown on the right-hand side of Fig. 2.9, where an instance of the metaclass StateMachine and an instance of its applied stereotype ExtStereotype are represented as an object model. The two implicitly defined *Properties* can be utilized in OCL expressions to navigate between stereotype and metaclass instances.

As a profile application shall not alter the UML model to which it is applied, the instances of applied *Stereotypes* cannot directly be contained in the model. Although the UML specification does not define how stereotype instances have to be represented and stored at runtime, it defines rules [116, p. 269] for serializing a UML model with applied UML profile using XMI. Due to these rules, an XMI file or resource first contains the serialized model followed by the stereotype instances. The model elements are contained as a nested structure, while the stereotype instances are represented as a flat list.

2.6 Methods for Verification and Validation

Metamodels and associated OCL constraints are the most outcomes of an MDE-based development process for DSLs. In addition, the creation of particular model transformations may also be required. Just as in the case of classical software development, metamodels and model transformations must be verified and validated by appropriate methods in the context of quality assurance. In the following, we briefly discuss the methods and approaches that are applicable for the verification and validation of our derivation approach, including the artefacts derived by this approach. Before we go into the details, we clarify the differences between validation and verification as applied in classical software development.

2.6.1 Background

In the classical software development, verification and validation play a key role in the quality assurance of the created software artefacts, such as executable programs. Since the objectives of verification and validation are different, we briefly discuss their differences.

Definitions. Apart from the various definitions regarding verification and validation given in the literature (e.g., [3, 110]) and standards (e.g., [62, 61]), one of the best known and probably shortest explanation for both terms was introduced by Boehm [18]:

Verification: Are we building the product right?

Validation: Are we building the right product?

Although the above statements make clear that validation and verification treat two different aspects in quality assurance, in our view they are too general. For instance, two much more concise definitions for both terms can be found in the IEEE guide to the project management body of knowledge [62]:

> *Verification: The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with validation.*
>
> *Validation: The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with verification.*

Based on both definitions above, we conclude that verification activities evaluate the fulfilment of functional requirements, while validation activities check whether the developed product met the expectations of users or other stakeholders. Although the definitions given make clear the objectives of verification and validation, they do not clarify the methods available for both aspects of quality assurance.

Static and dynamic techniques. Over the past decades, various methods and approaches for software verification and validation have been proposed in the literature. In general, these methods can be classified into static and dynamic techniques (e.g., see [3, 126]). The main difference between both categories is that techniques of the latter category require execution of the program under study, whereas static techniques are applied without program execution.

The category of static analysis techniques embraces different types of manual reviews, e.g., walk-throughs and inspections, and formal methods such as theorem proving [25, 132] and model checking [12, 27]. By contrast, the dynamic techniques typically consist of various types of software tests [3, 21, 126] that are performed at different stages of development and involve program execution.

As with the definitions for verification and validation, the literature has differing views on whether testing activities belongs either to validation or verification activities. For instance in [17, 105], dynamic software tests are considered as part of the software verification and are referred to as verification tests. By contrast, only reviews and formal methods are considered as verification tests. By contrast, another view is chosen in [126], where only reviews are considered as part of the verification, whereas dynamic software tests shall be part of validation activities.

In addition to rigorously classifying dynamic software tests to be part of validation or verification activities, they can also be regarded as part of both activities, as pointed out in [21]:

> *The observed behavior may be checked against user needs (commonly referred to as testing for validation), against a specification (testing for verification), or, perhaps, against the anticipated behavior from implicit requirements or expectations [...].*

We share the above view, and we therefore consider dynamic software tests as an integral part of both verification and validation. Thus, in the remainder of our work, we use the term *verification tests* to refer to dynamic software tests employed to check the functional requirements specified for our approach, while we consider the check of user-specific requirements as an objective of *validation tests*.

2.6.2 Metamodels and UML Profiles.

To verify metamodels and UML profiles created by applying our approach, we considered different approaches proposed in the literature. Since the transformation of UML class models into formal specifications for validation and verification is a very popular approach, we first considered the applicability of such approaches.

Transformation-based approaches. Several approaches (e.g., [54, 60, 154]) can be found in the literature, where the *UML-based Specification Environment (USE)* [53] is employed to transform UML models with OCL constraints into a relational logic language so that a model checker can perform the validation and verification. Currently, the USE tool has some restrictions concerning the supported features of UML and OCL. For instance, Gogolla and Hilken [54] state that the support of qualified associations, subset relations between association ends, and OCL operations on collection types is planned as future work.

Another model validation approach is proposed by Khan and Porres [81], where they transform a UML model with OCL constraints into the *Web Ontology Language (OWL) 2* [107]. Then, the result obtained is passed to a reasoner which processes the ontology to detect model inconsistencies. A disadvantage of the approach is that according to [81] only a subset of OCL is supported.

A further category of approaches treats the transformation of UML models into Alloy scripts, which can then be validated by an analyser tool. The Alloy language [74] is based on a first-order relational logic. For instance, approaches for transforming UML models towards Alloy scripts are presented in [10, 29, 102]. Just like for the OWL-based approach above, a common drawback of the approaches using Alloy is that only a subset of all OCL language constructs and not all features concerning class diagrams are supported.

Due to the limitations of the discussed approaches, we consider them as not applicable for our purposes, because metamodels that are created by applying our approach intensively use the CMOF concepts of subsetting and redefinition. In addition, we support all language constructs of OCL, while only a subset of them is supported by the approaches above. Therefore, we assume that OCL expressions of the metamodels we have created cannot be fully processed by these tools and approaches.

Manual approaches. In contrast to approaches treating the transformation of metamodels into a formal specification, the literature concerning other approaches for validating and verifying metamodels is rather limited. For example, the use of model fragments, also referred to as *test models*, as input for positive and negative test cases for testing metamodels is discussed in [99, 133]. The objective of such test cases is to verify or validate structural aspects and OCL constrains of a metamodel under test.

To avoid the creation of too many or too less test cases, the contribution of Andrew et al. [11] can be applied, where they propose various test coverage criteria for different types of UML diagrams. Even if the author's contribution is primarily intended for testing executable systems, the criteria defined for UML class diagrams can also be employed for the test of metamodels. The relevant criteria are as follows:

- ***Association-end multiplicity (AEM):*** Each multiplicity-pair has to be instantiated at least once.
- ***Class attribute (CA):*** Each attribute of a class has to be instantiated at least once.
- ***Generalization (GN):*** Each sub-class of a super-class has to be instantiated at leas once.

To obtain test cases that satisfy the AEM and CA criteria, Andrew et al. [11] propose to apply the category-partition testing [123] method which is well-known from the software testing discipline. All possible values for a class attribute or association end are categorized into equivalence classes. Then, exactly one representative value is selected for each equivalence class to define a test case based on that value. Appropriate equivalence classes shall be determined based on existing OCL constraints and the cardinalities of the attributes and association ends.

Conclusion. Due to the discussed shortcomings of formal approaches for verifying and validating CMOF-based metamodels, we also analysed manual techniques that use appropriate test models for positive and negative tests. Because no limitations on the test of CMOF-based metamodels and contained OCL expressions are reported for such techniques in the literature, we consider a manual testing approach to be applicable for verifying and validating the artefacts of our derivation approach.

2.6.3 Verification of Model Transformations.

In the past, various approaches for the (semi-)automatic verification of model transformations have been proposed. For instance, more than 50 tools and approaches for the validation and verification of model transformations are analysed in a survey of Ab. Rahim and Whittle [1]. We analysed the tools listed in the survey to determine those tools enabling an automated verification of model transformations specified by means of QVTo. Unfortunately, only a small number of coverage metrics are proposed but no tool or approach supporting an automatic verification.

In a next step, we analysed the more general approaches, which are not related to a particular transformation language and have to be applied semi-automatically or manual. For this purpose, we took into account that test models from our verification and validation activities of metamodels could be reused for transformation testing.

Even if the general approaches that are discussed in the literature differ in their realization, common challenges for verifying and validating model transformations can be identified. In the following, we summarize the crucial challenges as identified in [13, 14, 96, 139].

Test data. The challenge in creating test data is to generate a set of models, which are instances of the source metamodel of a transformation under test. These models are used as input for conducting transformation test cases. Among other terms, in the literature (e.g. see [14, 139]) such models are also referred to as *test models*. As pointed out by Gehan et al. [139] and Fleurey et al. [45], an important aspect is that a generation of test models covering all possible input data of a model transformation is infeasible. Therefore, they recommend to define appropriate criteria similar to those proposed by Andrew et al. [11] for the test of class diagrams.

Test oracle. The test oracle is a function or component that compares the obtained test result with the expected result. In the case of testing M2M transformation, the test result is always a model which is an instance of the target metamodel of the transformation under test. In the following, we refer to such models as *result models*. According to Mottu et al. [108], a test oracle can be implemented in terms of contracts, pattern matching, or model comparison.

A *contract-based* oracle consists of a set of assertions that are defined as pre-conditions and post-conditions. The pre-conditions are evaluated on a test model used as transformation input, while the post-conditions are evaluated on the result model after the test execution. If all assertions are satisfied, the test of a transformation is successfully passed.

A test oracle which implements the *pattern matching* approach consists of a set of patterns. This type of test oracle can be implemented in two ways. In the first case, the test oracle checks that a set of expected elements is contained in the result model after test execution. In the second case, a test oracle consists of a set of postconditions which must be met by the result model obtained.

A test oracle that is based on a *model comparison* compares the result model obtained after test execution with a model expected as test result. Two variants for implementing model comparison exist. The first variant compares the expected output model with the obtained result model. Typically, the expected result model can only be used in combination with a particular test model that is used as input for the transformation under test. Provided that an inverse transformation is present for the transformation under test, the inverse one can serve as test oracle. In this case, the transformation under test and its inverse counterpart are executed as a transformation chain. Thus, the test and the result model are instances of the source metamodel of the transformation under test. A test case using an inverse transformation as test oracle can be rated as passed if both models exactly match.

Test execution. As in the case of ordinary software test, the test cases for transformation testing can be conducted manually or automatically. Regardless of the test execution kind, the tests for model transformations are always conducted as follows:

1. Selecting an appropriate test model;
2. Executing the transformation under test with the selected test model as input;
3. Comparing the obtained result model with the expected result employing the test oracle;
4. Assessing the test case as passed or failed based on the result obtained from the test oracle.

Testing tools. In recent years, a few tools have been developed to compare two models among each other so similarities or differences between both models can be identified. Apart from other purposes, these tools can be employed to implement a comparison-based test oracle for transformation testing. A survey [144] conducted by Stephan and Cordy provides a good overview of existing approaches and tools in this area.

In a further work [143], Stephan and Cordy report on the results of a case study concerning the use of different model comparison techniques for the test of model transformations. To conduct this case study, they examined the applicability of five different tools for comparing the input and result models of model transformations. Based on their findings, Stephan and Cordy recommend the use of *EMF Compare* [23] or the *Epsilon Comparison Language (ECL)* [84]. An important feature of both tools for testing model transformations is the support of dynamic element identifiers. This is required because element identifiers are usually not preserved by out-place transformations. In addition, both recommended tools show the comparison results graphically, so the localization of differences can become more efficient. However, an important difference between both tools is that *ECL* has to be configured case-specific, while this is not required for *EMFCompare*.

Conclusion. As we highlighted in the first part of this section, automated verification approaches are not available for QVTo. Therefore, we can only apply a manual or semi-automated approach to validate and verify model transformations which are generated by applying our derivation approach. Test models originated from the verification and validation of metamodels already exist, so we can reuse them for testing model transformations. Due to this reason, we consider the creation of a test oracle that is based on a contract-based approach or that implements pattern-matching to be not required. Instead, we wish to employ a comparison-based test oracle that compares the transformation result to the expected one. To implement such a model comparison, we consider EMFCompare to be the most appropriate tool because its configuration, as explained above, is less complex than for ECL.

2.7 Summary

In the previous sections, we reviewed the fundamentals of an MDE-based development of DSLs, including a fundamental development process. We also regarded methods for verifying and validating the artefacts created during this process. Since we wish to research on an automated derivation approach for CMOF-compliant metamodels, UML profiles, and model transformations, we especially discussed the foundations of CMOF, UML profiles, and an OCL-based static semantics in detail. Based on the knowledge gained in this way, we analyse the detailed steps to derive the various artefacts that shall be automatically generate using our approach.

In addition to the foundations of an MDE-based development of DSLs, we presented an associated development process. In the following chapters, we use this process as a starting point to define a development process that can be applied in combination with our derivation approach. In addition, we wish to employ the analysed approaches for verification and validation of metamodels and model transformations to evaluate the artefacts generated by our derivation approach.

3 A New UML Profile for SDL – Lessons Learned

In this chapter, we discuss our contributions to a new version of the UML profile for SDL and how we specified it. First, we briefly outline the evolution of SDL and of its UML profile taking into account our contributions. Then, we give an introduction into the foundations of SDL. Building on these foundations, we detail the shortcomings of the old version of the UML profile for SDL and our contribution to remedy them. Thereafter, we treat the approach that we applied to test and verify the new version of the UML profile for SDL, and we present our prototypical implementation of this UML profile. Finally, we summarize the outcome of this chapter and argue in which way the experience gained while revising the UML profile for SDL inspired our research on an approach for the automatic derivation of UML profiles.

3.1 Evolution of SDL and our Contributions

Before we discuss the shortcomings of the UML profile for SDL and our contributions to their correction in the following sections in detail, we briefly outline the historical evolution of SDL and of its UML profile. Building on this, we then summarize our contributions to a new version of this UML profile, including its prototypical implementation.

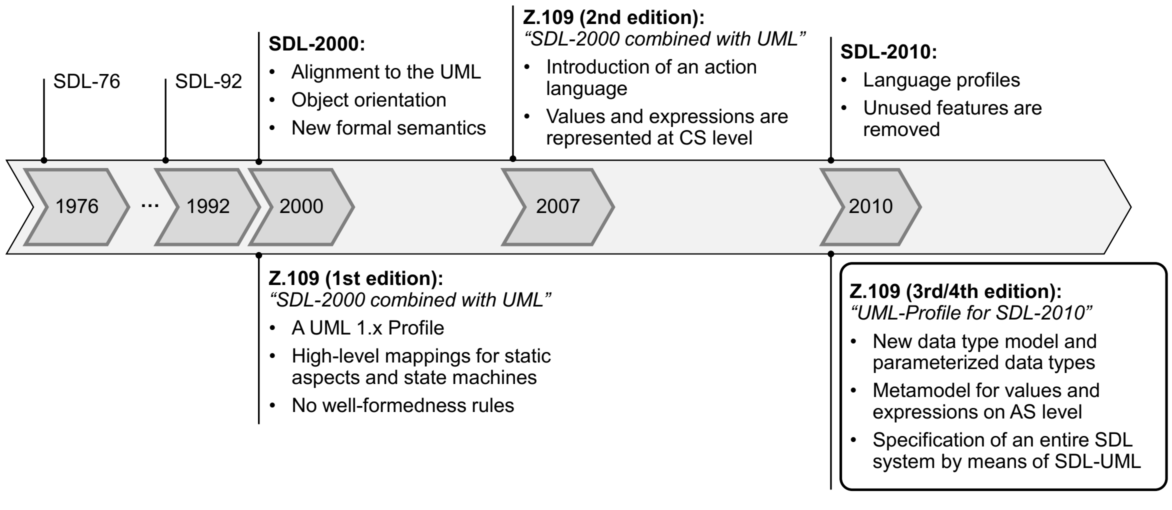

SDL-76
SDL-92
SDL-2000:
• Alignment to the UML
• Object orientation
• New formal semantics
Z.109 (2nd edition):
"SDL-2000 combined with UML"
• Introduction of an action language
• Values and expressions are represented at CS level
SDL-2010:
• Language profiles
• Unused features are removed
1976
...
1992
2000
2007
2010
Z.109 (1st edition):
"SDL-2000 combined with UML"
• A UML 1.x Profile
• High-level mappings for static aspects and state machines
• No well-formedness rules
Z.109 (3rd/4th edition):
"UML-Profile for SDL-2010"
• New data type model and parameterized data types
• Metamodel for values and expressions on AS level
• Specification of an entire SDL system by means of SDL-UML

Figure 3.1: The historical evolution of SDL and of its UML profile

3.1.1 Historical Evolution of SDL and of its UML Profile

Evolution of SDL. As pointed out in [129], the origins of SDL date back to 1976, when the first version of SDL was standardized as SDL-1976 (see Fig. 3.1). New versions of SDL were released every four years until 1992. After this period, a completely revised version of SDL was published in 1999 [64], referred to as SDL-2000. Fischer et al. [43] point out that the adaptation of the SDL language concepts to those of the UML and the introduction of a new formal semantics are the most important innovations of SDL-2000.

The last major revision of the SDL standard took place in 2010 and is referred to as SDL-2010 [70], where the language concepts of SDL-2000 were divided into different *language profiles* that build upon each other. Instead of implementing all SDL-2010 language features, a tool vendor now has the freedom to implement only the language concepts required for a particular language profile. In addition, some features introduced for SDL-2000 were removed in SDL-2010, because they were not implemented or used [129].

Evolution of the UML profile for SDL. In addition to aligning SDL-2000 to UML, the ITU recognized the potential of UML for the modelling of communication systems. Therefore, a first edition of the UML-1.3 profile for the combined use of SDL and UML was published as ITU Rec. Z.109 [65] in 1999. In addition to the UML profile, a high-level mapping of UML stereotypes to corresponding SDL language concepts was defined. The stereotypes defined by the first edition of the UML profile only supported the modelling of structural aspects and state machines, but not the specification of SDL statements and expressions. Thus, it was impossible to model an entire SDL specification using this UML profile. Furthermore, no well-formedness rules for the stereotypes of the UML profile were defined.

The main objectives of the second edition of Rec. Z.109 of 2007 were to address the above-mentioned drawbacks and to adapt the UML profile to UML 2.0. For this reason, additional stereotypes were introduced to represent SDL statements and expressions. In addition, constraints in natural language were specified to capture the static semantics of SDL. As reported by us in [85, 92], these changes were partially not appropriate and incomplete. Due to these reasons, the author of the thesis at hand created a new UML profile for SDL-2010, which solves the identified problems and introduces additional features (see Fig. 3.1). The latest outcome of this work is published as the fourth edition of Rec. Z.109 [69].

Notational conventions. In the remainder of this chapter, we use the acronym *SDL-UML* to refer to the *UML profile for SDL*, and the term *SDL-UML model* is employed as acronym for a UML model with applied SDL-UML profile. To distinguish between the different editions of SDL-UML, we employ the acronym *SDL-UML 2007* to refer to SDL-UML as specified in Rec. Z.109 of 2007 [66], while the acronym *SDL-UML 2013* points to the Rec. Z.109 released in 2013 [69]. Furthermore, we use the notations introduced in Sec. 2.5 to refer to stereotypes and UML metaclasses.

3.1.2 Our Contributions to a new UML Profile for SDL

Our work on a new edition of SDL-UML was motivated by open issues and errors identified based on a comparative case study between SDL and SDL-UML 2007. This case study was prepared as part of a master thesis by Patrick Rehm [130], which was supervised by the author of the thesis at hand. To enable an automatic compliance verification of a created model against the static semantics of SDL-UML, we translated the constraints specified in natural language into corresponding OCL constraints and queries. A summary of the results obtained from the case study was presented by us in [92]. Based on the case study, we identified the lack of OCL constraints and the time-consuming specification of SDL statements and expressions using UML elements and related stereotypes as the major drawbacks of SDL-UML 2007.

Because of the above-mentioned issues concerning SDL-UML 2007, we further investigated and compared the Rec. Z.109 [66] of 2007 with the specification of SDL-2000 [64]. We identified some issues related to the data type concept defined for SDL-UML 2007. A major shortcoming was that not all SDL language concepts relevant for the specification of data types were supported, or the language concepts were implemented only inadequately. We also identified issues related to the representation of SDL expressions. In addition, we noted that some well-formedness rules of SDL's static semantics had no corresponding constraints in SDL-UML 2007. The details of the problems identified by us are discussed in detail in Sec. 3.3.

To overcome the above shortcomings, we proposed a new data type concept and revised the representation of SDL expressions; our results were presented in [87]. Further inadequacies and issues of SDL-UML 2007 were directly reported to the ITU-T Study Group 17/Q.12[1] responsible for the standardization of SDL. Because of our various contributions to SDL-UML, the author of the thesis at hand was appointed as the author in charge for creating a new edition of SDL-UML. The most recent edition of our revised UML profile for SDL is SDL-UML 2013, which was released as Rec. Z.109 in 2013 [69]. The details of our key contributions to SDL-UML 2013 are discussed in Sec. 3.4.

To verify the changes and improvements we made to SDL-UML 2013, we developed a prototype tool that enables the modelling and verification of SDL-UML models. This prototype is briefly introduced in the following section.

[1] https://www.itu.int/en/ITU-T/studygroups/2017-2020/17/Pages/q12.aspx

3.1.3 The SDL-UML Modelling and Validation Framework

In order to verify the modifications and enhancements introduced by us for SDL-UML 2013, we implemented the *SDL-UML Modelling and Validation (SU-MoVal)* framework. This framework supports the graphical and textual specification of models that conform to SDL-UML 2013. In addition, based on a set of OCL constraints derived from Rec. Z.109 [69], the SU-MoVal framework can verify whether a created model violates the static semantics. Furthermore, SU-MoVal provides an editor for the textual notation that can be used to specify statements and expressions. The entire SU-MoVal framework is based on various model transformations. An overview of SU-MoVal is presented in [89].

All components of SU-MoVal are implemented as plug-ins for the *Eclipse Modeling Tools (MDT)* edition of Eclipse[2], because the default plug-ins of this Eclipse edition are required for realizing the framework. We made available our SU-MoVal framework as open source software, and it can be obtained via our homepage [156]. In the following, we give a brief introduction to the components of SU-MoVal shown in Fig. 3.2. Further details of the SU-MoVal framework are discussed in Sec. 3.6.

[2] https://eclipse.org/modeling/mdt/

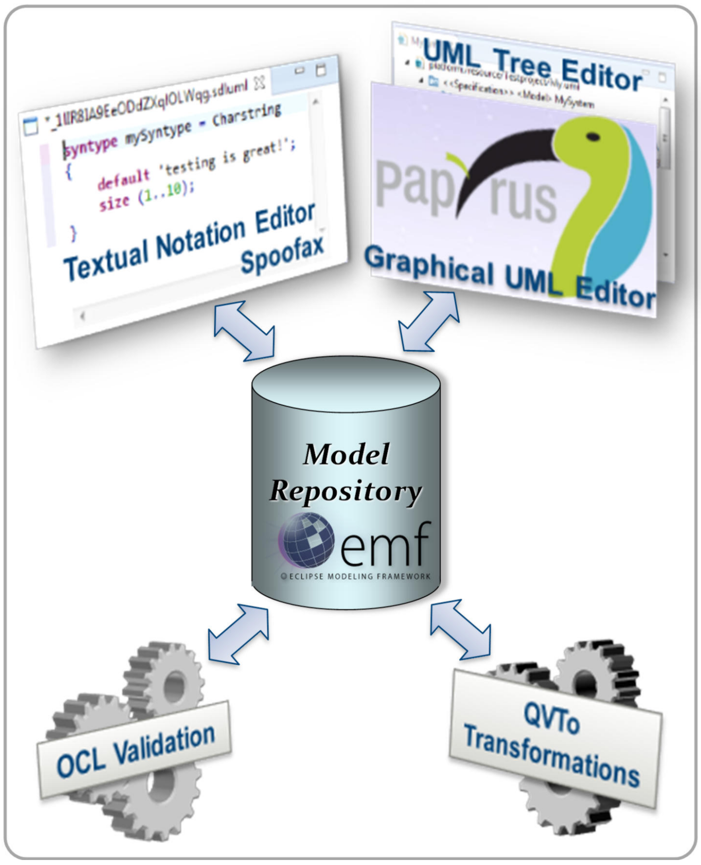

```
syntype mySyntype = Charstring
{
    default 'testing is great!';
    size (1..10);
}
```

Figure 3.2: The components of the SU-MoVal framework

Model validation One of the features provided by SU-MoVal is the validation of SDL-UML models, which is based on the OCL component of Eclipse. In order to validate models, OCL constraints and queries specified in the context of stereotypes and metaclasses of SDL-UML are evaluated. After validation, the result is displayed to the user in the 'Problems View' of Eclipse.

Modelling and textual notation For the specification of structural and behavioural aspects of an SDL-UML model, a user is free to choose the default UML tree editor of Eclipse or the Papyrus UML modelling tool[4]. Thanks to the extensibility mechanisms of EMF and UML, both tools can apply the SDL-UML profile to a UML model. However, the OCL validation component and the textual notation editor of SU-MoVal can only be invoked from the UML tree editor at present. The editor for the textual notation can be accessed from the context menu of the UML tree editor. The textual notation supports the specification of SDL statements and expressions, which are translated to corresponding SDL-UML elements. We employed a model-based approach to implement the textual notation for SDL-UML. The details of this approach are discussed in [86, 89].

Transformations Before the editor for the textual notation can be invoked, type definitions of an SDL-UML model are extracted and transformed into an internal format. For this purpose, a QVT transformation is conducted by the QVTo component of Eclipse[3]. In addition, further QVT transformations are implemented for mapping the textual notation to corresponding SDL-UML elements.

3.2 Foundations of SDL and of its UML Profile

In the following, we give a brief introduction to the foundations of SDL, its formal specification, and the UML profile for SDL as released in 2007.

3.2.1 Fundamental Parts of an SDL Specification

A model of a system or protocol that is created employing SDL is referred to as the *SDL specification*. The creation of an SDL specification can be realized either by means of the graphical or textual notation of SDL, where the graphical variant can be translated one-to-one into the textual notation.

SDL supports the language concepts of inheritance and refinement, as known from object-oriented languages. All entities of an SDL specification are instantiated based on type definitions. New subtypes can be derived from existing type definitions by inheritance. A subtype can introduce additional features or inherited

[3] https://wiki.eclipse.org/QVTo

ones can be redefined. Furthermore, SDL supports parametrized type definitions which can be employed to define generic types. Typically, an SDL specification defines structural and behavioural aspects of a system to be modelled.

Structural aspects. As of SDL-2000, an SDL model[4] consists of a set of hierarchically composed asynchronously communicating agents. Each agent is an instance of a particular agent type definition that specifies the agent's structure and can contain definitions of nested agent types, data types, interfaces, signals, procedures, timers and variables. To enable the specification of generic types, each SDL type definition may be associated with a set of formal context parameters. Apart from the contained type definitions, an agent type definition refers to a state machine which defines the behaviour of the agent.

The structure of an agent type is specified by means of a set of agent instances, which are interconnected via signal channels and gates. The set of all signals that an agent can send or receive is specified by an interface definition associated with an agent type definition. In addition to signals, an interface definition can declare remote procedures and remote variables. SDL treats interface definitions as a certain type of data type definition.

The following three types of SDL agents have to be distinguished: system agent, block agent, and process agent. In the hierarchical composition of an SDL specification, a system agent always represents the topmost agent of this hierarchy and it specifies the overall structure of the system to be modelled. The logical or physical entities of the system are defined as block agents contained in the system agent. The communication relationships of the block agents among each other and with the environment are defined by the signal channels of the system agent. A block agent can consist of additional block or process agents to specify its internal structure and behaviour, while a process agent may only contain process agents.

In SDL-2000 [64], data types could be defined as object or value data types, while SDL-2010 [70] only supports the latter kind. In addition to data types, SDL provides a language concept called syntype that can restrict the set of valid values for a data type according to specific rules. Although syntypes are not data types in the strict sense, they can be used in the same way as data types.

A data type definition can have a set of operations that consists of operators and methods. Each operation has a signature that may be associated with a behaviour definition. In contrast to a method, an operator of a data type definition must not have side-effects, i.e. it is not permitted to change any item of a data type. Thus, operators are comparable with static operations of GPLs.

By default, a data type owns only a set of operations. To enable the specification of data types with a specific characteristic, SDL provides a language concept named

[4] Although according to Rec. Z.100 [70] a model that is created using SDL is referred to as *SDL specification*, we use the term *SDL model* to avoid ambiguities in the remainder of the present work.

'constructor'. The following constructor types are available: literal, structure and choice. A literal constructor enumerates all valid literals of a data type. By contrast, the structure and choice constructors define of a list of data items, where each item has a distinct type. An instance of a data type with structure constructor contains values for all defined items of that type, whereas only a single value is present for an instance of a data type with choice constructor. Applying a constructor to a data type results in an implicit creation of constructor-specific operators and methods for the data type concerned.

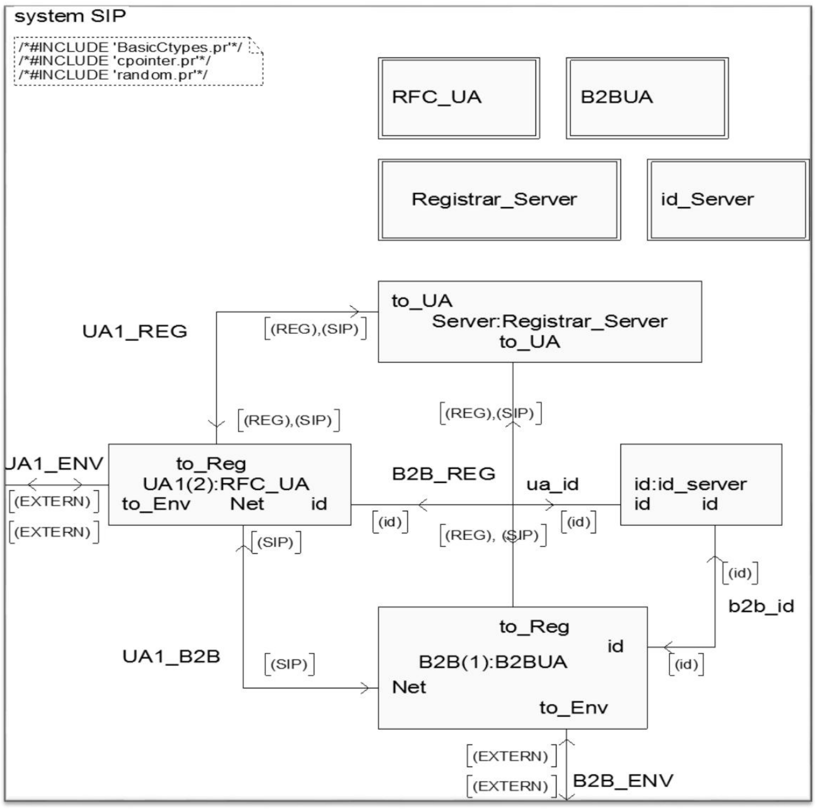
SDL provides a package of predefined data types, which contains 14 simple and 4 parametrised data types, e.g., Array. The latter types enable the definition of dynamic data structures. The simple types are defined as literal data types, syntypes or instances of a generic data type. Each predefined data type provides a set of operators which mainly represent arithmetic or logical infix operators. In contrast to user-defined data types, the behaviour of such an operator is not defined within the data type; instead, it is specified as part of SDL's dynamic semantic, as explained in the next section.

Fig. 3.3a shows an example of a system agent type definition from an SDL specification, which was created by us for a case study [92] on the *Session Initiation Protocols (SIP)*. In the upper part of the diagram, four references to block agent type definitions are displayed as doubly framed rectangles, while the bottom part defines the agent's structure consisting of four block agents. Each referenced agent type is instantiated in the form of a dedicated agent. In addition to signal channels that interconnect the agents, two channels between the agents and the environment of the system are defined.

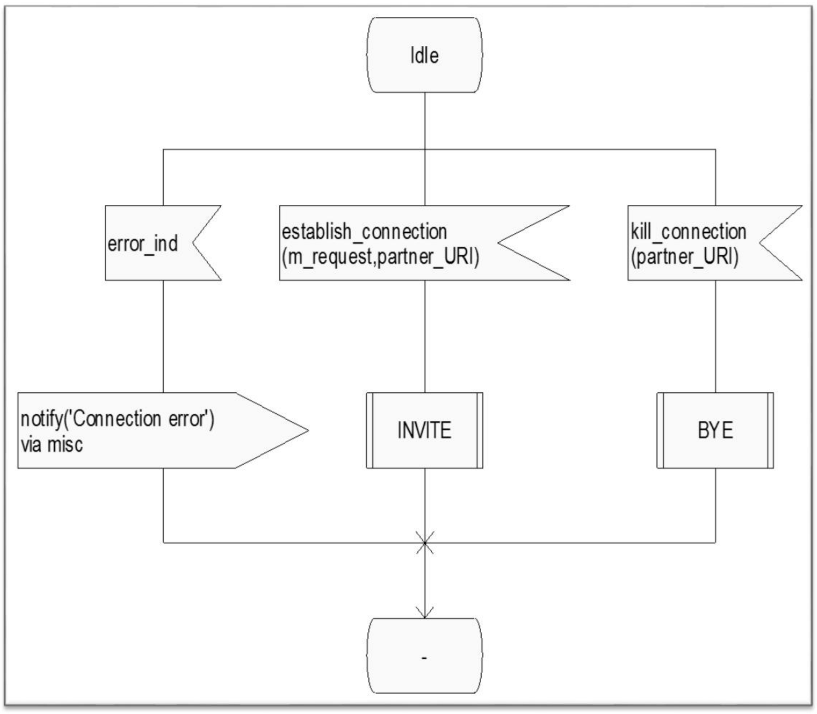
Behavioural aspects. Depending on the agent type, the state machine associated with an agent is executed in two different ways. The state machines of system or block agents are processed concurrently, while the state machines of process agents are executed alternately. The behaviour of an SDL state machine is specified by an associated composite state type definition, which is a particular type definition that serves as a container for states and transitions. Just as other type definitions of SDL, composite state type definitions can be specialized and redefined using inheritance.

Apart from agents, state machines also define the behaviour of SDL procedures, which can be invoked by state machines of other procedures or agents. For instance, procedures can be used to encapsulate a distinct part of the behaviour specification, so a reuse is enabled.

Although SDL state machines are similar to those of UML, they have a different syntactical structure. The most significant difference is that a SDL state machine is defined by a so-called state transition graph, which consist of a set of state nodes. A state node can be defined as either a base state or a composite state. The latter



(a) Agent type definition example



(b) SDL state machine example

Figure 3.3: SDL example from our case study [92] for the Session Initiation Protocol (SIP), adopted from [130]

enables the specification of hierarchical state machines in SDL. To define a composite state, a state node can refer to a composite state type definition.

Apart from other items, a state node owns a set of input nodes, where each of these nodes define an expected input signal and a transition to the next target state. When the signal associated with an input node is received, the transition of this node is fired. A list of statements defines the effect of a transition, and a terminating node specifies the target of the transition, e.g., another state node or a stop node. The statements used to define the transition effect are referred to as SDL's action language.

In addition to signals, SDL supports the specification of timers. Implicit signals are generated for expired timers, which can be referenced in the same way as ordinary signals in input nodes. Furthermore, SDL's action language provides dedicated statements to start and stop timers.

The SDL state machine example shown in Fig. 3.3b is used to establish and terminate a SIP connection. The state machine consists of two states: an Idle state node and a 'dash next state'. The latter kind of state is a short-hand notation for defining that a transition terminating at such a node returns to its source state. The Idle state node has defined three input nodes, which are triggered by different signals. The input node shown at the left side of the diagram is triggered by the reception of an error_ind signal, and an output node is defined as the effect of its outgoing transition. This output node specifies that a notify signal shall be send via a channel named misc. The input node located in the middle of the diagram is triggered by the signal establish_connection. When this signal is received, the procedure INVITE is invoked as specified by the call node of the outgoing transition. This procedure implements the details of the connection setup. The input node on the right side of the diagram implements the termination of an existing SIP connection. As in the former case, a procedure with a call node is invoked upon reception of the signal defined by the input node.

3.2.2 Formalization of SDL

The formal grammar and semantics of SDL is specified by three documents published as Annex F of Z.100 [70]. The formal specification of SDL can be divided into the following parts:

- grammar;
- well-formedness conditions;
- transformation rules;
- dynamic semantics.

Annex F.1 [71] provides a general overview of the formalisms employed for the specification. The static semantics of SDL is defined in Annex F.2 [72], which includes the first three parts listed above, and the dynamic semantics is treated by Annex F.3 [73]. In the following, we give a brief overview of the formal semantics of SDL. Even if the dynamic semantics of SDL is not relevant to our work, we discuss this topic briefly.

Static semantics. A concrete and abstract grammar is formally specified for SDL. The production rules of both grammars are defined using a modified variant [67] of the *Backus-Naur-Form (BNF)*. The abstract representation of an SDL specification is obtained from the concrete representation by applying transformations. However, the well-formedness of an SDL specification considering the grammar must be ensured, before the transformations can be applied. In the case of SDL, a verification of the syntactical correctness is insufficient because context-sensitive conditions, such as visibility rules for type definitions, have also to be taken into account. The required well-formedness conditions are specified in terms of a first-order predicate logic. The most important well-formedness conditions of SDL are as follows:

- *Scope/visibility rules:* This kind of rules is used to ensure that only the type definitions visible to a context or scope are referenced by statements and expressions.
- *Disambiguation rules:* In the case of ambiguities of identifiers, these rules are used to determine the correct type definition.
- *Data type consistency rules:* These rules ensure that the data types of the actual arguments of operation calls are compatible with those of the formal parameters. In the case of assignments, the rules ensure that the data type of a variable and that of the expression are compatible.

The first-order predicate logic used to define the static semantics of SDL distinguishes between domains, functions, and predicates. Domains can consist of explicitly defined unary predicates or non-terminal symbols of SDL's grammar. Predicates that are not unary, are defined as functions which have a Boolean result type and a name that is prefixed with is. Below, we show an example of a well-formedness condition for the abstract syntax, which ensures that the type of a constant expression of a variable definition is equal to the one denoted by sort reference identifier.

```
∀ d ∈ Variable−definition: d.s−Constant−expression ≠ undefined ⇒
  d.s−Constant−expression.staticSort1 =
  d.s−Sort−reference−identifier
```

As argued in [43, 37], an important objective for developing the formal semantics of SDL-2000 was to keep the scope of the abstract syntax small so its formalization could be realized. Therefore, not all language constructs of the concrete grammar of SDL have corresponding counterparts in the abstract grammar.

For instance, this is the case for SDL's short-hand notations or remote procedure calls. Thus, the concerned language constructs have to be replaced by other ones. Rewrite rules are employed to formalize the required transformations, which must be applied to the concrete representation of an SDL specification in a defined order.

For example, the shorthand notation for the value assignment of indexed data types, such as String, have to be transformed into a call to the implicitly defined Modify operation of the involved data type. The following rewrite rule is defined for this transformation in Annex F.2:

```
<assignment>(<indexed variable>(var, params), expr) =8=>
  <assignment>(
    var,
    <method application>(
      var,
      <identifier>(empty, "Modify"),
      expr
    )
  )
```

Let us assume that the following assignment statement is given and we apply the rewrite rule above to it:

```
myVar[1] = 10
```

We obtain the following result after applying the rewrite rule:

```
myVar = myVar.Modify(myVar, 1, 10)
```

After applying all rewrite rules specified in Annex F.2, the concrete representation of an SDL specification obtained as result can be transformed into the corresponding abstract representation using a one-to-one mapping.

Dynamic semantics. The formalization of the dynamic semantics of SDL is realized using an operational approach based on the formalism of an *abstract state machine (ASM)*, as discussed, e.g., by Borger and Stark [19]. This formalism is the mathematical foundation for the specification of the *SDL Abstract Machine (SDL)*. According to Eschbach [37], abstract code based on the *Abstract Syntax Tree (AST)* of an SDL specification can be derived and executed on the SAM.

The formal specification of the dynamic semantics is divided into the parts explained below, and Fig. 3.4 illustrates the relationship between these parts. We first consider the specification of the SAM as the most important component of dynamic semantics. The SAM specification is divided into three parts, which are as follows:

- *Signal flow:* This part specifies the basic concepts for signal flows as required for signals, timers, gates and channels;

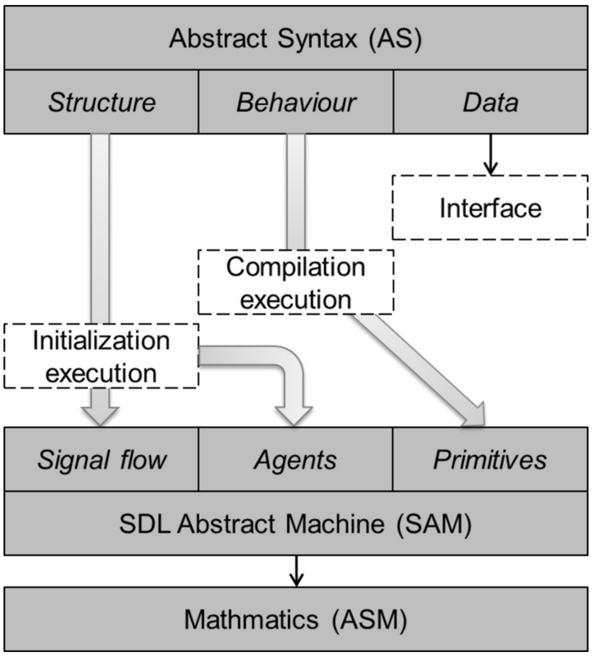


Figure 3.4: Formalization of the dynamic semantics of SDL according to Z.100 Annex F1 [71]

- *Agents:* Defines dedicated ASM agent types for each SDL agent type, e.g., block agent; and
- *Primitives:* This part embraces primitives for processing signals and behaviour. These primitives can be considered as the abstract machine instructions of the SAM.

The *compilation function* is employed to map the AST of an SDL specification to the SAM behavioural primitives. The compilation function can be regarded as an abstract compiler, which generates SAM statements from an AST.

The *initialization* defines how the initial system state (i.e., the static structure of the system) of the SAM is to be derived from an SDL specification. In particular, this applies to the SDL system agent, its signal flows, and its contained block agents.

The approach used to specify the data semantics differs from the other parts of the dynamic semantics, because the semantics of data is defined based on an *interface*. Thus, the data model of SDL can be changed to fulfil the particular requirements of a specific application domain, if required.

3.2.3 The UML Profile for SDL

The following provides a brief overview of the withdrawn edition of the *UML profile for SDL* released in 2007 [66]. To refer to this version of the UML profile, we use the acronym SDL-UML 2007. The stereotypes contained in SDL-UML 2007 map to the abstract syntax of SDL-2000 [64]. As in the case of models specified employing native SDL, structural and behavioural aspects must also be captured by a UML model with applied UML profile for SDL. Therefore, we divide our subsequent introduction into these two categories, and examples taken from our SIP case study [92] shall serve to illustrate the application of the SDL-UML stereotypes.

Structural aspects. In a native SDL model, type definitions (e.g., agent and data type definitions) and the system structure can be contained in the same diagram (see Fig. 3.3b). By contrast, two different diagram types have to be used in the case of SDL-UML models. Class diagrams contains the type definitions (see Fig. 3.5a), while composite structure diagrams (see Fig. 3.5b) define the structure and interconnections of a system to be modelled.

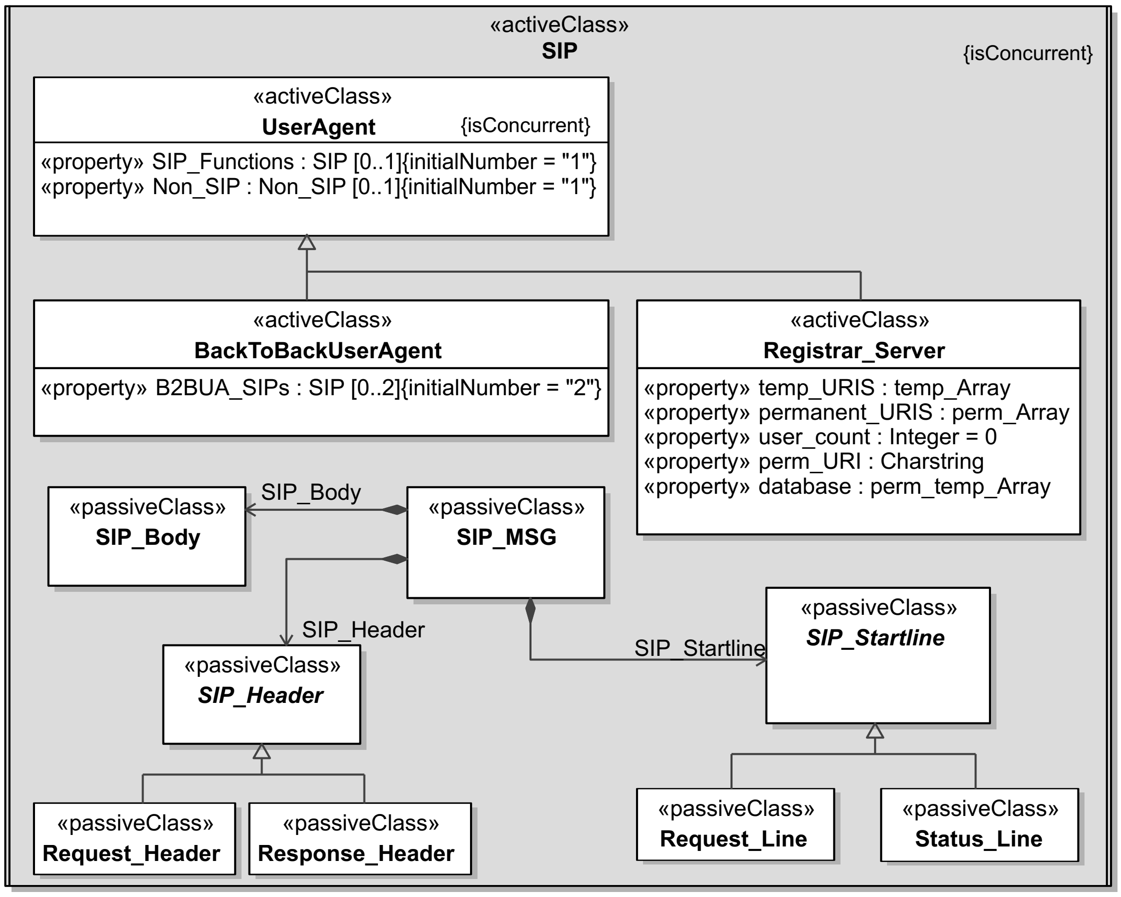
Class diagrams: In a class diagram of an SDL-UML model, a *Class* stereotyped with «ActiveClass» represents an SDL agent type definition, whereas an applied «PassiveClass» stereotype denotes an SDL object data type. The «ActiveClass» stereotype owns the isConcurrent attribute to indicate that a stereotyped *Class* contains further «ActiveClass» classes. If this attribute has assigned the value 'true', contained «ActiveClass» classes represent SDL block agent types, or otherwise SDL process agent types. The initialNumber attribute is another attribute of the «ActiveClass» stereotype, which determines the initial number of agents.

In addition to the aforementioned stereotypes, the following stereotypes that can be applied to certain *Classifier* types are available for specifying type definitions in a SDL-UML model:

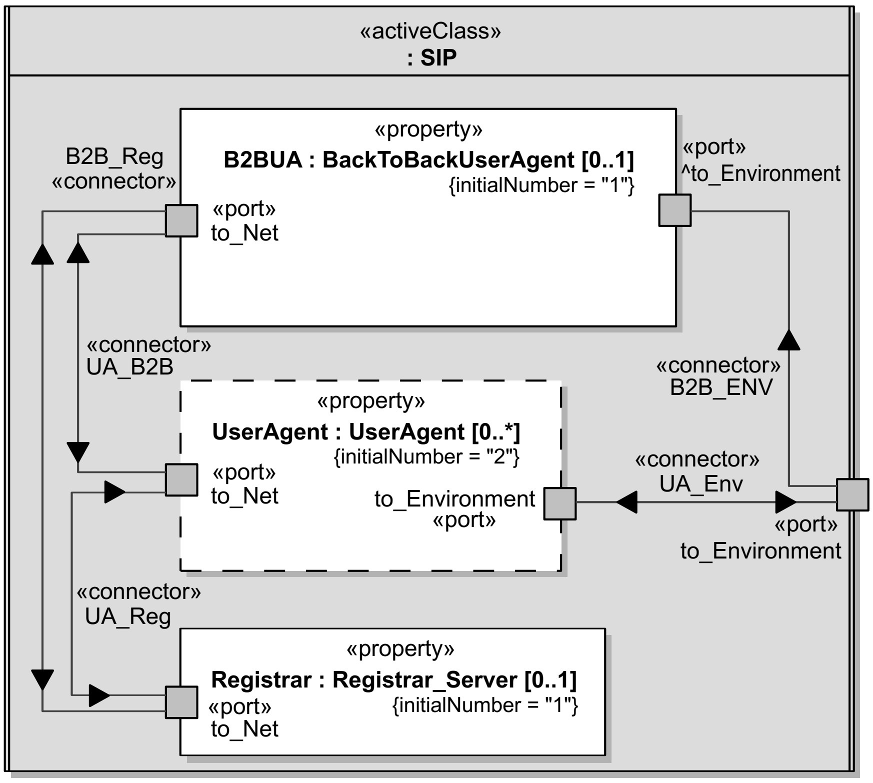
- «Signal» for representing an SDL signal definition;
- «Timer» that corresponds to an SDL timer definition;
- «Interface» that introduces an SDL interface definition; and
- the stereotypes «DataType», «PrimitiveType» and «Enumeration» for specifying different kinds of SDL value data type definitions.

Some of the above stereotypes constrain the *attribute* set of a *Classifier* to be empty; otherwise, the «Property» stereotype is automatically applied to each *Property* that is an *attribute* of a *Classifier*. Depending on the containing *Classifier*, a *Property* can represent various SDL language constructs. If the «Property» stereotype is applied to an *attribute* of an «ActiveClass», this either denotes an SDL variable definition or an SDL agent, which is determined by the *type* of a *Property*. If the *type* refers to an «ActiveClass», this corresponds to an SDL agent; whereas an SDL variable definition is represented in all other cases. By contrast, a «Property» that is owned by a «DataType» or «PassiveClass» corresponds to an item of an SDL data type definition.

An example class diagram is shown in Fig. 3.5a, where the SDL agents and data types required for our SIP case study are defined. The three *Classes*, stereotyped



(a) SDL-UML class diagram containing type definitions



(b) UML activity diagram

Figure 3.5: SDL-UML class and composite diagrams

with «ActiveClass» and located at the upper part of the diagram, correspond to the SDL agent types of the SDL model shown in Fig. 3.3a. The structure of the most important SIP messages is defined in the bottom part the diagram. An object-oriented modelling approach is applied to specify all type definitions contained in the diagram. For example, BackToBackUserAgent and Registrar_Server inherit from UserAgent.

Since the *type* properties of the attributes owned by the UserAgent class refer to «ActiveClass» classes, these attributes correspond to the specification of SDL agents. By contrast, all attributes of the Registrar_Server represent SDL variable definitions because their *type* properties refer to «PassiveClass» classes.

Composite structure diagrams: The structure and communication channels of a system have to be modelled using UML composite structure diagrams, where SDL agents are represented as instances of *Classes* stereotyped with «ActiveClass». The communication relationships between the different class instances are defined by *Ports* and *Connectors*. A *Port* that has applied the «Port» stereotype corresponds to an SDL gate definition, and a *Connector* that is stereotyped with «Connector» represents an SDL channel definition.

The communication relationships of the SIP system of our case study are illustrated in Fig. 3.5b. The instance of the UserAgent corresponds to a system agent, and the instances contained in it represent the block agents of the modelled system.

Behavioural aspects. As in the case of the structural aspects of a system that is modelled using SDL-UML, the behavioural aspects are also specified using two diagrams types, whereas one diagram type is sufficient in native SDL. A UML state machine diagram corresponds in a broader sense to that of SDL. However, no details of the transition effect can be specified in the UML variant. Instead, one of the UML behaviour diagrams, such as the activity diagram, has to be used for this purpose. In the case of SDL-UML, only UML activity diagrams can be employed, and the stereotypes defined for this kind of diagram represent the statements of SDL's action language.

In addition to the behaviour specification of «ActiveClass» classes, state machine diagrams of SDL-UML can define the behaviour of *Operations*, which represent SDL procedures. A significant difference between the state machine diagrams of SDL and UML is that the number of element types available for the UML variant is lower than that of the SDL variant. This circumstance is also reflected in SDL-UML because exactly one stereotype is defined for each UML element type (e.g., *Transition*) usable to model UML state machines.

State machine diagrams: In SDL-UML state machine diagrams, SDL state nodes are represented by the «State» stereotype, while the «Transition» stereotype corresponds to an SDL transition. The various types of SDL terminating nodes for an

SDL transition are represented by the «Pseudostate» stereotype, where the concrete variant is determined by the *kind* property of the *Pseudostate* that has applied the stereotype.

As we already pointed out, the most important properties of a *Transition* are *trigger*, *guard* and *effect*. In SDL-UML models, the «Transition» stereotype has to be applied to each *Transition* of a *StateMachine*. The *effect* of such a *Transition* can only be defined employing activity diagrams. Each action of an *Activity* maps to the list of graph nodes of an SDL transition. The purpose of the *trigger* property is comparable to that of the SDL input node, which triggers an SDL transition. Various *Event* types can be used to define the *trigger* of a *Transition* in UML. By contrast, only the *SignalEvent* is valid for SDL-UML, and thus, it corresponds to the SDL input node.

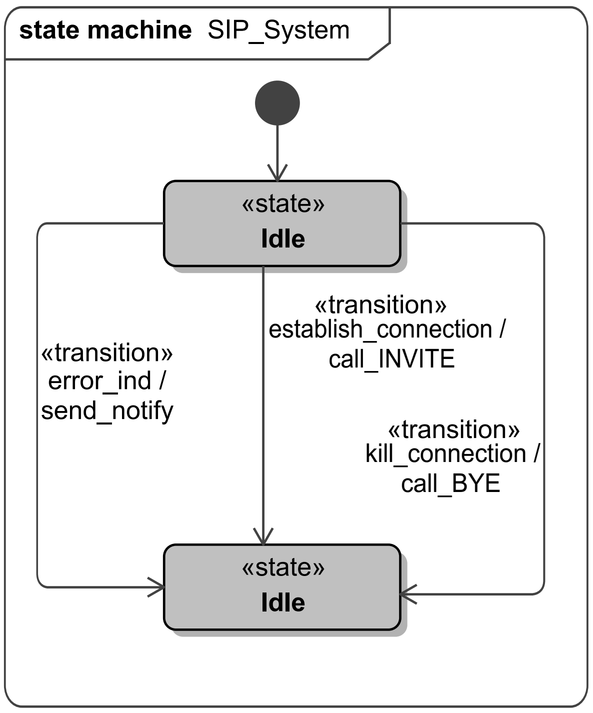
In addition to simple states, SDL-UML supports the use of submachine states that correspond to SDL's composite states. Thus, the specification of hierarchical state machines is enabled in SDL-UML models. To define a submachine «State», its *submachine* property has to refer to another *StateMachine*.

An example of an SDL-UML state machine is shown in Fig. 3.6a, which corresponds to the SDL state machine depicted in Fig. 3.3b. When comparing both diagrams, it becomes obvious that the details of the *effect* of a «Transition» are not displayed in the diagram. Instead, this aspect is captured by an activity diagram referenced by the *effect* property. Furthermore, there are no graphical symbols representing the trigger property of a «Transition».

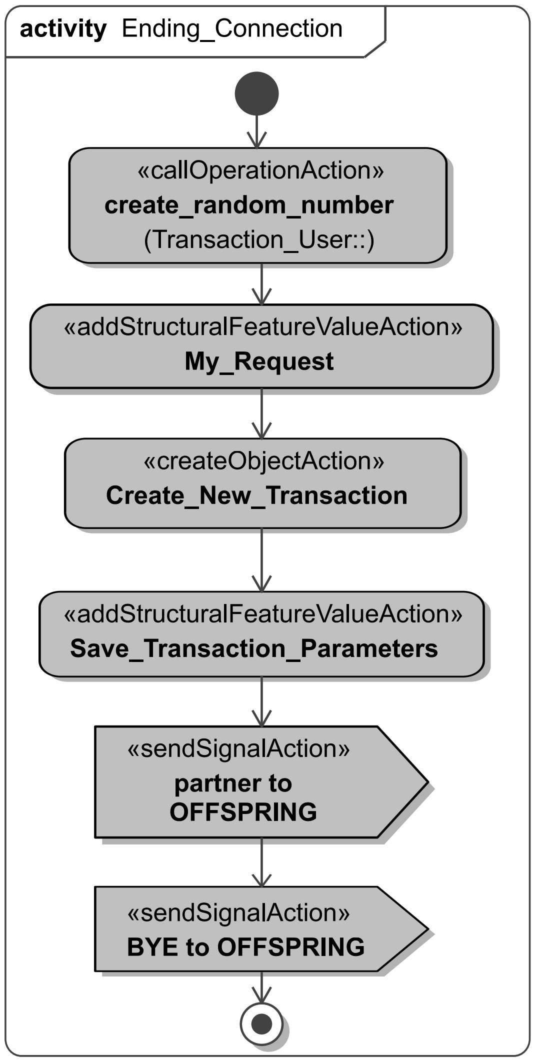
Activity diagrams: Apart from the specification of transition effects, activity diagrams define the behaviour of *Operations* owned by «DataTypes». Various UML element types are available for modelling an *Activity*, where most of them are subtypes of *Action*. However, only a subset of these element types is required to represent the statements of SDL's action language. The execution order of the *Actions* contained in an SDL-UML activity diagram is specified using *ControlFlows*. Because the input values of *Actions* are passed using variables and *ValuePins*, *ObjectFlows* are not required in activity diagrams of SDL-UML models.

Fig. 3.6b shows an activity diagram example, which defines the *effect* of a «Transition» of a state machine in our SIP case study. Based on this example, we explain the most important stereotypes applicable to *Actions*.

Starting from the *ActivityStartNode*, all actions shown in the diagram are processed in the order defined by the *ControlFlows* contained in the diagram. The first action is a «CallOperationAction» that invokes a procedure. This corresponds to an SDL call node used to invokes an SDL procedure. Then, a value is assigned to a «Property» of an «ActiveClass» by employing an «AddStructuralFeatureValueAction» which represents an SDL assignment. Thereafter, the «CreateObjectAction» instantiates an «ActiveClass». This type of action represents an SDL create request node. The fourth action is another value assignment followed by



(a) State machine diagram



(b) Activity diagram as effect of a transition

Figure 3.6: SDL-UML state machine and activity diagram examples.

two «SendSignalActions» which define the creation and transmission of signals. Both actions correspond to SDL output nodes.

In addition to the stereotypes above, SDL-UML provides further stereotypes which are applicable to *Actions* and *StructuredActivityNodes*, e.g, for starting and stopping timers or for manipulating the control flow. The expressions and values of SDL are specified using *Expressions* and other subtypes inheriting from *ValueSpecifiation*. However, UML does not provide any graphical representation for these elements, so they cannot be displayed in diagrams.

3.3 Identified Shortcomings of SDL-UML 2007

In this section, we detail the major issues identified by our comparison of SDL-UML 2007 with the formal specification of SDL-2000. First, we discuss the disadvantages of the data type concept defined for SDL-UML 2007. Then, we analyse

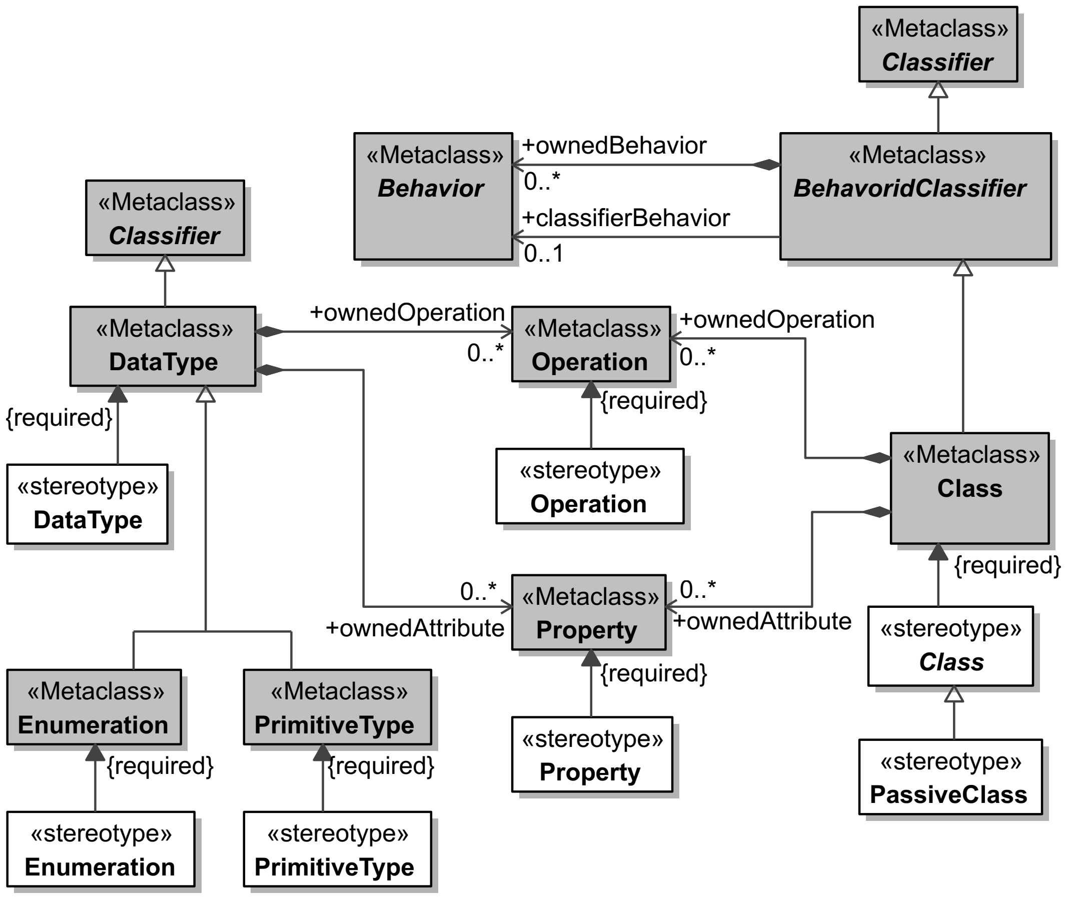


Figure 3.7: Abstract syntax of SDL-UML 2007 [66] for specifying data types

problems related to the representation of SDL expressions, and finally, we discuss the drawbacks of UML actions that represent statements of SDL's action language.

3.3.1 Drawbacks of the Data Type Model

Before we analyse the drawbacks of SDL-UML 2007 for specifying data types, we briefly introduce the stereotypes that define the associated abstract syntax of SDL-UML 2007 as shown in Fig. 3.7. In addition to these stereotypes, the figure also contains the extended UML metaclasses.

The «PassiveClass» stereotype represents an SDL object data type definition, while SDL value data type definition are captured by the stereotypes «DataType», «PrimitiveType» and «Enumeration». Thus, the first obvious difference between the representation of object and value data types is the number of available stereotypes. Due to this fact, one could assume that SDL-UML 2007 offers more language concepts for value data types than for object data types. However, the latter are no longer supported by SDL-2010. Therefore, we do not analyse the limitations of object data types in SDL-UML 2007.

As discussed in the foundations chapter, the concrete syntax of SDL embraces more constructs than the abstract syntax. In addition, the stereotypes of SDL-UML 2007 map directly to constructs of SDL's abstract syntax. This raises the

```
/* Literal data type */
value type Monochrom { literals black, white = 256; }

/* Literal data type with inheritance and default value */
value type Greyscale inherits Monochrom {
  literals darkgrey = 64, mediumgrey = 128, lightgrey = 192 ;
  default white;
}

/* Syntype definitions */
syntype DarkColours = Greyscale constants <= 127;
syntype BritghtColours = Greyscale constants > 127;

/* Structure data type with operations */
value type PixelType {
  struct
    xpos, ypos Integer;
    colour Greyscale;
    optional isActive Boolean;  //Optional structure field
  /* Operation signature declarations */
  operators
    equalColours(PixelType, PixelType) −> Boolean;
  methods
    virtual setPixel(Integer, Integer, Greyscale);
  /* Operation behaviour specifications */
  operator equalColour(pixA PixelType, pixB PixelType) {
    return pixA = pixB;
  }
  virtual method setPixel(x Integer, y Integer, c Greyscale) {
    this.xpos := x;
    this.ypos := y;
    this.colour := c;
  }
}

/* Choice data type */
value type ColoursType {
  choice
    mono Monochrom;
    grey Greyscale;
}
```

Figure 3.8: SDL data type definitions specified in SDL's textual notation

question of whether this approach results in restrictions concerning the language concepts applicable to data types in SDL-UML models. We take a closer look at this topic employing exemplary data type definitions contained in Fig. 3.8. The given examples make use of the various data type constructors and other language concepts applicable to data types. Based on this example, we now discuss the major limitations of SDL-UML 2007 identified by us [85, 87].

Static and dynamic operations. A data type definition in the concrete syntax of SDL may have operation signatures for operators and methods. Provided an operation is not captured by the data interface of SDL's dynamic semantics (see Sec. 3.2.2), its behaviour must be specified in a data type definition. For example, take the equalColour operator of the PixelType data type shown in Fig. 3.8. The signature of this operator is specified at Line 22, and the behaviour definition starts at Line 26.

The operator signatures map to the set of static operation signatures of a value data type definition in the abstract syntax of SDL, while method signatures map to the set of dynamic operation signatures. By contrast, an «Operation» of a «DataType» in SDL-UML 2007 is always mapped to a static operation signature in the abstract syntax of SDL. Thus, the definition of dynamic operations is impossible in SDL-UML models.

Behaviour of operations. As mentioned above, operations may be associated with behaviour specifications, which are always located below the declarations for operation signatures, e.g., see the behaviour specifications below Line 25 in Fig. 3.8. Regardless of whether such a specification represents the behaviour of an operation or method, it is always mapped to the set of procedure definitions of a data type definition in SDL's abstract syntax.

In an SDL-UML model, the behaviour of a data type operation is represented by an «Activity», which is referenced by the *method* of an *Operation*. As shown in Fig. 3.7, the metaclass *DataType* directly inherits from the metaclass *Classifier*. Consequently, the metaclass *DataType* does not have an *ownedBehavior* property as it is the case for the metaclass *Class*, and thus, an «Activity» cannot be part of a *DataType*. Due to this reason, the «Activity» has to be stored elsewhere, rather than inside the enclosing *DataType*. However, this contradicts the visibility rules of SDL because only certain entities that are defined outside of a «DataType» may be accessible for such an «Activity».

Choice data types. In the concrete syntax of SDL, a data type definition can have a **choice** constructor that defines a list of items, where each item is associated with a certain type. A value assignment is only possible for the item that was selected during the instantiation of such a data type. An example for a data type definition with choice constructor is shown at Line 37 in Fig. 3.8. SDL-UML 2007 does not support the definition of data types with present choice constructor.

Optional fields of structured data types. The field of a data type definition with **struct** constructor can be defined as optional, e.g., see the isActive field at Line 19 in Fig. 3.8. Optional fields that have not assigned a value are considered to be not present. This condition can be tested using an implicitly introduced method. Although the specification of structured data types is possible in SDL-UML 2007, optional fields are not supported.

Syntypes. A **syntype** constrains an existing data type definition, and it can be referenced in the same manner as an ordinary data type. For instance, the two syntype declarations shown below Line 10 in Fig. 3.8 restrict the number of applicable literals of the Greyscale data type. SDL-UML 2007 does not specify a stereotype or a mapping for syntypes.

Data types with literal constructor. An SDL data type definition with a **literal** constructor enumerates all literals defined for that data type. Each literal is implicitly associated with an integer value representing its position in the list of literals of a data type definition. However, the number of a literal can also be specified explicitly, e.g., see the white literal of the Monochrome data type in Line 2 of Fig. 3.8.

In SDL-UML 2007, the «Enumeration» stereotype represents a data type definition with literal constructor. Each *EnumerationLiteral* contained in the *ownedLiteral* property of an *Enumeration* maps to a literal of the corresponding SDL data type. However, SDL-UML 2007 provides no possibility to explicitly define the position of a literal within the literal list of a data type. Consequently, literal types with non-consecutive numbering (e.g., the Greyscale type in Fig. 3.8) cannot be specified.

Default values. The concrete syntax of SDL enables the definition of default values for data type definitions. For instance, such a default value is defined for the Greyscale data type in Line 7 of Fig. 3.8. SDL-UML 2007 does not support the specification of default values.

Predefined data types. The predefined data types of SDL can be divided in simple and generic types. The first category is specified using data types with literal constructor, syntypes or concrete instances of generic data types, while the latter category consists of parametrized data type definitions. By contrast, all predefined simple data types of SDL-UML 2007 are specified as *PrimitiveType* instances with applied «PrimitiveType» stereotype, regardless in which manner their counterparts of SDL are specified. Furthermore, the generic data types are represented as *DataType* instances with applied «DataType» stereotype. All these types own certain *TemplateParameters* that correspond to SDL context parameters.

The representation of all predefined simple data types as instances of *PrimitiveType* raises some issues concerning the semantics. The first issue concerns the rules for inheritance, because specific rules apply to each kind of SDL data type. In addition, the representation approach used for SDL-UML 2007 does not consider the implicitly introduced operators and methods associated with the different kinds of data type constructors of SDL.

Conclusions. Due to the drawbacks we have identified above, we consider the stereotypes of SDL-UML 2007 for modelling data types to be inadequate. In addition, to enable the definition of choice data types and syntypes in SDL-UML models, new stereotypes that capture both language concepts of SDL shall be introduced. Due to the difference between the concepts for representing the predefined data types in native SDL and SDL-UML 2007, the concept of SDL-UML should be aligned to that of SDL. Only in this way can we ensure an error-free mapping of SDL-UML models to constructs of the abstract syntax of SDL.

3.3.2 Issues Concerning the Representation of SDL Expressions

In the following, we discuss the issues we have identified regarding the representation of SDL expressions in SDL-UML 2007, as reported in [85, 87]. However, before we go into the details, we briefly summarize the relationship between the syntax and semantics of expressions in SDL and the corresponding stereotypes of SDL-UML 2007.

SDL provides various expression types that are typically employed to calculate the values required for statements in an SDL model. The top-level constructs of SDL's abstract syntax for expressions is shown in Fig. 3.9a, where the expressions are categorized into constant and active expressions. The semantics of SDL distinguishes between the *static type* of an expression, which is determined by the static analysis, and the *dynamic type*, which is obtained by evaluating an expression at runtime. Both the static and the dynamic type of a constant expression are equal, while they can differ for an active expression. An SDL expression is considered to be constant, if it does not contain any kind of *imperative expression*, *create expression*, or a *value returning call node*. Therefore, some expression types shown in Fig. 3.9a may belong to both the category of constant and active expressions.

SDL-UML 2007 specifies nine stereotypes that extend subclasses of the UML *ValueSpecification* metaclass. The abstract syntax of these stereotypes and their relation to the extended UML metaclasses is shown in Fig. 3.9b. Except for the literal and the variable access expression of SDL, almost all expression types defined in SDL's abstract syntax map to the «Expression» stereotype. Literal values for user-defined data types and for almost all predefined data types of SDL-UML are captured by the «InstanceValue» stereotype. The second purpose of this stereotype is to represent an SDL variable access expression.

An exception to the specification of literal values in SDL-UML are those for the four predefined Boolean, String, Integer, and Natural data types, because each is represented by a dedicated stereotype. For example, an Integer value is captured by the «LiteralInteger» stereotype.

The «Expression» stereotype. As we justified above, the «Expression» stereotype represents almost all expression types of SDL.

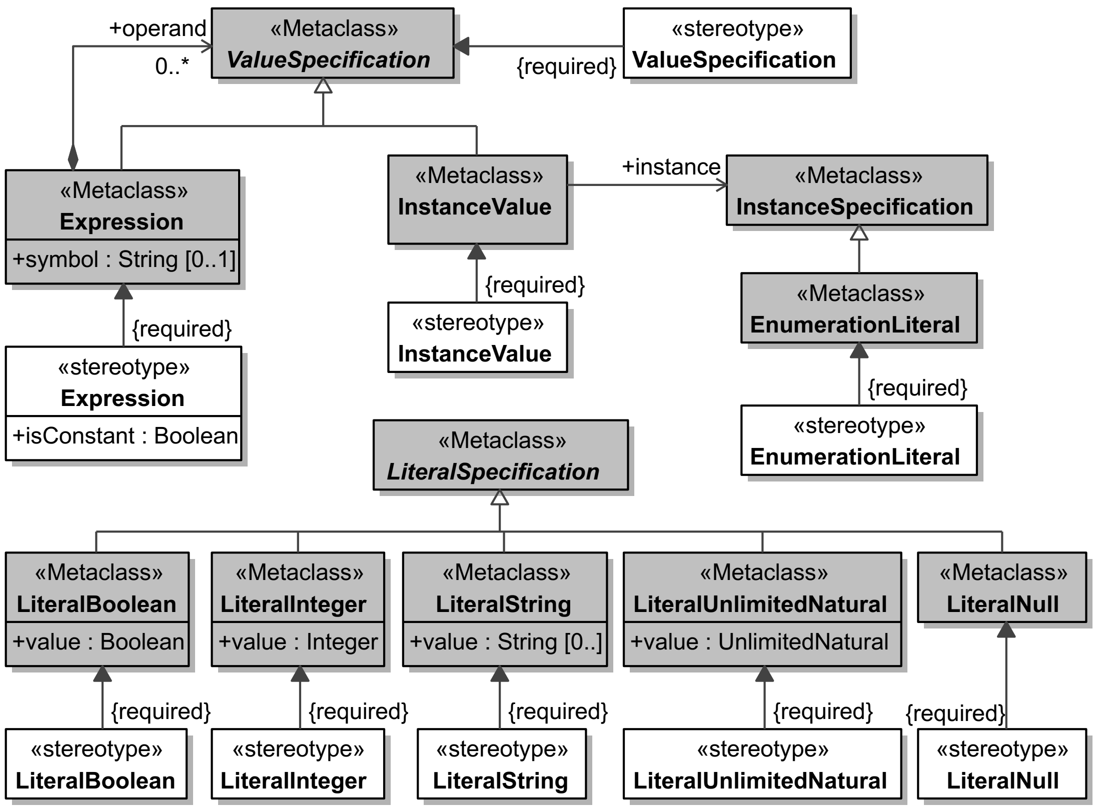
Static semantics: One might expect that all constraints of SDL's static semantics concerning expressions, as defined in Rec. Z.100 [64], are also specified for the «Expression» stereotype. But this is not the case; in fact, only four constraints

```
Expression =
  Constant−expression | Active−expression

Constant−expression =
  Literal | Conditional−expression | Equality−expression | Operation−application |
    Range−check−expression

Active−expression =
  Variable−access | Conditional−expression | Operation−application |
    Equality−expression | Imperative−expression | Range−check−expression |
    Value−returning−call−node | State−expression
```

(a) Abstract syntax of SDL-2000 [64]



(b) Abstract syntax of SDL-UML 2007 [66]

Figure 3.9: Abstract syntax of SDL-2000 and SDL-UML 2007 for expressions

are present, as shown in Fig. 3.10. For this reason, we consider SDL's static semantics to be incorrectly captured by the «Expression» stereotype, and thus, the static semantics of expressions in SDL-UML models cannot be verified.

Representation of concrete syntax: A second drawback of the stereotype «Expression» concerns the approach applied to represent the different SDL expression types. In contrast to other stereotypes of SDL-UML 2007, the syntax of the «Expression» stereotype is closely linked to SDL's concrete syntax for expressions. This is because the «Expression» stereotype introduces a textual notation that is

10.1 Expression

The stereotype «Expression» extends the metaclass *Expression* with multiplicity [1..1].

10.1.1 Attributes

1. isConstant: Boolean — true if the expression is a constant expression. This is a derived attribute.

10.1.2 Constraints

1. The «PassiveClass» *Class* for a create request shall have an operator with the name Make that has, as a result, an instance of the *PassiveClass Class*.
2. In the *operand* list of an «Expression» *Expression* that is mapped to an operation application, each *operand* shall match the type of the corresponding parameter of the operation.
3. In the *operand* list of an «Expression» *Expression* that is mapped to a conditional expression, the first *operand* shall be a Boolean and each of the other *operands* shall be of the same type, that is the type (of the «Expression» *Expression*).
4. The *type* (of the «Expression» *Expression*) shall match the type required in the context of the «Expression» *Expression*.

10.1.3 Semantics

...

- An «Expression» *Expression* that is a constant expression (*isConstant* is true) is mapped to an Expression that is a Constant-expression.
- An «Expression» *Expression* that is not a constant expression (*isConstant* is false) is mapped to an Expression that is an Active-expression.
- In the following, Expression is used to mean Constant-expression or Active-expression depending on the value of *isConstant*.
- Unless explicitly stated otherwise, the «Expression» *Expression* is mapped to the Operation-application alternative of Expression.
- When the «Expression» *Expression* is mapped to an Expression that is an Operation-application, the *symbol* is used to determine the Operator-identifier of the Operation-application.
- The *operand* list maps to the Expression list of the Operation-application.

...

10.1.4 Notation

...

The *symbol* in an <operand> with an <implies sign> is the text string for the <implies sign> qualified by the type for the context and the operand set is the «ValueSpecification» *ValueSpecification* pair for <operand> and <operand0>.

```
<operand> ::= <operand0>
  | <operand> <implies sign> <operand0>
```

...

Figure 3.10: Excerpt of the syntax and semantics of the «Expression» stereotype as specified in Rec. Z.109 [66] of 2007

similar to that of SDL expressions. The terminal and non-terminal symbols of each production rule map to the *operand* list and the *symbol* of a UML *Expression* according to a specific mapping rule. For instance, consider the production rule shown in the 'notation' section of Fig. 3.10. The <implies sign> defines the *symbol*, while the <operand> and <operand0> symbols map to items of the *operand* list of an *Expression*.

The crucial point is that the *symbol* shall be *"qualified by the type for the context"*, where we have to take into account that the *symbol* property is of String type. Thus, the *symbol* is a concatenated String consisting of the fully-qualified name of a UML element that determines the context and a part representing the operator sign. This rule applies not only to the given example, but also to almost all other production rules of the textual notation defined for the «Expression» stereotype.

One drawback of the approach above is that elements referred by the *symbol* of an *Expression* cannot be employed for property navigations in OCL expressions. Thus, the definition of OCL constraints for the «Expression» stereotype becomes nearly impossible.

Another disadvantage concerns the mapping towards constructs of SDL's abstract syntax. Since the *operand* list and the *symbol* represent terminal and non-terminal symbols of production rules at concrete syntax level, the values held by these *Expression* properties have to be transformed to corresponding constructs in the abstract syntax. However, this requires the application of various rewrite rules (see Sec. 3.2.2) defined by the formal semantics before the transformation to abstract syntax constructs can be conducted. Thus, the mapping defined for the «Expression» stereotype is rather complex and not straightforward.

The «InstanceValue» stereotype. As mentioned above, the «InstanceValue» stereotype is intended for two different purposes. According to the semantics defined for the «InstanceValue» stereotype, an *InstanceValue* whose *instance* property refers to an *EnumerationLiteral* represents an SDL literal expression. The *qualifiedName* property of such an *InstanceValue* shall be used to derive the SDL literal. By contrast, an *InstanceValue* represents an SDL variable access expression, if its *instance* property points to an *InstanceSpecification* for a *Class* or *DataType*. In this case as well, the *qualifiedName* property of an *InstanceValue* shall be employed to derive the SDL variable access expression.

The semantics and mapping rules of the «InstanceValue» stereotype as pointed out above raises some issues related to the semantics of UML [116] as well as that of SDL-UML 2007. In the following paragraphs, we discuss the identified issues.

The qualifiedName property: According to the UML specification [116], the value of the *qualifiedName* property of an *InstanceValue* is automatically derived at runtime based on the containing *Namespaces*. Therefore, the *qualifiedName* property does not contain any information concerning the *EnumerationLiteral* or instance of a *Class* or *DataType* represented by an *InstanceValue*. This information can only be obtained through the *instance* property of an *InstanceValue*. For this reason, we consider the *qualifiedName* property as an inappropriate source for the mapping to an SDL variable access or literal expression.

Values of predefined data types: A dedicated subtype of *LiteralSpecification* must be used to specify a value for UML's predefined *PrimitiveTypes*. For instance, a value for the Integer *PrimitiveType* is represented as an instance of *LiteralInteger*.

However, the UML specification [116] leaves open how to specify values for an user-defined *PrimitiveType*. Based on UML's semantics for the *InstanceSpecification*, one might conclude that the major objective of this language concept is to represent instances of *Classifiers* that have defined structural features, such as owned attributes. Even if a *PrimitiveType* is a certain subtype of *DataType*, and thus also of *Classifier*, the UML specification [116] gives the following statement:

A PrimitiveType defines a predefined DataType, without any substructure.

Due to the statement above, we assume that a *PrimitiveType* shall not have any structural feature, so that values for such a data type have no substructure. Due to the non-existent structural features, we consider the use of an *InstanceValue* to specify *PrimitiveType* values to be unsound. Consequently, the semantics of the «InstanceValue» stereotype in SDL-UML 2007 is inappropriate and should be revised.

Four PrimitiveTypes as exception. As we justified at the beginning of this section, values of four predefined data types are captured by dedicated stereotypes shown in Fig. 3.7, whereas values of all other data types shall be represented by *InstanceValues* stereotyped with «InstanceValue». The underlying motivation is neither known to the author of the present work, nor is a corresponding justification given in the 2007 edition of Rec. Z.109 [66].

The above fact leads to a rather complex abstract syntax for the value specification in SDL-UML 2007. To reduce this complexity, it should be considered to use only a single stereotype that is applicable to represent values for all predefined and user-definable data types in SDL-UML models.

Conclusions. Based on our analysis, we consider the mix of concrete and abstract syntax constructs for the definition of the «Expression» stereotype as one of the major drawbacks of SDL-UML 2007. Furthermore, almost all expression types of SDL are captured by the «Expression» stereotypes, while only a small subset of SDL's static semantics is captured by the constraints defined for the stereotype.

Another disadvantage of SDL-UML 2007 is that values for predefined and user-defined data types are represented by different stereotypes, resulting in increased complexity. In particular, the «InstanceValue» stereotype contributes to this because it is employed for both the representation of values and for variable access expressions.

3.3.3 Shortcomings of SDL's Action Language

Since the expressions of SDL are mainly used in combination with statements of SDL's action language, we wonder whether the syntax and semantics of stereotypes representing SDL statements are sound in SDL-UML 2007. To further analyse this subject, we exemplary examined the syntax and semantics of the two stereotypes «SendSignalAction» and «SetAction» in relation to their counterparts of SDL-2000 [64].

The abstract syntax of both stereotypes is shown in Fig. 3.11. To improve the comprehension of this aspect, not only the extended UML metaclasses but also their most important superclasses are included in the figure. In addition, all *Associations* and *Properties* of the metaclasses and stereotypes that are relevant for the syntax and semantics are displayed in the figure.

The «SendSignalAction» stereotype. An SDL output node is represented in SDL-UML 2007 by the «SendSignalAction» stereotype that extends the UML metaclass of the same name, as shown in the upper part of Fig. 3.11. Because this metaclass is a specialization of *InvocationAction*, it inherits the two properties *onPort* and *argument*. The *onPort* property is optional and defines the *Port* of a *Classifier* to be used to transmit an event that is originated by the current *InvocationAction*. The values that shall be passed as input to an event are specified by the *InputPins* of the *argument* property of an *InvocationAction*. In addition to the inherited properties, the *SendSignalAction* introduces the *signal* and *target* properties. The first property defines the *Signal* for which an event shall be created, while the *target* property specifies the receiver for the transmitted event.

Apart from the above properties of the extended metaclass, the «SendSignalAction» does not introduce additional attributes. Thus, all information required for the mapping towards an SDL output node have to be determined based on a *SendSignalAction* instance that has the «SendSignalAction» stereotype applied. The defined mapping rules are listed in clause '9.20.3 Semantics' of Fig. 3.12, and the two constraints defined by the stereotype are shown in clause '9.20.2 Constraints'.

If we compare the constraints and mapping rules of the «SendSignalAction» stereotype, we find that only two metaclass properties are covered by the constraints, while four properties are captured by the mapping rules. Strictly speaking, the two metaclass properties *onPort* and *target* are captured by the two constraints, but not the *signal* and *argument* properties. Especially for the last two properties, it is questionable whether constraints are missing when compared to the static semantics of SDL. The following two constraints concerning the SDL output node are defined in the SDL-2000 specification [64]:

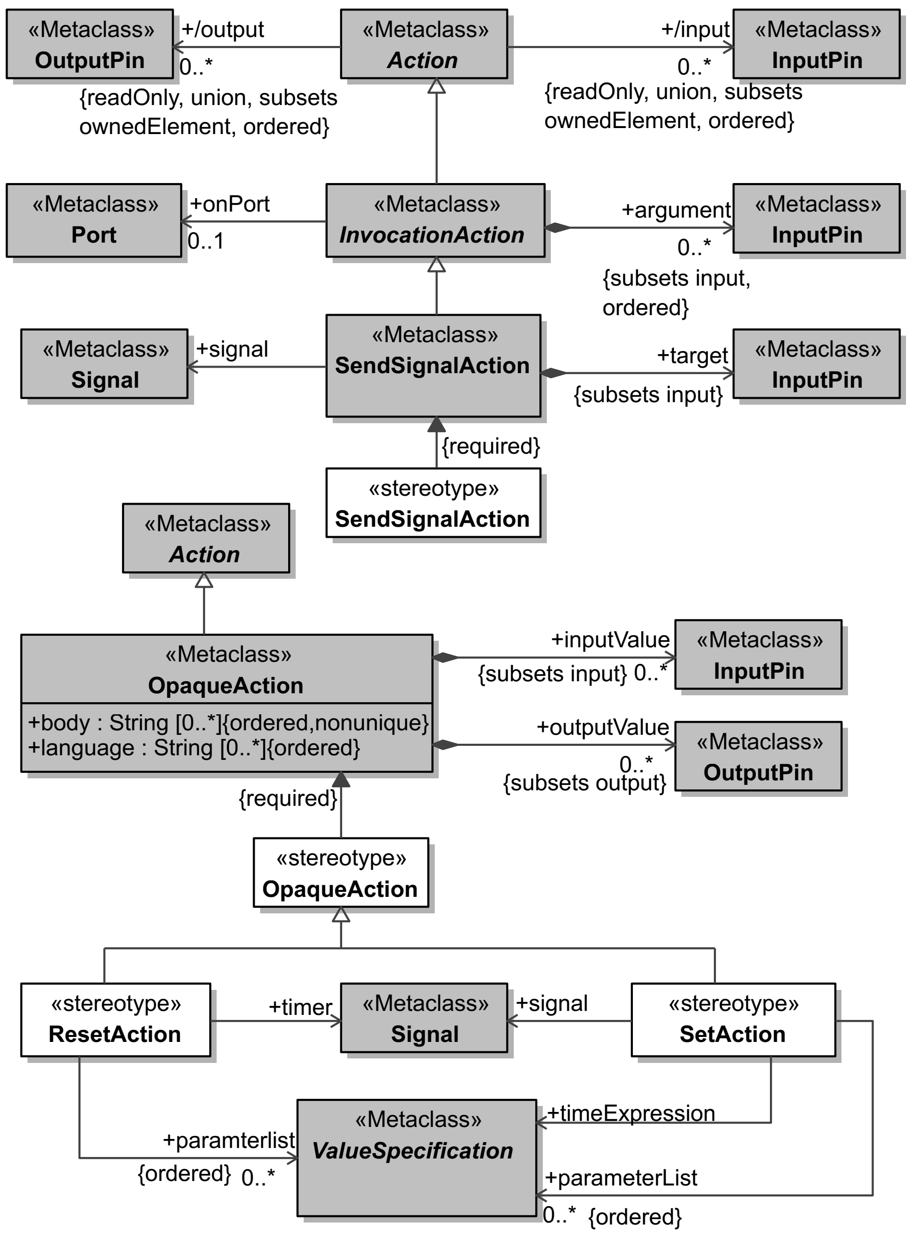


Figure 3.11: Abstract syntax of exemplary stereotypes representing SDL statements in SDL-UML 2007 [66]

1. *The length of the list of optional Expressions must be the same as the number of Sort-reference-identifiers in the Signal-definition denoted by the Signal-identifier.*

2. *Each Expression must be sort compatible with the corresponding (by position) Sort-identifier-reference in the Signal-definition.*

According to the mapping rules of the «SendSignalAction» stereotype, the expression list of an SDL output node corresponds to the *argument* property of the extended UML metaclass. Thus, we conclude that both constraints of SDL cited above are missing for the stereotype.

Apart from the missing constraints, when we consider the «SendSignalAction» stereotype in Fig. 3.12, the question arises whether the specification of Constraint 1 is sound. This constraint specifies that the *target* property shall consist of a *ValuePin*, but no rules concerning the *value* to be represented by this *ValuePin* are made. Thus, we can only guess whether this value should be defined by an «Expression» and which result *type* it should have.

The «SetAction» stereotype. A timer is started in SDL by a set node statement. In addition to the initial start value, this statement can define values that are passed to the timer via defined parameters. An SDL set node is represented by the «SetNode» stereotype that inherits from the «OpaqueAction» stereotype. Thus, «SetNode» indirectly extends the UML metaclass *OpaqueAction*, as shown in Fig. 3.11. For the sake of completeness, this figure also contains the «ResetAction» stereotype, which can be employed to reset a started timer.

The *OpaqueAction* metaclass provides the two properties *inputValue* and *outputValue*, where only the latter is relevant for the stereotypes discussed above. In addition to these properties, the «SetAction» has defined the attributes *timer*, *timeExpression*, and *parameterlist*. The *timer* attribute refers to a *Signal* with applied «Timer» stereotype, which represents the timer to be started, and the initial value for this timer is determined by the *timeExpression* attribute. In addition, the values for the optional timer parameters can be defined with the *parameterList* attribute.

The specification of the «SetAction» stereotype, as defined in Rec. Z.109 [66] of 2007, is shown in the bottom part of Fig. 3.12. The constraints introduced by this stereotype are listed in clause '9.21.2', and the mapping rules are specified in clause '9.21.3'.

If we consider the mapping rules of the stereotype, we can see that all attributes of the stereotype above are mapped to corresponding items of the SDL set node. Therefore, we conclude that the mapping rules are sound and complete. However, this is not given for the constraints of the stereotype because of the issues analysed below.

Since the *timer* attribute is of type *Signal*, it can refer to any kind of *Signal* without taking an applied stereotype into account. However, because of the given description (see Fig. 3.12) for this attribute, we assume that only a *Signal* with applied «Timer» stereotype is a valid value for it. For this reason, we conclude that a constraint capturing this condition is missing.

Another issue concerns the manner in which Constraint 1 of the «SetAction» stereotype is defined. The textual specification of this constraint contains the term *"... shall match ..."*, which is very vague and can be misleading. The entire Rec. Z.109 of 2007 [66] does not contain any further definition for this term. Hence, we consider the static semantics [64] defined for the SDL set node, and there we can find the following counterpart for Constraint 1 of the «SetAction» stereotype:

9.20 SendSignalAction

The stereotype «SendSignalAction» extends the metaclass *SendSignalAction* with multiplicity [1..1].

9.20.1 Attributes

No additional attributes.

9.20.2 Constraints

1. The *target* property shall reference a *ValuePin*.
2. The *onPort* property shall reference a *Port* of the container «ActiveClass» *Class* of the «SendSignalAction» *SendSignalAction*.

9.20.3 Semantics

1. A «SendSignalAction» *SendSignalAction* is mapped to an Output-node.
2. The *qualifiedName* of *signal* property maps to the Signal-identifier.
3. The *target* property maps to the Signal-destination.
4. The *onPort* property maps to the Direct-via.
5. The *argument* property maps to the Expression list.

9.21 SetAction

A timer is started with a set action represented by a «SetAction» stereotype. The «SetAction» stereotype is a concrete subtype of the stereotype «OpaqueAction».

9.21.1 Attributes

1. parameterlist: *ValueSpecification* [*] — The expressions that correspond to the actual parameters of the timer.
2. timer: *Signal* — The «Timer» *Signal* that represents the timer that is started by the action.
3. timeExpression: *ValueSpecification* — The duration that determines when the timer will expire.

9.21.2 Constraints

1. Each item in the *parameterlist* shall match the corresponding *ownedAttribute* of the timer.
2. The *timeExpression* shall be of the Time type.

9.21.3 Semantics

1. A «SetAction» *OpaqueAction* is mapped to a Set-node.
2. The *timer* maps to the Timer-Identifier.
3. The *parameterlist* maps to the Expression list and *timeExpression* maps to Time-expression.

Figure 3.12: Syntax and semantics of the stereotypes «SendSignalAction» and «SetAction» as specified in Rec. Z.109 [66] of 2007

The sorts of the list of Expressions in the Set-node and Reset-node must correspond by position to the list of Sort-reference-identifiers directly following the Timer-name identified by the Timer-identifier.

When translated into the syntax of SDL-UML, the above condition means that the *type* of an item of the *parameterlist* must be the same as the *type* of the corresponding attribute of the «Timer» *Signal*. In addition, the number of items contained in the *parameterlist* must be equal to the number of attributes defined for the «Timer» *Signal*. Thus, we conclude that Constraint 1 of the «SetAction» stereotype is not correctly defined and has to be replaced by a new one capturing the above conditions.

Because SDL expressions in SDL-UML are only represented by certain subtypes of *ValueSpecification*, which have applied one of the previously discussed stereotypes (see Sec. 3.3.1), a further constraint is required that ensures this condition for the *parameterlist* attribute.

Conclusions. Based on the two stereotypes «SendSignal-Action» and «SetAction», we have shown that their syntax and semantics have gaps and errors with regards to referenced expressions. This applies not only to these stereotypes but to all other stereotypes that represent statements of SDL's action language. Therefore, apart from the stereotypes for representing SDL expressions, the stereotypes that correspond to SDL statements must also be revised. Otherwise, the creation of SDL-UML models that are compliant to the static semantics of SDL is impossible.

3.4 Introduced Enhancements

Our proposals [92, 85] to remedy the drawbacks of SDL-UML 2007 as pointed out above are discussed in this section. These proposals also formed the foundation for our revision of the UML profile for SDL, which was released as a new edition of the Rec. Z.109 in 2013 [69]. In the following, this UML profile is referred to as *SDL-UML 2013*.

Because our work on SDL-UML 2013 took about four years, the final version of SDL-UML 2013 contains a few differences compared to our initial proposals. Most of these differences are originated from experience gained through our prototypical implementation of SDL-UML 2013, which is presented in Sec. 3.6. Further differences resulted from a different view of the ITU working group responsible for the standardisation of SDL, which does not necessarily reflect our opinion. Therefore, we comment on the concerned parts of our proposals.

3.4.1 Specification of Data Types

Based on our findings discussed in Sec. 3.3.1, a new concept for representing data type definitions in SDL-UML 2013 is introduced by us. The final abstract syntax of the stereotypes involved in specifying data types is shown in Fig. 3.13a. This abstract syntax differs in a few details when compared to our initial proposal presented in [87]; if necessary, we briefly explain the reason for a modification in the context of the affected stereotype.

The «DataTypeDefinition» stereotype. The «DataTypeDefinition» stereotype is the most general stereotype for representing SDL value data type definitions. It captures the general set of mapping rules and constraints that apply to all kinds of data type definitions. Therefore, all other stereotypes representing certain SDL data type variants inherit from the «DataTypeDefinition» stereotype, as is shown in Fig. 3.13a. Three of them represent the different data type constructors of SDL and one corresponds to the SDL syntype.

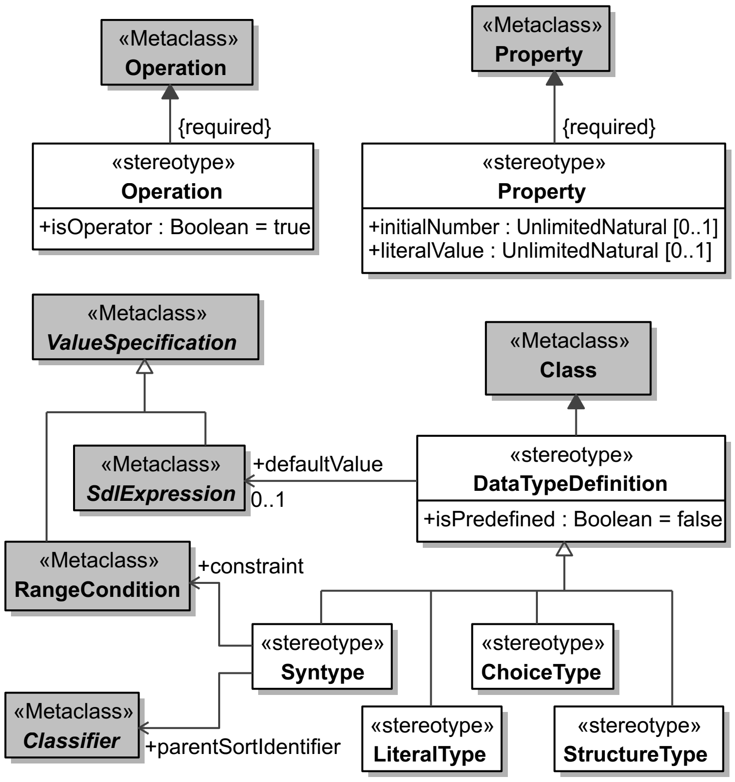
To avoid semantic ambiguities in SDL-UML models, the stereotype «DataType-Definition» and all its subtypes must not be used in combination with other stereotypes defined for SDL-UML 2013. Thus, we have not defined this stereotype as 'required'; otherwise, it would automatically be applied to all instances of *Class*, including instances of those metaclasses inheriting from *Class*, e.g., *StateMachine*.

Simple data types: As pointed out in the previous section, an SDL data type definition without a constructor is the simplest data type variant. We employ the «DataTypeDefinition» stereotype to specify this kind of SDL data type, and thus, it is not defined to be 'abstract' because, otherwise, it could not be applied to any model element.

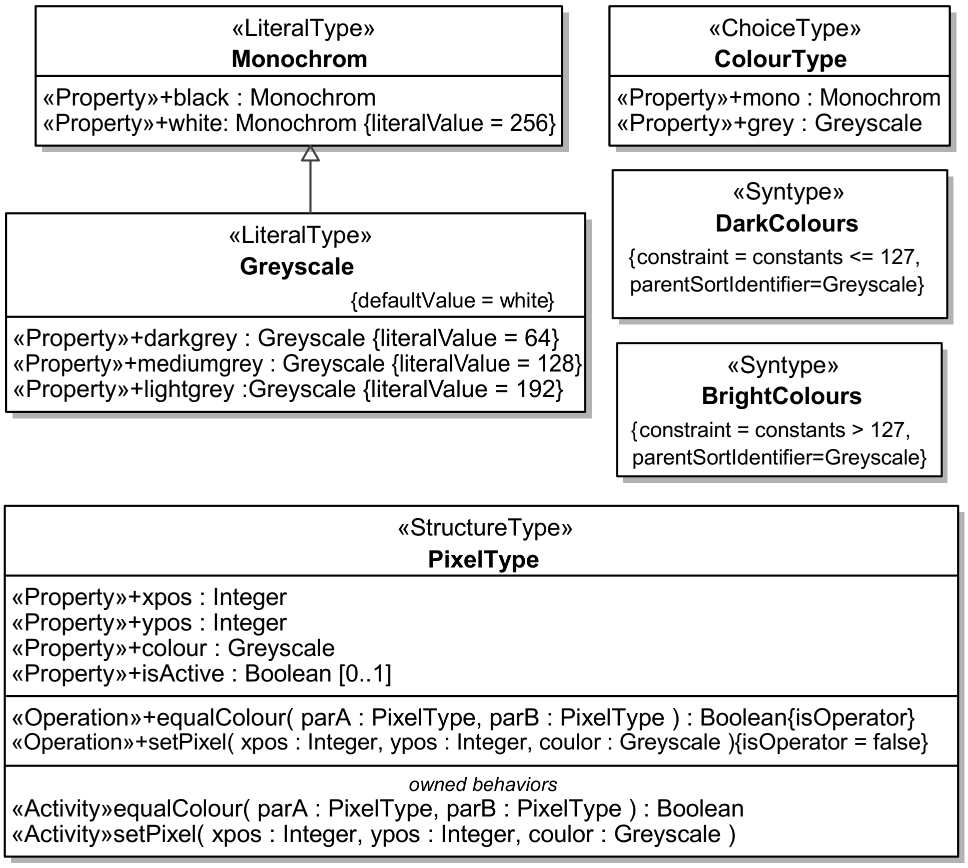
Behaviour specifications for Operations: Although it would be obvious to extend the *DataType* metaclass to enable the specification of data types, the «DataType-Definition» stereotype extends the *Class* metaclass of UML, as shown in Fig. 3.13a. Our major criterion for this choice is the ability of a *Class* to directly contain behaviour specifications for *Operations*, which is enabled through the *ownedBehavior* property. This solves the missing possibility to define behaviour specifications for data type operations in SDL-UML 2007. Furthermore, this approach ensures that the visibility of type definitions in the behaviour specifications of *Operations* complies with the visibility rules of SDL. For instance, the data type PixelType shown in Fig. 3.13b owns two «Activity» specifications that define the behaviour of its operations.

From a semantical point of view, one could argue that the *Class* metaclass is inappropriate for specifying data types, because of the different definitions for the *Class* and *DataType* metaclasses given in the UML specification [116]:

- *A Class classifies a set of objects and specifies the features that characterize the structure and behavior of those objects.*



(a) Revised abstract syntax of SDL-UML 2013 [69]


«LiteralType»
Monochrom
«Property»+black : Monochrom
«Property»+white: Monochrom {literalValue = 256}
«ChoiceType»
ColourType
«Property»+mono : Monochrom
«Property»+grey : Greyscale
«LiteralType»
Greyscale
{defaultValue = white}
«Property»+darkgrey : Greyscale {literalValue = 64}
«Property»+mediumgrey : Greyscale {literalValue = 128}
«Property»+lightgrey :Greyscale {literalValue = 192}
«Syntype»
DarkColours
{constraint = constants <= 127,
parentSortIdentifier=Greyscale}
«Syntype»
BrightColours
{constraint = constants > 127,
parentSortIdentifier=Greyscale}
«StructureType»
PixelType
«Property»+xpos : Integer
«Property»+ypos : Integer
«Property»+colour : Greyscale
«Property»+isActive : Boolean [0..1]
«Operation»+equalColour(parA : PixelType, parB : PixelType) : Boolean{isOperator}
«Operation»+setPixel(xpos : Integer, ypos : Integer, coulor : Greyscale){isOperator = false}
owned behaviors
«Activity»equalColour(parA : PixelType, parB : PixelType) : Boolean
«Activity»setPixel(xpos : Integer, ypos : Integer, coulor : Greyscale)

(b) Data type definition examples

Figure 3.13: Data type definitions and their abstract syntax in SDL-UML 2013

- *A DataType is a type whose instances are identified only by their value.*

Apart from the aspect of behaviour specifications, the main difference between a *Class* and a *DataType* is the manner how their instances are represented at runtime. The instances of a *Class* represent objects, while values are represented in the case of a *DataType*. In our view, this difference would be important if UML models were instantiated and executed directly, so the dynamic semantics of UML would have to be considered. Due to the translational semantics of SDL-UML, which is defined by a mapping to constructs of SDL's abstract grammar, we consider the dynamic semantics of UML to be less relevant for SDL-UML. The creation of a UML profile is always a trade-off between the semantics of UML and that of the DSL to be represented by a UML profile.

Data type specialization: An SDL data type definition can specialize a more general supertype by adding additional operations or type-related items, such as additional fields for data types with structure constructor. In SDL-UML, we can define a generalization relationship between data type definitions using the *general* property of a *Class* with applied «DataTypeDefinition» stereotype. Since SDL does not support multiple inheritance, we have constrained the *general* property so that at most one «DataType» can be referenced. For instance, the «LiteralType» Greyscale shown in Fig. 3.13b inherits from the «LiteralType» Monochrom.

Redefinition: Provided that two SDL type definitions have an inheritance relationship, an inheriting subtype can redefine type definitions contained in the supertype. Just as an SDL type definition, a UML *Classifier* also supports the ability to redefine another *Classifier*. The *redefinedClassifier* property of a *Classifier* is used to refer to all redefined *Classifiers*. In SDL-UML, a data type definition that is represented as a *Class* with applied «DataTypeDefinition» stereotype can redefine another data type definition by employing the *redefinedClassifier* property. Because multiple inheritance is not supported in SDL, the *redefinedClassifier* property of a «DataType» is constrained so it can have at most one reference to a redefined data type.

Default values: Because the specification of default values for data types is not supported in SDL-UML 2007, we have added the *defaultValue* attribute to the «DataTypeDefinition» stereotype, shown in Fig. 3.13a. The *defaultValue* has to be defined in terms of an *SdlExpression*, which we discuss in more detail in the next section. For instance, see the «LiteralType» Greyscale contained in Fig. 3.13b.

The «Operation» stereotype. As in the case of SDL-UML 2007, we use the «Operation» stereotype to represent operations of data type definitions. The stereotype «Operation» extends the metaclass *Operation* and is defined as 'required', so it is applied automatically. This stereotype is not only used in the context of data type definitions but also for *Operations* contained in an «ActiveClass». However, the semantics and constraints discussed in the following apply only for operations contained in data type definitions.

Operators and methods: SDL-UML 2007 does not support the explicit distinction between operators and methods. Apart from implicitly introduced operations for data type constructors, e.g., the structure constructor, operations can also be defined by a user. We employ the *isOperator* property of the «Operation» stereotype to make a distinction between an operator and a method. If the *isOperator* property is true, an *Operation* maps to an SDL static operation signature, and otherwise, to an SDL dynamic operation signature.

For instance, the «Operation» equalColour() of the «StructureType» PixelType shown in Fig. 3.13b represents an operator, whereas the «Operation» setPixel() defines a method.

Redefinition of methods: In SDL, a method contained in a supertype can be redefined by a method of an inheriting subtype. This feature is denoted as *method virtuality*. Just as an SDL method, a UML *Operation* can also redefine another *Operation* using the *redefinedOperation* property.

Since redefinition is only applicable to methods in SDL, only an «Operation» whose *isOperator* property is 'false' is permitted to be redefined in an SDL-UML model. In addition, we constrain the *redefinedOperation* property to contain at most one reference, because multiple inheritance is not supported by SDL data type definitions.

The «StructureType» stereotype. To enable a clean separation between simple data types specified using the «DataTypeDefinition» stereotype and data types with structure constructor, we have introduced the «StructureType» stereotype that inherits from the «DataTypeDefinition». The «StructureType» stereotype introduces additional constraints and mapping rules that capture the syntax and semantics of SDL structure data type definitions.

Structure fields: As in the case of SDL-UML 2007, the fields of a structure type are represented by the *ownedAttribute* property of a *Class* with applied «StructureType». According to SDL's semantics, the type of a structure field shall only be a data type definition. Therefore, we constrain the *type* property of an *ownedAttribute* so that it can only refer to a «DataTypeDefinition».

An *ownedAttribute* of a «StructureType» with a multiplicity of [0..1] denotes an optional data field. Since SDL does not support the specification of multi-valued structure fields, we proposed to restrict the multiplicity of *ownedAttribute* in an appropriate manner, but the ITU working group did not agree. Therefore, SDL-UML 2013 defines an in-place transformation that introduces a new data type based on the predefined generic data types of SDL-UML for each multi-valued *ownedAttribute*. Then, the new data types are used instead of the original ones. In our view, this approach increases the complexity of the mapping towards SDL.

Just as in the case of SDL, two implicitly defined methods are created for each *ownedAttribute* before a «StructureType» is mapped to corresponding SDL construct. The methods are required to access and modify values of the structure fields. In addition, another method is introduced for each optional field to test the presence of this field. For example, the «StructureType» PixelType shown in Fig. 3.13a owns four structure fields, where the isActive field is defined as optional.

The «ChoiceType» stereotype. We introduced a new stereotype «ChoiceType» to represent SDL data type definitions that have a choice constructor. A «ChoiceType» comprises a set of different data types, but only one of these types can be used as its actual type. The ColourType shown in Fig. 3.13b is an example of a choice type definition employing the new stereotype. In the following, we discuss the details of the constraints and mapping rules of the «ChoiceType» stereotype, introduced in addition to those inherited from the «DataTypeDefinition» stereotype.

Choice variants: Each *ownedAttribute* of a *Class* with applied «ChoiceType» stereotype defines a certain choice variant, and its *type* property determines the associated data type. In SDL, a data type definition with choice constructor implies the creation of three new data type definitions that are mapped to the abstract syntax. Therefore, we have defined a set of in-place transformations in SDL-UML 2013, which have to be applied to a «ChoiceType» before the mapping towards SDL can be conducted.

The «LiteralType» stereotype. Because the *Enumeration* metaclass inherits from the *DataType* metaclass, it cannot own behaviour specifications as discussed above. Therefore, we have replaced the «Enumeration» stereotype of SDL-UML 2007 by the «LiteralType» stereotype, which is used to represent an SDL data type definition that has a literal constructor. As shown in Fig. 3.13a, the «LiteralType» stereotype inherits from «DataTypeDefinition», and thus, it extends the UML metaclass *Class*. Each item of the *ownedAttribute* property of a *Class* with applied «LiteralType» stereotype maps to an item in the literal signature set of an SDL value data type definition.

Literal names and named numbers: A «Property» that is an *ownedAttribute* of a «LiteralType» represents a particular SDL literal signature, which can be defined in terms of a 'named number', a 'literal name' or a 'name class'. However, the latter variant is only employed to define the literals of SDL's predefined data types, and SDL-UML 2007 only supports the specification of literal names.

To increase the expressiveness of the «LiteralType» stereotype, we have enabled the specification of named numbers, so that a user-defined number can be explicitly assigned to a literal. Therefore, we added the *literalValue* attribute to the «Property» stereotype, which enables the distinction between SDL named numbers and literal names.

When the *literalValue* attribute is present, a «Property» of a «LiteralType» represents an SDL literal signature that is defined as a named number; otherwise, it represents a literal name. The possibility to specify a literal in terms of a name class is not supported because this is only employed in the context of the predefined data types.

For instance, all *ownedAttributes* of the «LiteralType» Greyscale shown in Fig. 3.13b represent literals which are defined as named numbers. In contrast, the black attribute of the «LiteralType» Monochrom corresponds to a literal that is specified in terms of a literal name.

The «Syntype» stereotype. As argued in the section above, SDL syntypes are not supported in SDL-UML 2007. Therefore, we introduced the new stereotype «Syntype» that inherits from the «DataTypeDefinition» stereotype. Thus, the «Syntype» stereotype extends the UML metaclass *Class* as shown in Fig. 3.13a. An SDL syntype has a reference to the data type definition constrained by it and a range condition that defines the constraint. Since a syntype of SDL only consists of the items above, the *ownedAttribute* and *ownedOperation* properties of a *Class* with applied «Syntype» shall be empty.

Syntype constraint: We introduced the *constraint* attribute of the «Syntype» stereotype to represent the constraint of an SDL syntype, which has to be specified as a *RangeCondition*. For instance, the «Syntype» BrightColours shown in Fig. 3.13b has a constraint that restricts the set of literals of the «LiteralType» Greyscale.

Constrained data type: The *parentSortIdentifier* attribute of the «Syntype» stereotype shall reference the data type definition to be constrained. Our BrightColour example «Syntype» shown in Fig. 3.13b refers to the «LiteralType» Greyscale.

The «Property» stereotype. The stereotype «Property» extends the UML metaclass *Property* and is defined as 'required'. As discussed above, the «Property» stereotype is used in the context of various data type related stereotype. In addition, it is also applied to the attributes of an «ActiveClass». For to this reason, various constraints and mapping rules are defined for the «Property» stereotype. Since all rules of this stereotype related to data types are already discussed in connection with the above stereotypes, no further details need to be given here.

Predefined data types. The simple predefined data types of SDL-UML 2007 are defined as *PrimitiveType* instances, regardless in which way they are represented in SDL. As argued in the previous section, this approach has some drawbacks concerning the semantics and the definition of values. Therefore, we removed the «PrimitiveType», and instead we specify the simple primitive types as *Class* instances to which one of the above stereotypes is applied. The advantage of this approach is that the primitive data types of SDL-UML 2013 can be used in the same way as user-defined data types.

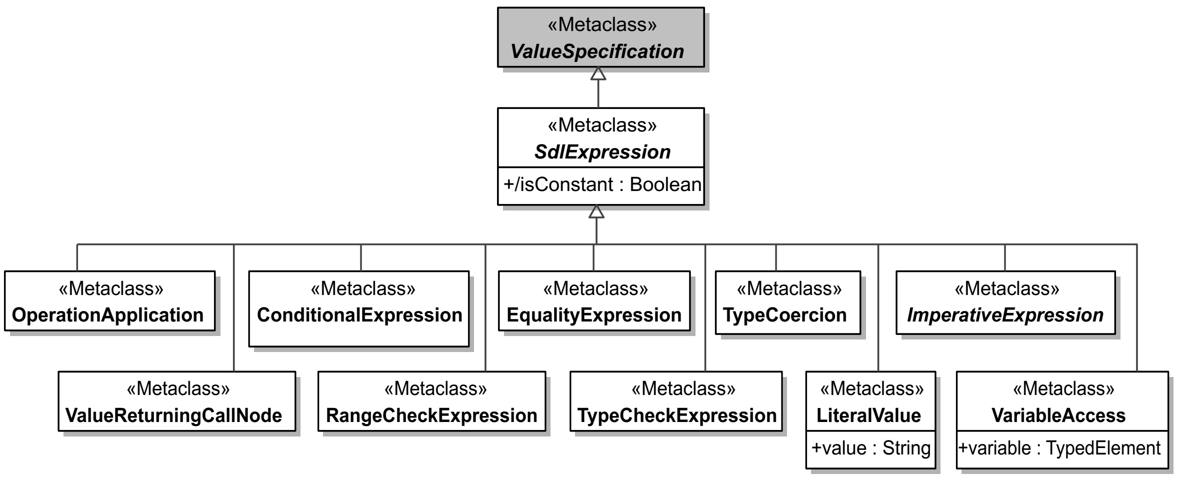


Figure 3.14: Excerpt of the abstract syntax of SDL-UML 2013 [69] for representing expressions

The four generic predefined data types are represented as *Class* instances with applied «DataTypeDefinition» stereotype. As in the case of SDL-2007, a set of formal parameters is defined for each of the generic types. A data type definition that inherits from one of the generic types has to replace the formal parameters with actual parameters.

Conclusions. At the beginning of this section, we pointed out that the final concept of representing data type definitions in SDL-UML 2013 [69] differs from our initial proposal in [87]. As discussed, this is mainly caused by the behaviour specifications for *Operations*. However, the essential point is that our final concept for representing data type definitions remedies all issues of SDL-UML 2007 identified in Sec. 3.3.1.

In our view, one drawback concerning the «Property» stereotype still remains, because this stereotype is automatically applied to all *Properties* without regarding the containing *Classifier*. Thus, the mapping rules and constraints defined for the «Property» stereotype have to take into account the containing *Classifier* of a *Property*, and consequently, their complexity increases. For this reason, we propose to introduce additional stereotypes that extend the metaclass *Property*, where each stereotype is used for a specific purpose, e.g., to represent a structure field or a selection variant. However, the responsible ITU working group did not agree.

3.4.2 Representation of SDL Expressions

To solve the issues of SDL-UML 2007 regarding the representation of SDL expressions as discussed in Sec. 3.3.2, we initially proposed a new abstract syntax [85, 87] for the affected parts of SDL-UML. However, our initial proposal did not completely remedy all identified issues, so some modifications were required. Below, we first briefly present our initial proposal, and then we discuss which issues could not be solved in an appropriate way. Finally, we present the abstract syntax introduced by us for expressions in SDL-UML 2013.

The initial proposal. The first objective of our initial proposal for representing SDL expressions was to reduce the complexity concerning the specification of data type values, and as a second objective, the shortcomings of the «InstanceValue» stereotype should be remedied.

To achieve the first objective, we proposed to introduce the «LiteralValue» stereotype to specify values for each kind of predefined and user-defined data type. Thus, when compared to SDL-UML 2007, the four stereotypes (e.g., «LiteralInteger») that extend the metaclasses inheriting from *LiteralSpecification* and the «InstanceValue» stereotype are no longer required.

Furthermore, we proposed to introduce the «VariableAccess» stereotype that extends the metaclass *OpaqueExpression* for capturing variable access expressions. This stereotype should replace the «InstanceValue» stereotype of SDL-UML 2007, which we considered to be inappropriate, as pointed out in the section above.

Apart from literal and variable access expressions, all remaining SDL expression types should be represented by the «Expression» stereotype as in SDL-UML 2007. Therefore, we did not address the issues concerning this stereotype analysed in the previous section. Thus, no separation between concrete and abstract syntax could be achieved. Exactly this fact led to problems in the specification of OCL constraints, which should capture the static semantics of SDL expressions. Therefore, we revised our initial proposal as discussed below.

The new approach to represent expressions. In contrast to our initial proposal, our final approach for representing expressions in SDL-UML 2013 is not based on stereotypes. Instead, we use a total of 25 metaclasses that directly or indirectly inherit from the UML metaclass *ValueSpecification*. Each of these metaclasses corresponds to exactly one expression type of SDL's abstract syntax (see Fig. 3.9a). As shown in Fig. 3.14, the *SDLExpression* metaclass is the supertype of all other metaclasses that represent SDL expressions.

Our decision to define the abstract syntax of expressions for SDL-UML 2013 as an extension to the UML metamodel is motivated by the fact that this approach is also employed for other languages that supplement the UML, such as OCL or the *Value Definition Language (VDL)* of the MARTE profile [117]. An advantage of this approach is that metaclass instances for nested SDL expressions can be created in a single pass, while a stereotype-based approach requires two passes because of the separate instances that have to be created for metaclasses and stereotypes (see Sec. 2.5.2). In the first pass, metaclass instances are created so that the stereotypes can be applied to them in the second pass. In addition, appropriate values have to be assigned to the stereotype attributes as a part of the second pass.

In contrast to stereotypes that are a light-weight extension to the UML, our metaclass-based approach is a middle-weight extension to the UML metamodel.

As a disadvantage, it can be argued that additional effort is required to customize a modelling tool to support SDL-UML 2013. Since SDL expressions can only be specified textually, a parser and editor for this notation has to be developed anyway, which requires higher effort than implementing the additional metaclasses. Because the *ValueSpecification* metaclass does not have a graphical or textual notation, existing diagram editors of a modelling tool must not be modified.

New syntax and semantics. As pointed out above, we represent each expression type as a dedicated metaclass in SDL-UML 2013, and the syntax and semantics of each metaclass is specified as part of the Rec. Z.109 released in 2013 [69]. As an example, Fig. 3.15 shows the syntax and semantics of the metaclasses *SdlExpression* and *LiteralValue*.

Since the *SdlExpression* metaclasses is the supertype for all other expression types, its specification shown in Fig. 3.15 embraces fewer rules and constraints than the corresponding «Expression» stereotype of SDL-UML 2007 (see Fig. 3.10). Instead, the metaclasses inheriting from the *SdlExpression* metaclass capture the static semantics defined for the corresponding SDL expression type, e.g., see the specification of the *LiteralValue* metaclass shown in Fig. 3.15. Furthermore, the specification of each metaclass defines a set of specific mapping rules to a corresponding counterpart of SDL's abstract syntax.

Conclusions. Because of our alignment of the metaclasses to corresponding constructs of SDL's abstract syntax, their specifications do not include any production rules for a textual notation, as is the case for SDL-UML 2007 (see Fig. 3.10). Thus, we could meet our objective to separate the abstract syntax of expressions in SDL-UML from the concrete syntax because a missing separation between both syntax kinds was identified as a drawback of SDL-UML 2007.

Just as the «LiteralValue» stereotype of our initial proposal, the *LiteralValue* metaclass is used to represent values for each kind of data type. Thus, our new approach also addresses the problems of SDL-UML 2007 for representing data type values. In addition, we introduced the *VariableAccess* metaclass to represent SDL variable access expressions, so this aspect is separated from the specification of values.

3.4.3 Improving and Formalizing the Static Semantics

After we presented our proposals for improving the data type concepts of SDL-UML and the representation of SDL expressions, we discuss below our proposal for formalizing the static semantics of SDL-UML using OCL. Consequently, we defined OCL *Constraints* in the context of the stereotypes or additional metaclasses of SDL-UML. We identified the following sources for the specification of constraints:

10.17 SdlExpression

The metaclass *SdlExpression* is a specialization of the UML metaclass *ValueSpecification*. This metaclass is abstract, and the metamodel diagram for the metaclass is defined in Figure 10-1. Subtypes of the metaclass are the metaclasses *RangeCheckExpression* (see clause 10.15), *ConditionalExpression* (see clause 10.5), ...

Depending on the *isConstant* property, the *SdlExpression* metaclass represents a Constant-expression or an Active-expression alternative of an Expression. For both alternatives of an Expression in the SDL-2010 abstract syntax further alternatives exist. The semantics and mapping rules for the different alternatives of an Expression depends on the above listed subtypes of the *SdlExpression* metaclass.

10.17.1 Attributes

1. isConstant : Boolean — True, if the expression is a constant expression, otherwise it is an active expression.

10.17.2 Constraints

1. The *type* property of an *SdlExpression* shall not be empty.

10.17.3 Semantics

The subtypes of *SdlExpression* specify the semantics and mapping rules.

10.9 LiteralValue

The metaclass *LiteralValue* is a subtype of the metaclass *SdlExpression*. The metamodel diagram for the metaclass is defined in Figure 10-1. The metaclass *LiteralValue* represents a Literal in the SDL-2010 abstract syntax.

10.9.1 Attributes

1. value: String — This represents the concrete value for a data type.

10.9.2 Constraints

1. The *type* property shall reference a «DataTypeDefinition» *Class*.
2. The *isConstant* property shall be true.

10.9.3 Semantics

1. The *LiteralValue* maps to a Literal (a Constant-expression) in the SDL-2010 abstract syntax.
2. The *qualifiedName* of the *type* property of a *LiteralValue* maps to the Qualifier part of the Literal-identifier.
3. The *value* property of a *LiteralValue* maps to the Name part of the Literal-identifier.

Figure 3.15: Syntax and semantics of the metaclasses *SdlExpression* and *LiteralValue* as specified in Rec. Z.109 [69] published in 2013

1. Syntactic and semantic gaps between SDL and UML;
2. Static semantics of SDL;
3. Conditions implied by the syntax of SDL.

In the following, we employ the stereotype «SendSignalAction» as an example to discuss the details of our approach to specify OCL *Constraints* that capture the three aspects above. Analogously, we created or revised the constraints of all other stereotypes defined for SDL-UML 2013.

Syntactic and semantic gaps between SDL and UML. As we emphasized in the motivation for our thesis at hand in Sec. 1.1, syntactic and semantic gaps may exist between modelling languages. This is the case between SDL and UML because UML provides more language constructs than SDL. In the following, we explain this fact using the example of the «SendSignalAction» stereotype and the UML metaclass *SendSignalAction* that is extended by this stereotype. We then introduce our approach to close this gap.

To model input and output parameters, all UML *Actions* provide a set of *InputPins* and a set of *OutputPins*. However, in the case of SDL-UML, only *ValuePins* are used to define values for input parameters, whereas output values are assigned to class attributes or local variables. Thus, the use of *ObjectFlows* between *Actions* is not required in SDL-UML. In contrast, no language concept comparable to *ValuePins* is available in SDL. Instead, values for SDL statements have to be specified using expression lists. Consequently, a syntactic gap regarding the UML language concept of *ValuePin* exists.

One option to close the gap is to define a stereotype that extends the UML metaclass *ValuePin*. Because no SDL counterpart exists, we do not have to define mapping rules for this stereotype. However, it is required to define an OCL constraint that restricts the *value* property of a *ValuePin* in such a way that only *SdlExpressions* can be assigned to this property. In addition, we have to ensure that only *ValuePins* can be used as *InputPins* for *Actions*. Consequently, further OCL *Constraints* capturing these conditions need to be defined for all stereotypes that extend the various types of UML *Actions*.

As a second option to close the syntactic gap, we can define all the above-mentioned OCL constraints directly in the context of stereotypes that extend certain UML *Actions*, e.g., the *SendSignalAction*. The advantage of this solution is that no additional stereotype needs to be introduced for *ValuePin*. However, we have to specify the natural language descriptions of the constraints in such a way so that it becomes evident that they do not concern the UML *Action* but one of the *ValuePins*.

To avoid additional overhead by introducing a new stereotype, we have chosen the second option. For example, to ensure well-formedness of the *target* property of the extended UML *SendSignalAction* metaclass, we introduced the OCL *Constraint*

Constraint 1: The *target* property shall reference a *ValuePin*, and its *value* property shall be an *SdlExpression*.

```
base_SendSignalAction.target.oclIsTypeOf(ValuePin)
and base_SendSignalAction.target.oclAsTypeOf(ValuePin).value.oclIsKindOf(SdlExpression)
```

Constraint 2: The *value* of the *target* property shall have the *type* Predefined::Pid.

```
base_SendSignalAction.target.oclIsTypeOf(ValuePin) implies
  base_SendSignalAction.target.oclAsType(ValuePin).value.type.qualifiedName = `Predefined::Pid'
```

Constraint 3: The *onPort* property shall reference a *Port* of the containing «ActiveClass» of the «SendSignalAction».

```
let container : Class = base_SendSignalAction.allNamespaces()
  −>select(n| n.isStereotypedBy(`SDLUML::ActiveClass'))−>first().oclAsType(Class)
in base_SendSignalAction.onPort.owner = container
```

Constraint 4: If present, the *type* of the *signalPriority* property shall be Predefined::Natural.

```
self.signalPriority <> null implies
  self.signalPriority.type.qualifiedName = `Predefined::Natural'
```

Constraint 5: If present, the *type* of the *activationDelay* property shall be Predefined::Duration.

```
self.activationDelay <> null implies
  self.activationDelay.type.qualifiedName = `Predefined::Duration'
```

Constraint 6: The *argument* list shall only contain *ValuePin* items with a *value* property that is of kind *SdlExpression*.

```
base_SendSignalAction.argument−>forAll(v|
  v.oclIsTypeOf(ValuePin) and v.oclAsType(ValuePin).value.oclIsKindOf(SdlExpression) )
```

Constraint 7: The number of items in the *argument* list shall be equal to the number of *ownedAttributes* of the associated *signal*.

```
base_SendSignalAction.argument−>size() = base_SendSignalAction.signal.ownedAttribute−>size()
```

Constraint 8: Each *ValuePin* in the *argument* list shall have a *value* (except of *Undefined* values) whose *type* is type-compatible by order with its corresponding *ownedAttribute* of the associated *signal*.

```
let sig : uml::Signal = base_SendSignalAction.signal,
  args : OrderedSet(ValuePin) = base_SendSignalAction.argument.oclAsType(ValuePin)
in
args−>size() = sig.ownedAttribute−>size() implies
  Sequence{1..args.value−>size()}−>forAll(i|
    args.value−>at(i).type.conformsTo(sig.ownedAttribute−>at(i).type)
    or args.value−>at(i).oclIsTypeOf(Undefined)
  )
```

Figure 3.16: OCL *Constraints* of the «SendSignalAction» stereotype of SDL-UML 2013

1 shown in Fig. 3.16. This constraint captures that the *target* property must consist of a *ValuePin* with an *SdlExpression* as *value*. Thus, we define how to close the syntactic gap implied by *ValuePin*.

Static semantics of SDL. The well-formedness rules of SDL's static semantics served as source for us to introduce corresponding OCL *Constraints* for stereotypes of SDL-UML 2013. As we argued in the section concerning the foundations of SDL (see Sec. 3.2.2), the formalization of the static semantics is based on a first-order predicate logic. In the following, we explain our approach for specifying constraints for SDL-UML 2013 based on the well-formedness rule defined for the SDL output node, shown at the top of Fig. 3.17. As is apparent from the textual description of this well-formedness rule, the length of the expression list of an output node on the one hand and the type-compatibility of the expressions contained in this list on the other hand are constrained according to defined criteria. In order to better distinguish between these two conditions, we introduced two separate constraints for the «SendSignalAction» stereotype. Constraint 7 shown in Fig. 3.16 is used to verify the length of the expression list, while Constraint 8 ensures the type-compatibility of the list items.

Textual description: When translating the well-formedness rules of the static semantics of SDL into corresponding constraints of the stereotypes, a distinction must be made between the descriptions of the constraints in natural language and the corresponding OCL *Constraints*. As far as possible, we replaced the names of syntactic constructs of SDL with its corresponding counterparts of the UML or the stereotypes of SDL-UML. If a translation was not applicable due to syntactic gaps between SDL and UML, we appropriately extended the description concerned.

A good example of such an extension is the textual description of Constraint 8 in Fig. 3.16. As we justified above, the expression list of an SDL output node consists of SDL expressions, whereas the *argument* list in SDL-UML contains *ValuePin* items, which have a *value* property that is an *SdlExpression*. Exactly this relation was taken into account by us for the translation the textual description.

OCL Constraints: Before we introduce our approach to translate well-formedness rules of SDL to OCL Constraints, we briefly describe the function of the well-formedness rule shown in Fig. 3.17, because this is used as a running example below. Line 1 of the rule specifies that at least one signal definition for each ouput node must exist, and the conditions defined in the two subsequent lines must apply to this output node. The $\mathrm{getEntityDefinition1}()$ function invoked in Line 2 determines the $\mathrm{Signal\text{-}definition}$ referenced by the $\mathrm{Signal\text{-}identifier}$ of an output node. Then, the $\mathrm{isActualAndFormalParameterMatched1}()$ predicate called in Line 3 verifies whether the list of parameters of the output node and the list of formal parameters of the signal definition match.

The length of the list of optional Expressions *must be the same as the number of* Sort-reference-identifiers *in the* Signal-definition *denoted by the* Signal-identifier. *Each* Expression *must be sort compatible to the corresponding (by position)* Sort-identifier-reference *in the* Signal-definition.

```
∀ n ∈ Output−node : ∃ d ∈ Signal−definition :
  (d=getEntityDefinition1(n.s−Signal−identifier, signal))
  ∧ isActualAndFormalParameterMatched1(n.s−Expression−seq, d.foramlParameterSortList1)
```

Determine if the sort list of Expressions *corresponds by position to the* Sort-reference-identifier *list.*

```
isActualAndFormalParameterMatched1(expl: Expression*, fpsl: Sort−reference−identifier) : BOOLEAN = def
  (expl.length = fpsl.length)
  ∧ ( ∀i ∈ 1..expl.length:expl[i] = undefined ∨ isCompatibleTo1(expr[i].staticSort1, fpsl[i]) )
```

The function isCompatibleTo1 *determines if a* Sort-reference-identifier *is compatible to the other.*

```
isCompatibleTo1(id1: Sort−reference−identifier, id2: Sort−reference−identifier) : BOOLEAN = def
  let d1 = getEntityDefinition1(id1, sort) in
    let d2 = getEntityDefinition1(id2, sort) in
      d1 = d2 ∨ isSuperType1(d2, d1)
    endlet
  endlet
```

The function staticSort1 *returns the static sort of* e.

```
staticSort1(e: Expression) : Sort−reference−identifier = def
case e of
  | Literal => getEntityDefinition1(e.s−Literal−identifier, literal).s−Result
  | Variable−identifier => getEntityDefinition1(e, variable).s−Sort−reference−identifier
  | Equality−expression ∪ Range−check−expression ∪ Timer−active−expression =>
        mk−Identifier(mk−Qualifier("Predefined"), "Boolean")
  | Now−expression => mk−Identifier(mk−Qualifier("Predefined"), "Time")
  | Pid−expression => mk−Identifier(mk−Qualifier("Predefined"), "Pid")
  | Any−expression => e.s−Sort−reference−identifier
  | Operation−application =>
        getEntityDefinition1(e.s−Operation−identifier, operation).s−Result
  | Value−returning−call−node =>
        getEntityDefinition1(e.s−Procedure−identifier, procedure).s−Result
  | Conditional−expression => staticSort1(e.s−Consequence−expression)
otherwise undefined
endcase
```

Figure 3.17: Excerpt of the static semantics of the SDL output node as defined in Annex F2 of Rec. Z.100 [72]

As we argued for the textual description, we split the well-formedness rule into the two Constraints 7 and 8 shown in Fig. 3.16. Since the well-formedness rule applies to SDL output nodes, we specified Constrains 7 and 8 in the context of the «SendSignalAction» stereotype. Entity definitions in SDL are referenced using textual identifiers, whereas attributes are employed to reference elements of a UML model. Therefore, in the case of SDL-UML, one is not required to resolve textual identifiers by employing an OCL query similar to the getEntityDefinition1() function of SDL. Consequently, we did not translate this function to a corresponding OCL query. Instead, we used the *signal* property of the *SendSignalAction* metaclass. In the same way, we translated all other SDL well-formedness rules that include identifiers.

The length of the expression list and the type-compatibility are not directly verified in the body of the well-formedness rule shown in Fig. 3.17. Instead, both verifications are part of the isActualAndFormalParameterMatched1() predicate. If we had not split the well-formedness rule into two OCL constraints, we would have translated that predicate into an OCL query. The length check defined in Line 2 of the predicate corresponds exactly to the OCL *Constraint* 7 shown in Fig. 3.16, and the type-compatibility specified in Line 3 of the predicate is implemented by OCL *Constraint* 8.

As shown in Line 3 of the isActualAndFormalParameterMatched1() predicate, if an item of the expression list is not defined as 'undefined', its type-compatibility is checked by employing the isCompatibleTo1() predicate. Because the conformsTo() query of the UML *Classifier* metaclass corresponds exactly to the isCompatibleTo1() predicate of SDL, we considered that a translation into OCL is not required. Therefore, we invoke the conformsTo() query to check the type compatibility in Line 6 of OCL *Constraint* 8. In this *Constraint*, we determine the types to be compared via the *type* of a *Property* or *SdlExpression*. By contrast in the case of SDL, the staticSort1() function called in Line 3 of the isActualAndFormalParameterMatched1() function is used to determine a data type identifier for a given expression.

OCL Queries: After we outlined how to translate well-formedness rules of SDL's static semantics into OCL *Constraints*, we explain the translation of functions and predicates below. We illustrate our translation approach on the example of the staticSort1() function shown in Fig. 3.17. The translated counterpart is the OCL query staticSort() contained in Fig. 3.18. Since the staticSort1() function is associated with SDL expressions, we defined the OCL query in the context of the *SdlExpression* metaclass, and instead of an identifier, this query returns a *Type* instance. We mainly use this OCL query to compute the value of the redefined *type* property of the *SdlExpression* at runtime.

The staticSort1() function determines the sort reference identifier based on the kind of an SDL expression. In the same way, we distinguish between the various subtypes of *SdlExpression* in the OCL query staticSort(). When we compare the staticSort1() function and the OCL query, it becomes obvious that the OCL query is listing more expression types than its SDL counterpart. This is caused by an incomplete specification of the staticSort1() function. We have already reported this issue to the responsible ITU working group.

```
 1 context SdlExpression::staticSort() : Type
 2 body:
 3   if self.oclIsTypeOf(LiteralValue) then
 4     self.type
 5   else if self.oclIsTypeOf(VariableAccess) then
 6     self.oclAsType(VariableAccess).variable.type
 7   else if self.oclIsTypeOf(TypeCheckExpression)
 8     or self.oclIsTypeOf(TimerActiveExpression)
 9     or self.oclIsTypeOf(RangeCheckExpression)
10     or self.oclIsTypeOf(EqualityExpression) then
11       self.getPredefinedType(`Boolean')
12   else if self.oclIsTypeOf(NowExpression) then
13     self.getPredefinedType(`Time')
14   else if self.oclIsTypeOf(PidExpression) then
15     self.getPredefinedType(`Pid')
16   else if self.oclIsTypeOf(AnyExpression) then
17     self.oclAsType(AnyExpression).sortReference
18   else if self.oclIsTypeOf(OperationApplication) then
19     self.oclAsType(OperationApplication).operation.type
20   else if self.oclIsTypeOf(ValueReturningCallNode) then
21     self.oclAsType(ValueReturningCallNode).procedure.ownedParameter
22     −>any(direction = ParameterDirectionKind::return).type
23   else if self.oclIsTypeOf(ConditionalExpression) then
24     self.oclAsType(ConditionalExpression).consequenceExpression.type
25   else if self.oclIsTypeOf(ActiveAgentsExpression) then
26     self.getPredefinedType(`Natural')
27   else if self.oclIsTypeOf(StateExpression) then
28     self.getPredefinedType(`Charstring')
29   else if self.oclIsTypeOf(TimerRemainingDuration) then
30     self.getPredefinedType(`Duration')
31   else if self.oclIsTypeOf(TypeCoercion) then
32     self.oclAsType(TypeCoercion).sortReference
33   else
34     null
35   endif endif endif endif endif endif endif endif endif endif endif endif endif
```

Figure 3.18: The OCL-defined query staticSort() of the metaclass *SdlExpression*

Since some SDL expression kinds have a fixed result type, the staticSort1() function returns an identifier that refers to an expression-specific predefined data type. For instance, an identifier pointing to the predefined Boolean type is returned for an equality expression. To return such an identifier, the staticSort1() function creates a new identifier syntax node using the mk-Identifier() function, e.g., see Line 6 of the staticSort1() function. In the case of the staticSort() query, we instead use the getPredefinedType() query to determine an appropriate predefined data type, e.g., see Line 11 in Fig. 3.18.

If the result of an SDL expression has to be determined based on a particular symbol or sub-expression, the staticSort1() function invokes the getEntityDefinition1() function discussed above. By contrast, the OCL staticSort() query determines the *Type* to be returned based on a *type* property. For instance, such a type-resolution is applied for the *LiteralValue* expression (see Line 4 in Fig. 3.18).

Conditions implied by the syntax of SDL. The formal specification of the concrete syntax of SDL is based on production rules defined in a particular BNF variant. To capture that an SDL expression shall have a specific static data type, the <expression> symbol is preceded by the name of the data type, which is additionally underlined. For example, this notation form is used for the two following symbols of the SDL output node:

$\langle\text{activation delay}\rangle :: \langle\underline{\text{Duration}}\langle\text{expression}\rangle$
$\langle\text{signal priority}\rangle :: \langle\underline{\text{Natural}}\langle\text{expression}\rangle$

In the above example, the first line defines that an expression that defines the activation delay of an SDL output node must always return a Duration value as result. The signal priority in the second line of the example must always return a Natural value.

Because this particular form of syntactic constraint is not captured by well-formedness rules of SDL's static semantics, we introduced appropriate OCL *Constraints* for the stereotypes of SDL-UML 2013. For instance, we created the OCL *Constraints* 4 and 5 for the «SendSignalAction» stereotypes (see Fig. 3.16) based on the above production rules. Both constraints are constructed according to the same pattern, and they ensure that the *type* property of the affected stereotype attribute refers to the required predefined data type.

Conclusions. In this section, we presented our approach to formalizing the static semantics of SDL-UML by means of OCL constraints and queries on the example of the «SendSignalAction» stereotype. In the same way, we applied this approach to all other stereotypes and metaclasses defined for SDL-UML 2013. Thus, all language constructs of SDL-UML 2013 have a formally defined static semantics. Similar to the OCL constraints, we also revised their description in natural language.

All of the above-mentioned artefacts were submitted to the ITU working group responsible for SDL, so they could become part of SDL-UML 2013. Because the members of the working group had different point of views how the OCL constraints and queries should be published, only the descriptions in natural language became part of the Rec. Z.109 of 2013 [69]. Regardless of the standardization process of the ITU, we provide an implementation of SDL-UML 2013 as part of our SU-MoVal framework [156], and this implementation includes all the OCL artifacts created by us.

A general challenge in translating SDL's static semantics into OCL constraints and queries was to identify matching stereotype and metaclass attributes for symbols of the production rules of SDL's abstract syntax. Because of same or similar names for stereotype attributes and corresponding symbols of production rules, the translation was straightforward for almost all stereotype attributes. By contrast, such a match did not exist for metaclass attributes, so the creation of OCL constraints referring to these attributes was more difficult. In addition, the syntactic gaps between UML and SDL discussed above made it difficult to identify appropriate UML counterparts. This issue was a potential source of errors when manually translating OCL constraints and requests. We were only able to identify and remedy these errors by employing an iterative specification in conjunction with appropriate validation and verification activities. The details of our approach to verify OCL constraints and queries is explained in the following section.

3.5 Validation and Verification of the UML Profile

As mentioned, the iterative verification of OCL constraints and queries of the SDL-UML 2013 stereotypes is an important activity to ensure proper translation of the static semantics of SDL. Because of the close relationship between stereotypes and the OCL artefacts they contain, we found a combined validation and verification of these two artefact types obvious. This statement also applies to the additional metaclasses of SDL-UML 2013.

In Sec. 2.6.2, we pointed out that automated verification approaches for CMOF-based metamodels, which typically include OCL constructs, are of limited use. Either the approaches we studied did not support all language concepts of CMOF or only a subset of OCL. Since UML profiles are modelled using particular class diagrams, which in a broader sense correspond to those employed for the specification of CMOF-based metamodels (see Sec. 2.5.2), the restrictions of automated verification approaches also apply to UML profiles. Therefore, we decided to verify the revised UML profile for SDL using manually specified test cases, which are created according to the methodology described in Sec. 2.6.2. The test cases are executed using our SU-MoVal framework.

Before we explain our verification of the OCL constructs of SDL-UML 2013, we first discuss the validation of structural aspects of this UML Profile.

3.5.1 Validation of Structural Aspects

As discussed in the literature, e.g., in [99, 133], the main goal of validating a class diagram is to verify structural aspects and OCL constraints contained in this diagram. To validate the structural aspects of a class diagram means to verify whether classes, class attributes and association ends can be instantiated as expected. When we apply this validation method to UML profiles, we do not instantiate classes but stereotypes. In Sec. 2.6.2, we pointed out that structural aspects shall be verified employing positive and negative test cases, which are specified by means of appropriate model fragments and are referred to as ‘test models’.

In the context of UML profiles, the test objective of positive tests is to verify the successful instantiation of stereotypes, including their attributes or association ends. By contrast, the objective of negative tests is to verify that no instantiation is feasible for model fragments that do not conform to the abstract syntax of the UML profile. For example, a negative test might verify that an Integer value cannot be assigned to an attribute of Boolean type. Furthermore, a negative test could verify that an abstract stereotype cannot be applied to any model element.

As our SU-MoVal framework is implemented based on Eclipse EMF [142], we did not have to conduct negative tests for structural aspects. This was possible due to the mechanisms of Eclipse EMF, which prevent the previously discussed error scenarios. Therefore, we performed only positive tests to validate the stereotypes of SDL-UML 2013. Since the additional metaclasses of SDL-UML 2013 extend the EMF-based implementation of UML, we applied the same test procedure to them as in the case of stereotypes.

In order to reduce the amount of positive test cases, we applied the *Category Partition Testing* method combined with the test coverage criteria discussed in Sec. 2.6.2. Consequently, we designed our test cases in such a way that each attribute and association end of a stereotype was assigned a value at least once, and each stereotype was instantiated at least once.

3.5.2 Verification of the Static Semantics

In addition to the structural aspects discussed above, the vast majority of the tests we performed consisted in the verification of the OCL constraints and queries specified for the stereotypes and additional metaclasses of SDL-UML 2013. We applied the same methodology for preparing these test cases as we did for the validation of the structural aspects of SDL-UML 2013. To create test cases for the verification of OCL constructs, required equivalence classes are not only determined based on attributes and association ends, but also OCL expressions have to be regarded.

The *Constraints* of a metaclass or stereotype represent invariants that must be satisfied at any time, i.e., the evaluation of an OCL expression that defines such a *Constraint* must always return the Boolean value 'true'. We consider positive test cases as those that expect the Boolean value 'true' as result of an OCL expression. By contrast, test cases that expect the Boolean value 'false' as result are regarded to be negative test cases. Accordingly, we specified appropriate test models, so that the evaluation of a *Constraint* either returned the Boolean value 'true' or 'false' as result. To create these test models, we took into account the model elements and their attributes referenced in the OCL *Constraint* to be tested.

Below we discuss our approach for creating test models on the example of Constraints 6 – 8 of the «SendSignalAction» stereotype, shown in Fig. 3.16. When we consider the OCL specification of these constraints, we find that the *argument*

Table 3.1: Test cases for verifying OCL *Constraints* of the «SendSignalAction» stereotype

			Expected evaluation results:									
			true				***false***					
Element	**Property**	**Test Case** / **values**	**TC 1**	**TC 2**	**TC 3**	**TC 4**	**TC 5**	**TC 6**	**TC 7**	**TC 8**	**TC 9**	**TC 10**
Signal	***ownedAttribute***	A *Signal* with no *ownedAttributes*.	**X**							**X**		
		A *Signal* with one *ownedAttributes*.		**X**			**X**	**X**	**X**		**X**	**X**
		A *Siganl* with two *onedAttributes*.			**X**	**X**						
SendSignalAction	**valid *arguments***	Empty *argument* list	**X**									
		One *ValuePin* with an *SdlExpression* as *value*; The *SdlExpression* has a *type* that is compatible to its corresponding *ownedAttribute* of the *Signal*.		**X**								
		Two *ValuePins* with an *SdlExpression* as *value*; One *SdlExpression* has a *type* that is is equal to the corresponding *ownedAttribute* of the *Signal*, and the other one has a *type* that is compatible to its corresponding *ownedAttribute*.			**X**					**X**	**X**	
		One *ValuePin* with an *SdlExpression* as *value* that is compatible to its corresponding *owned-Attribute* of the *Signal*; One *ValuePin* with an *Undefined* as *value*.				**X**						
	invalid *arguments*	One *InputPin*					**X**					
		One *ValuePin* with an empty *value* property						**X**				
		One *ValuePin* with a *value* that is not an *SdlExpression*.							**X**			
		One *ValuePin* with an *SdlExpression* that is not compatible to its corresponding *ownedAttribute.*										**X**
		Tested OCL *Constraints*	6, 7, 8	6, 7, 8	6, 7, 8	6, 7, 8	6, 8	6, 8	6, 8	7	7	8

property of the «SendSignalAction» stereotype is constrained based on the *onwedAttribut* property of the referenced *Signal*. Therefore, we determined appropriate equivalence classes for these two properties. To avoid the creation of redundant test models, we determined the equivalence classes considering all three constraints. We used a tabular approach, where the identified equivalence classes are listed as table lines, and the table columns represent the determined test cases.

Table 3.1 shows the equivalence classes and the resulting test cases identified by us for Constraints 6 – 8 of the «SendSignalAction» stereotype. The first three table lines represent valid values for the equivalence class 'ownedAttribute', which corresponds to the identically named property of the *Signal* referenced by a «SendSignalAction». Because the items of the *ownedAttribute* of the *Signal* serve only as reference values to verify the type-compatibility or the number of items contained in the *argument* property, we did not specify an equivalence class that embraces invalid values.

In contrast to the *ownedAttribute* property, we distinguish between valid and invalid values for the *argument* property of the «SendSignalAction» stereotype. Thus, we defined the two equivalence classes 'valid argument' and 'invalid argument' for this property. The values we identified for the first equivalence class are listed in rows 4–7 of the table, whereas values of the 'invalid argument' class are specified in rows 8–11.

When considering the columns of Table 3.1, we see that 10 test cases (TC 1–10) are specified based on the defined values of the three equivalence classes discussed above. Apart from the test case identifier, the *Constraints* to be verified by a test case are listed in round brackets in the header of a table column. Test cases TC 1–4 expect the Boolean value 'true' as result of the Constraints 6–8; thus, they correspond to positive test cases. In contrast, test cases TC 5–10 represent negative test cases, because they expect the Boolean value 'false' as result.

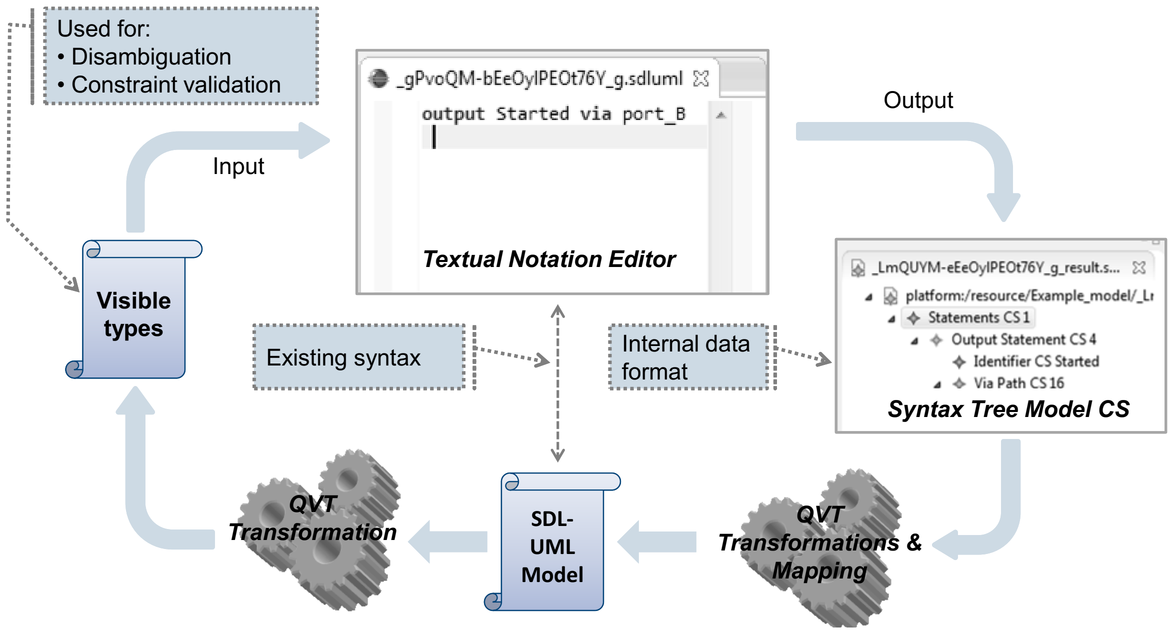
In the matrix resulting from the rows and columns of Table 3.1, the values of an equivalence class selected for a test case are marked with an X. The test model intended for a particular test case must contain exactly the values (i.e., model elements) identified in the table column associated with this test case. Instead of creating a separate test model for each test case, we combined multiple test models into a single model.

As is apparent from Table 3.1, the four values of the 'valid argument' equivalence class are mainly required to verify the different type-compatibility variants of Constraint 8. In addition, they also serve to positively verify Constraints 6 and 7. In contrast, we use the four values of the 'invalid argument' equivalence class to verify that Constraints 6 and 8 return the Boolean value 'false' for invalid values.

3.5.3 Conclusions

In this section, we explained how we applied *Category Partition Testing* not only to validate the structural aspects of the UML profile for SDL, but also to verify the stereotype-specific OCL constraints. The latter aspect has been exemplified by three *Constraints* of the «SendSignalAction» stereotype. Although we have not addressed the verification of OCL queries, they can be verified in a similar way to OCL constraints. However, in the case of OCL queries, equivalence classes are not only determined based on model elements and attributes, but also query parameters have to be considered.

Since the method we used for validation and verification was not specifically developed for SDL-UML 2013, it can also be applied to other UML profiles. In



```
output Started via port_B
```

Figure 3.19: SU-MoVal Editor for the textual notation

particular, this should be an option if, as in the case of SDL-UML 2013, the use of automated verification methods is not or only partially applicable.

If a corresponding metamodel exists for a UML profile to be validated, and M2M transformations are to be developed to achieve model interoperability, a reuse of already created test models is a good option, because they can serve as input for the test of the transformations. Thus, a redundant creation of test models can be avoided.

3.6 Details of the SU-MoVal Framework

A brief introduction to our SU-MoVal framework, which implements SDL-UML 2013 prototypically, is given in Sec. 3.1.3. In the following, we discuss the details of implementing the textual notation editor and the model transformations used by SU-MoVal. However, we do not treat the OCL-based verification of the static semantics of SDL-UML models because we use standard functionalities of Eclipse for this purpose. The details of the OCL constraints and queries that we have specified can be found in Sec. 3.4.3.

3.6.1 Editor for the Textual Notation

The textual notation editor of SU-MoVal supports a subset of the concrete syntax of SDL, so that the specification of SDL statements and expressions is enabled. We generated this editor using the *Spoofax Language Workbench* [157]. In addition to familiar features provided by other editor construction kits, the Spoofax Language Workbench is based on a *Scanner-less Generally Left-Right (SGLR)* parser. This type of parser enables an efficient handling of SDL's syntax ambiguities (see Table 3.2) by creating a parse forest rather than just a syntax tree. If an ambiguity exists, a sub-tree for each syntactic alternative is produced, so that the correct alternative can be selected after parsing.

Input and output of the editor. The textual notation editor has no direct access to an SDL-UML model, so that the required input and output of the editor must be transformed. The simplified workflow and involved artefacts are illustrated in Fig. 3.19.

The editor utilizes type definitions extracted from an SDL-UML model to disambiguate the syntax tree and to verify the conformance to the well-formedness rules specified for the textual notation. (see Fig. 3.19). These type definitions are extracted by invoking a dedicated QVT transformation before the editor is opened. Starting from the element in an SDL-UML model for which a textual notation is specified, the transformation collects all visible type definitions and transforms them to an editor internal format. Later, this data can be accessed by the various editor components.

When the editor content shall be saved, the parsed syntax tree cannot be directly mapped to an SDL-UML model. Instead, it is mapped to a syntax tree model MM_{CS} that is passed as input to a QVT transformation chain. After several transformation steps, the output of the transformation chain consists of SDL-UML elements stored in the current SDL-UML model. The syntax tree model is not accessible by the user, because it is only intended for the internal data exchange.

In addition to the created SDL-UML elements, the editor saves the entered textual notation in terms of a UML *Comment* owned by the model element for which the notation is specified. Therefore, this content can be loaded from the *Comment* when the editor is opened once again for the same model element.

Disambiguation of the textual notation. As we discussed in [86], the concrete syntax of SDL contains ambiguities that are relevant for the textual notation editor of SU-MoVal. Disambiguation is possible only with context-sensitive information provided by the QVT transformation mentioned above.

Table 3.2: Different ambiguities of SDL's concrete syntax

Alternative 1	Alternative 2
VariableAccess(**id**)	Literal(**id**)
TimerActiveExpression()	ActiveAgentsExp(**id**)
CallStatement(ProcCallBody(**id**, ...))	CallStatement(RPCallBody(**id**, ...))
Destination_agentId(**id**)	Destination_pid(VariableAccess(**id**))

The ambiguities relevant to the textual notation are listed in Table 3.2. All ambiguities between the listed syntax alternatives are caused by identifiers (bold printed). If an ambiguity is identified, the disambiguation algorithm of the editor resolves the identifier within the list of visible type definitions.

Constraints for the textual notation. Instead of utilizing an OCL-based approach as we first proposed in [86], the verification of constraints on the textual notation of SU-MoVal relies on the built-in capabilities of the Spoofax editor framework. The disadvantage of an OCL-based solution is a bad performance, because each time when the validation is invoked, the current parse tree has to be mapped to a new syntax tree model. By contrast, the constraint validation of Spoofax operates directly on the internal parse tree. Hence, we specified appropriate constraints by means of the Spoofax syntax. Below, we give an example of such a constraint on the textual notation.

```
ResetClause(identifier, _)−>(identifier, $['[timerName]' must denote a timer!])
where
  timerName := <ID−to−fullQualifiedName> identifier;
  not(<is−timer−definition> identifier)
```

The purpose of our example constraint is to assure that the identifier of a reset clause refers to a visible timer definition. Therefore, we define a so-called matching rule (e.g., $\mathrm{ResetClause(identifier,\ \_)}$) that matches against a distinct syntax tree node. If the condition of such a matching rule is fulfilled, an error message is created and displayed to the user. In addition, the corresponding location in the editor is highlighted. As in the case of disambiguation, the component for constraint validation utilizes the list of visible type definitions to determine correct type definitions.

3.6.2 QVT-based Transformations

Due to the positive result of our case study [86] concerning the general applicability of QVT's *Operationl Mappings* language [113] for transforming short-hand notations of SDL, we employed this language to implement the model transformations required for SU-MoVal. However, this case study covered only a small subset of all implemented model transformations. The main challenges we had to consider for creating the transformations were the following:

- *Transformation models for data types*: A data type concept similar to that of the concrete syntax of SDL is defined for SDL-UML. Hence, relevant transformation models specified for SDL also apply for SDL-UML.
- *Expansion of short-hand notations*: As we pointed out in Sec. 3.2.2, so-called short-hand notations of SDL's concrete syntax must be expanded to simpler constructs. Because SU-MoVal's textual notation is based on a subset of this syntax, a subset of the short-hand notations is also relevant for SDL-UML.

- *Name resolution*: Identifiers in the textual notation have to be resolved to full qualified identifiers, taking into account type definitions contained in an SDL-UML model.

Data type transformations. Since the data type concept of SDL-UML 2013 involves a few language concepts of SDL's concrete syntax, the relevant transformation models specified for SDL data types have to be applied to SDL-UML models. For this reason, the SU-MoVal framework employs a transformation chain consisting of three in-place transformations that are invoked before an SDL-UML model is mapped to SDL's abstract syntax or before the textual notation editor is opened. We use these transformations for the following purposes:

1. *Generic and implicit data type operations* are added by a transformation that implements the relevant transformation models specified in Z.101 [70]. Therefore, the two generic operations equal and copy are added to each «DataTypeDefinition».

2. *Multi-valued properties* are not supported in SDL. Therefore, we apply a further transformation to all elements that are stereotyped by «Property», «Parameter» or «Variable». Provided one of these elements is multi-valued, a new «DataTypeDefinition» which inherits from an appropriate generic data type is introduced by the transformation.

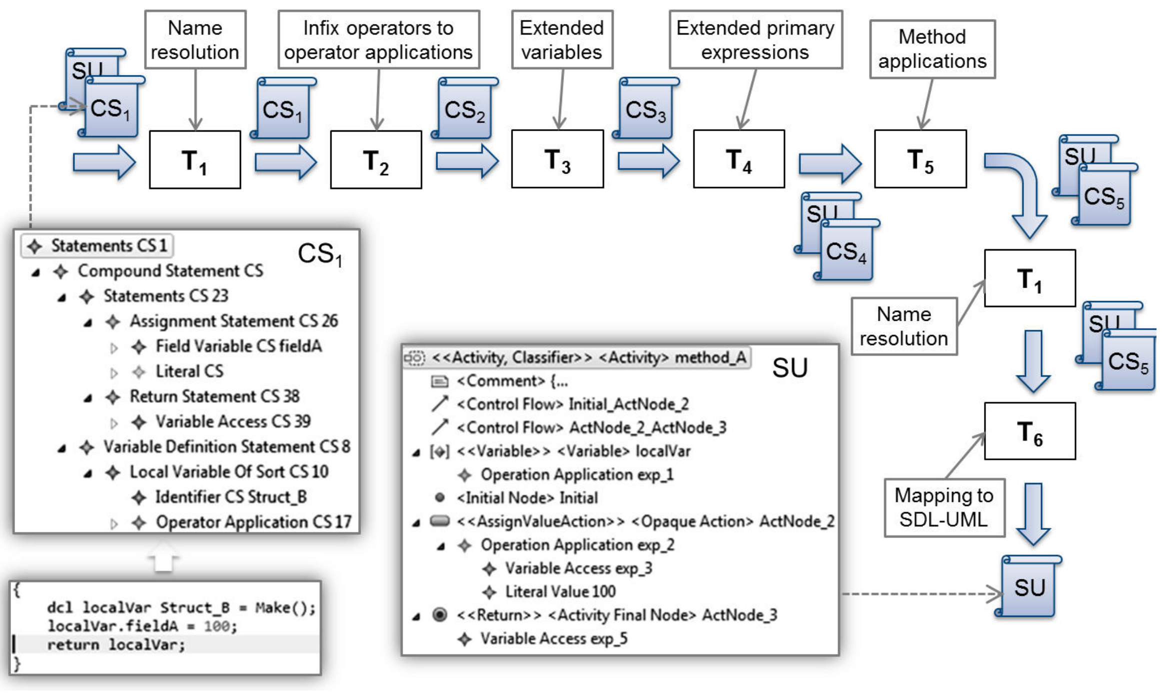
3. *Definition of inherited operations* is implemented by the third transformation, which realizes SDL's transformation model for inheritance. Therefore, this transformation identifies the set of inheritable «Operation»s. Then, these operations are added to all subtypes of the «Classifier» in question.

Mapping of the textual notation to SDL-UML elements. A further transformation chain (shown in Fig. 3.20) implements the mapping of the textual notation to corresponding elements of an SDL-UML model. Even though this chain is composed of several transformations invoked in sequential order, it can be divided into the following functional parts:

- Name resolution of identifiers (transformation T_1)

- Transformation of short-hand notations (transformations T_2–T_5)

- Mapping of the textual notation to corresponding SDL-UML elements (transformation T_6)



```
Statements CS 1
  Compound Statement CS
    Statements CS 23
      Assignment Statement CS 26
        Field Variable CS fieldA
        Literal CS
      Return Statement CS 38
        Variable Access CS 39
    Variable Definition Statement CS 8
      Local Variable Of Sort CS 10
        Identifier CS Struct_B
        Operator Application CS 17
```

```
{
    dcl localVar Struct_B = Make();
    localVar.fieldA = 100;
    return localVar;
}
```

```
<<Activity, Classifier>> <Activity> method_A
  <Comment> {...
  <Control Flow> Initial_ActNode_2
  <Control Flow> ActNode_2_ActNode_3
  <<Variable>> <Variable> localVar
    Operation Application exp_1
  <Initial Node> Initial
  <<AssignValueAction>> <Opaque Action> ActNode_2
    Operation Application exp_2
      Variable Access exp_3
      Literal Value 100
  <<Return>> <Activity Final Node> ActNode_3
    Variable Access exp_5
```

Figure 3.20: Input and output of the transformation chain that transforms concrete syntax elements to corresponding SDL-UML elements

Name resolution: All identifiers must be resolved before short-hand notations are expanded or the textual notation is mapped to SDL-UML elements. This is realized by transformation T_1, which implements the name resolution as specified in Z.100 [70]. Because type definitions in SDL-UML are specified by means of model elements, we had to modify the name resolution algorithm. The different parts of transformation T_1 are executed in the following order:

1. *Collection of visible type definitions:* Starting from the model element in an SDL-UML model for which textual notation is specified, all visible model elements that represent type definitions (e.g., «DataTypeDefinition») are collected according to SDL's visibility rules.

2. *Resolution by container:* This part resolves all identifiers that are not referring to literals, operators or methods. First, we determine the 'container' that encloses the model element for which an identifier is to be resolved. In SDL-UML, all subtypes of *Classifier* represent a 'container'. Then, an identifier is resolved based on the visible type definitions resulting from transformation part 1.

3. *Resolution by context*: All identifiers that refer to literals, methods or operators are resolved by this transformation part. First, we determine the 'context' of the resolution, which can be an assignment statement, a decision node or an expression. After that, the possible model elements to which an

Table 3.3: Short-hand notations to be expanded by model transformations

Transfor-mation	Purpose	Example input	Example output
T_5	Infix operators	(1 + 1) = 2	equal(add(1,1),2)
T_6	Extended variables	myVar[1] = 10	myVar = Extract(myVar, 10)
T_7	Extended primary	myVar[0]	Extract(myVar, 0)
T_8	Method applications	myVar.method_A()	method_A(myVar)

identifier may refer are computed according to the SDL's rules for resolution by context.

Short-hand expansion: As we pointed out at the beginning of this section, a few short-hand notations of SDL's concrete syntax are also supported by the textual notation of SU-MoVal. Therefore, they must be expanded to corresponding simpler constructs before other transformations or mappings can be applied. This task is realized by the model transformations T_2–T_5, which are shown in Fig. 3.20. All the short-hand notations supported by SU-MoVal and the associated transformations employed to expand them are shown in Table 3.3. As is apparent from this table, only four short-hand notations of SDL's concrete syntax are relevant in the case of our textual notation, and they are all associated with SDL expressions. In order to give a better impression of the purpose of a transformation, we provide exemplary inputs and outputs for all transformations listed in Table 3.3.

Mapping to SDL-UML elements: Before the final mapping to elements of an SDL-UML model can occur, transformation T_1 must be applied once again. This is required because various operator application expressions are created when expanding the short-hand notations, so the identifiers of these expressions must be resolved. Then, each node of the syntax tree is mapped to a corresponding SDL-UML element by transformation T_6. All elements are added below that element of an SDL-UML model for which the textual notation is specified. An example of the mapping of the textual notation to corresponding SDL-UML elements is given in Fig. 3.20.

3.7 Related Work

In the literature, many contributions on various topics related to SDL can be found. However, only few relate to a model-based specification of the SDL or to a UML profile for SDL. For example, Prinz et al. [128] and Scheidgen [135] present a metamodel for SDL. Both contributions propose to define the static semantics in terms of OCL *Constraints*, whereas the ASM formalism [128] or a dedicated action language [135] is used to formalize the dynamic semantics. Although many aspects are covered, both contributions do not treat the formalization of a UML profile for SDL.

A formalization of the UML profile for SDL, as published in Rec. Z.109 of 2007 [66], is analysed by Grammes in [56, 57]. Based on the UML profile's static semantics that is specified in natural language, he proposes to create corresponding OCL *Constraints*. However, this approach does not ensure that the given descriptions match the formally defined static semantics [72] of SDL. Therefore, this approach leaves open the gaps in the static semantics of SDL-UML 2007 that we identified in Sec. 3.3.

Furthermore, Grammes does not investigate whether the language concepts provided by SDL-UML 2007 are suitable for specifying a complete SDL model using SDL-UML. By contrast, we studied this topic intensively and identified some open issues and drawbacks related to the data type concept and the representation of SDL expressions (see Sec. 3.3).

The *UML Profile for Communicating Systems (UML-CS)* is proposed by Werner [150, 151] as an alternative to the standardized UML profile for SDL. The static semantics of UML-CS is defined in terms of OCL *Constraints*, and a mapping to the abstract syntax of SDL is specified employs the *eXtensible Stylesheet Language Transformations (XSLT)*[5], which is an XML-based transformation language.

Although UML-CS embraces stereotypes for specifying data types, some issues concerning the data type concept of UML-CS exist. For instance, all predefined data types of SDL are represented in UML-CS as instances of the *PrimitiveType* metaclass. This also applies to the generic data types, such as Array or Vector. Since the data type concept of SDL-UML 2007 is pretty similar to that of UML-CS, the same drawbacks we identified for SDL-UML 2007 also apply to UML-CS. In addition, the question remains open how the generic data types of UML-CS are mapped to the abstract syntax of SDL.

Another issue related to data types concerns the mapping of structured data types. In UML-CS, it remains unclear how data type fields are to be mapped to SDL because data types cannot have fields in the abstract syntax of SDL. Instead, data type fields have to be transformed to dedicated operations, and such transformations are not specified for UML-CS. By contrast, our revised SDL-UML edition of 2013 [69] defines in-place transformations that must be applied before mapping an SDL UML model to the abstract syntax of SDL.

A further drawback of UML-CS is that expressions can only be specified as plain text. Thus, the static semantics of expressions cannot be verified based on UML-CS models. Instead, the textual expressions must first be parsed, after which the entire UML-CS model has to be transformed into a corresponding SDL model. Only then is a verification of the static semantics possible at the level of SDL's

[5] http://www.w3.org/TR/xslt20/

abstract syntax. Because of our proposal to explicitly represent SDL expressions as dedicated metaclasses, the static semantics of expressions can be verified directly for an SDL-UML 2013 model. This eliminates a prior transformation step as is required for UML-CS models.

Apart from UML-CS discussed above, an activity was started by an ETSI working group to provide an alternative to the UML profile for SDL as published in 1999 [65]. The UML profile proposed by the ETSI working group should also be called 'UML Profile for Communicating Systems'. However, the specification [38] of this UML profile never went beyond the draft status and was never completed.

Compared to our contribution, the works discussed above have gaps or do not remedy the problems we have identified in Sec. 3.3. A key point is their lack to represent SDL expressions, so that not all aspects of the static semantics of an SDL-UML model can be verified at UML level. This drawback is resolved by our metaclass-based representation of SDL expressions and the static semantics defined in OCL.

3.8 Summary and Conclusions for an Automatic UML Profile Derivation

Summary. As pointed out in the introduction to this chapter, our goal for the revision of SDL-UML was to specify a UML profile that can be employed to model an entire SDL model using UML. To enable a sound transformation into an SDL model, the conformance of a UML model that has applied the UML profile for SDL, to the static semantics of SDL must be ensured. Typically, this kind of verification is realized using well-formedness rules defined by means of OCL *Constraints*.

To determine whether SDL-UML 2007 fulfilled the above objectives, we initially conducted a comparative case study between SDL and SDL-UML. As a first drawback, we found that the constraints of this UML profile were specified in natural language, so we had to translate them into OCL.

Another disadvantage of SDL-UML 2007 is the time-consuming specification of SDL statements and expressions using UML elements and related stereotypes. Because of these initial identified problems, we systematically compared the formal specification of SDL's syntax and static semantics [72] with that of SDL-UML 2007 [66]. As argued in Sec. 3.3, we identified three major problem areas: the data type concept and the representation of SDL statements and expressions. Overall, we found that stereotypes related to these language constructs have different drawbacks and gaps in terms of syntax and static semantics. For instance, some syntactic constructs such as stereotypes or their attributes are missing. Furthermore, the constraints specified in natural language are partially faulty or missing.

In Sec. 3.4, we discussed our approach to solve the drawbacks in the above areas of SDL-UML. We introduced a new data type concept and a new representation of SDL expressions, using dedicated metaclasses that extend the UML metaclass *ValueSpecification*. Furthermore, we showed how to employ OCL to formalize the static semantics of SDL-UML. To validate the applicability of our proposed modifications and enhancements, we developed the SU-MoVal framework, which we presented in Sec. 3.6. According to the approach discussed in Sec 3.5, we employed SU-MoVal to iteratively verity the OCL constraints and queries created by us for SDL-UML.

All proposed modifications were submitted to the ITU working group responsible for the standardization of SDL, so that they could be incorporated into SDL-UML 2013 [69]. Most of our proposed modifications were accepted by the working group. However, our OCL constraints did not become part of the standard as there were different views on how to publish them.

Conclusions for an automatic UML profile derivation. In the thesis's motivation, we emphasized that the experience we gained while working on SDL-UML 2013 inspired us to explore an automated derivation approach for UML profiles. Based on the points above, we argue below how we derived this research objective.

As we analysed in this chapter, a major disadvantage of SDL-UML 2007 was its lack of syntactic constructs compared to the abstract syntax of SDL. In our opinion, this originated from the manual creation of the UML profile without conducting a proper verification. This source of error is not only specific to SDL-UML, but may also occur with any other DSL for which a UML profile is to be created manually. To avoid such error, a UML profile for an existing DSL should always be derived from its grammar. As in the case of metamodels, a derivation of UML profiles based on production rules would be conceivable. In such a scenario, however, MDE approaches cannot be applied. For this reason, we do not consider to directly derive UML profiles based on production rules of an existing DSLs; instead, we derive metamodels in a first step. Then, we can automatically derive UML profiles based on such metamodels in a second step. Due to this two-staged derivation, the approach could also be used for new DSLs, where metamodels are to be created directly.

The revision and extension of stereotype constraints defined by means of natural language and their subsequent translation into OCL constraints and queries took high efforts for us while revising SDL-UML and creating our SU-MoVal framework. This was caused by inaccurately specified constraints in natural language and missing OCL counterparts in SDL-UML 2007. Therefore, we used the formally-defined static semantics of SDL as a starting point for introducing constraint descriptions in natural language and for creating corresponding OCL constraints.

A challenge in translating SDL's static semantics into OCL constraints and queries was to identify matching metaclass attributes for SDL production rule symbols. This was partly because the names of the metaclass attributes and the corresponding production rule symbols were not equal or similar. In contrast, finding matching stereotype attributes was trivial because their names matched those of the production rule symbols. In addition, the syntactic and semantic gaps between UML and SDL made it difficult to identify appropriate UML counterparts. Both discussed aspects were potential error sources when we manually translated SDL's static semantics. We were only able to identify and correct these errors through an iterative specification in conjunction with appropriate verification tests.

To avoid potential errors and high efforts when translating the static semantics, the OCL constraints and queries of a metamodel for a DSL should be automatically transferable to a derived UML profile, without requiring a manual revision. However, a prerequisite is that a matching counterpart for each syntactic construct of the DSL must exist in a derived UML profile. In addition, we have to bridge the syntactic and semantic gaps between the DSL of interest and the UML. For instance, we could employ the approach chosen by us for SDL-UML 2013, as explained in Sec. 3.4.3. Accordingly, we might introduce OCL expressions at locations where syntactic or semantic gaps exist in a derived UML profile. Instead of inserting these OCL expressions for OCL constraints or queries to be transferred from a metamodel, we consider it as more appropriate to define stereotype attributes as 'derived' and to create OCL expressions that compute the attribute values at runtime. Thus, we avoid a complex algorithm for modifying OCL constrains and queries to be transferred from a metamodel to a UML profile.

Although the derivation of UML profiles for DSLs and the transfer of static semantics can be achieved, this does not replace the validation and verification of a derived UML profile. On the one hand, we assume that the mapping of metaclasses to stereotypes needs to provide information that defines which UML metaclasses should be augmented with derived stereotypes. On the other hand, the OCL expressions for bridging the syntactic and semantic gaps cannot be generated automatically; instead, they must be manually specified and provided in an appropriate manner. Due to both reasons, we consider reviewing and validating derived UML profiles as essential.

Because we consider to validate and verify metamodels for DSLs by employing the same methodology as applied to SDL-UML 2013, we have to create appropriate test models. A reuse of these test models to validate and verify derived UML profiles would be an asset. However, the test models created for metamodels must be transformed into equivalent UML models with applied stereotypes. To automate this activity, we can create appropriate M2M transformations. However, a manual creation of such transformations is associated with additional effort. Therefore, if possible, we wish to derive these transformations based on metamodels for DSLs. In addition, these transformations can be used later to achieve model interoperability between the DSL of interest and UML.

4 An Automatic Derivation Approach

In the previous chapter, we identified potential drawbacks of manually creating UML profiles for DSLs using the example of SDL. A major disadvantage is that appropriate verification and validation activities are required to ensure that, for each syntactic construct of a DSL, a corresponding counterpart is present in the created UML profile. Another challenge concerns the manual transfer of the static semantics of a DSL to a corresponding UML profile.

As highlighted in the introduction to the thesis, the above points motivated us to develop an approach to automatically derive UML profiles based on CMOF metamodels of DSLs. The processing of CMOF language concepts, e.g., attribute subsetting and redefinition, is one of the challenges that we address with our approach. Furthermore, our approach supports the automatic transfer of the OCL-defined static semantics of a metamodel to a derived UML profile.

To support grammar-based DSLs, our approach enables the automated derivation of CMOF-based metamodels based on production rules. In addition, we support model interoperability between DSL models and corresponding UML models with applied UML profile. For this purpose, our approach offers the automatic derivation of appropriate M2M transformations.

In the first section of this chapter, we give an overview of our approach and the *DSL Metamodelling and Derivation Toolchain (DSL-MeDeTo)* that implements it. In addition, we introduce an appropriated MDE-based development process that can be combined with our derivation approach. Then, we present the derivation of each artefact type in a specific section. Finally, we analyse the related work and point out our conclusions.

4.1 Overview of the Approach

This section provides an overview of our overall approach to derive a metamodel, a UML profile, and model transformations that can transform a domain model into a UML model, and vice versa. Furthermore, we propose an MDE-based development process that shall be applied combined with our approach. In addition, we give a short introduction to our DSL-MeDeTo that implements all aspects of our approach.

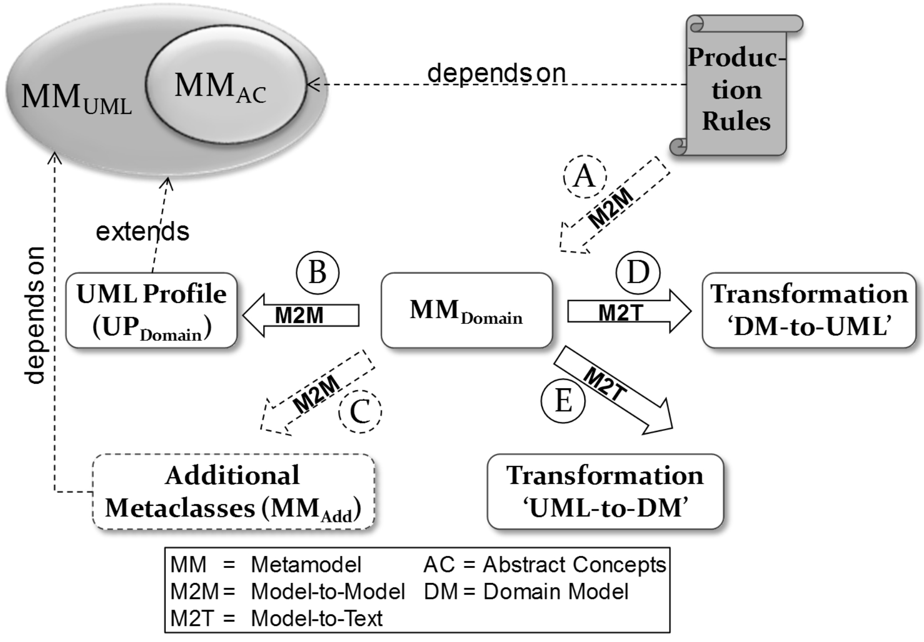


Figure 4.1: Transformation steps and their derived artefacts

4.1.1 Our Overall Approach

Our holistic derivation approach shown in Fig. 4.1 consists of several steps, some of which are optional. The DSL-specific metamodel MM_{Domain} is the central artefact for almost all derivations. If production rules of a grammar-based DSL are available, the metamodel can be generated automatically; otherwise, it has to be created manually.

To obtain a metamodel that does not require too much effort for further refinement, we reuse 'Abstract Concepts' that are defined by an existing metamodel (MM_{AC}), as proposed in [44, 135]. Apart from these works, the reuse of constructs of existing languages is also recommended by other works (e.g.,[26, 77]). In contrast to the approach proposed in [44, 135], we use particular annotations for a given DSL's production rules so that relationships between generated metaclasses and the 'Abstract Concepts' must not be created manually.

We define 'Abstract Concepts' as a set of generic language concepts that are commonly shared across several DSLs and not only applicable to a specific language. For example, language concepts such as generalization or redefinition can be regarded as 'Abstract Concepts'. Furthermore, we presume that each 'Abstract Concept' is represented by a particular metaclass contained in MM_{AC}.

If MM_{Domain} is derived based on production rules in Step (A), a few manual refinements have to be made before it can be used as input for Steps (B)–(E). In particular, OCL constraints have to be specified to meet the static semantics of the DSL. Otherwise, MM_{Domain} has to be created from scratch using the 'Abstract Concepts' in MM_{AC}, and the relation between metaclasses in MM_{Domain} and the 'Abstract Concepts' is defined through inheritance.

After creating MM_{Domain}, we can automatically derive a DSL-specific UML profile UP_{Domain} in Step (B). If required, additional metaclasses (contained in MM_{Add}) that extend the MM_{UML} are derived in Step (C). The derivation of additional metaclasses may be an option if stereotypes cannot be employed due to their restrictions as defined by the UML [116]. For instance, such an approach is applied for the value and expression languages of the SDL-UML profile [69] and of the MARTE profile [117]. Because the input and output artefacts of Steps (B) and (C) are models, we realize both derivations by two dedicated M2M transformations.

In Steps (D) and (E), we derive two M2M transformations that can be used to obtain model interoperability between DSL-specific models and UML models with applied UML profile. In contrast to the aforementioned artefacts, we develop two *Model-to-Text (M2T)* transformations to generate the source code of the M2M transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$.

4.1.2 The 'Abstract Concepts'

As pointed out above, the metamodel MM_{AC} holds a key role for our entire derivation approach, because metaclasses of the metamodel MM_{Domain} inherit from 'Abstract Concepts' defined by MM_{AC}. An important prerequisite for MM_{AC} is that it has to match with a subset of MM_{UML}. Otherwise, a straightforward mapping of MM_{Domain} to UP_{Domain}, as implemented by our approach, is impossible.

Constraints. We consider a metamodel MM_{AC} to be matching with MM_{UML}, if the following constraints are fulfilled:

Constraint 1 *For each metaclass* `MC` *of* MM_{AC}*, a corresponding metaclass* `MC'` *with an equal name shall be present in* MM_{UML}*. In addition,* `MC` *shall have an equal or lesser number of attributes than* `MC'`*.*

This constraint is essential for the derivation of UML profiles according to our approach, because the UML metaclasses to be extended by *Stereotypes* are determined based on the correlation between 'Abstract Concepts' and UML metaclasses. To determine such a correlation, we employ the *name* property of the metaclasses. Even though an 'Abstract Concept' has fewer attributes than the corresponding UML metaclass, we consider a correlation as given. For instance, such a situation may occur if some attributes of an 'Abstract Concept' are removed because they are not required to define the syntax of a DSL.

Constraint 2 *For each attribute* `att` *of a metaclass* `MC`*, a corresponding attribute* `att'` *with an equal name shall be present in metaclass* `MC'`*. In addition,* `att` *and* `att'` *shall have the same properties, e.g., the same type and cardinality.*

The condition imposed by Constraint 2 is important because OCL constraints and queries in MM_{Domain} may capture attributes of 'Abstract Concepts'. When deriving a UML profile, such an attribute access must be translated into an access to a

UML metaclass attribute. In addition, attributes of metaclasses in MM_{Domain} may redefine or subset 'Abstract Concept' attributes. During the UML profile derivation, we have to translate such attribute relationships into appropriate constructs that access UML metaclass attributes.

Constraint 3 *A data type of MM_{AC} shall have a corresponding data type in MM_{UML}.*

In addition to metaclasses, the 'Abstract Concepts' may contain *DataTypes*. Therefore, each of these *DataTypes* must have a corresponding type in MM_{UML}. This condition is ensured by Constraint 3.

Creating the metamodel MM_{AC}. Different approaches can be applied to obtain a metamodel MM_{AC}. Apart from creating it from scratch, Clark et al. [26] argue that also a reuse of metaclasses of an existing metamodel, by copying or importing them, can be considered for establishing a new metamodel.

As MM_{AC} shall match with MM_{UML}, we consider a creation of MM_{AC} from scratch to be too error-prone and expensive. Another option is to use the MOF or the UML Infrastructure Library [114]. Because the metaclasses of these metamodels are primarily employed to define UML's 'Kernel' package, they could be reused to create an MM_{AC} that only supports structural language concepts (e.g., *Classifier*). Finally, also the reuse of parts of MM_{UML} could be considered if language concepts for behavioural specifications (e.g., *StateMachines*) are required.

Our approach does not support the import of metaclasses; otherwise, there would be a dependency between MM_{Domain} and the metamodel from which the metaclasses are imported. Therefore, we assume that MM_{AC} is created manually as a copy or automatically using the Package Merge feature provided by the CMOF.

4.1.3 Our MDE-based Development Process

In the following, we propose an MDE-based development process that is applicable to our derivation approach. Due to the reasons highlighted in Sec. 2.1, we wish to adapt the process proposed by Strembeck and Zdun [145].

The main process that we have adapted is shown as an activity diagram in Fig. 4.2. The details of the sub-activities involved are discussed in the next sections of this chapter. When compared to the original process shown in Fig. 2.1, the process proposed by us embraces two additional and one modified action. The affected Actions (1) – (3) in Fig. 4.2 are coloured white, while the unmodified Actions (4) – (6) are coloured grey. Instead of using the input and output artefact names of the original process, we employ the artefact names as already introduced for our derivation approach, e.g., MM_{Domain} instead of 'core language model'.

In Action (1) of our process, the metamodel MM_{Domain} is manually created for a new DSL or derived from production rules of an existing DSL. The latter case corresponds to Step (A) of our derivation approach. Then, the metamodel MM_{Domain} is enriched with meta-information in Action (2), so that a UML profile for a DSL

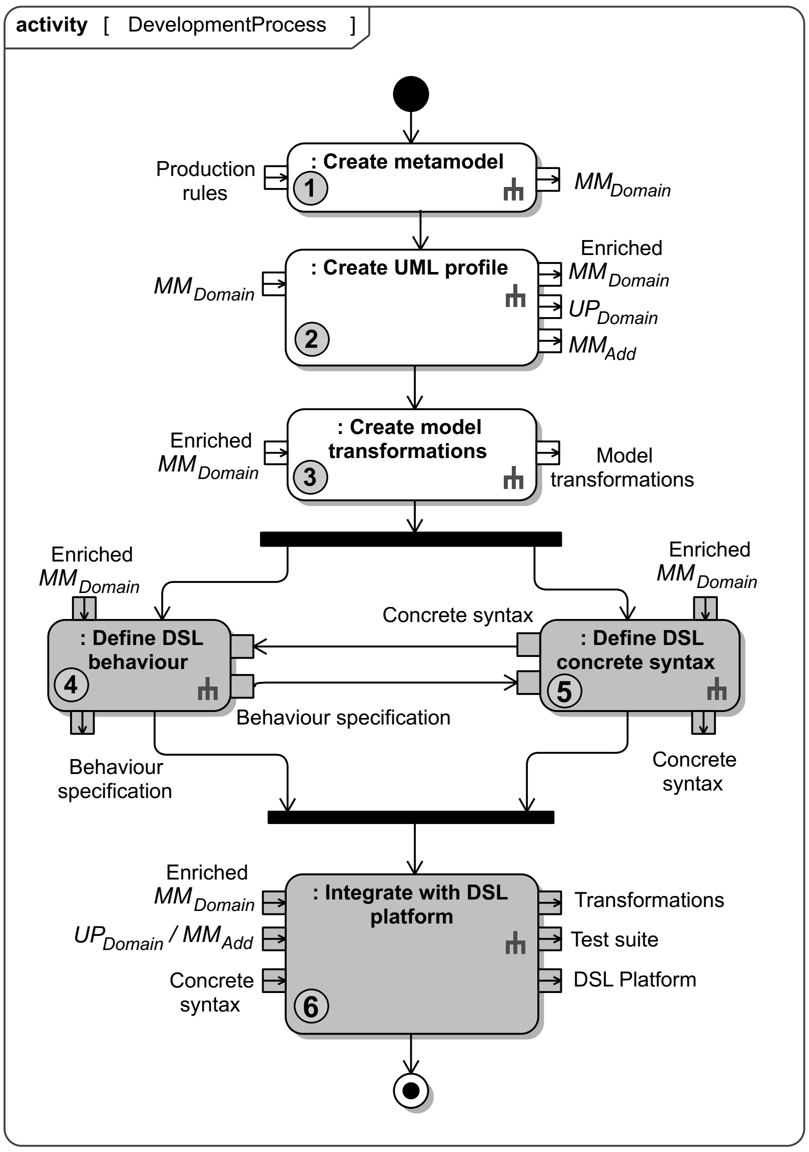


Figure 4.2: Our MDE-based development process for DSLs (a modified variant of the process proposed in [145])

can be derived from it. If required, the additional metaclasses contained in metamodel MM_{Add} must also be derived in Action (2). Thus, this corresponds to Steps (B) and (C) of our approach. Action (3) represents our Steps (D) and (E), where the two model transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ are derived. To conduct Action (3), the enriched metamodel MM_{Domain} of Action (2) must be passed to it.

Finally, Actions (4) – (6) can be executed, which correspond to the actions of the original process shown in Fig. 2.1. Depending on the *integrated development environment (IDE)* or UML tool used, a task of Action (6) is to integrate either the metamodel MM_{Domain} or the UML profile UP_{Domain} and, if applicable, the metamodel MM_{Add}.

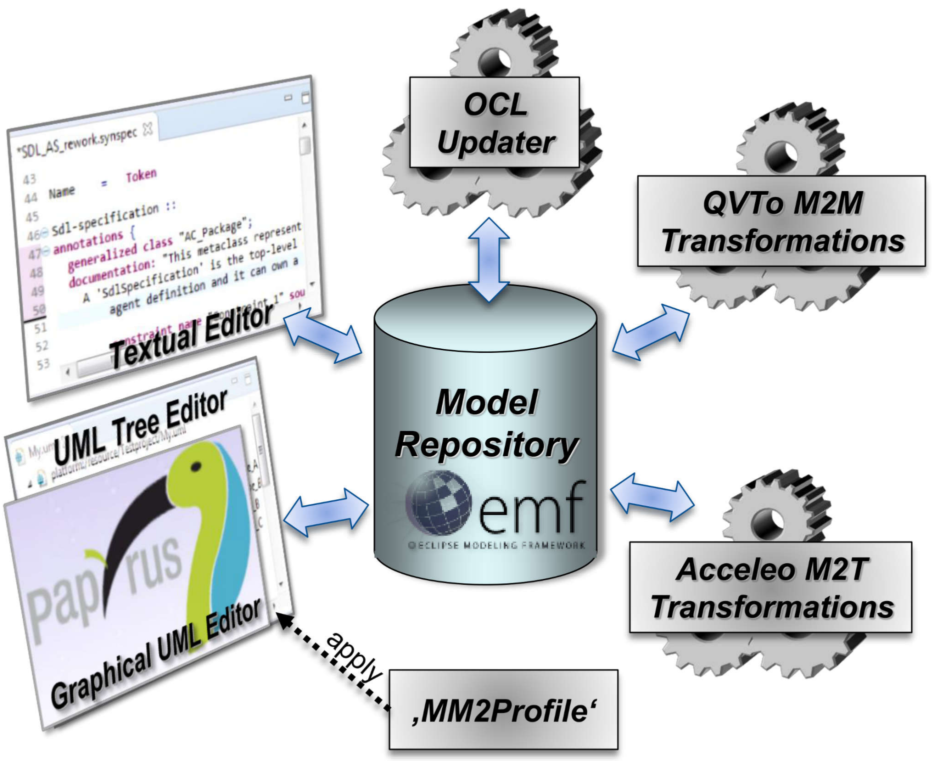


Figure 4.3: The Eclipse-based toolchain and its components

4.1.4 Implementation of the DSL Metamodelling and Derivation Toolchain

The following briefly explains the most important components of our *DSL Metamodelling and Derivation Toolchain (DSL-MeDeTo)* [156] that implement all aspects of our approach. Essential objectives for the toolchain design are the utilization of well-established standards and technologies and the use of open-source components. We prefer standardized technologies because one of our goals is to minimize the training effort for users of our toolchain. Furthermore, we utilize open source components to reduce implementation effort.

Accordingly, we choose the *Model Development Tools (MDT)* edition of Eclipse[2] to realize our toolchain, which consists of a set of particular plug-ins as shown in Fig. 4.3. To create a metamodel or to derive other artefacts such as UML profiles, the components of our toolchain have access to a common model repository that contains the UML model representing the metamodel. Because our derivation approach is designed for CMOF-based metamodels, we have to employ UML models instead of Ecore models. The code generators of Eclipse-MDT can handle both formats. The most important plug-ins of our toolchain are:

Textual editor. The textual editor of our toolchain is used to specify the production rules for the DSL or computer language of interest. In addition, the production rules can be associated with different types of annotations, e.g., to define a relationship to the 'Abstract Concepts'. The parsed production rules are stored by the textual editor in a intermediate model-based format; thus, they can be transformed to a metamodel by employing an M2M transformation. As we already made good experiences with Spoofax for creating our SU-MoVal framework (see Sec. 3.6), we have also used this language workbench to create the textual editor for DSL-MeDeTo.

UML Profile 'MM2Profile'. The UML profile 'MM2Profile' is used to enrich a metamodel MM_{Domain} with additional information, which is processed by different components of the toolchain. The application of this UML profile to a metamodel MM_{Domain} is a prerequisite for the derivation of a UML profile UP_{Domain} and of the additional metaclasses MM_{Add}. In addition, 'MM2Profile' defines a set of OCL *Constraints* that must be met by MM_{Domain} so that its processing by our toolchain is sound. More details of 'MM2Profile' are discussed in Sec. 4.3.3.

OCL Updater. Before the metamodel MM_{Domain} with applied 'MM2Profile' can be used for the derivation of other artefacts, all its OCL-defined *Operations, Properties* and *Constraints* have to be adapted in such a way that they can be utilized for a derived UML profile. This adaptation is implemented by the 'OCL Updater' component, which consists of an OCL parser and a pretty printer. The update is based on the abstract syntax tree (AST) of a parsed OCL expression, as argued in Sec. 4.5

M2M Transformations. We employ the operational language of the *Query/View/Transformation* specification (QVT) [113] for implementing the following M2M transformations of our toolchain:

- $T_{SR\text{-}to\text{-}MM}$: This transformation implements Step (A) of our approach; its output is the CMOF-based metamodel MM_{Domain}. The transformation requires two different input models. The first is the intermediate model that represents the parsed production rules, and the second model MM_{AC} contains the associated 'Abstract Concepts'.

- $T_{MM\text{-}to\text{-}Profile}$: In Step (B) of our approach, the UML profile UP_{Domain} is derived based on MM_{Domain}. This derivation is implemented by the $T_{MM\text{-}to\text{-}Profile}$ transformation.

- $T_{MM\text{-}to\text{-}AddMC}$: This transformation implements the optional Step (C) of our approach, where additional metaclasses contained in metamodel MM_{Add} are derived.

Our toolchain employs the Eclipse plug-in QVTo for executing the M2M transformations. Each transformation can directly be invoked from an Eclipse context menu, without specifying any parameters, because they are extracted from the model that is used as input.

M2T Transformations. We use the *MOF M2T Language (MTL)* [111] for realizing two M2T transformations that implement Steps (D) and (E) of our approach. The M2T transformation $T_{GenDM\text{-}to\text{-}UML}$ is employed to generate the M2M transformation $T_{DM\text{-}to\text{-}UML}$ in Step (D), while $T_{GenUML\text{-}to\text{-}DM}$ generates $T_{DM\text{-}to\text{-}UML}$ in Step (E).

The source code of both M2M transformations is generated in terms of the operational language of QVT. We utilize the Acceleo[1] component of Eclipse to execute both M2T transformations, which can be invoked from Eclipse's context menu.

Model editors. Apart from the discussed components, either Eclipse's UML tree editor or the UML modelling tool Papyrus[2] can be used for creating or modifying the metamodels that are processed by our toolchain (see Fig. 4.3). Hence, one of these tools also have to be utilized for applying the UML profile 'MM2Profile' to a metamodel.

4.2 Metamodel Derivation

The details of our approach for deriving CMOF-based metamodels from production rules are discussed in this section. First, we define a process to be used with our approach for deriving metamodels. Then, we analyse the requirements concerning the derivation of metamodels from production rules, so that we can define a set of design decisions that form the basis of our derivation approach. We also introduce a running example that is intended to illustrate the different aspects of our approach.

4.2.1 Process for Creating Metamodels

The actions for creating a metamodel for a DSL are captured by the activity shown in Fig. 4.4, which is part of our MDE-based development process introduced in Sec. 4.1.3. Our Actions (1) – (3) and (5) are coloured white, while the actions taken from the original process [145] are coloured grey.

Metamodel for an existing DSL. In the case of an existing DSL, the activity starts with Action (1), whereby 'Abstract Concepts' are identified based on the language concepts of the DSL of interest. Thereafter, appropriate metaclasses of the UML metamodel or the UML infrastructure library are selected and copied to the metamodel MM_{AC}. Finally, non-required features of the metaclasses contained in MM_{AC} are removed, if necessary. The resulting metamodel MM_{AC} is passed as an input to Action (2), where the production rules of the DSL are enriched with annotations so as to relate them to the 'Abstract Concepts'. In Action (3), the metamodel MM_{Domain} is derived automatically based on the annotated production rules

[1] http://www.eclipse.org/acceleo/
[2] https://www.eclipse.org/papyrus/

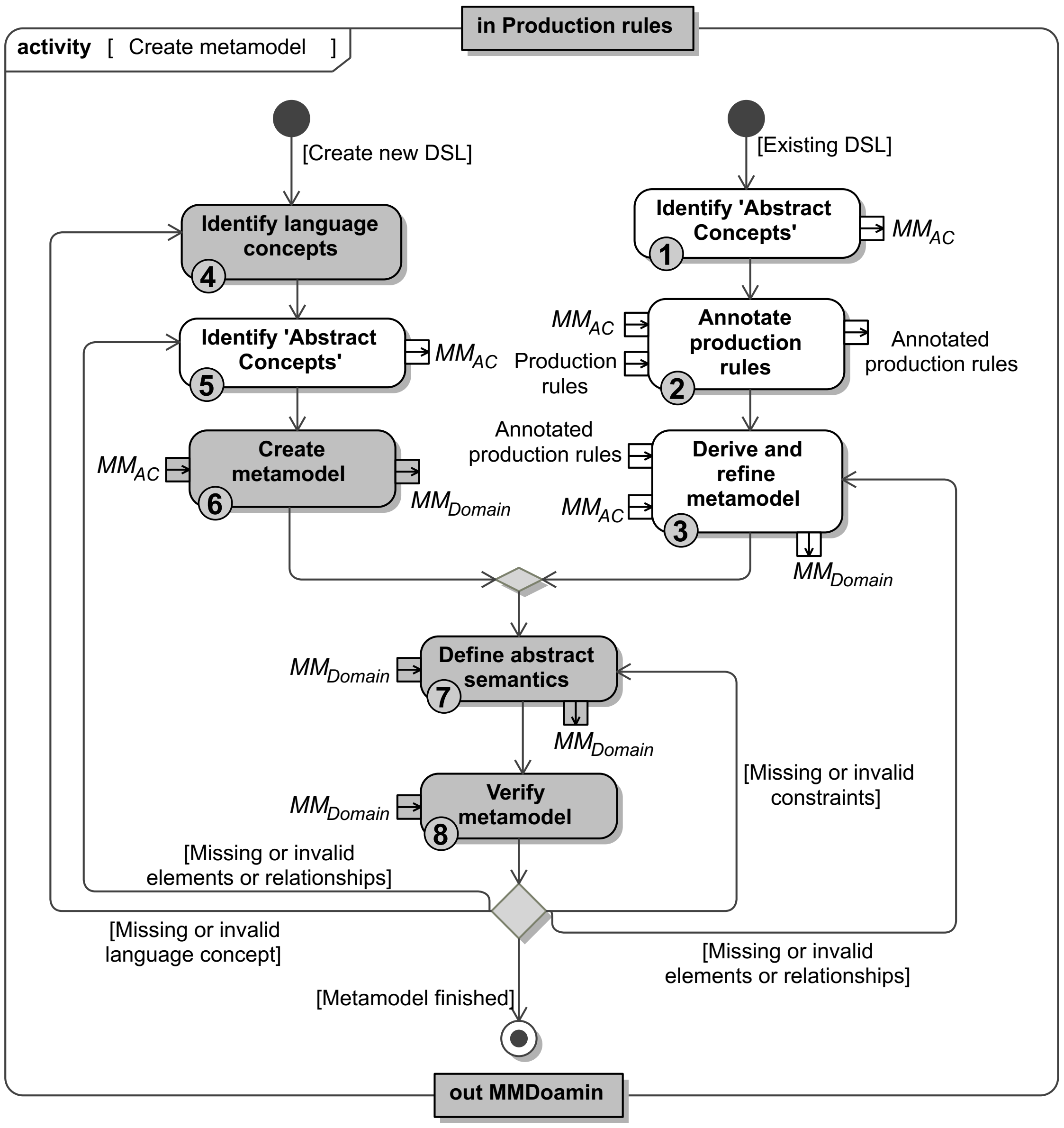

in Production rules
activity [Create metamodel]
[Create new DSL]
[Existing DSL]
Identify language concepts
4
Identify 'Abstract Concepts'
1
MM_AC
Identify 'Abstract Concepts'
5
MM_AC
MM_AC
Production rules
Annotate production rules
2
Annotated production rules
MM_AC
Create metamodel
6
MM_Domain
Annotated production rules
MM_AC
Derive and refine metamodel
3
MM_Domain
MM_Domain
Define abstract semantics
7
MM_Domain
MM_Domain
Verify metamodel
8
[Missing or invalid constraints]
[Missing or invalid elements or relationships]
[Missing or invalid language concept]
[Missing or invalid elements or relationships]
[Metamodel finished]
out MMDoamin

Figure 4.4: Activity for manually creating or deriving a metamodel

of Action (2) and the metamodel MM_{AC}. Action (3) corresponds to Step (A) of our derivation approach.

Metamodel for a new DSL. If a metamodel shall be created for a new DSL, the activity starts with Action (4), where the necessary language concepts have to be analysed. Thereafter and as in Action (2), the 'Abstract Concepts' required for a DSL are determined in Action (5), and the metamodel MM_{Domain} is created manually in Action (6).

Regardless whether the metamodel MM_{Domain} is derived automatically or created manually, the static semantics of the DSL using OCL constraints and queries has to be defined in Action (7). Because our approach creates CMOF-based metamodels, the static semantics is part of the metamodel, which is in contrast to the original process proposed in [145].

Finally, the metamodel and its static semantics are verified in Action (8). This can be conducted by employing the approach we applied to the UML profile for SDL (see Sec. 3.5). Because the verification of metamodels is not part of our derivation approach, we do not provide any details here.

4.2.2 Design Decisions for the Metamodel Derivation

The *Backus-Naur Form (BNF)* was developed in the late 1950s for the specification of the programming language ALGOL [4]. BNF is a so-called metalanguage that can be used to formally define the syntax of context-free grammars in the form of production rules. Many extensions have been proposed for BNF (e.g., [28, 63, 67]), which are generally summarized under the term *Extended BNF (EBNF)*. Before we introduce our design decisions that are the foundation stone for our approach to derive CMOF-based metamodels, we briefly discuss the commonalities of all EBNF-based notations.

The syntax of a language defined by means of production rules consists of a set of symbols that can be divided into terminal and non-terminal symbols. A non-terminal symbol is replaced during the syntax analysis by applying the defined production rules until only terminal symbols that cannot be further decomposed are present. A non-terminal symbol is introduced by a production rule, which defines how it can be replaced by other symbols. For this purpose, a production rule consists of a left-hand part, which defines the name of the non-terminal symbol, and a right-hand part, which consists of one or more expressions. Typically, all EBNF variants provide various expression types, so that terminal symbols, references to non-terminal symbols, alternative symbols, symbol compositions, optional symbols, and symbol repetitions can be defined.

The translation of production rules into EMOF-based metamodels has been explored in several works (e.g., [6, 44, 94, 153]), which provides a good foundation for deriving metamodels from production rules. However, these works do not cover the derivation of CMOF-based metamodels, which is an objective of our derivation approach.

Because the language concepts of the EMOF are a subset of those provided by CMOF, we can build on the approaches proposed in the literature to derive the general elements of metamodels. Therefore, we first discuss the parts that we wish to adopt from the approaches presented in the literature. Thereafter, we analyse the CMOF-specific language concepts that must be supported by our derivation approach. Based on the insights gained through this analysis, we specify design decisions that define the CMOF-related aspects of our approach.

Fundamental mapping rules. Although the approaches proposed in [6, 44, 94, 153] differ in their details, they use similar rules for mapping production rules to metaclasses of a metamodel. We can summarize these rules as follows:

- For each production rule of a grammar, a corresponding metaclass P is created in the derived metamodel.
- If a production rule contains an alternative expression, the corresponding metaclass AP is marked as abstract.
- An expression that references a non-terminal symbol is mapped to an *Association* between a metaclass P and the metaclass introduced for the non-terminal symbol.
- An expression representing a terminal symbol is mapped to an attribute att of a metaclass P. In addition, a *DataType* S is created based on the terminal symbol, if it is not already present in the metamodel. Moreover, the *type* property of the attribute att must reference *DataType* S.
- For each symbol listed in an alternative expression AE, its associated metaclass AS is determined. Then, a *Generalization* from AS to the metaclass that represents the production rule containing AE is introduced.
- Each sub-expression in a composite expression is mapped to an attribute or association end in the metaclass P, which is derived from the production rule that contains the composite expression.
- If an expression is inside an optional expression, the attribute or association end derived from it is defined as optional.
- If an expression is inside a repetition expression, the attribute or association end derived from it is specified to be a sequence.
- All attributes and association ends introduced for production rule expressions must be created as compound aggregation so that their values become part of metaclass instances at runtime.

Design Decision 1 *The essential elements for CMOF-based metamodels, such as metaclasses and data types, shall be derived from production rules by employing the fundamental mapping rules above.*

CMOF-specific and other considerations. The major drawback of the metamodels created by applying the above mappings is that they can only be considered as primitive or initial [44, 94]. The reason is that a metamodel can only be as expressive as the grammar (i.e., EBNF) from which it is obtained. Thus, a manual revision of metamodels derived from production rules is necessary. For example, the following problem categories are identified in [44] as the most important ones that need to be revised manually.

1. Reuse of abstract language concepts by generalization,

2. Chained production rules, each with an expression that points to a non-terminal symbol, which causes the generation of a high number of metaclasses. For example, consider the following production rules:

   ```
   SymoblA := SymbolB
   SymbolB := SymbolC
   SymbolC := TOKEN
   ```

   Using the mapping rules introduced above, we obtain three metaclasses connected by two associations, and the last metaclass has an attribute of String type. This construct would be manually reworked so that only a single metaclass with one attribute is present after the revision.

3. Another problem in EBNF concerns unsupported features of *Property* and *Association*, such as navigable associations and the ability to define association ends or metaclass attributes as a reference rather than an aggregation.

4. Typically, languages that have a textual notation provide the language concepts Identifier and Qualifier to refer to type definitions. After applying associated production rules, both concepts are represented as strings or string sequences in an abstract syntax tree (AST). By contrast, type definitions in metamodels can be referenced directly via metaclass attributes, e.g., using the type property in UML.

As we highlighted in the previous section, a metamodel MM_{Domain} derived using our approach makes use of 'Abstract Concepts' and thus, we address the first problem category. However, production rules used as input to the derivation must be enriched with meta-information so that relationships between them and the 'Abstract Concepts' can be specified. During metamodel derivation, this information is processed to create *Generalizations* between generated metaclasses and the identified 'Abstract Concepts'.

In addition to the generalization of 'Abstract Concepts', it may be required that certain generated metaclasses can inherit from each other. We achieve this by defining another meta-information category, which makes it possible to refer to production rules instead of 'Abstract Concepts'. Then, we can introduce *Generalizations* for metaclasses generated from production rules that are provided with the above meta-information.

Design Decision 2 *Based on meta-information, the derivation approach shall support the creation of Generalizations either between generated metaclasses or between these ones and 'Abstract Concepts'.*

To implement the concept of abstraction, it is useful for generic metaclasses at the top of an inheritance hierarchy being defined as abstract, so they cannot be instantiated. Although the use of 'Abstract Concepts' implicitly supports abstraction, we consider it essential that metaclasses derived from production rules can explicitly be defined to be abstract.

Design Decision 3 *A meta-information type for production rules shall be introduced, which enables one to define a metaclass derived from a production rule as abstract.*

The generation of interlinked metaclasses, as discussed for the second category of problems, can be avoided when employing an appropriate resolution algorithm. When processing chained production rules, we resolve the linked production rules until we determine one that does not have an expression pointing to a non-terminal symbol. Then, we use the metaclass derived from the determined production rule as *type* of a metaclass attribute or association end.

Design Decision 4 *To avoid the creation of interlinked metaclasses, the derivation approach shall implement the above algorithm for determining the type of a metaclass attribute or association end.*

According to the third problem category, a metaclass attribute or association end generated for an expression in a production rule cannot be specified as a reference to a *Type*, because it is created as aggregation by default. To overcome this drawback, we introduce an appropriate meta-information so one can define that a metaclass attribute or association end shall be generated as a reference rather than an aggregation.

If we extend the meta-information so that the *Type* to be referenced can be specified explicitly, we also eliminate the fourth problem category regarding Qualifiers and Identifiers. Without this meta-information, an aggregation metaclass attribute or association end would be generated for an expression in a production rule, and its *type* property would refer to a metaclass that represents the Qualifier or Identifier. By contrast, if we provide the meta-information, the default generation shall be overridden so that an attribute or association end is created as reference to the specified *Type*.

Design Decision 5 *A particular type of meta-information shall be applicable to expressions of production rules so that a metaclass attribute or an association end can be created as reference to the specified Type.*

As we discussed for the third category of problems, EBNF grammars offer only a small number of language concepts. This applies in particular to expressions of production rules from which associations or attributes are derived, but without supporting the language concepts provided by CMOF such as subsetting, redefinition or derived union. We wish to provide the missing information by specifying certain meta-information for production rule expressions, so that the CMOF language concepts can automatically be applied to metaclass attributes and associations during metamodel derivation.

Design Decision 6 *Production rule expressions shall be annotatable with meta-information, so that the CMOF language concepts subsetting, redefinition, and derived union can automatically be applied to metaclass attributes and association ends.*

In addition to the above language concepts, the CMOF provides the ability to specify metaclass attributes and association ends as derived, so that their values are computed at runtime, e.g., based on OCL expressions. In this context, metaclass attributes or association ends are often defined as read-only, thus no values can be assigned manually at runtime. Both language concepts shall be applicable at the level of production rule expressions by means of meta-information.

Design Decision 7 *Production rule expressions shall be annotatable with meta-information so that metaclass attributes or association ends can be specified as derived and read-only.*

The CMOF supports the specification of *Constraints* on metaclasses which represent well-formedness rules of the static semantics. To ensure the traceability of the static semantics defined for a computer language, we wish to provide a possibility to capture the static semantics in a natural language at the level of production rules. Based on this information, *Constraints* can be created for the metaclasses derived from production rules.

Design Decision 8 *Constraints shall be created based on meta-information of production rules.*

The EMOF already offers the possibility to create *Comments* on all types of model elements. In our view, *Comments* on metaclasses, attributes, and associations can increase the comprehensibility of a metamodel. Therefore, we wish to provide a certain type of meta-information that can be used for this purpose.

Design Decision 9 *A certain type of meta-information shall be provided that can be used to create comments on production rules and expressions.*

4.2.3 Production Rules and Annotations

In the following, we introduce our notation for production rules, which is created based on our Design Decisions 1 – 9 postulated in the section above. This notation is implemented by our DSL-MeDeTo presented in the overview section. In addition to language concepts for specifying production rules, our notation provides annotations that enable the specification of the various types of meta-information as required by our design decisions.

The textual notation. Because our approach has initially been used to derive a metamodel for SDL [90], the notation supported by our DSL-MeDeTo is aligned to ITU-T Rec. Z.111 [67]. This recommendation defines language concepts for specifying the abstract syntax of computer languages such as SDL. Apart from

```
Grammar =
  Rule*
Rule =
  Definition−rule | Equivalence−rule
Definition−rule =
  Name "::" [ Annotations ] [Expression]
Equivalence−rule =
  Name "=" [Annotations] Expression
Expression =
  (Alternatives | Composition | Option | List | Set | Domain) [ Annotations ]
Alternatives =
  "(" Expression {"|" Expression }+ ")"
Composition =
  "{" Expression+ "}"
Option =
  "[" Expression "]"
List =
  Expression ( "*" | "+" )
Set =
  Expression ( "-set" | "+set" )
Domain =
  Non−terminal−domain | Elementary−domain | Enumeration−domain
Annotations =
  "annotations" "{" { Annotation ";" }+ "}"
Annotation =
  Documentation | RuleAnnotation | ExpAnnotation
RuleAnnotation =
  Constraint | AbstractClass | Generalization
ExpAnnotation =
  ReferencedRule | ReferencedType | CompositeType
  | Subsetted | Redefined | DerivedUnion | DerivedProperty | ReadOnly
```

Figure 4.5: Extract of the concrete syntax of the adapted production rule notation

small differences, these concepts are the same as those provided by other EBNF grammars. Thus, our notation is not only applicable for SDL, but can also be used for the specification of other computer languages or DSLs.

Two types of production rules are supported by our notation: a Definition-rule defines a particular non-terminal node, and an Equivalence-rule can be used to introduce an alias definition. If an Equivalence-rule is referenced on the right-hand side of another rule, this reference is replaced by the content of the Equivalence-rule. Apart from the mandatory Name, each Rule may consist of a particular Expression and a set of Annotations. Because the production rule expressions match those of other EBNF grammars, we not go into their details and refer to the previous section. The concrete syntax of the notation is summarized in Fig. 4.5, where keywords are marked by quotation marks.

Production rule annotations. The Annotation types provided by our notation are introduced based on Design Decisions 2 – 9. During the transformation of a Rule to a corresponding metaclass, its attached Annotations are evaluated to trigger various types of mapping operations. Thus, different kinds of relationships between metaclasses contained in MM_{Domain} can be established.

Except for the Documentation annotation, the defined annotation types can be divided into two groups. The first group consists of the set of RuleAnnotations that are only applicable to Rule nodes, whereas the second group comprises the set of ExpAnnotations that are employed for Expression nodes. In contrast, the Documentation annotation is applicable to Rules and Expressions. The purpose of the different Annotation types is as follows:

Documentation maps to a *Comment* that is owned by a *Class* or *Property* (Design Decision 9).

GeneralizedClass introduces a *Generalization* between either two generated *Classes* or a *Class* and an 'Abstract Concept' (Design Decision 2).

Constraint introduces a *Constraint* for a *Class* that is generated from an annotated Rule (Design Decision 8).

AbstractClass defines that a generated *Class* shall be abstract, i.e., the *isAbstract* property has the value 'true' (Design Decision 3).

DerivedUnion defines that a generated *Property* is a derived union, where value 'true' is assigned to the properties *derived* and *derivedUnion* (Design Decision 6).

RedefinedProperty introduces a redefinition relation of a generated *Property* to another *Property* of a super class (Design Decision 6).

SubsettedProperty specifies that a generated *Property* shall subset another *Property* (Design Decision 6).

DerivedProperty defines that the value of a generated *Property* is computed at runtime, so the *derived* property has value 'true' (Design Decision 7).

ReadOnly defines that a generated *Property* shall be read-only, i.e. the *isReadOnly* has value 'true' (Design Decision 7).

ReferencedType overrides the *type* of a generated *Property* and defines that it is a reference, so the *isComposite* property has value 'false' (Design Decision 5).

ReferencedRule has the same effect as ReferencedType, but this annotation type is used to refer to a specific Rule (Design Decision 5).

CompositeType overrides the *type* of a generated *Property* and specifies it to be a composite part of the owning metaclass (Design Decision 5).

Intermediate representation. Because our DSL-MeDeTo presented in the overview section above employs an M2M transformation to derive a metamodel MM_{Domain}, a model is required as input. Therefore, the abstract syntax tree of the parsed production rules is stored as an intermediate model-based representation. For this purpose, we define a metamodel MM_{Syntax}, which contains a corresponding metaclass for each symbol type of our textual notation. Thus, the transformation of parsed production rules into a model M_{Syntax} that conforms to metamodel MM_{Syntax}, is straightforward. Due to the one-to-one relation between our textual notation and the metamodel MM_{Syntax}, its details are not discussed here.

Using a model-based representation enables the extensibility of our toolchain so that users can employ another editor, or they can also implement a different notation, if required. For example, the *Text-to-Model (T2M)* transformation tool xText [159] could be used instead of the editor of our toolchain.

4.2.4 Running Example

We adopt a simple state machine DSL[3] as a running example[4] to illustrate and discuss our derivation approach. Because the original DSL has a textual notation, only production rules for the concrete syntax are available, but not for the abstract syntax. Terminal symbols representing keywords have been stripped from the original production rules, so we can specify the syntax shown in Fig. 4.6 using our notation for production rules.

The 'Abstract Concepts' employed by us for creating the running example consist of metaclasses contained in the 'StateMachine' package of the UML metamodel [116]. We consider the following metaclasses: *StateMachine*, *Event*, *Signal*, *State*, *Behaviour*, and *Transition*. The running example utilizes seven **generalized class** annotations to define an inheritance relationship to these 'Abstract Concept' metaclasses.

A 'Statemachine' of the example DSL consists of an 'Event' set, a 'ResetEvent' set, a 'Command' set, and the 'State' set. Many of the non-terminal symbols of the production rules are annotated with **derived property**, which means that their corresponding metaclass attributes in MM_{Domain} are also defined to be *derived*.

The difference between a **referenced type** and a **referenced rule** annotation becomes obvious based on the 'Transition' rule of our example. The **referenced type** annotation of the 'Event' expression points to the metaclass AC_Event of MM_{AC}, whereas the **referenced rule** annotation of the 'State' expression refers to a production rule. However, both annotation types are mapped in the same way to metaclass attributes; these are used as references to types (i.e., metaclasses or data types).

[3] https://www.eclipse.org/community/eclipse_newsletter/2014/august/article1.php
[4] Note that we employ this example only to illustrate the different derivations; it cannot be used to derive a syntactically complete metamodel.

```
Statemachine :: annotations {generalized class "AC_StateMachine";
  constraint name "Constraint_1": body "A StateMachine shall contain one Region." }
{
  Event* annotations { derived property }
  Reset−event* annotations { derived property }
  Command* annotations {
    subsetted property "AC_Class::nestedClassifier" }
  State* annotations { derived property }
}
Event :: annotations { generalized class "AC_Event" }
  { Name Code }
Reset−event :: annotations { generalized class "AC_Event" }
  { Name Code }
Command :: annotations { generalized class "AC_Signal" }
  { Name Code }
State :: annotations { generalized class "AC_State" }
{
  Name
  Action+ annotations { derived property; referenced rule Command }
  Transition*
}
Action :: annotations { generalized class "AC_Behavior" }
  Command+ annotations { referenced rule Command }
Transition :: annotations { generalized class "AC_Transition" }
{
  [Event] annotations { derived property; referenced type "AC_Event" }
  [State] annotations { derived property; referenced rule State }
}
Name = Token
Code = Nat
```

Figure 4.6: Syntax of a state machine DSL used for the running example

The **subsetted property** annotation attached to the 'Command' expression of the 'Statemachine' rule is significant for our approach, because this annotation type maps to a subsetting metaclass attribute; such attributes are treated in a particular way during the derivation of a UML profile, as presented in Sec. 4.3.

4.2.5 Transformation of Production Rules

We employ the M2M transformation $T_{SR\text{-}to\text{-}MM}$ of our toolchain to create a metamodel MM_{Domain}. The parsed production rules represented by the intermediate model M_{Syntax} and the 'Abstract Concepts' contained in the MM_{AC} metamodel serve as input for this transformation.

Overall, our metamodel derivation via transformation $T_{SR\text{-}to\text{-}MM}$ consists of four processing steps. In Step (1), the 'Abstract Concepts' metaclasses of metamodel MM_{AC} are copied to metamodel MM_{Domain}. Afterwards, the metaclasses of MM_{Domain} are generated in Step (2), whereas production rule annotations are pro-

```
Rule ⇒ toAbstractClass() : Class
when { self.expression.isType(Alternative) }
  result.name := self.name;
  result.isAbstract := true;

Rule ⇒ toNonAbstractClass() : Class
when { !self.expression.isType(Alternative) ∧ !self.expression.isKind(Domain) }
  result.name := self.name;
  result.isAbstract := false;
  // Referring to abstract metaclasses
  var superTypes := input.objectsOfKind(Alternative)
    −>select(alternatives.Name = self.Name).owningRule;
  superTypes−>forEach(st) {
    result.superClass += st.map toAbstractClass()
  };
  // Mapping of expressions to attributes
  if self.expression.isType(Composite) then
    self.expression.asType(Composite).expression−>forEach(exp){
      result.ownedAttribute += exp.map toAttribute()
    }
  else
    result.expression := self.expression.map toAttribute()
  endif;

Rule ⇒ toEnumeration() : Enumeration
when { self.expression.isType(EnumerationDomain)) }
  result.name := self.name;
  self.ownedNodes.asType(EnumerationDomain).quotations−>forEach(quot) {
      result.ownedLiteral += object EnumerationLiteral{ name := quot }
  }
```

Listing 1: Mapping operations toAbstractClass(), toNonAbstractClass() and toEnumeration()

cessed in Step (3) in order to modify the previously created metaclasses depending on the respective annotation type. Finally, *Associations* for all metaclass attributes that refer to metaclasses are created in Step (4).

Step (2) is comparable to existing approaches [6, 44, 94, 153], where EMOF-based metamodels are generated based on production rules (see Sec. 4.2.2). In contrast, Step (3) is novel as it enables the application of language concepts provided by the CMOF; hence, we can derive a CMOF-compliant metamodel.

We now explain the detailed mappings as pseudocode employing the notation defined in Sec. 2.4.3.

1. Copying the 'Abstract Concepts'. In the first step, all metaclasses and data types contained in MM_{AC} are copied to MM_{Domain}. In addition, their names are prefixed with 'AC_' to avoid name clashes and distinguish between metaclasses of 'Abstract Concepts' and generated metaclasses. Finally, all copied metaclasses are

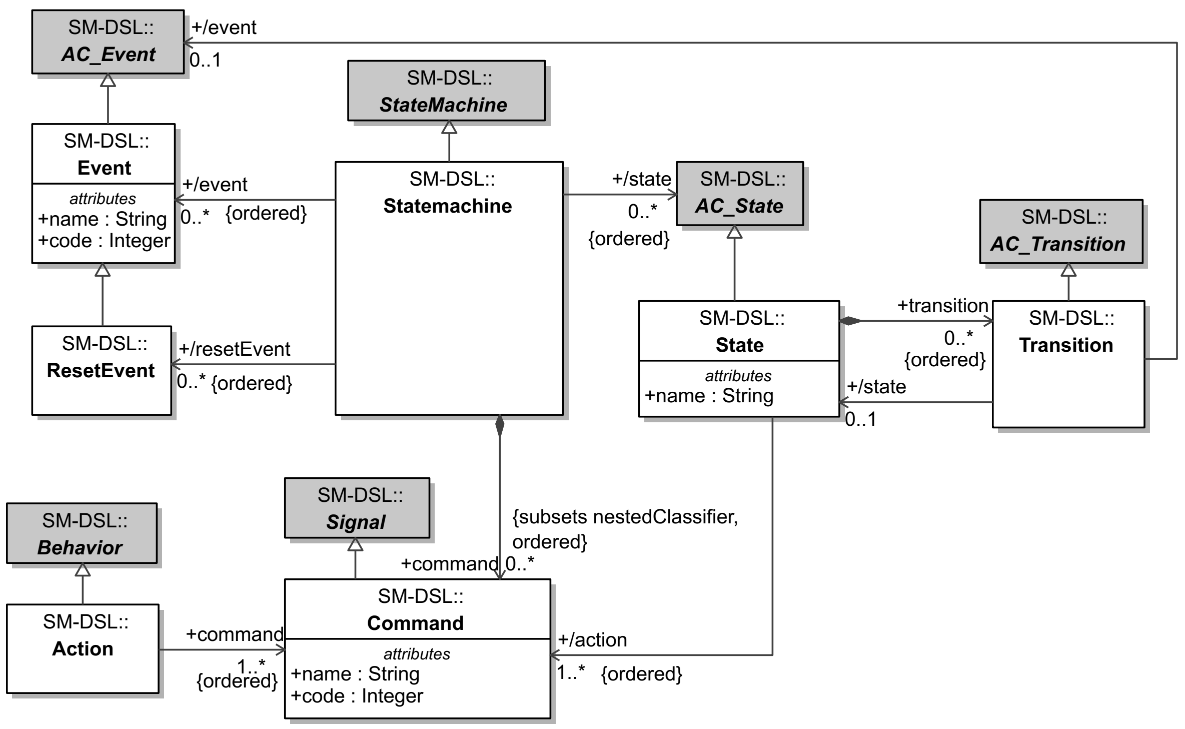


Figure 4.7: Metamodel MM_{Domain} that is derived from the production rules in Fig. 4.6

defined to be abstract, because 'Abstract Concepts' shall not be instantiable; otherwise, this would lead to syntactically incorrect models.

2. Mapping the production rules. The second step consists of the mapping of production rules to corresponding metaclasses of MM_{Domain}. According to Design Decision 1, we adopt the mappings proposed in the literature for creating EMOF metamodels based on production rules. Therefore, all mappings below implement the rules summarized in the context of Design Decision 1.

Mapping of Rules: In general, we map a Rule to a metaclass (represented as *Class*) or *Enumeration* in the metamodel MM_{Domain}, which depends on the type of an Expression in the Rule. To capture all mapping variants for a Rule, we defined the three mapping operations shown in Listing 1.

The toAbstractClass() mapping operation maps a Rule that has an Alternative expression to an abstract *Class*. One might assume that a corresponding metaclass attribute is introduced for each item of the alternative expression. However, according to the rules summarized in the context of Design Decision 1, we have to create inheritance relationships to metaclasses introduced for the production rules referenced in an alternative expression. These relationships are introduced by the mapping operation below.

A Rule with an Expression that is not an Alternative or a Domain is mapped to a non-abstract *Class*, as implemented by the toNonAbstractClass() mapping operation in Listing 1. This operation determines the *ownedAttributes* of a created *Class* from the Expression of a Rule.

A Composite expression maps to one or more *ownedAttributes,* while all other expression types are mapped to exactly one *ownedAttribute* (Lines 17 – 23 in Listing 1). Moreover, if the current input Rule is referred as member of an Alternative expression in another Rule, we map the latter to a *superClass* of the created *Class* (see Lines 11 to 14 of Listing 1). This establish the inheritance relationships to abstract metaclasses created by the toAbstractClass() mapping operation.

In contrast to the previous mappings, a Rule with an EnumerationDomain is mapped to an *Enumeration* in metamodel MM_{Domain}, as realized by mapping operation toEnumeration() in Listing 1. This operation maps each item of an EnumerationDomain to a corresponding *EnumerationLiteral* of the *Enumeration* to be created.

In the following, we explain the mapping of the different Rule types using our running example shown in Fig. 4.6. The Statemachine rule is mapped to a non-abstract metaclass by applying the mapping operation toNonAbstractClass(), and the *name* of this metaclass is equal to that of the production rule (see Fig. 4.7).

By contrast, the Name rule that has a Domain expression is not mapped by this operation; such a construct can be considered as a reference to another type. We discuss this topic more closely in the context of the mapping of an Expression. The advantage of this is that the number of metaclasses generated for MM_{Domain} is reduced.

Mapping of Expressions: In general, we map the Expression of a Rule to an *ownedAttribute* (i.e., a *Property*) of a metaclass according to the rules discussed in the context of Design Decision 1. Because the particular mapping depends on the type of an Expression, we introduce a specific mapping operation for each Expression type so that the *name,* the *type* and the cardinality[5] of a *Property* are mapped in a specific manner. Overall, we specify six out-place mapping operations shown in Lines 1 – 45 of Listing 2.

According to the general rules for deriving EMOF metamodels (see Design Decision 1), *Associations* shall be introduced for non-terminal symbols of production rules. To enable a better comprehensibility of our metamodel derivation, we create these *Associations* separately in Step (4), instead of introducing them in the context of the mappings below.

As we pointed out in Sec. 4.2.2, an element to be mapped can in certain cases not be derived from a source element; thus, a mapping operation is not applicable in such situations. However, we can determine the required data from the source or target model employing the following query and helper operations shown in Lines 46 to 66 of Listing 2:

[5] The cardinality of a *Property* is defined by its *lower* and *upper* properties (see Sec. 2.2.2).

```
NonTerminalDomain ⇒ toAttribute() : Property
  result.name := self.getAttributeName();
  result.aggregation := AggregationKind::composite;
  result.type := getTypeForName( self.getAttributeName() );

Option ⇒ toAttribute() : Property
  when { self.expression.isType(EnumerationDomain)
    ∧ self.expression.asType(EnumerationDomain).quotations−>size() = 1
  }
  var aName := self.expression.asType(EnumerationDomain).quotations−>first();
  aName := "is" + aName.toLower().firstToUpper();
  result.name := aName;
  result.type := getUmlPrimitiveType("Boolean");
  result.lower := 1;
  result.upper := 1;

Option ⇒ toAttribute() : Property
  result.name := self.getAttributeName();
  result.type := getTypeForName( self.getAttributeName() );
  result.lower := 0;
  result.upper := 1;
  result.aggregation := AggregationKind::composite;

List ⇒ toAttribute() : Property
  result.name := self.getAttributeName();
  result.type := getTypeForName( self.getAttributeName() );
  result.lower := self.min;
  result.upper := self.max;
  result.isOrdered := true;
  result.isUnique := true;
  result.aggregation := AggregationKind::composite;

Set ⇒ toAttribute() : Property
  result.name := self.getAttributeName();
  result.type := getTypeForName( self.getAttributeName() );
  result.lower := self.min;
  result.upper := self.max;
  result.isOrdered := false;
  isUnique := true;
  result.aggregation := AggregationKind::composite;

ElementaryDomain ⇒ toAttribute() : Property
  result.name := self.getAttributeName();
  result.type := self.getUmlType();
  result.aggregation := AggregationKind::composite;
```

Listing 2: Mapping operations that map *Expression* elements to metaclass attributes

```
query SR::Expression getAttributeName() : String
  if self.isType(NonTerminalDomain) then
    return self.asType(NonTerminalDomain).Name
  else if self.isType(ElementaryDomain) then
    return self.asType(ElementaryDomain).kind.repr()
  else
    return self.ownedExpressions.asType(NonTerminalDomain)−>any(true).Name
endif endif;

helper getTypeForName( ruleName : String ) : UML::Type
  var rule := input.objectsOfKind(Rule)−>any(name = ruleName);
  if rule.ownedNodes−>one(isType(NonTerminalDomain)) then
  {
    var refRuleName := rule.ownedNodes−>any(true).Name;
    return getTypeForName(refRuleName);
  }
  else if rule.ownedNodes−>one(isType(ElementaryDomain)) then
    return rule.ownedNodes−>any(true).getUmlType();
  else
    return output.objectsOfKind(Classifier)−>any(name = ruleName);
  endif endif;

Class ⥫ annotationsToGeneralization()
when { self.resolve(SR::Rule).annotations
    −>select(isType(SR::GeneralizedClass))−>notEmpty()
  }
  var srcRule := self.resolve(SR::Rule);
  var annon := srcRule.annotations.asType(SR::GeneralizedClass);
  annon−>forEach(ga)
  {
    var resType := self.getTypeByQName(ga._class);
    self.general += resType.asType(UML::Classifier)
  };

Property ⥫ annotationsToSubsettedProperty()
when { self.resolve(SR::Expression).annotations
    −>select(isType(SR::SubsettedProperty))−>notEmpty()
  }
  var srcExp := self.resolve(SR::Expression);
  var annon := srcExp.annotations.asType(SubsettedProperty);
  annon−>forEach(sp)
  {
    var resProp := self.getPropertyByQName(rp._property);
    self.subsettedProperty += resProp
  };
```

Listing 2: Mapping operations that map *Expression* elements to metaclass attributes (continued)

- The helper function getUmlPrimitiveType() is used to obtain one of the five predefined *PrimitiveTypes* of UML. Due to the simplicity of this helper function, we omit its details.

- The getAttributeName() query in Listing 2 determines the name of a *Property*. Provided that the context element of the query is not an ElementaryDomain, we directly or indirectly derive the name from a NonTerminalDomain expression.

- The helper function getTypeForName() resolves a *Type* in the target model based on the name of a Rule. Furthermore, this function implements our algorithm for eliminating interlinked metaclasses, as we have discussed in the context of Design Decision 4. The function is called recursively (see Line 60) for a a Rule with NonTerminalDomain expression, until a Rule with another expression type is present. Then, we resolve the *Type* based on the recursively determined Rule.

3. Processing the annotations. The various meta-information types introduced by our Design Decisions 2 – 9 are captured by different annotation types that are processed in Step (3) of transformation $T_{SR\text{-}to\text{-}MM}$. For this purpose, we define 12 in-place mapping operations that are applied to either *Class* or *Property* elements of MM_{Domain}, which are created in Steps (1) and (2). Because we apply these mapping operations in-place, the elements concerned are updated but not recreated. Due to the high number of mapping operations and their similar structure, we only treat two of them here:

1. We use the mapping operation annotationToGeneralization() shown in Listing 2 to establish a *Generalization* relationship between the current context *Class* and that identified by the GeneralizedClass annotation, as required by Design Decision 2. To ensure that the mapping operation can only be invoked for an element that has applied this annotation type, we introduce an appropriate guard condition in Line 70.

   Because the mapping is applied in-place to an already generated *Class* in MM_{Domain}, we cannot directly determine the annotation type that is applied to the source Rule. Therefore, we resolve this Rule in M_{Syntax} based on the current context *Class* of the mapping operation (see Lines 70 and 72). Then, the guard condition can check whether the resolved Rule has the GeneralizedClass annotation applied.

   If the guard condition is successfully evaluated, the *Class* to be generalized is determined in MM_{Domain} by employing the getTypeByQName() query. Finally, this *Class* is assigned to the *general* property of the current context *Class* that is represented by the **self** variable.

For example, the *Generalization* between the Statemachine and the AC_StateMachine *Classes* shown in Fig. 4.7 originates by a GeneralizedClass annotation that is processed by the mapping operation discussed above.

2. The annotationsToSubsettedProperty() mapping operation in Listing 2 is used to process the SubsettedProperty annotations of an Expression. Its structure is similar to the mapping operation discussed above. Apart from the guard condition and the processed annotation type, the main difference is the assignment to the *subsettedProperty* of the current context *Property*.

   As shown in Fig. 4.7, the command attribute of the Statemachine *Class* subsets the nestedClassifier attribute Event, which is the result of the annotationsToSubsettedProperty() operation.

4. Introduction of 'Associations'. In class diagrams used to design MOF-compliant metamodels, relationships between metaclasses must be specified by employing binary *Associations*, which must respect the MOF-specific constraints (see Sec. 2.2.2). This particular *Association* type has the same semantic meaning as specifying an *ownedAttribute* for a *Class*. This is because the creation of an *Association* also induces the implicit creation of *Property* items for each association end.

Because we do not introduce *Associations* in transformation Steps (1) – (3), we create them in Step (4) of transformation $T_{SR\text{-}to\text{-}MM}$. We define an in-place mapping operation that introduces unidirectional binary *Associations* for those metaclass attributes in MM_{Domain} that have a *type* referring to a *Class*. However, we do not create *Associations* to *DataType* because this is not permitted due to the constraints defined by the MOF. For example, all *Associations* shown in Fig. 4.7 are introduced by the above mapping operation.

Manual refinements of MM_{Domain}. Although we can automatically derive many aspects of the metamodel MM_{Domain} by employing the mappings discussed above, this does not apply for `Constraint` annotations attached to production rules (e.g., see Constraint_1 in Fig. 4.6). Because constraints that define the static semantics of a computer language are often expressed in natural language, we only support this kind of specification for our textual notation. Hence, the textual description of a Constraint annotation is copied to a *Constraint* element of a metaclass in MM_{Domain}, but the required OCL specification has to be implemented manually.

Another refinement concerns the *opposite* feature of an *ownedAttribute* that is derived from the *opposite* end of an *Association*. If the *ownedAttribute* redefines or subsets another *Property*, then the *opposite* feature also has to redefine or subset a *Property* in an appropriate manner. Although this kind of refinement could be implemented as annotation for our textual notation, it is a design decision that a manual refinement shall be done in MM_{Domain} using a modelling tool. We opted for this because the MOF and UML define several complex constraints on the redefinition and subsetting of *Associations*. The implementation of these constraints in the textual editor of our DSL-MeDeTo would only be feasible with greater effort.

4.3 Approach for Deriving UML Profiles

As we claimed in the introduction to the present chapter, we can automatically derive a UML profile based on a metamodel for a DSL in Step (B) of our approach. In the following, we discuss the details of this derivation step.

First, we introduce our process that captures all actions and artefacts involved in deriving UML profiles. Then, we discuss the design decisions of our derivation approach. Based on this information we then treat the prerequisites for the usability of a metamodel for our UML profile derivation, and we also consider the enrichment of a metamodel with information that is required for the derivation. Thereafter, we present the details of our derivation approach.

4.3.1 Process for Deriving UML Profiles and Additional Metaclasses

The automated derivation of UML profiles based on metamodels for DSLs represents a new activity relative to the process proposed in [145], because that process is only intended for developing metamodels. In addition to a UML profile, additional metaclasses can be derived as part of the activity, if required. The process steps are defined as actions of the activity shown in Fig. 4.8: an existing metamodel MM_{Domain} is passed as input to this sub-activity, and a derived UML profile UP_{Domain}, a metamodel MM_{Add}, and an enriched metamodel MM_{Domain} are the output.

The activity starts with Action (1), where the metamodel MM_{Domain} is enriched with the meta-information required for our derivation approach. For this purpose, stereotypes of the UML profile 'MM2Profile' have to be applied to metaclasses of MM_{Domain}. Before the automatic derivation of the UML profile UP_{Domain} and the additional metaclasses for metamodel MM_{Add} can be started, the OCL-defined static semantics of the enriched metamodel MM_{Domain} must be updated in Action (2). This update is because of the separate instantiation of stereotypes and UML metaclasses as explained in Chap. 2.

The updated metamodel MM_{Domain} obtained from Action (2) serves as input for Actions (3) and (4). Provided that additional metaclasses shall be created, Action (3) has to be conducted before the UML profile UP_{Domain} is derived in Action (4). Action (3) corresponds to Step (C), and Action (4) implements Step (B) of our derivation approach. The derivation of MM_{Add} before UP_{Domain} is essential because additional metaclasses in MM_{Add} may be referenced by stereotypes of UP_{Domain}, and therefore, they must be present before UP_{Domain} is created. The

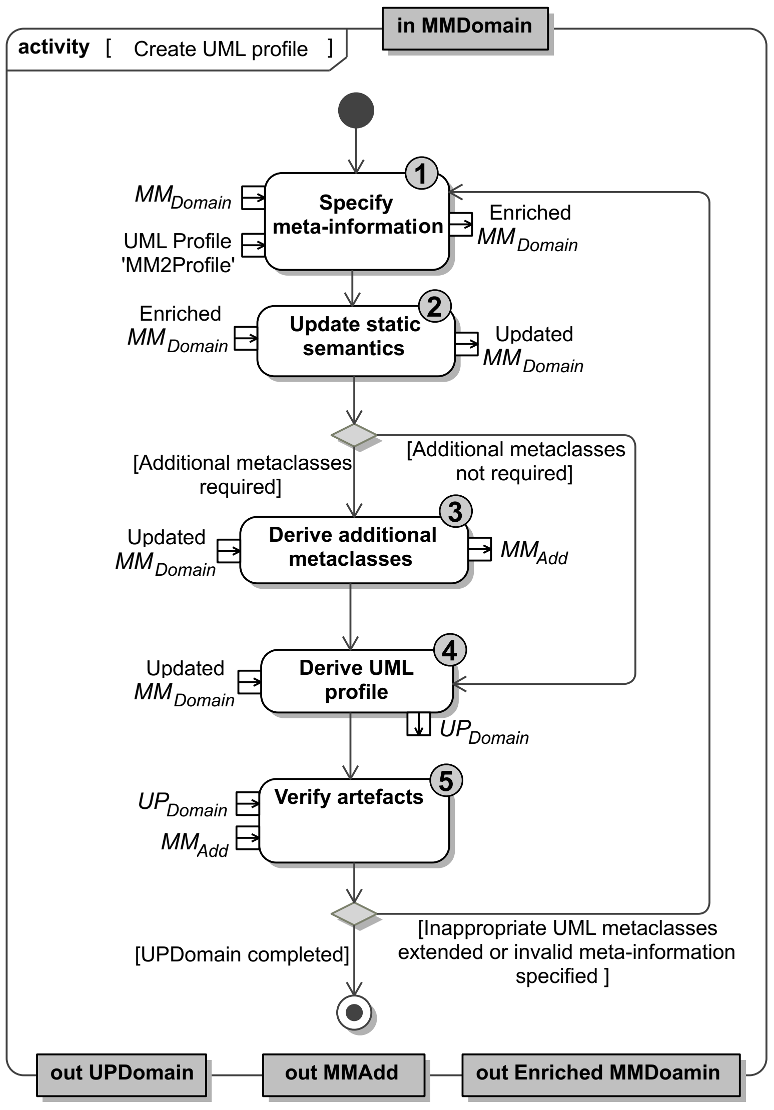


Figure 4.8: Activity for deriving UML profiles and addtional metaclasses

details for the derivation of a UML profile UP_{Domain} are discussed in this section, whereas the derivation of additional metaclasses in MM_{Add} is covered in Sec. 4.4.

Finally, the syntax and static semantics of UP_{Domain} and MM_{Add} are verified in Action (5). For this purpose, we employ the approach that we have successfully applied for verifying the UML profile for SDL.

4.3.2 Design Decisions for the Profile Derivation

The limitations of UML profiles as highlighted in Chap. 2 have significantly influenced the design of our derivation approach for UML profiles based on metamodels.

Because our objective is to derive a UML profile and optionally 'additional' metaclasses, the metaclasses contained in the metamodel MM_{Domain} have to be processed in different ways. In addition, we have to consider that MM_{Domain} also

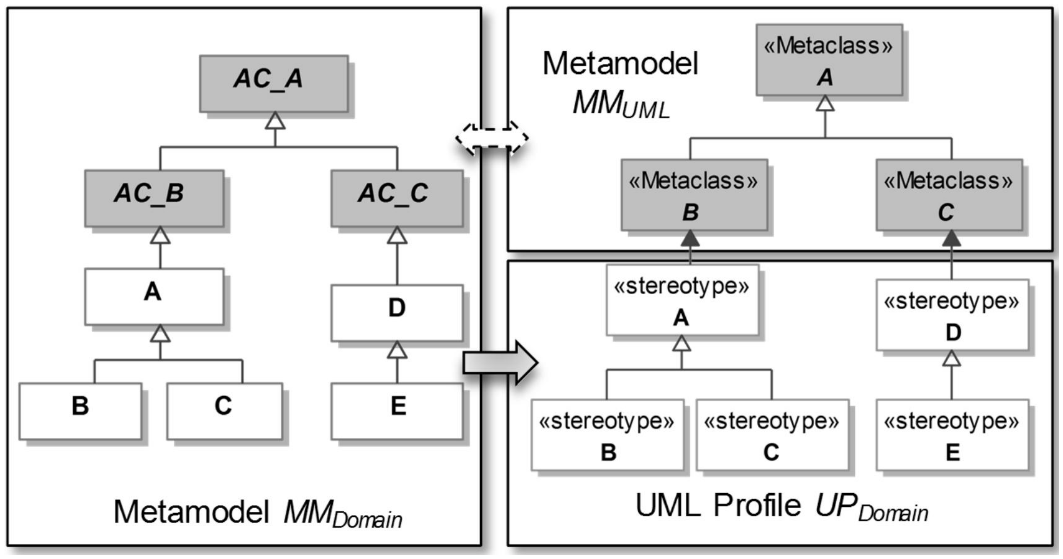


Figure 4.9: Default mapping of a metamodel to a corresponding UML profile

contains metaclasses that represent 'Abstract Concepts'. Hence, we presume that the metaclasses contained in MM_{Domain} can be divided into three different sets of metaclasses. The first set represents 'Abstract Concepts'; a metaclass of this set is denoted as MC_{AC}. The second set embraces those metaclasses that shall be mapped to *Stereotypes* of the derived UML profile UP_{Domain}. We use the abbreviation MC_{St} to refer to a metaclass of this set. If 'additional' metaclasses for a metamodel MM_{Add} shall be derived, the third set contains metaclasses that shall be mapped to metaclasses of MM_{Add}. A metaclass of this set is denoted as MC_{AMC}.

Design Decision 10 *A metamodel MM_{Domain} consists of a set of MC_{AC} metaclasses, a set of MC_{St} metaclasses, and an optional set of MC_{AMC} metaclasses.*

As argued in Sec. 4.1.2, each 'Abstract Concept' metaclass has a 'matching' UML metaclass in MM_{UML}, and their names are prefixed with 'AC_' to avoid name clashes with those metaclasses generated based on syntax rules. Hence, we do not to have map MC_{AC} metaclasses contained in the metamodel MM_{Domain}. In addition, we can employ the name prefix to identify that a metaclass of MM_{Domain} is an MC_{AC} metaclass.

Design Decision 11 *An 'Abstract Concept' metaclass MC_{AC} is identified by a name prefix* 'AC_'*, and it is not mapped to any kind of element.*

Hence, all metaclasses without a name prefix are MC_{St} or MC_{AMC} metaclasses. In order to make a clear distinction between these two metaclass types possible, an additional qualifier is required. As the derivation of 'additional' metaclasses is only optional, we consider it sufficient to use such a qualifier only for MC_{AMC} metaclasses. For this reason, we define metaclasses that have no special qualifier or name prefix as MC_{St} metaclasses. By default, we use this type of metaclasses of MM_{Domain} to derive *Stereotypes* for a UML profile, as shown in Fig. 4.9.

Design Decision 12 *A metaclass of MM_{Domain} without a qualifier is assumed to be an MC_{St} metaclass, whereas a metaclass with existing qualifier represents an MC_{AMC} metaclass.*

As we emphasized in Sec. 2.5.2, an extension of the UML using UML profiles is incomparable to a direct extension of its metamodel. Therefore, the following two constraints are defined in the UML specification, which prevent the generalization of UML metaclasses by elements in a UML profile:

> *"An element imported as a metaclassReference is not specialized or generalized in a Profile."* [116, Sec. 12.4.7.5]

> *"A Stereotype may only generalize or specialize another Stereotype."* [116, Sec. 12.4.9.6]

Hence, a *Stereotype* is not permitted to generalize a UML metaclass. Instead, an *Extension* association has to be used for extending a UML metaclass by a *Stereotype*. Consequently, we introduce an *Extension* for each *Stereotype*, which is derived from an MC_{St} metaclass that inherits directly from an MC_{AC} metaclass in MM_{Domain}. Because *Stereotypes* defined as 'required' can cause syntactic problems as argued in Sec. 2.5.2, we consider this feature to be inapplicable to our derivation approach. Thus, we do not label the *Extensions* for *Stereotypes* as 'required'.

For example, according to the above rule, an *Extension* is introduced only for *Stereotypes* derived from the metaclasses A and D in Fig. 4.9.

Design Decision 13 *An Extension shall be introduced for each Stereotype, which is derived from an MC_{St} metaclass that directly inherits from an MC_{AC} metaclass.*

In contrast to the previous case, a *Stereotype* can inherit from another *Stereotype* without restrictions. Hence, we can introduce a *Generalization* between two *Stereotypes* that are derived from two MC_{St} metaclasses that are in a *Generalization* relationship (e.g., metaclasses `E` and `D` in Fig. 4.9).

Design Decision 14 *A Generalization is introduced for each Stereotype that is derived from an MC_{St} metaclass that inherits directly from another MC_{St} metaclass.*

As mentioned in the introduction, one of our key objectives for the derivation of a UML profile is the preservation of the syntactic structure defined by a metamodel. This is a prerequisite for transferring the OCL-defined static semantics of the metamodel to a UML profile. Therefore, we map each *Property* that is an *ownedAttribute* of an MC_{St} metaclass to a corresponding *Property* of a *Stereotype*. However, we have to obey the following rule of the UML specification:

> *"The type of a composite aggregation Stereotype Property cannot be a Stereotype (since Stereotypes are owned by their Extensions) or a metaclass (since instances of metaclasses are owned by other instances of metaclasses)."* [116, Sec. 12.3.3.4]

Therefore, we map an MC_{St} attribute whose *type* property refers to a metaclass, to a stereotype attribute whose *aggregation* property has value 'none'. Consequently, this kind of stereotype attribute represents a reference to a metaclass instance.

Design Decision 15 *Each Property that is an ownedAttribute of an MC_{St} metaclass is mapped to a corresponding ownedAttribute of a Stereotype. In addition, if the type property of an ownedAttribute of an MC_{St} refers to a metaclass, the aggregation property of the mapped stereotype attribute shall have value 'none'.*

If an *ownedAttribute* of an MC_{AC} metaclass is redefined or subsetted by an *ownedAttribute* of an MC_{St} metaclass, this kind of relationship cannot be preserved for a derived UML profile. This is caused by the two UML constraints above, and the fact that redefinition and subsetting can only be employed for *Classes* that are in a direct or indirect inheritance relationship.

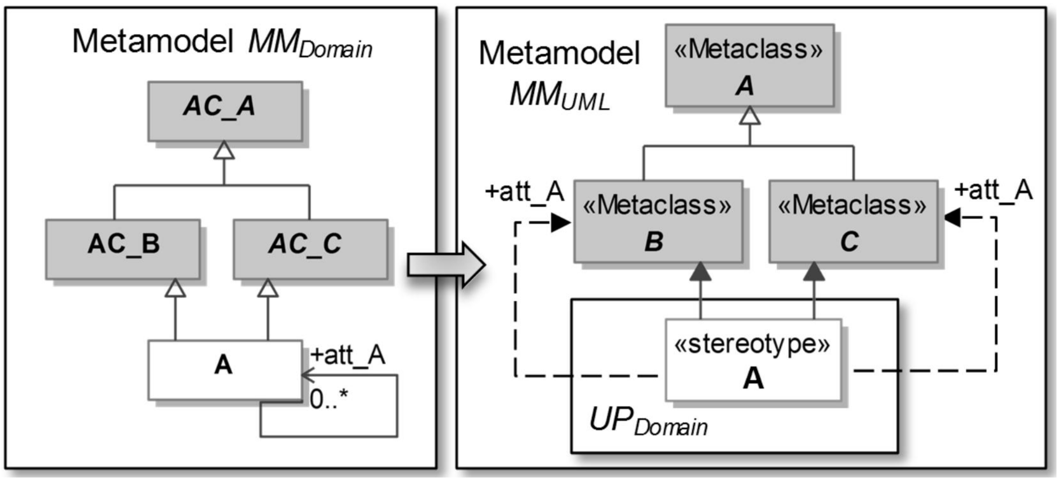
In general, this kind of property could be mapped to stereotype attributes in two different ways. The first possibility is to map such an attribute one-to-one to a stereotype attribute, where existing redefinition/subsetting relationships are removed. However, the drawback of this approach is that values for stereotype attributes have to be assigned manually. The second option is to map a metaclass attribute to a 'derived' stereotype attribute whose value is computed automatically based on an OCL expression at runtime and, thus, no manual value assignment is required. Because of this advantage, we choose the second option for our UML profile derivation.

Design Decision 16 *An ownedAttribute of an MC_{St} metaclass that redefines or subsets an ownedAttribute of an MC_{AC} metaclass shall be mapped to a derived and read-only stereotype attribute, and an OCL expression is introduced as its defaultValue.*

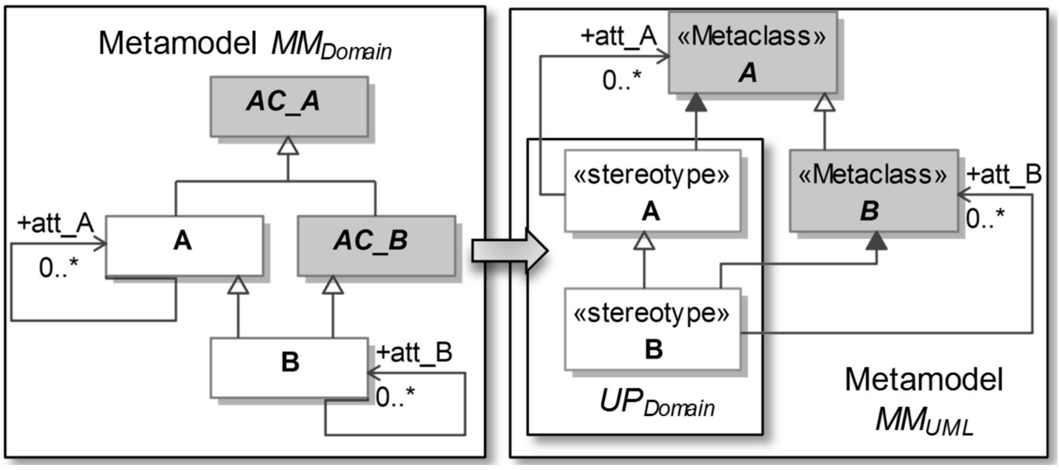
When creating a UML model, a model element usually has to be defined in a first step, and then a specific stereotype can be applied to it in a second step. While the model element is always located at a certain position in the UML model, the instance of its applied stereotype is not directly contained in the model. However, the UML model and associated stereotype instances are contained in the same resource or container (see Sec. 2.5.2).

This fact could become a problem when the *type* of an *ownedAttribute* of a *Stereotype* refers to another *Stereotype*. In this case, the designer of a UML model has to identify a valid *Stereotype* instance that shall be assigned as value for a *Stereotype* attribute, which is often non-trivial because *Stereotype* instances usually have no unique identification feature.

Consequently, we do not use a *Stereotype* as the *type* of a stereotype attribute; instead, we refer to a UML metaclass that is extended by a *Stereotype*, or to an 'additional' metaclass contained in the MM_{Add} metamodel.



(a) An MC_{St} metaclass that inherits from two MC_{AC} metaclasses



(b) An MC_{St} metaclass that inherits from an MC_{St} and an MC_{AC} metaclass (Case B)

Figure 4.10: Multiple inheritance and the *type* of a stereotype attribute

Design Decision 17 *The type property of an ownedAttribute of a Stereotype shall not refer to a Stereotype. Instead, the type shall either refer to a UML metaclass that is extended by a mapped Stereotype, or to an 'additional' metaclass contained in MM_{Add}.*

The recomputation of the *type* property of stereotype attributes as discussed before also affects the modelling of inheritance relationships between MC_{St} and MC_{AC} metaclasses in MM_{Domain}. As long as an MC_{St} metaclass only inherits from a single MC_{AC} metaclass, there are no restrictions on the modelling of MM_{Domain}. But this is not the case for multiple inheritance.

Assume that an MC_{St} metaclass A that inherits from at least two MC_{AC} metaclasses is used as the *type* of a metaclass attribute att_A, and the MC_{St} is mapped to a *Stereotype* A, as shown in Fig. 4.10a. According to Design Decision 13, this stereotype would then have two *Extension* associations to different UML metaclasses. Due to Design Decision 17, two possibilities would exist as *type* of the mapped attribute att_A.

To avoid this ambiguity, it must be ensured that a derived *Stereotype* has only an *Extension* to a single UML metaclass. One possible solution would be that an MC_{St} metaclass of MM_{Domain} inherits from at most one MC_{AC} metaclass, so that only one *Extension* would be derived for a *Stereotype*. However, this approach would require a modification of an existing metamodel MM_{Domain}, or its modelling would become too restrictive. Therefore, we consider this approach to be inappropriate.

Another solution would be to introduce a specific type of meta-data that can be attached to MC_{St} metaclasses, so that the UML metaclass to be extended by a *Stereotype* can explicitly be defined. Because this approach does not require a modification of MM_{Domain}, we prefer this solution to avoiding ambiguities for the *type* property of stereotype attributes.

For the sake of completeness, it should be mentioned that an MC_{St} metaclass can also inherit from an MC_{AC} and one or more MC_{St} metaclasses, as shown in Fig. 4.10b. In contrast to the previous scenario, no ambiguities for the determination of the *type* property of a stereotype attribute exist here, because a derived *Stereotype* has only a single *Extension* to a UML metaclass. Due to Design Decision 17 *Generalization* relationships to other *Stereotypes* do not affect type determination.

Design Decision 18 *A particular type of meta-data shall be applicable to an MC_{St} metaclass so that the UML metaclass to be extended by a Stereotype can explicitly be defined.*

According to the UML [116], a metaclass attribute can redefine and subset other attributes at the same time. In addition, a metaclass attribute cannot only redefine or subset a single attribute, but also several attributes. We have to consider these scenarios when introducing OCL expressions that are employed as substitute for redefining or subsetting attributes (see Design Decision 16).

Because subsetting or redefinition of several attributes usually occur only in the case of metaclasses with multiple inheritance, we must pay particular attention to Design Decision 18, which requires that a derived stereotype extends only a single UML metaclass. Thus, only attributes of this UML metaclass can be invoked in generated OCL expressions for stereotype attributes. Hence, in the case of multiple inheritance, we must either be able to specify which UML metaclass attribute should be invoked in OCL expressions, or there must be a possibility to specify OCL expressions manually.

To decide which of both solutions shall be implemented by our derivation approach, we have analysed the UML metamodel concerning the utilization of 'subsetting' and 'derivation'. The results are summarized in Table 4.1. Based on these we suppose that the 'subsetting' of a single attribute (75.7%) is much more common than that of multiple attributes (13.7%). A similar situation exists for 'redefinition', whereas a combined use of 'redefinition' and 'subsetting' (1%) can be considered as a rarely used special case.

Table 4.1: *Properties* that redefine and/or subsets other properties of UML metaclasses (analysis based on the UML metamodel [115])

Number of 'Properties'		**Number of occurrence in MM_{UML}**
subsetted	redefined	
1	0	**437** (75,7%)
≥ 2	0	**79** (13,7%)
0	1	**53** (9,1%)
0	≥ 2	**3** (0,5%)
≥ 1	≥ 1	**6** (1,0%)

In summary, we conclude that redefinition or subsetting of a single attribute is the default. Hence, we automatically introduce OCL expressions only for such stereotype attributes derived from MC_{St} metaclass attributes, which only 'redefine' or 'subset' a single attribute. In all other cases, we prefer a manual specification of OCL expressions.

In addition, it may be useful to manually introduce OCL expressions for stereotype attributes, so the values of these attributes can be computed at runtime based on attributes of other model elements. Therefore, a possibility should exist to specify an OCL expression as a specific type of meta-data for an *ownedAttribute* of an MC_{St} metaclass.

Design Decision 19 *An ownedAttribute of an MC_{St} metaclass can have an alternative OCL expression, which is used as defaultValue of a corresponding ownedAttribute of a derived Stereotype.*

Due to Design Decision 17, we have to recompute the *type* property of a *Stereotype* attribute so that it refers to a UML metaclass. A disadvantage is that syntactically invalid values can also be assigned to such kind of attribute. For example, a value assigned to a stereotype attribute may have the correct UML type, but it may also have applied an invalid stereotype. Therefore, we introduce OCL constraints to ensure that only those UML elements that have applied a particular stereotype can be assigned to such stereotype attributes. However, we do not generate OCL constraints for 'derived' and 'read-only' stereotype attributes, because no values can be assigned to them manually (see Design Decision 16).

Design Decision 20 *An OCL Constraint shall be created for each Stereotype attribute that is not defined as 'derived' and 'read-only', and that is mapped from an MC_{St} attribute with a type property that refers to an MC_{St} metaclass. The Constraint shall ensure that only UML elements having applied a certain Stereotype can be assigned to a stereotype attribute.*

Even though we do not introduce OCL constraints for 'derived' and 'read-only' stereotype attributes, this does not apply for those UML metaclass attributes that are employed as computational basis for the *defaultValue* of stereotype attributes (see Design Decision 16). Because of the same reason as above, only UML elements having applied a particular stereotype shall be assignable to these UML metaclass attributes. Hence, we introduce appropriate OCL constraints. Based on the redefinition and subsetting relationships of MC_{St} attributes, we identify the UML metaclass attributes that shall be constrained.

Design Decision 21 *An OCL Constraint shall be created for each UML metaclass attribute that is required as computation base for the value of a 'derived' and 'read-only' stereotype attribute. The Constraint shall ensure that only UML elements having applied a particular Stereotype can be assigned to this UML metaclass attribute.*

4.3.3 Assumption on and Enrichment of the Source Metamodel

Provided that a metamodel MM_{Domain} is generated based on the production rules as discussed in Sec. 4.2, this metamodel can directly be used as input for our derivation of a UML profile. By definition, we consider that such a metamodel contains a set of MC_{AC} metaclasses and a set of MC_{St} metaclasses (see Design Decisions 11 and 12). In this case, the UML metaclass to be extended by a derived *Stereotype* can be determined based on the inheritance relationships of the source MC_{St} metaclass in MM_{Domain}.

However, the direct use of MM_{Domain} to derive a UML profile is not always applicable. For example, it cannot be defined that instead of the 'matching' UML metaclass, one of its subtypes shall be extended by a derived *Stereotype*. Furthermore, without additional information, it cannot be specified that a metaclass of MM_{Domain} should be mapped to an 'additional' metaclass of MM_{Add} (see Design Decision 12). Therefore, the stereotypes of our UML profile 'MM2Profile' presented below must be applied to a metaclasses of the metamodel MM_{Domain} before this metamodel can be employed as derivation source.

The UML profile 'MM2Profile'. As shown in Fig. 4.11, this profile consists of five stereotypes that extends four different UML metaclasses. Except for stereotype «MM2Profile» that is automatically applied to a *Package* element, the application of all other stereotypes is optional. We thus enable the application of these stereotypes only in cases where additional information is required for the derivation. In addition to the stereotype attributes shown in Fig. 4.11, we introduce OCL constraints to ensure the well-formedness of the metamodel MM_{Domain}.

The purpose of the stereotypes we have defined for the UML profile 'MM2Profile' is as follows:

The «MM2Profile» stereotype: Because this stereotype has an *Extension* that is defined as required, it is automatically applied to the *Package* that defines the metamodel MM_{Domain}. Most of the attributes of this stereotype define input parameters for the model code generator of Eclipse and therefore are passed directly

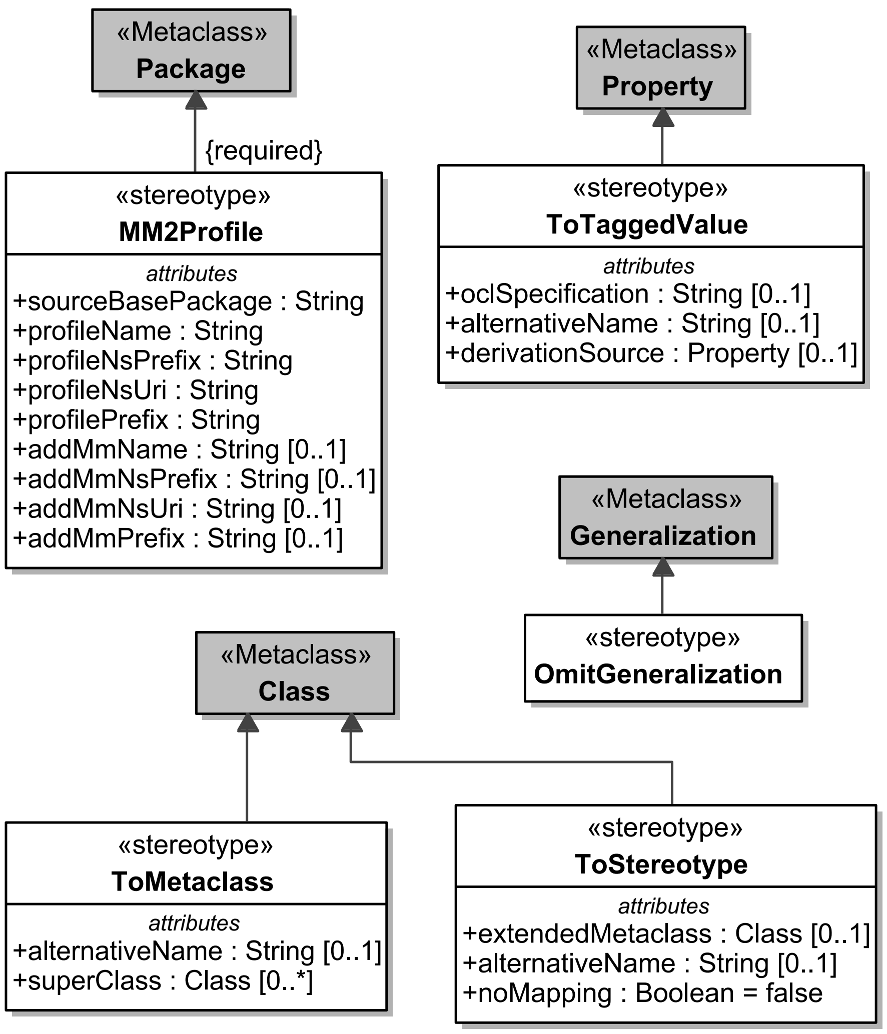

«Metaclass»
Package
{required}
«stereotype»
MM2Profile
attributes
+sourceBasePackage : String
+profileName : String
+profileNsPrefix : String
+profileNsUri : String
+profilePrefix : String
+addMmName : String [0..1]
+addMmNsPrefix : String [0..1]
+addMmNsUri : String [0..1]
+addMmPrefix : String [0..1]
«Metaclass»
Property
«stereotype»
ToTaggedValue
attributes
+oclSpecification : String [0..1]
+alternativeName : String [0..1]
+derivationSource : Property [0..1]
«Metaclass»
Generalization
«stereotype»
OmitGeneralization
«Metaclass»
Class
«stereotype»
ToMetaclass
attributes
+alternativeName : String [0..1]
+superClass : Class [0..*]
«stereotype»
ToStereotype
attributes
+extendedMetaclass : Class [0..1]
+alternativeName : String [0..1]
+noMapping : Boolean = false

Figure 4.11: The 'MM2Profile' UML Profile that is used to enrich a metamodel

to the derived UML profile and to the metamodel MM_{Add}. Our tool chain thereby determines only the names for the profile UP_{Domain} and the metamodel MM_{Add}, as well as the qualified names of contained elements.

While attributes prefixed with 'profile' are passed to the derived UML profile UP_{Domain}, all attributes with the 'add' prefix are passed to the metamodel MM_{Add}, which contains the 'additional' metaclasses. Because the generation of MM_{Add} is optional, the latter attributes are also defined as optional. The purpose of the attributes that we have defined for the «MM2Profile» stereotype is as follows:

sourceBasePackage defines the qualified name of the Java package that implements the metamodel MM_{Domain}.

profileName determines the name of UP_{Domain}.

profileNsPrefix specifies the namespace prefix for the XML namespace of UML profile UP_{Domain}, which is used in XMI documents for referring to elements contained in UP_{Domain}.

profileNsUri specifies the *Unique Resource Identifier (URI)*, which defines the XML namespace of UP_{Domain}.

profilePrefix defines a prefix that becomes part of names introduced for Java artefacts that are generated based on UP_{Domain}.

Because the attributes prefixed with 'add' of the stereotype «MM2Profile» have the same purpose as those listed above, we do not dive into their details. In addition to the above attributes, the «MM2Profile» stereotype defines OCL constraints that capture the well-formedness rules for MC_{AC} metaclasses, as discussed in Sec. 4.1.2.

The «ToStereotype» stereotype: If the default derivation for a *Stereotype* according to our approach is not applicable, the attributes of the «ToStereotype» can be employed as follows:

extendedMetaclass overrides the automatically determined UML metaclass to be extended by a derived *Stereotype*.

alternativeName defines an alternative name for a derived *Stereotype*. Otherwise, the *name* of an MC_{St} metaclass is used.

noMapping determines whether a *Stereotype* is generated for the current MC_{St} metaclass. If the value of this attribute is *'true'*, no stereotype is generated.

The above attributes provide a possibility to override the default derivation of a stereotype from an MC_{St} metaclass in MM_{Domain}. However, to ensure the applicability of our derivation approach, we have introduced the following constraints:

Constraint 4 *The UML metaclass specified by the attribute* extendedMetaclass *shall directly or indirectly inherit from the 'matching' UML metaclass determined based on the MC_{AC} metaclasses associated with the current MC_{St} metaclass.*

This constraint ensures that the UML metaclass referenced by the extendedMetaclass attribute inherits from the 'matching' UML metaclass, which is determined based on the MC_{AC} metaclass associated with the MC_{St} metaclass that has applied the ««ToStereotype»» stereotype. Only in this way it can be ensured that the alternatively specified UML metaclass provides the same attributes through inheritance as the 'matching' metaclass. We thus guarantee that the constraints defined for 'Abstract Concepts' are not violated.

Constraint 5 *Neither a super-class nor a sub-class of the current MC_{St} metaclass shall have an applied* «ToMetaclass» *stereotype.*

Stereotypes are not permitted to inherit from metaclasses by *Generalization*. The above constraint ensures that no MC_{AMC} metaclass can exist in an inheritance hierarchy of MC_{St} metaclasses. Otherwise, our derivation approach would create *Stereotypes* that inherit from 'additional' metaclasses derived from MC_{AMC} metaclasses.

Constraint 6 *The* noMapping *attribute shall only have value true if neither the current MC_{St} nor its inherited MC_{St} metaclasses introduce Properties, Operations, or Constraints.*

The noMapping attribute can be used to prevent a corresponding stereotype from being generated in UP_{Domain} for an MC_{St} metaclass. If this were applied to an MC_{St} that provides *Properties, Operations* or *Constraints,* such elements would be missing in a generated UML profile and thus, the static semantics and syntax of UP_{Domain} might be corrupted. To prevent such a situation, we introduce the above constraint.

The «ToMetaclass» stereotype: According to Design Decision 12, an additional qualification for an MC_{AMC} metaclass in MM_{Domain} is required. Hence, we employ the stereotype «ToMetaclass» to define that a metaclass of MM_{Domain} represents an MC_{AMC} metaclass. The «ToMetaclass» stereotype provides the following attributes:

alternativeName defines an alternative name for an 'additional' metaclass contained in metamodel MM_{Add}. Otherwise, the *name* of the current MC_{AMC} metaclass is used.

superClass overrides the automatically determined UML metaclass from which an 'additional' metaclass shall inherit.

As in the case of the «ToStereotype» stereotype, the above attributes provide the ability to override the default mapping. Therefore, constraints are again required to ensure the applicability of our derivation procedure. Hence, we introduce the following constraints for the «ToMetaclass» stereotype:

Constraint 7 *The UML metaclass specified by attribute* superClass *shall directly or indirectly inherit from the 'matching' UML metaclass that is determined based on the MC_{AC} metaclasses associated with the current MC_{St} metaclass.*

In contrast to a *Stereotype,* an 'additional' metaclass extends a UML metaclass using *Generalization,* so that both metaclasses are in an inheritance relationship. We determine the 'matching' metaclass of UML for an 'additional' metaclass based on the MC_{AC} metaclass associated with an MC_{AMC} metaclass. If the default mapping for an MC_{AMC} metaclass is skipped using the superClass attribute, then the referenced UML metaclass must be in a direct or indirect inheritance relationship to the original 'matching' metaclass of UML. Only thus, we can map the subsetting and redefinition relations between MC_{AMC} and MC_{AC} metaclass attributes to corresponding ones in MM_{Add}. To capture the discussed conditions, we have introduced the above constraint for the «ToMetaclass» stereotype.

Constraint 8 *Neither a super-class nor a sub-class of the current MC_{AMC} metaclass shall have an applied* «ToStereotype» *stereotype.*

As for the «ToStereotype» stereotype, we introduce the above constraint for the «ToMetaclass» stereotype in order to ensure that no inheritance relationships between *Stereotypes* and 'additional' metaclasses are created during the derivation.

The «ToTaggedValue» stereotype: We employ this stereotype to explicitly define a *name* or to specify an alternative OCL expression (used as *defaultValue*) for an attribute of a derived *Stereotype*. To support these features, the following attributes are defined for the «ToTaggedValue» stereotype:

alternativeName defines an alternative name for an *ownedAttribute* of a derived *Stereotype*.

oclSpecification defines an alternative OCL expression that is used as *defaultValue* for an *ownedAttribute* of a derived *Stereotype*.

derivationSource specifies the computation source for the oclSpecification. The attribute shall only be defined if a oclSpecification is present.

Because we employ the «ToTaggedValue» stereotype only in the context of the stereotype attribute mapping, we introduce the following constraint, which ensures that the «ToTaggedValue» stereotype can only be applied to MC_{St} attributes.

Constraint 9 *The* «ToTaggedValue» *stereotype shall be applied only to an ownedAttribute of an MC_{St} metaclass.*

The «OmitGeneralization» stereotype: In general, a metaclass can have more than one *Generalization* relation to super-classes. However, for the derivation of a UML profile, it may be required that a *Generalization* to a particular metaclass is not taken into account. This can be specified by applying the «OmitGeneralization» stereotype to a *Generalization* contained in the MM_{Domain} metamodel.

For example, assume an MC_{St} A is given that inherits from a MC_{AC} at a high level of abstraction (e.g., AC_Namespace) and another MC_{AC} (e.g., AC_DataType). In this case, we want to prevent that the stereotype derived from A extends the UML metaclass DataType. To achieve this, we apply the «OmitGeneralization» stereotype to the *Generalization* from A to DataData.

4.3.4 The UML Profile Derivation

Based on our design decisions, we now discuss the details of our approach for deriving a UML profile UP_{Domain}. As input for this derivation, we employ a metamodel MM_{Domain} enriched with the UML profile 'MM2Profile' introduced in the previous section. Apart from a description at an abstract level, we use pseudocode to explain the most important mapping operations of the $T_{MM\text{-}to\text{-}Profile}$ transformation in a comprehensible manner, employing the notational conventions introduced in Sec. 2.4.3.

Usually, the attributes of a *Stereotype* can already be introduced when creating a stereotype, but this is not feasible because of Design Decisions 16 and 17. To implement these design decisions in an efficient manner, the stereotype attributes must already exist in the derived UML profile. Hence, we realize transformation $T_{MM\text{-}to\text{-}Profile}$ using different groups of 'out-place' and 'in-place' mapping operations, which are processed in sequential order.

More generally, we can assume that our derivation consists of the following processing steps:

1. Creation of the *Stereotypes* and *DataTypes* of a UML profile by utilizing a set of 'out-place' mapping operations.

2. Introduction of the *Generalization* relationships between *Stereotypes* and the *Extension* associations to UML metaclasses.

3. Creation of the *ownedAttributes* of the *Stereotypes*.

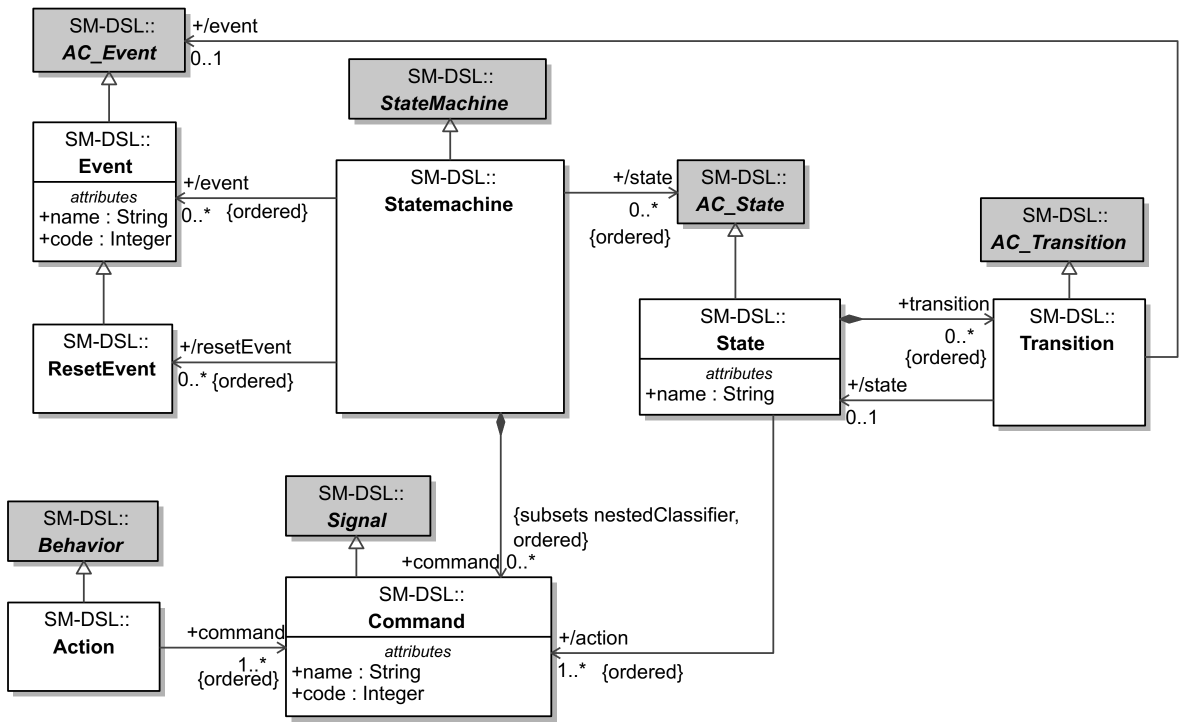
4. Introduction of OCL expressions that define the *defaultValues* of stereotype attributes, and OCL *Constraints* that preserve the well-formedness of the derived UML profile.

Apart from the first processing step, which is implemented with 'out-place' mapping operations, all other steps are realized with 'in-place' mapping operations that are applied to the UML profile generated in the first step.

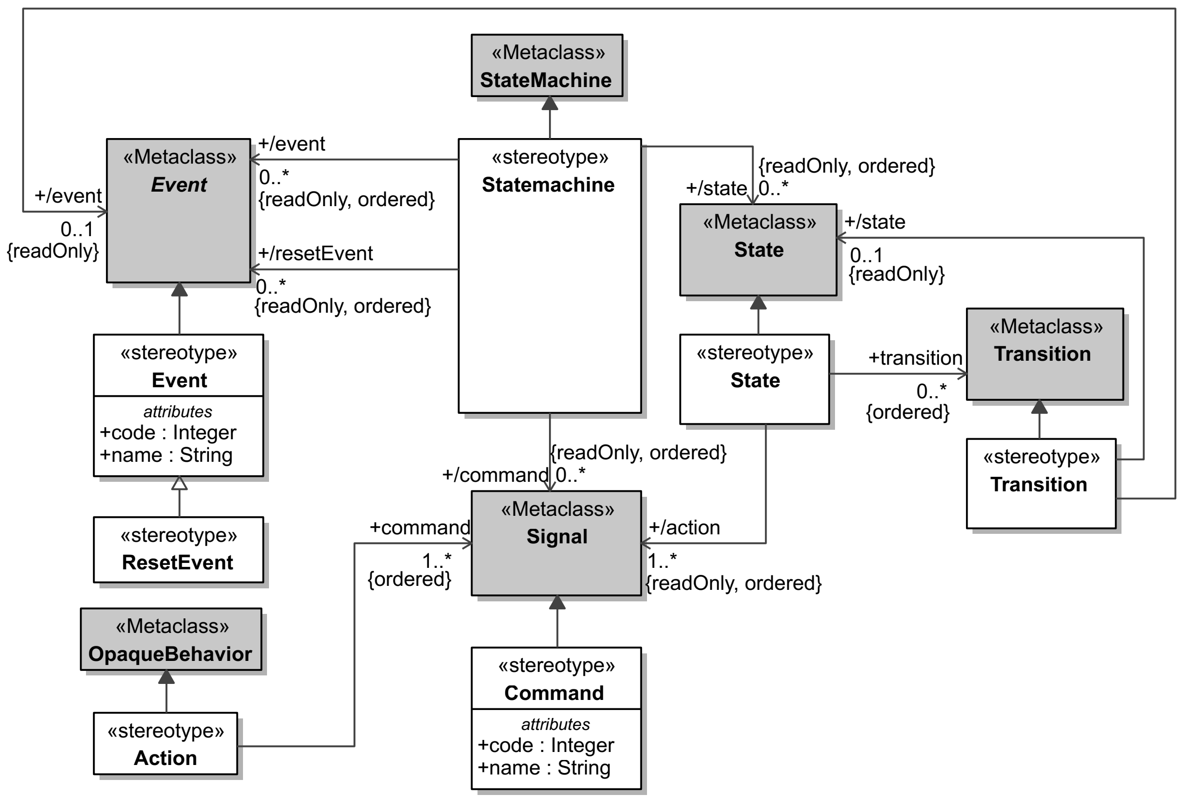
In order to ensure comprehensibility, we explain all mappings below using 'out-place' operations, which map elements of the metamodel MM_{Domain} to corresponding elements of the UML profile UP_{Domain}. In addition, we employ our running example of a state machine DSL to discuss the mappings. For this purpose, we assume that the metamodel MM_{Domain} generated for this DSL serves as input for the mapping operations below. Both the UML profile UP_{Domain} and its source metamodel are shown in Fig. 4.12.

Determining the metaclass kind. Before we detail our mapping operations, we introduce three important query operations, shown in Listing 3, which determine the kind of a metaclass contained in MM_{Domain}.

Because only an MC_{St} metaclass maps to a *Stereotype* (Design Decision10), we employ the $\mathrm{mapsToStereotype}()$ query to determine whether a metaclass is of that type. According to Design Decisions 11 and 12, this query returns value 'true' only if a metaclass is not an 'Abstract Concept' and does not map to an 'additional' metaclass. Because a metaclass can explicitly be defined as MC_{St}, value 'true' is



(a) Metamodel MM_{Domain} that is used as input for the UML profile derivation



(b) UML profile UP_{Domain} that is derived from the metamodel MM_{Domain} in Part (a)

Figure 4.12: The derived UML profile UP_{Domain} and its source metamodel MM_{Domain}

```
query Class::mapsToStereotype() : Boolean
  return (
    !self.isAbstractConcept() ∧ !self.mapsToAdditionalMetaclass() )
    ∨ self.isStereotypedBy("ToStereotype")
    ∨ self.allParents()−>any( isStereotypedBy("ToStereotype") )−>notEmpty();

query Class::isAbstractConcept() : Boolean
  return self.name.startsWith("AC_");

query Class::mapsToAdditionalMetaclass() : Boolean
  return self.isStereotypedBy("ToMetaclass") ∨ self.allParents()
    −>any(c| c.isStereotypedBy("ToMetaclass") )−>notEmpty();
```

Listing 3: The queries mapsToStereotype(), isAbstractConcept() and mapsToAdditionalMetaclass()

also returned if the «ToStereotype» stereotype is applied either to a metaclass itself or to one of its super-classes.

We use the query isAbstractConcept() in Listing 3 to decide whether a metaclass is of type MC_{AC}, i.e., an 'Abstract Concept'. Due to Design Decision 11, we only have to check whether the *name* of a metaclass is prefixed with 'AC_'.

In order to determine whether a metaclass is of type MC_{Add} that maps to an 'additional' metaclass, we employ the query mapsToAdditionalMetaclass() shown in Listing 3. According to Design Decision 12, a metaclass of type MC_{Add} must always explicitly qualified be as such. Hence, the query determines whether the «ToMetaclass» stereotype is applied to a metaclass itself or to one of its super-classes. If this is the case, the query returns value 'true'.

Mapping to 'Stereotypes'. Owing to Design Decision 12, we always map an MC_{St} metaclass to a corresponding *Stereotype* of the UML profile UP_{Domain}. We implement this by the toStereotype() mapping operation of Listing 4, where the guard condition in Line 2 ensures that only metaclasses of type MC_{St} are mapped. Furthermore, if the «ToStereotype» stereotype is applied to an MC_{St}, the mapping to a *Stereotype* can be omitted due to the noMapping attribute. This condition is also verified in the guard condition by invoking the noStereotypeMapping() query. Because of its simplicity, we leave out the details of this query.

In the first part of the the toStereotype() mapping operation (see Lines 4 – 9), we map some properties of an MC_{St} one-to-one to corresponding *Stereotype* properties, while items of the *ownedAttribute* and *ownedOperation* properties are mapped in a particular manner, which is discussed later.

Thereafter, we introduce inheritance relationships (i.e., a *Generalization*) to super-types of a mapped stereotype. According to Design Decision 14, a *Generalization* must be introduced only for those *Stereotypes* created from MC_{St} metaclasses that inherit from other MC_{St} metaclasses. This fact is illustrated in Fig. 4.13,

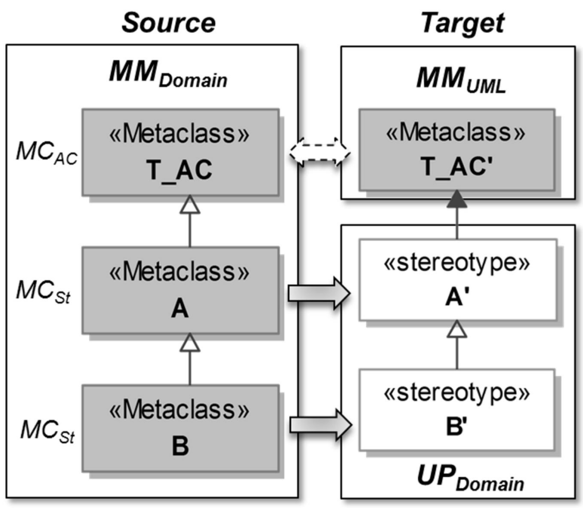


Figure 4.13: Mapping of MC_{St} metaclasses to *Stereotypes*

where only *Stereotype* `B'` has a *Generalization* relationship to *Stereotype* `A'`, and both stereotypes are derived from an MC_{St} metaclass of MM_{Domain}. Hence, we determine all those MC_{St} metaclasses from which the current context MC_{St} of the mapping operation inherits, by invoking the superStereotype() query in Line 12. Then, we assign the *Stereotypes* mapped from these metaclasses to the *superClass* property. This also implies an implicit creation of *Generalization* relationships between the currently mapped *Stereotype* and its super-types. An example for this mapping is shown in Fig. 4.12b, where the «ResetEvent» stereotype inherits from the «Event» stereotype.

In the third part of the toStereotype() mapping operation (see Lines 15 – 29), we introduce *Extension* associations between the mapped *Stereotype* and the UML metaclass to be extended. However, due to Design Decision 13 an *Extension* is introduced only for those *Stereotypes* created from MC_{St} metaclasses that directly inherit from MC_{AC} metaclasses. For a better comprehensibility, we illustrate the above processing step using Fig. 4.13.

Assume that the shown stereotypes `A'` and `B'` are mapped from the MC_{St} metaclasses `A` and `B` in MM_{Domain}. Then, according to the rule above, an *Extension* is introduced between *Stereotype* `A'` and the 'matching' UML metaclass T_AC', but not between the *Stereotypes* `A'` and `B'`.

In Line 15 of the toStereotype() mapping operation, we call the extensionTargets() query for determining the set of UML metaclasses that shall be extended by a mapped *Stereotype*. Then, based on the metaclasses thus obtained, an *Extension* and a *Property* are introduced for each identified UML metaclass. We assign the created *Property* to the set of *ownedAttributes* of the mapped *Stereotype* (see Line 20). Because it represents an implicit reference to the extended UML metaclass, we compose its *name* using the prefix 'base_' and the name of the UML metaclass. As the created *Extension* cannot be owned by a *Stereotype*, we attach it to the en-

```
Class ⇒ toStereotype() : Stereotype
when { self.mapsToStereotype() ∧ !self.noStereotypeMapping()}
  // 1. Mapping of the metaclass properties
  result.name := self.name;
  result.isAbstract := self.isAbstract;
  result.ownedComment := self.ownedComment−>map toComment();
  result.ownedRule := self.ownedRule−>map toConstraint();
  result.ownedAttribute−>map toProperty();
  result.ownedOperation := self.ownedOperation−>map toOperation();

  // 2. Adding intheritance relationships
  result.superClass := self.superStereotypes()−>map toStereotype();

  // 3. Adding 'Extension' relationships to UML metaclasses
  self.extensionTargets()−>forEach(mc) {
    // Creating the 'base_xxx' stereotype attribute
    var baseProp := object Property {
      name := "base_"+ mc.name;
      type := mc; };
    result.ownedAttribute += baseProp;
    // Creating the 'Extension'
    profile.packagedElement += object Extension {
      name := self.name +"_"+ mc.name;
      ownedEnd := object ExtensionEnd {
        name := "extension_"+ self.name;
        type := self;
        lower := 0; upper := 1; };
      memberEnd += baseProp; };
  };

  // 4. Creating additional OCL 'Constraints'
  var oclSpecs := self.genOclSpecForSrcAttributeConstraints()
    −>union(self.genOclSpecForExplicitPropConstraints());
  oclSpecs−>forEach(spec) {
    result.ownedRule += object Constraint {
      var number := result.ownedRule−>size() + 1;
      name := "Constraint_"+ number.repr();
      specification := object OpaqueExpression {
        language := "OCL";
        body := spec; }; };
  };
```

Listing 4: The toStereotype() mapping operation and associated queries

```
42 query Class::extensionTargets() : Set(Class)
43    if self.isStereotypedBy("ToStereotype") then
44       return self.getExplicitExtensionTargets() endif;
45    var res : Set(Class) := Set{};
46    var generalizations := self.generalization−>select( !isOmitted());
47    var mc_ac := generalizations.general
48       −>select( !mapsToStereotype() );
49    mc_ac−>forEach(ac) {
50       res += getUmlTypeByName(ac.name).asType(Class) };
51    return res;
52
53 query Class::superStereotypes() : Set(Class)
54    var generalizations := self.generalization−>select( !isOmitted() );
55    return generalizations.general−>select( mapsToStereotype() );
```

Listing 4: The toStereotype() mapping operation and associated queries (continued)

closing *Profile* (see Line 22). For example, all *Extensions* between *Stereotypes* and UML metaclasses shown in Fig. 4.12b are the result of the explained mapping.

In the last part of the toStereotype() mapping operation, we introduce additional OCL constraints for a mapped *Stereotype* according to Design Decisions 20 and 21. We collect the OCL specifications for the new constraints by means of two queries in Line 32 and create a *Constraint* element that is added to the existing items of the *ownedRule* property of a *Stereotype*. The details of both employed queries are treated later in this section.

Determining super-types: As mentioned above, we determine the MC_{St} metaclasses that are super-types of a given MC_{St} using the superStereotypes() query shown in Lines 42 – 42 of Listing 4. The super-types are identified based on the *generalization* property of the context MC_{St}, but all items of this property with applied «OmitGernalization» stereotype are omitted. Then, we select all MC_{St} metaclasses and return them.

Determining UML extension targets: The UML metaclasses to be extended by a mapped *Stereotype* are identified by the query extensionTargets() shown in Lines 53 – 55 of Listing 4. First, we check whether UML metaclasses have been explicitly defined using the extendedMetaclass attribute of the «ToStereotype» stereotype, and if so, these metaclasses are returned as result. Otherwise, we determine the UML metaclasses based on the *Generalizations* of the current context MC_{St} of the query. However, all *Generalizations* with applied «OmitGernalization» stereotype are sorted out, which is realized using the isOmitted() query. Then, all MC_{AC} metaclasses are determined so that their 'matching' UML metaclasses can be resolved by invoking the getUmlTypeByName() function.

Mapping of *DataTypes.* Apart from *Stereotypes*, a UML profile can also introduce different kinds of data types, which can be *DataTypes*, *PrimitiveTypes*, or *Enumerations*. For the derivation of a UML profile, we determine those data types of MM_{Domain} that do not represent 'Abstract Concepts' and that are not mapped to data types of MM_{Add}. Then, we copy these data types one-to-one into the UML profile. Because this mapping is straightforward, its details are not discussed here.

Mapping to stereotype attributes. According to Design Decision 15, we map an *ownedAttribute* of an MC_{St} metaclass to a corresponding attribute of a *Stereotype* contained in UP_{Domain}. However, we must also respect the extension mechanism of *Stereotypes*, because subsetting and redefinition are only applicable to stereotype attributes as long as no attribute of a UML metaclass is involved. Due to this reason and according to Design Decision 16, an MC_{St} metaclass attribute that is redefining or subsetting an attribute of a UML metaclass is mapped to a 'derived' and 'read-only' stereotype attribute. Except of those attributes that have a manually specified OCL expression (see Design Decision 19), all other attribute types of an MC_{St} metaclass are mapped one-to-one to corresponding stereotype attributes.

Determining the type of an attribute: Before we detail the employed mapping operations, we briefly explain the isDerivedProperty() query, which determines whether an MC_{St} attribute shall be mapped to a 'derived' and 'read-only' stereotype attribute. As shown in Listing 5, we distinguish four cases for which an MC_{St} attribute shall be mapped to a derived stereotype attribute.

In the first two cases captured in Lines 3 – 9, we check whether a *Property* in MM_{Domain} is already specified as 'derived', or whether an alternative OCL expression (see Design Decision 19) is provided by means of the «ToTaggedValue» stereotype, which is done by employing the hasAdditionalOclSpec() query.

In the third and fourth case (Lines 13 – 19), we verify whether an MC_{St} attribute is redefining or subsetting an MC_{AC} attribute, because we have to map such an attribute to a 'derived' stereotype attribute as defined by Design Decision 16. For this purpose, we determine the set of MC_{AC} attributes that are directly or indirectly redefined or subsetted by an MC_{St} attribute. Only if at least one of the determined attributes is not specified to be 'read-only', we assume that an MC_{St} attribute must be mapped to a 'derived' stereotype attribute. This is because a model element must be assignable to at least one of those UML metaclass attributes that are invoked by an OCL expression to compute a value of a 'derived' stereotype attribute at runtime; otherwise, this computation would return an empty set.

In both cases, we do not regard MC_{St} attributes that are defined to be a superset of other attributes, i.e., its *isDerivedUnion* property has value *'true'*. Because we wish to preserve the superset relationships of such attributes, they are mapped one-to-one instead of creating 'derived' stereotype attributes for them.

```
query Property::isDerivedProperty() : Boolean
  // 1. An ‘additional’ OCL specification is present
  if self.hasAdditionalOclSpec() then
    return true endif;

  // 2. A derived property with OCL specification is already present
    if self.isDerived ∧ self.isReadOnly ∧ self.defaulValue <> null
      ∧ !self.isDerivedUnion then
      return true endif;

  // 3. An ’ownedAttribute’ of an ’AbstractConcept’ is redefined
  if !self.isDerivedUnion ∧ self.allRedefinedProperties()
    −>forAll(class.isAbstractConcept())−>any(!isReadOnly) then
      return true endif;

  // 4. An ’ownedAttribute’ of an ’AbstractConcept’ is subsetted
  if !self.isDerivedUnion ∧ self.allSubsettedProperties()
    −>forAll(class.isAbstractConcept())−>any(!isReadOnly) then
      return true endif;

  // In all other cases ...
  return false;
```

Listing 5: The isDerivedProperty() query

Derived and read-only attributes: Based on Design Decisions 16 and 19, we map an MC_{St} metaclass attribute to a ‘derived’ and ‘read-only’ stereotype attribute. As argued above, we distinguish between four different scenarios to which this mapping applies. However, the most important application scenario is the substitution of ‘redefinition’ and ‘subsetting’ relationships between attributes of MC_{St} and MC_{AC} metaclasses. For the sake of clarity, we employ the example given in Fig. 4.14 to explain these two cases in more detail.

In Case (A) of our example, an MC_{St} metaclass A with an attribute a and an MC_{AC} metaclass T_AC are given. Furthermore, attribute a redefines attribute t_ac of T_AC, and its *type* property also refers to T_AC. We obtain a ‘read-only’ and ‘derived’ attribute a of stereotype A' as mapping result, and the *type* property of a now refers to the ‘matching’ UML metaclass T_AC’. In addition, an OCL expression is introduced to define the *defaultValue* of a.

Case (B) differs from Case (A) only in the detail that attribute a of MC_{St} A is subsetting attribute t_ac, instead of redefining it. However, the mapping result is the same as in Case (A).

We realize the mapping towards a ‘derived’ and ‘read-only’ stereotype attribute by the toDerivedProperty() mapping operation shown in Listing 6, which is divided into three parts. The guard condition of this mapping operation invokes the already introduced query isDerivedProperty() to determine whether a *Property* has to be mapped.

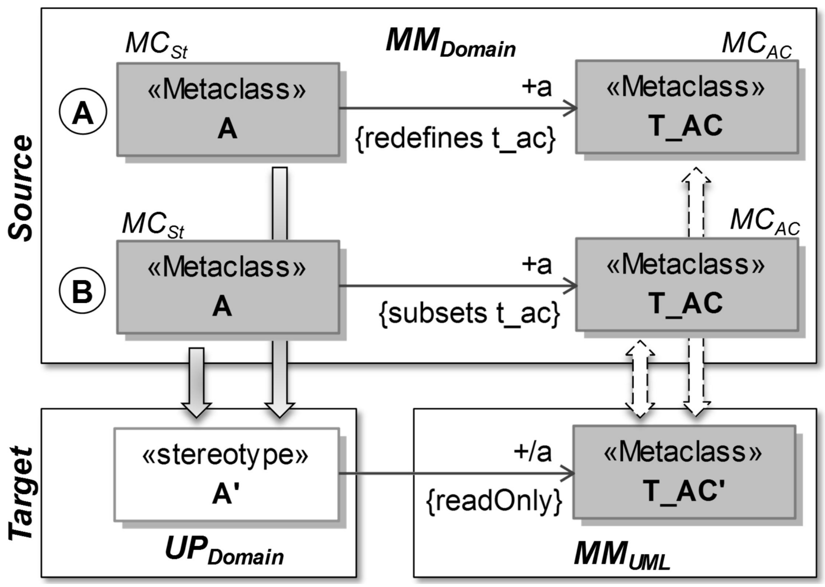


Figure 4.14: Mapping to derived attributes.

In the first part of the operation (Lines 4 – 15), we process those properties, e.g., the *name*, that can be mapped one-to-one to a stereotype. An exception are the *type* and *aggregation* properties. The *type* property has to be recomputed due to Design Decision 17, and we employ the getNewType() query in Line 15 to determine the recomputed value. Furthermore, due to Design Decision 15 the *aggregation* property of a stereotype attribute, whose *type* refers to a metaclass, must be assigned value 'none', as shown in Lines 5 to 8.

In the second part of the mapping operation, we define a stereotype attribute as 'derived' and 'read-only', as shown in Lines 18 – 20.

Finally, we create the *defaultValue* that is defined by means of an OCL expression in the third part (Lines 23 – 35). To do so, we first check whether an OCL expression is manually provided by an applied «ToTaggedValue» stereotype, and if present, we assign it to the *defaultValue* property. Alternatively, we try to generate an OCL expression based on 'redefined' or 'subsetted' attributes. Otherwise, we map the *defaulValue* of the MC_{St} source attribute.

For example, assume that the attributes of the MC_{St} metaclass Statemachine of our running example (see Fig. 4.12a) are passed to the toDerivedProperty() mapping operation. As all attributes meet the discussed criteria, they are processed by this operation. In particular, the command attribute of the MC_{St} is mapped because it subsets the nestedClassifier attribute of an MC_{AC}, whereas the remaining attributes are mapped because they are already defined as 'derived'. Hence, all attributes of the «Statemachine» stereotype (shown in Fig. 4.12b) are 'derived' and 'read-only'.

Other types of attributes: In contrast to MC_{St} attributes that are mapped to 'derived' stereotype attributes, we map the remaining attribute kinds one-to-one. This also applies to those MC_{St} attributes that are redefining/subsetting other MC_{St} at-

```
Property ⇒ toDerivedProperty() : Property
when { self.isDerivedProperty() }
  // 1. Mapping of general attribute properties
  result.name := self.name;
  if (isTypeOf(Class)) then
    result.aggregation := AggregationKind::none
  else
    result.aggregation := self.aggregation
  endif;
  result.lower := self.lower;
  result.upper := self.upper;
  result.isUnique := self.isUnique;
  result.isOrdered := self.isOrdered;
  result.ownedComment := self.ownedComment;
  result.type := self.type.getNewType();

  // 2. Defining a stereotype attribute as 'derived'
  result.isReadOnly := true;
  result.isDerived := true;
  result.isDerivedUnion := false;

  // 3. Creating the OCL defined 'defaultValue'
  var oclExp : String := null;
  if self.hasAdditionalOclSpec() then
    oclExp := self.getAdditionalOclSpec()
  else
    oclExp := self.generatedOclSpec()
  endif;
  if oclExp <> null then {
    result.defaultValue :=
      new OpaqueExpression( self.getAdditionalOclSpec() );
      result.aggregation := self.aggregation endif;
  }
  else
    result.defaultValue := self.defaultValue.map toValueSpecification()
  endif;
```

Listing 6: The toDerivedProperty() mapping operation

tributes, or that are specified to be a superset. Because only stereotype attributes are involved, it is possible to preserve such kinds of relationships after the mapping, and therefore no UML restrictions apply in this case.

The toNonDerivedProperty() mapping operation shown in Listing 7 maps MC_{St} attributes to 'non-derived' stereotype attributes. In the guard condition of the operation, we invoke the isDerivedProperty() to evaluate whether the conditions for a mapping to a 'derived' stereotype attribute are violated, and if so, an MC_{St} attribute is mapped to a 'non-derived' stereotype attribute.

Most of the properties of an MC_{St} attribute are mapped one-to-one to corresponding properties of a stereotype attribute. However, due to Design Decisions 15

```
 1 Property ⇒ toNonDerivedProperty() : Property
 2 when { !self.isDerivedProperty() }
 3   result.name := self.name;
 4   if (isTypeOf(Class)) then
 5     result.aggregation := AggregationKind::none
 6   else
 7     result.aggregation := self.aggregation endif;
 8   result.lower := self.lower;
 9   result.upper := self.upper;
10   result.isUnique := self.isUnique;
11   result.isOrdered := self.isOrdered;
12   result.ownedComment := self.ownedComment;
13   result.type := self.type.getNewType();
14   result.isReadOnly := self.isReadOnly;
15   result.isDerived := self.isDerived;
16   result.isDerivedUnion := self.isDerivedUnion;
17   self.redefinedProperty−>forEach(rp) {
18     if !rp._class.isAbstractConcept() then
19       result.redefinedProperty += rp.map toProperty() endif;
20   };
21   self.subsettedProperty−>forEach(sp) {
22     if !sp._class.isAbstractConcept() then
23       result.subsettedProperty += sp.map toProperty() endif;
24   };
```

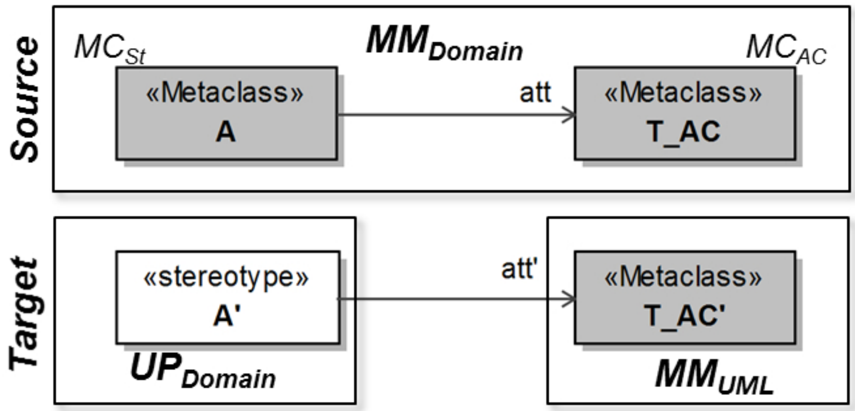
Listing 7: The toNonDerivedProperty() mapping operation

and 17, we recompute the *type* property and set the value for the *aggregation* property as explained for the toDerivedProperty() mapping operation.

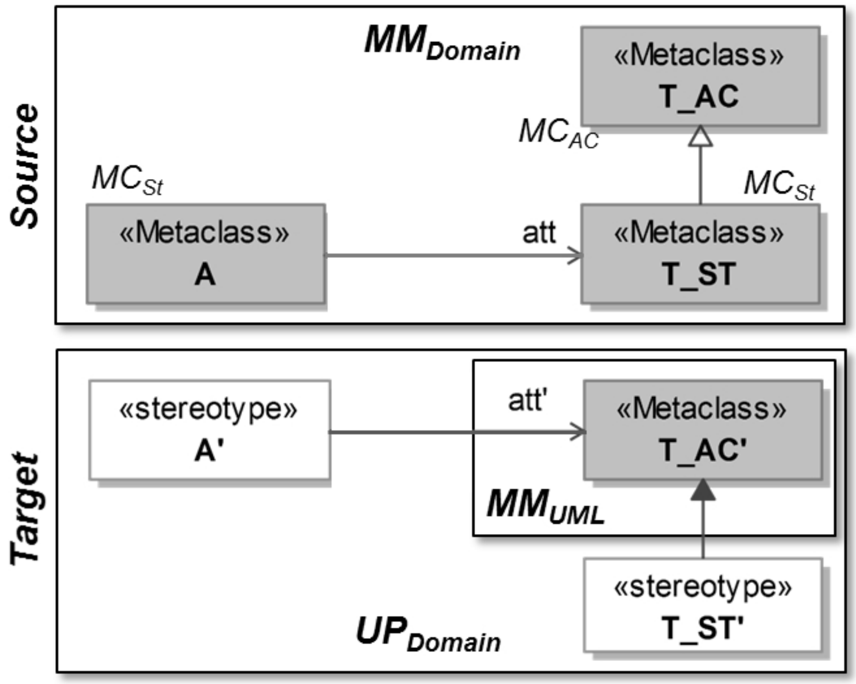
Another exception to the default mapping concerns the properties *subsettedProperty* and *redefinedProperty*. Because attribute subsetting or redefinition is only feasible between stereotype attributes, we map only those items that refer to MC_{St} attributes (Lines 17 – 24).

Computation of attribute types. A central design decision for our derivation approach is that the metamodel MM_{Domain} can contain three different types of metaclasses (see Design Decision 10), which are mapped in different ways. In addition, the *type* property of a metaclass attribute has to be recomputed during the mapping, as specified by Design Decision 17. Because different types of metaclasses can be contained in MM_{Domain}, we must consider this fact for recomputing the *type* property of a metaclass attribute during its mapping.

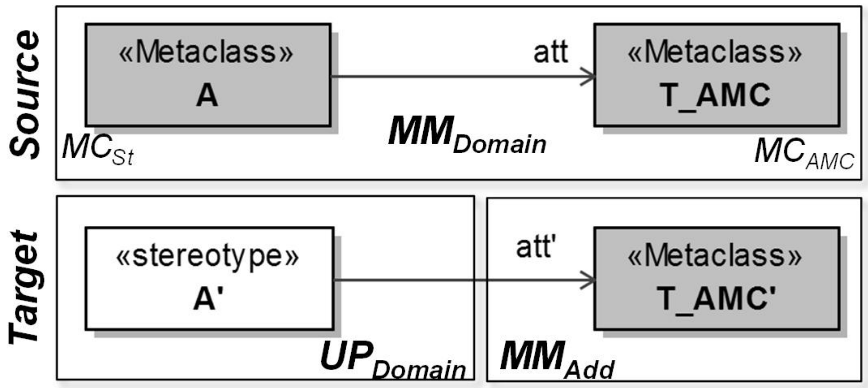
In the following, we analyse the different cases. Let `A` be an MC_{St} metaclass in MM_{Domain} and `A'` its corresponding *Stereotype* in UP_{Domain}. In addition, assume that MC_{St} `A` has an attribute att that maps to a corresponding attribute att' of `A'`, and the *type* property of att can refer to an MC_{AC} (Case A), an MC_{St} (Case B), or an MC_{AMC} (Case C) metaclass, as shown in Fig. 4.15. We determine the *type* property of att' as follows:



(a) An MC_{AC} metaclass is used as attribute type



(b) An MC_{St} metaclass is used as attribute type



(c) An MC_{AMC} metaclass is used as attribute type

Figure 4.15: Variants for recomputing the *type* property of stereotype attributes

Case A: Based on Design Decision 11, we assume that a 'matching' UML metaclass exists for each MC_{AC} metaclass of MM_{Domain}. Hence, when we recompute a *type* property that refers to an MC_{AC} metaclass, we determine the 'matching' UML metaclass and use it as the new *type*.

For example, consider attribute att of MC_{St} `A` shown in Fig. 4.15a as input, and assume that the *type* property of att refers to an MC_{AC} metaclass T_AC in MM_{Domain}. Then, the recomputed *type* of stereotype attribute att' refers to metaclass T_AC' of MM_{UML}.

Case B: According to Design Decision 10, MC_{St} metaclasses in MM_{Domain} are mapped to *Stereotypes*, which have either *Extensions* to UML metaclasses or *Generalizations* to other *Stereotypes*. If the *type* of a metaclass attribute refers to an MC_{St} metaclass, the *type* of the mapped attribute can refer either to a *Stereotype* or its extended UML metaclass.

Our derivation approach employs references to UML metaclasses, as we have argued for Design Decision 17. Thus, when we recompute a *type* property that refers to an MC_{St} metaclass, we determine a UML metaclass rather than the *Stereotype* that extends it.

For instance, consider attribute att of MC_{St} `A` shown in Fig. 4.15b as input, and assume the *type* property of att refers to the MC_{St} metaclass T_ST in MM_{Domain}. Then, the recomputed *type* of stereotype attribute att' refers to metaclass T_AC' of MM_{UML}.

Case C: An MC_{AMC} metaclass of MM_{Domain} maps to an 'additional' metaclass in MM_{Add}, as specified by Design Decision 10. If such an MC_{AMC} is referenced as a *type* of a metaclass attribute in MM_{Domain}, then only the corresponding metaclass in MM_{Add} has to be determined and used as *type* of the mapped attribute.

For example, let attribute att of MC_{St} `A` shown in Fig. 4.15c be the input for the recomputation, and assume that the *type* of att refers to the MC_{AMC} metaclass T_AMC in MM_{Domain}. Then, the recomputed *type* of stereotype attribute att' refers to metaclass T_AMC' of MM_{Add}.

In our running example, all the attributes of the «Statemachine» stereotype shown in Fig. 4.12b are an example of Case B because their *type* properties refer to UML metaclasses extended by *Stereotypes*.

In addition to metaclasses, metamodel MM_{Domain} may contain different kinds of data types, which we copy one-to-one to the UML profile UP_{Domain} or the metamodel MM_{Add}. Because these data types can also be referenced by the *type* property of a metaclass attribute in MM_{Domain}, we have to recompute the *type* of a mapped attribute in this case as well. But unlike metaclasses, we only have to determine

```
 1 query Type::getNewType() : Type
 2   // 1. Resolving metaclasses
 3   if self.isType(Class) then {
 4     var cls := self.asType(Class);
 5     switch {
 6       // Case A: Type is an ‘Abstract Concept’
 7       case ( cls.isAbstractConcept() )
 8         return getUmlTypeByName( self.name ) endif;
 9       // Case B: Type is a MC_St
10       case ( cls.mapsToStereotype() ) {
11         var baseType := cls.getBaseMetaclass();
12         return getUmlTypeByName( baseType.name ); };
13       // Case C: Type is a MC_Add
14       case ( cls.mapsToAdditionalMetaclass() )
15         return getAddMcByName( cls.name )
16     };
17   } endif;
18   // 2. Resolving data types
19   if self.isKind(DataType) then
20     switch {
21       case ( self.name.startsWith("AC_") )
22         return getUmlTypeByName( self.name );
23       case ( self.isProfileDataType() <> null )
24         return getProfileDataType( self.name );
25       case ( self.isAddDataType() <> null )
26         return getAddDataType( self.name );
27     };
28   endif;
29   return null;
```

Listing 8: The getNewType() query

whether the data type is contained in UP_{Domain}, MM_{Add} or MM_{UML}; then we use it as the new *type* of a mapped attribute.

We employ the getNewType() query to determine the *type* property of a mapped attribute of a *Stereotype* or an ‘additional’ metaclass, and we also use it to resolve the *type* of an *Operation Parameter*[6]. As shown in Listing 8, the query consists of two parts: The different types of metaclasses of MM_{Domain} are processed in the first part, while *DataTypes* are resolved in the second part.

The first part of the query is structured analogously to the three metaclass types already discussed. In Case A (see Line 6), we verify whether the metaclass passed as input is of type MC_{AC}, and if so, we invoke the getUmlTypeByName() helper function to return the ‘matching’ metaclass in MM_{UML} as result.

Provided the passed metaclass is an MC_{St}, the UML metaclass that shall be extended by a *Stereotype* is determined in Case B (see Line 9). For this purpose, we invoke the getBaseMetaclass() query to resolve the MC_{AC} metaclass, which is

[6] Both MOF concepts *Property* and *Parameter* are a specialization of *TypedElement*; thus, both inherit the *type* property.

```
Operation ⇒ toOperation() : Operation
  result.name := self.name;
  result.isQuery := self.isQuery;
  result.ownedComment := self.ownedComment;
  result.bodyCondition := self.bodyCondition.map toConstraint();
  self.redefinedOperation−>forEach(rop) {
    if !rop._class.isAbstractConcept() then
      result.redefinedOperation += rop.map toOperation() endif;
  };
  result.ownedParameter := self.ownedParameter−>map toParameter();

Parameter ⇒ toParameter() : Parameter
  result.name := self.name;
  result.direction := self.direction;
  result.lower := self.lower;
  result.upper := self.upper;
  result.isOrdered := self.isOrdered;
  result.isUnique := self.isUnique;
  result.type := self.type.getNewType();
```

Listing 9: The mapping operations $\mathrm{toOperation}()$ and $\mathrm{toParameter}()$ for processing MC_{St} operations

directly or indirectly specialized by the current MC_{St}. To do so, all parent metaclasses of this MC_{St} are recursively visited until a parent is reached who is an MC_{AC} metaclass or who has the «ToStereotype» stereotype applied. Then, either the MC_{AC} metaclass or the UML metaclass explicitly specified by «ToStereotype» is returned, and the 'matching' UML metaclass is resolved by invoking the $\mathrm{getUmlTypeByName}()$ helper function.

In Case C (see Line 13), we check whether the input is an MC_{AMC} metaclass, and if so, we resolve the corresponding metaclass in the metamodel MM_{Add} using the helper function $\mathrm{getAddMcByName}()$.

Data types are recomputed in the second part of the $\mathrm{getNewType}()$ query, where we distinguish three cases. We check if the input *DataType* has a 'matching' one in the UML metamodel or whether it corresponds to a data type in UP_{Domain} or MM_{Add}; if so, the resolved *DataType* is returned.

Mapping of *Operations*. The $\mathrm{toOperation}()$ mapping operation shown in Listing 9 maps an *Operation* of an MC_{St} metaclass to a corresponding *Operation* of a *Stereotype*. Because of the restrictions specified for UML profiles, a particularity of this mapping operation is that not all items of the *redefinedOperation* property can be mapped. If an operation of an MC_{AC} metaclass is referenced by the *redefinedOperation* property, this would imply that a corresponding operation of a 'matching' UML metaclass would have to be referenced after the mapping. But a *Stereotype* is not permitted to redefine a feature of a UML metaclass. Hence, we map

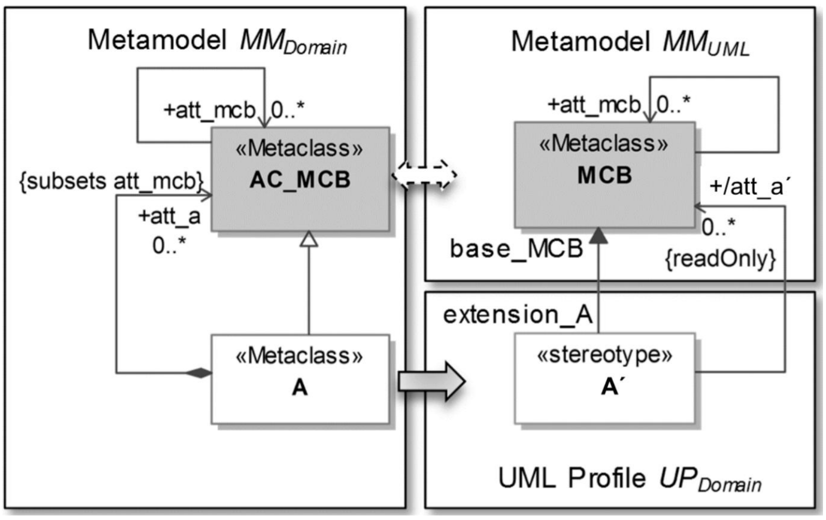


Figure 4.16: Subsetting MC_{St} attribute and the corresponding 'derived' and 'read-only' stereotype attribute

only those items of the *redefinedOperation* property that are owned by an MC_{St} metaclass, as implemented in Line 7.

According to Design Decision 17, the *type* of a stereotype attribute has to be recomputed. Because the *Parameter* of an *Operation* also has a *type*, we recompute this property in the same way. Therefore, the toProperty() mapping operation invokes the query getNewType() in Line 19 to determine the new value for the *type* property of a *Parameter*.

4.3.5 OCL Expressions for Stereotype Attributes

Due to Design Decision 16, we map a redefined or subsetted attribute of an MC_{St} metaclass to a 'derived' and 'read-only' stereotype attribute. Consequently, the value of this attribute is computed at runtime based on its *defaultValue* property. To enable this computation, we introduce an OCL expression for each of these attributes. However, as we have argued for Design Decision 19, the automatic introduction of OCL expressions is limited to the subsetting or redefinition of a single attribute only. In all other cases, an OCL expression has to be manually specified by applying the «ToTaggedValue» stereotype.

Assume we have given an MC_{St} metaclass `A` that inherits from an MC_{AC} metaclass AC_MCB, and `A` has an attribute att_a that is subsetting an attribute att_mcb of metaclass AC_MCB, as shown in Fig. 4.16. According to our mapping rules, `A` is mapped to a *Stereotype* `A'`, and att_a maps to a 'derived' and 'read-only' stereotype attribute att_a'. Because metaclass AC_MCB has a 'matching' UML counterpart MCB, metaclass AC_MCB must not be mapped. In addition, we introduce an *Extension* between *Stereotype* `A'` and metaclass MCB, which also implies the creation of the attributes extension_A and base_MCB. However, the subsetting relationship of att_a is not preserved for att_a'; instead, an OCL expression is introduced so that the attribute's value can be computed at runtime.

For this purpose, we first navigate from the instance of stereotype `A'` to the instance of MCB by employing attribute base_MCB. Then, we access attribute att_MCB to collect all items required to compute the value of attribute att_a'. We select these items either based on their applied stereotypes – as in the present case – or according to their type. The latter option is applied if the *type* property of att_a' refers to an 'additional' metaclass. Finally, we typecast the set of all determined values so that they match the type and cardinality of att_a'.

The substitution described in the example above can also be applied to other subsetting or redefining MC_{St} attributes; these can be replaced by an OCL expression in the same way. A generated OCL expression can always be partitioned into the following parts:

- navigation to UML metaclass MCB that is extended by a *Stereotype* ST;
- navigation to attribute att_src of MCB, so this attribute serves as the source for the value computation;
- selection of all relevant items of att_src, based on their applied stereotype or element type;
- type-cast of the selected items to match the *type* and cardinality of the stereotype attribute att_st.

We than generate all OCL expressions according to two patterns. The first pattern is used for stereotype attributes that have an upper cardinality one, while the second pattern is employed in all other cases. The patterns are as follows:

```
(1) self.base_<MCB>.<att_src>
        −>any(<sel_exp>).<type−cast>
(2) self.base_<MCB>.<att_src>
        −>select(<sel_exp>)−><type−cast>
```

For example, when we employ Pattern (2) for the command attribute of the «Statemachine» stereotype (see Fig. 4.12b), we obtain the following OCL expression:

```
self.base_StateMachine.nestedClassifier
  −>select(isStereotypedBy('Command')).oclAsType(UML::Signal)−>asOrderedSet()
```

In this example, StateMachine is the extended metaclass <MCB>, and nestedClassifier corresponds to the source attribute <att_src>. In the second code line, all items of the nestedClassifier that are stereotyped with «Command» are selected. Finally, these items are type-casted to an ordered set of Signal elements, because this is the defined *type* of the command attribute.

```
 1 query Property::generateOclSpec() : String
 2   if self.isDerivedProperty()
 3     ∧ ( self.redefinedProperty−>size() = 1 ∨ self.subsettedProperty−>size() = 1) then
 4       return null endif;
 5
 6   // 1. Gathering the different parts of the expression
 7   var relatedProp := self.subsettedProperty−>union(self.redefinedProperty)−>any(true);
 8   var src_exp := self.oclContext( relatedProp );
 9   var sel_exp := self.oclSelectExp( relatedProp );
10   var cast_exp := self.oclTypeCast( relatedProp );
11
12   // 2. Assembling the OCL expression according to the patterns
13   var oclString := if srcProp.upper = 1 then
14       "<SRC_EXP>->any(<SELECT_EXP>).<CAST>"
15     else
16       "<SRC_EXP>->select(<SELECT_EXP>).<CAST>"
17     endif;
18   oclString := oclString.replaceAll("<SRC_EXP>", src_exp);
19   oclString := oclString.replaceAll("<SELECT_EXP>", sel_exp);
20   oclString := oclString.replaceAll("<CAST>", cast_exp);
21
22 query Property::oclContext(relatedProp : Property) : String
23   var extMCs := self._class.allBaseMetaclasses();
24   var baseMC := extMCs−>any(mc| mc.ownedMember−>includes(relatedProp) );
25   return if baseMC <> null then
26     "self.base_" + getUmlTypeByName(baseMC.name).name + "." + relatedProp.name
27     else "" endif;
```

Listing 10: The queries generateOclSpec() and oclContext()

We implement the generation of OCL expressions for 'derived' and 'read-only' stereotype attributes by the generateOclSpec() query, whose pseudocode is shown in Listing 10. First, we evaluate whether a generation is supported for the current *Property* according to the above conditions, and if not, the query returns value 'null'. Then, in the first part of the query (Lines 6 – 10), we invoke additional queries to collect the information required to compile the OCL expression in Lines 13 – 20.

The relatedProp variable corresponds to the attribute <att_src>, which is determined based on the subsetted or redefined properties of the current context *Property* of the query. Because of the defined rules for the introduction of 'derived' and 'read-only' stereotype attributes, exactly one related property exists. Finally, the OCL expression is assembled in the second part by applying one of the generation patterns introduced before.

The oclContext() query shown in Listing 10 generates the OCL fragment that is employed to navigate to the <att_src> attribute. First, we call the allBaseMetaclasses() query to determine all those MC_{AC} metaclasses from which the container MC_{St} metaclass of the current context *Property* inherits. Then, we ascertain which of these metaclasses owns the relatedProp, so that we can use the *name* of this metaclass to construct the first part of the source navigation expression. The second part of this expression is created based on the *name* of the relatedProp.

To create the OCL expression fragment for selecting all relevant items of the <att_src> attribute, we employ the oclSelectExp() query shown in Listing 11. Based on the *type* of the current context *Property*, we distinguish two cases. If the *type* is not an MC_{St} metaclass, then we return the oclIsKindOf() operation as a selection criterion; otherwise, the user-defined isStereotypedBy() operation is returned, which determines whether a particular stereotype is applied to an element. In Line 9, we derive the qualified name of the *Stereotype*, which has to be verified using the isStereotypedBy() operation, from the global variable PROFILE_NAME and the *name* of the MC_{St} metaclass. The variable contains the current value of the profileName attribute of the «MM2Profile» stereotype applied to MM_{Domain}.

The oclTypeCast() query shown in Listing 11 is employed to generate the <type-cast> fragment of the OCL expression. In the first part of the query, we create the type-cast that consists of an invocation of the oclAsType() operation; the argument of this operation call is derived from the *type* of the current context *Property*, i.e., if the context *Property* is a collection type as indicated with an upper cardinality greater than one, an additional type-cast in the second part of the query (Lines 18 – 32) is appended, so that the collection matches the required collection type. For instance, the operation asSet() is employed for such an type-cast.

4.3.6 Additional OCL Constraints

Because the *type* of a stereotype attribute is recomputed according to Design Decision 17, we must introduce additional OCL constraints to preserve the static semantics defined by metamodel MM_{Domain}. Because of Design Decisions 20 and 21, we distinguish between two categories of OCL constraints. The first category is introduced to ensure the well-formedness of stereotype attributes that are not 'derived' and 'read-only'. By contrast, the second category is employed to ensure the well-formedness of UML metaclass attributes that are relevant for computing the *defaultValues* of 'derived' and 'read-only' stereotype attributes.

OCL constraints for stereotype attributes. As argued for Design Decision 20, a syntactically invalid value can be assigned to a stereotype attribute that is not defined as 'derived' and 'read-only'. Hence, we introduce OCL constraints to ensure that only UML elements having applied particular stereotypes can be assigned to such stereotype attributes. We determine these stereotypes as explained below.

Assume we are given an MC_{St} attribute att, and its *type* property refers to an MC_{St} metaclass `A`. According to our derivation approach, this attribute is mapped to a corresponding attribute att', and a *Stereotype* `A'` is derived from `A`. In addition, this stereotype has an *Extension* to a UML metaclass MCB. Furthermore, the *type*

```
query Property::oclSelectExp(relatedProp : Property) : String
  var oclString := "";
  if !self.type.mapsToStereotype() then {
    var uml_mc := getUmlTypeByName( self.type.name );
    oclString :=
      "oclIsKindOf("+ uml_mc.qualifiedName +")";
  }
  else if self.type.mapsToStereotype() then
    oclString := "isStereotypedBy('" + PROFILE_NAME +"::"+ self.type.name +"')";
  endif endif;
  return oclString;

query Property::oclTypeCast(relatedProp : Property) : String
  // 1. Determining the required type
  var oclString := "oclAsType(" + self.type.getNewType().qualifiedName +")";

  // 2. Determining the required type cast for collection types
  if self.upper > 1 ∨ self.upper = * then {
    switch {
      case (self.isUnique ∧ !self.isOrdered)
        oclString := oclString +"->asSet()"
      case (self.isUnique ∧ self.isOrdered)
        oclString := oclString +"->asOrderedSet()"
      case (!self.isUnique ∧ self.isOrdered)
        oclString := oclString +"->asSequence()"
      case (!self.isUnique ∧ self.isOrdered)
        oclString := oclString +"->asSequence()"
      else
        oclString := oclString +"->asBag()"
    };
  } endif;
  return oclString;
```

Listing 11: The queries $\mathrm{oclSelectExp}()$ and $\mathrm{oclTypeCast}()$

property of $\mathrm{att'}$ is recomputed so that it refers to MCB. Thus, any kind of MCB instance can be assigned to $\mathrm{att'}$; but $\mathrm{att'}$ is only well-formed if all elements assigned to it have $\mathrm{A'}$ applied. To ensure this, we create an OCL constraint for each stereotype attribute that is not defined as ‘derived’ and ‘read-only’. Such an OCL constraint consists of the following parts:

- Navigation to stereotype attribute $\mathrm{att\_st}$;
- Verification that all items of $\mathrm{att\_st}$ have stereotype ST applied.

Apart from the components above, the cardinality of the stereotype attribute must also be respected because different constraint kinds have to be used for single-valued and multi-valued attributes. Hence, we use the following patterns to generate OCL constraints. Pattern (1) is employed for a single-valued stereotype attribute, whereas Pattern (2) is used in the case of a multi-valued attribute.

```
(1) self.<att_st>.isStereotypedBy(<ST>)
(2) self.<att_st>->forAll(isStereotypedBy(<ST>))
```

OCL constraints for UML metaclass attributes. We also introduce a second category of OCL constraints for ensuring the well-formedness of UML metaclass attributes. The constrained attributes serve as computation basis for the *defaultValue* of 'derived' and 'read-only' stereotype attributes (Design Decision 21).

Suppose that the same excerpt of a metamodel MM_{Domain} (see Fig. 4.16) is given as employed to explain the generation of OCL expressions. After applying our derivation approach, we obtain a 'derived' and 'read-only' stereotype attribute $\mathrm{att\_a}$' that has an OCL expression as *defaultValue*. Moreover, an attribute $\mathrm{att\_mcb}$ of a UML metaclass MCB serves as basis for computing this OCL expression.

To preserve well-formedness, we have to ensure that only permitted elements can be assigned to metaclass attribute $\mathrm{att\_mcb}$. For the given example, only the assignment of elements with an applied stereotype `A'` is permitted. Hence, we introduce an appropriate OCL *Constraint* that constrains attribute $\mathrm{att\_mcb}$ accordingly.

Apart from the given example, a UML metaclass attribute can also be used to compute several stereotype attributes. This is always the case if several MC_{St} attributes in the source model MM_{Domain} are subsetting or redefining the same MC_{AC} attribute. In such a scenario, we obtain a set of valid stereotypes that can be applied to items of the attribute $\mathrm{att\_mcb}$, and therefore, these also must be verified via a generated OCL constraint.

Furthermore, the *type* properties of MC_{St} attributes can also refer to MC_{AMC} metaclasses. This implies that the *type* properties of the corresponding stereotype attributes refer to 'additional' metaclasses. In this case, we have to verify that the items assigned to the UML metaclass attribute $\mathrm{att\_mcb}$ are instances of 'additional' metaclasses. Consequently, an OCL *Constraint* for a UML metaclass attribute consists of the following parts:

- Navigation to attribute $\mathrm{att\_mcb}$ of UML metaclass MCB, which is extended by *Stereotype* ST;
- Verification that each item of $\mathrm{att\_mcb}$ has one of the permitted stereotypes applied, or whether it is an instance of an 'additional' metaclass.

We employ the following two patterns to generate an OCL *Constraint*. As in the case of the previous constraint category, we must obey the attribute cardinality here.

```
(1) self.base_<MCB>.<att_mcb>.<verify_exp>
(2) self.base_<MCB>.<att_mcb>->forAll(verify_exp)
```

For instance, we employ the second pattern to generate an OCL constraint for the nestedClassifier attribute of the «Statemachine» stereotype (see Fig. 4.12b). Thus, the resulting OCL constraint restricts the elements that can be contained in the nestedClassifier attribute to those with applied «Command» stereotype.

```
self.base_StateMachine.nestedClassifier
  −>forAll(isStrictStereotypedBy('Command'))
```

Implementation. We implement the generation of OCL constraints as discussed above using the two queries shown in Listing 12. The OCL specification for constraints that ensure the well-formedness of stereotype attributes are created by the genOclSpecsForExplicitePropConstraints() query, while those that capture UML metaclass attributes are generated by the genOclSpecsForSrcAttributeConstraints() query. Both queries are invoked from the toStereotype() mapping operation in Listing 4, which creates a *Constraint* element for each returned OCL specification.

Even if the queries generate different types of OCL specifications, they invoke the same set of queries to obtain the different parts of an OCL specification. Therefore, we first discuss both generation queries before we go into the details of the sub-queries invoked by them.

In the first step of the genOclSpecsForExplicitPropConstraints() query, we determine the set of MC_{St} metaclass attributes that are not mapped to a 'derived' and 'read-only' stereotype attribute. For each attribute of this set, the constraintSpec() query is then invoked to generate an OCL specification. Finally, we return all OCL specifications created in this way.

Based on the set of all MC_{AC} metaclasses from which the current context MC_{St} of the genOclSpecsForSrcAttributeConstraints() query inherits, we collect the set of MC_{AC} attributes needed for computing stereotype attributes (Lines 12 – 16). Hence, we only consider MC_{AC} attributes that are not defined as 'read-only' or 'derivedUnion', and which are redefined or subsetted by at least one attribute of MC_{St}. The selected attributes are stored in variable relSrcAttrs.

For each attribute contained in the variable, we invoke the associatedProperty() query to determine the set of all MC_{St} attributes that are subsetting or redefining the current attribute. The returned set is stored in variable assocProps, which is then passed to the constraintSpec() query in order to generate the OCL specification.

The OCL specifications for both *Constraint* categories are constructed by the constraintSpec() query in Listing 13. A difference between both categories exists only for the argument assocProps that has to be passed to the query. If an OCL specification is to be generated for a *Constraint* of a stereotype attribute, the current context *Property* of the query is also needed as argument. In contrast, all subsetting or redefining attributes of the current context *Property* are passed as argument in case of a *Constraint* for a UML metaclass attribute.

```
1  query Class::genOclSpecsForExplicitPropConstraints() : Set(String)
2    // The set of all explicit properties of a Stereotype
3    var props := self.ownedAttribute−>select(p| !p.isDerivedProperty());
4    var res := Set(String) := Set{};
5    props−>forEach(p) {
6      res := res−>including( p.constraintSpec(Set{p}) )
7    };
8    return res;
9
10 query Class::genOclSpecsForSrcAttributeConstraints() : Set(String)
11   // Collecting all MC_ac attributes that serve as computation base
12   var relSrcAttrs := self.allBaseMetaclasses().attribute−>select(a |
13         ( !a.isReadOnly ∨ !a.isDerivedUnion )
14      ∧ ( self.ownedAttribute.redefinedProperty−>includes(a)
15         ∨ self.ownedAttribute.subsettedProperty−>includes(a) )
16   )−>asSet();
17   var res := Set(String) := Set{};
18   relSrcAttrs−>forEach(ra) {
19     var assocProps := ra.associatedProperties(self);
20     if assocProps−>isEmpty() then
21       continue endif;
22     res := res−>including( ra.constraintSpec(assocProps) );
23   };
24   return res;
25
26 query Property::associatedProperties(mc_st : Class) : Set(Property)
27   var classes := mc_st.allParents()−>select(c|
28     c.mapsToStereotype() )−>including(mc_st);
29   return classes.ownedAttribute−>select(a|
30     ( a.subsettedProperty−>includes(self) ∨ a.redefinedProperty−>includes(self) )
31     ∧ a.isDerivedProperty()
32   )−>asSet();
```

Listing 12: Queries for creating OCL constraints on *Stereotypes*

In the first part of the query, we create the OCL expression fragment that is required for navigating to the attribute that shall be constrained. We obtain this fragment by calling the oclNavigationExp() query. The OCL fragments for verifying the type-conformance of attribute items are collected and stored in the typeChecks variable in the second part. To obtain these fragments, we invoke query oclTypeVerificationExp() for each item of argument assocProps.

Depending on the cardinality of the current context *Property* of the constraintSpec() query, we apply two different generation patterns, and distinguish two cases in the third part of the query. In the first case (Line 14), we apply the generation pattern for a single-valued attribute; in the second case (Line 18), we use the pattern for multi-valued attributes.

All fragments stored in the typeChecks variable are assembled into a single OCL expression by employing QVT's predefined joinfields() operation. The first

```
query Property::constraintSpec(assocProps : Set(Property)) : String
  // 1. Creating the attribute navigation expression
  var nav_exp := self.oclNavigationExp();

  // 2. Collecting the verification expressions
  var typeChecks : Dict(String, String);
  assocProps−>forEach(ap) {
    var selection := ap.oclTypeVerificationExp();
    typeChecks−>put(selection, selString);
  };

  // 3. Assembling the OCL specification
  var res : String;
  if (self.upper = 1) then
    res := nav_exp + "␣<>␣null␣implies\n"
      + typeChecks−>values()−>joinfields("␣or␣"+nav_exp+".", nav_exp+".", "")
  else
    res := nav_exp + "->notEmpty()␣implies\n"
      + nav_exp + "->forAll("
      + typeChecks−>values()−>joinfields( "␣or␣", "", ")" )
  } endif;
  return res;

query Property::oclNavigationExp() : String
  if self._class.mapsToStereotype() then
    return "self." + self.name endif;

  return if !self._class.mapsToStereotype() then
    "self.base_"
    + self.getExtensionTargets()−>any(true).name
    + "." + self.name
  else "";

query Property::oclTypeVerificationExp() : String
  return if self.type.mapsToStereotype() then
    "isStereotypedBy('"
    + PROFILE_NAME + self.type.name + "')"
  else
    "oclIsKindOf("
    + self.type.getNewType().qualifiedName + ")"
  endif;
```

Listing 13: The queries constraintSpec(), oclNavigationExp() and oclTypeVerificationExp()

parameter of this operation defines the separator for linking two list elements. The second parameter specifies the initial string, while the end string is defined by the third parameter. We use the joinfields() operation for prepending the navigation expression to each verification expression contained in the typeChecks variable. If more than one verification expression is obtained, all expressions are linked together to a single verification expression by inserting 'or' operators.

A particular navigation expression is required to access a metaclass or stereotype attribute. This expression is generated using the oclNavigationExp() query in Listing 13, which creates two different kinds of navigation expressions. If the owner of the context *Property* is an MC_{St} metaclass, we create an expression to access a stereotype attribute; otherwise, we generate an expression to access a metaclass attribute. We always assume that a stereotype instance is used as a starting point for a navigation expression.

Our oclTypeVerificationExp() query in Listing 13 creates the operations call expression used to verify the type conformance of the items of a metaclass or stereotype attribute. For a context *Property* that has an MC_{St} metaclass as owner, we generate a call expression to invoke the user-defined isStereotypedBy() operation. In all other cases, we return a call expression for the predefined operation oclIsKindOf(). The isStereotypedBy() operation verifies whether a particular stereotype or one of its subtypes has been applied to a model element. In contrast, the oclIsKindOf() operation is used to check whether a model element is an instance of the specified metaclass or a subtype of that metaclass.

4.4 Approach for Deriving 'Additional' Metaclasses

As we claimed in the introduction to our overall approach and apart from a metamodel-based derivation of UML profiles, our approach can also be applied for deriving 'additional' metaclasses from the same metamodel MM_{Domain}. In the following, we provide details for generating these artefacts. Because the generation of additional metaclasses is part of the process for the derivation of a UML profile, we do not treat this aspect again, but refer the reader to Sec. 4.3.1.

First, we analyse the prerequisites for deriving 'additional' metaclasses, so we can specify further design decisions that supplement those given in Sec. 4.3.2. We then discuss our derivation approach in detail, by employing mapping operations that are specified as pseudocode and by illustrating the derivation on our running example of a state machine DSL.

Because the metaclasses of MM_{Domain} that are to be mapped to 'additional' metaclasses have to be labelled with the «ToMetaclass» stereotype, we cannot use the metamodel introduced in the previous section (see Fig. 4.12a). Instead, we assume that the metamodel shown in Fig.4.17a serves as an input to the mapping operations discussed in this section. The only difference between both metamodels is that metaclasses of the present version of metamodel MM_{Domain} have the «ToMetaclass» stereotype applied.

4.4.1 Design Decisions for the Derivation of Metaclasses

Based on the design decisions defined in Sec. 4.3.2, further design decisions for the derivation of 'additional' metaclasses are specified below. This is required because the existing design decisions primarily address rules for deriving a UML profile, but not the generation of 'additional' metaclasses that inherit from UML metaclasses. As we extend an existing metamodel by inheritance, we must conform to the rules specified by the MOF [121], but not to those concerning UML profiles [116].

Because our derivation approach is based on the assumption that 'Abstract Concepts' have 'matching' UML counterparts, MC_{AC} metaclasses of MM_{Domain} are not mapped (see Design Decisions 10 and 11). Hence, MC_{AMC} metaclasses inheriting from MC_{AC} metaclasses have to be mapped to 'additional' metaclasses that inherit from UML metaclasses. These metaclasses must be imported by MM_{Add} because they are not part of the metamodel MM_{Add}.

Design Decision 22 *All UML metaclasses that are extended by 'additional' metaclasses using inheritance relationships must be imported from metamodel* MM_{Add}.

For the same reason, references and relationships to 'Abstract Concepts' cannot be preserved for 'additional' metaclasses. Therefore, during the mapping of MC_{AMC} metaclasses, all their references to MC_{AC} metaclasses must be modified in such a way that corresponding UML metaclasses are referenced after the mapping. This modification must be applied to the *type* property of an *Operation* or *Property*. In addition, the *redefinedProperty* and *subsettedProperty* properties of a *Property*, as well as the *redefinedOperation* property of an *Operation*, have to be modified in case they refer to a feature of a MC_{AC} metaclass.

Design Decision 23 *Any kind of references and relationships of* MC_{AMC} *to* MC_{AC} *metaclasses of* MM_{Domain} *must be mapped in such a way that the corresponding 'additional' metaclasses of* MM_{Add} *refer to 'matching' UML metaclasses.*

Because metamodels are defined by means of language concepts of the MOF, they cannot contain or refer to *Stereotypes*. This is because *Stereotypes* are a language concept of the UML and not of the MOF. Thus, the *type* property of a *TypedElement* (i.e., a *Property* or *Parameter*) that defines a feature of an 'additional' metaclasses cannot refer to a *Stereotype*. Therefore, the *type* property of a feature of an 'additional' metaclass must be recomputed in the same way as we do for *Stereotypes*. Consequently, the *type* property refers to a UML metaclass that is extended by a *Stereotype* instead of referring to that *Stereotype*.

Design Decision 24 *The type property of a feature of an 'additional' metaclass must not refer to a Stereotype. Instead, the type shall refer to a UML metaclass extended by that Stereotype.*

The explained recomputation makes an assignment of invalid items to the attribute of an 'additional' metaclass possible. Therefore, we must introduce an OCL constraint for each metaclass attribute mapped from an MC_{AMC} attribute with an *type* property that refers to an MC_{St} metaclass. The constraint ensures that only items that have a proper *Stereotype* applied can be assigned to an attribute of an 'additional' metaclass.

Design Decision 25 *If the type property of an attribute of an 'additional' metaclass is recomputed to refer to a UML metaclass, an OCL Constraint must be introduced to ensure the well-formedness of that attribute. The created constraint shall verify that all items of the constrained attribute have the proper Stereotype applied.*

4.4.2 Derivation of 'Additional' Metaclasses

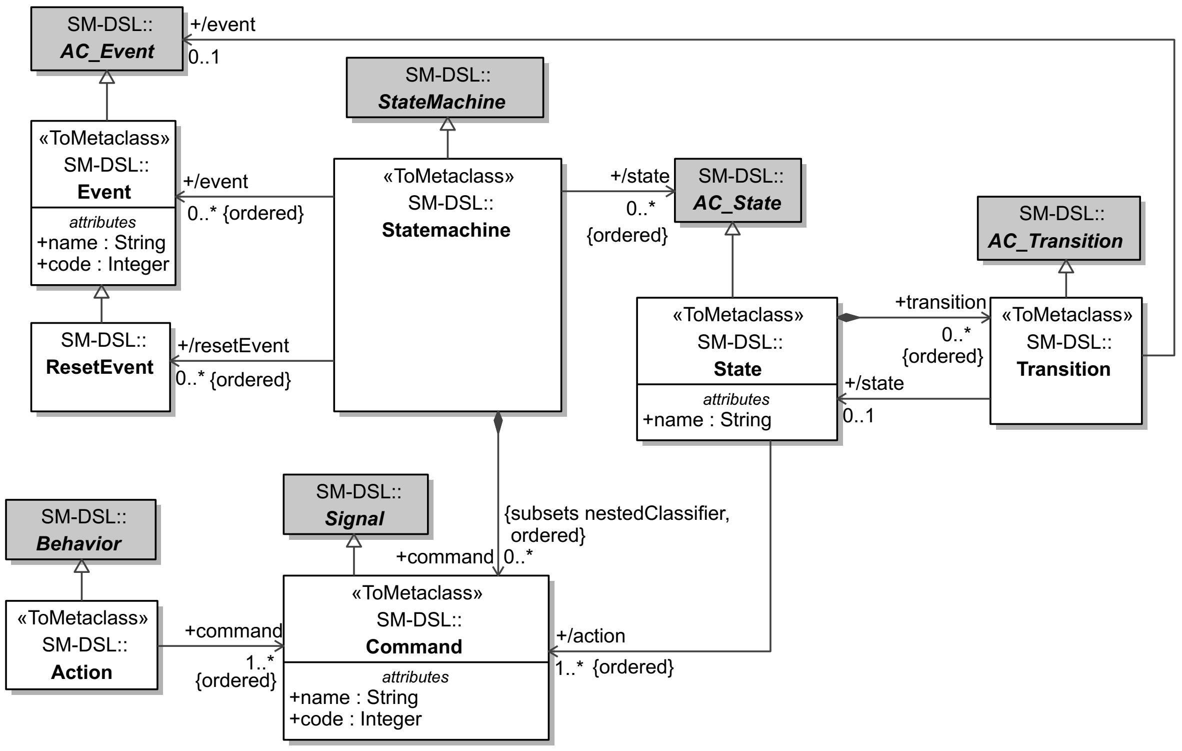
We now treat the details of our approach for deriving 'additional' metaclasses contained in the metamodel MM_{Add}. Because this derivation is captured by Step (B) of our overall approach, we use metamodel MM_{Domain} as input. However, this metamodel has to be enriched with «ToMetaclass» stereotypes, as shown in the example of the state machine DSL metamodel (see Fig. 4.17a).

The 'additional' metaclasses generated based on the enriched metamodel MM_{Domain} of our state machine DSL are shown in Fig. 4.17b. Only those metaclasses prefixed with 'AddMC::' represent 'additional' metaclasses, while all metaclasses with a 'UML::' prefix are imported from the UML metamodel.

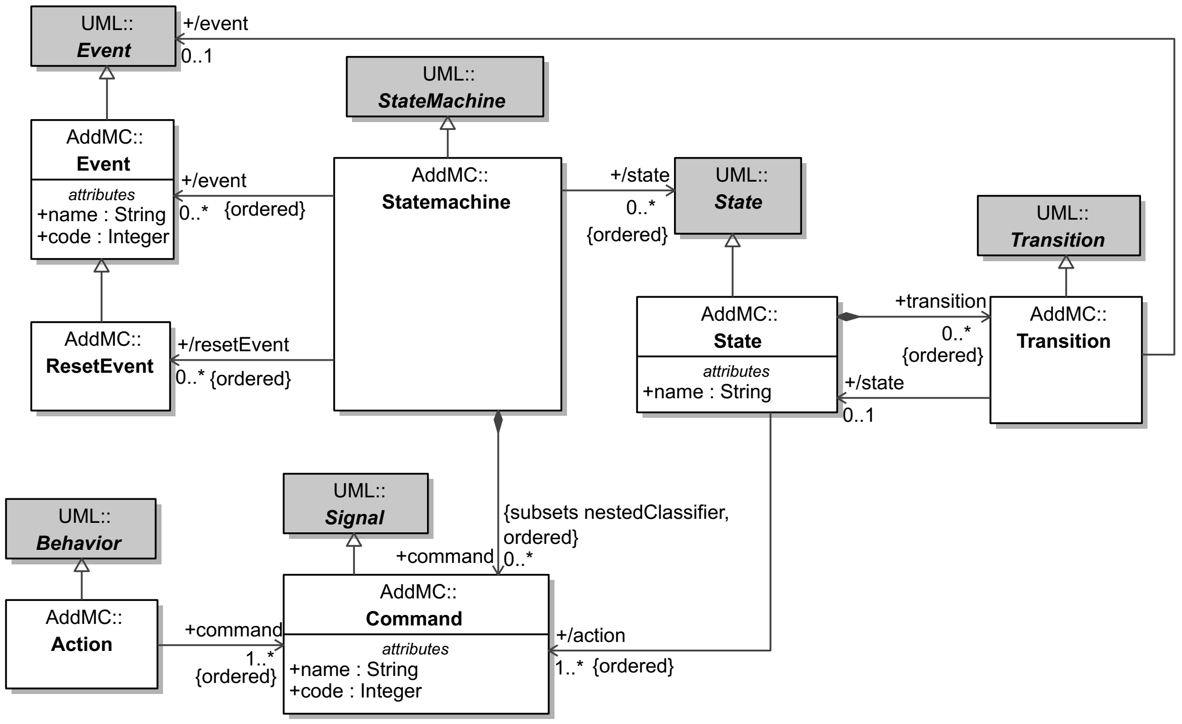
We use the enriched MM_{Domain} and the derived MM_{Add} metamodels to discuss the mapping rules for 'additional' metaclasses. Furthermore, the most important parts of our transformation $T_{MM\text{-}to\text{-}AddMC}$, which implements the derivation of 'additional' metaclasses, are explained using pseudocode. For this purpose, we apply the notational conventions introduced in Sec. 2.4.3.

Creating the MM_{Add} metamodel. According to the MOF [121], a metamodel is defined by means of a *Package* that contains all metaclasses, associations, and data types defining that metamodel. Hence, we first create a *Package* containing all those elements that are mapped from the metamodel MM_{Domain} to corresponding elements of the metamodel MM_{Add}.

Because we extend an existing metamodel – the UML metamodel – by means of further metaclasses, we have to import those metaclasses that are to be extended (Design Decision 22). Hence, we create an *ElementImport* relationship between each UML metaclass to be imported and the *Package* that represents the MM_{Add} metamodel. The relevant metaclasses of UML are determined either based on those MC_{AC} metaclasses that serve as super-classes of MC_{AMC} metaclasses, or



(a) A modified version of metamodel MM_{Domain} shown in Fig. 4.12a



(b) The 'additional' metaclasses that are derived from MM_{Domain} shown in Part (a)

Figure 4.17: A modified version of metamodel MM_{Domain} and the 'additional' metaclasses that are derived from it

based on the optional superClass attributes of the «ToMetaclasses» stereotype applied to MC_{AMC} metaclasses.

In the next step, we map all relevant elements of metamodel MM_{Domain} to corresponding items of metamodel MM_{Add}. These are MC_{AMC} metaclasses, *DataTypes* that are referred to by MC_{AMC} metaclass attributes, and *Associations* between MC_{AMC} metaclasses. Except for *DataTypes*, element types must be mapped in a particular manner because of our design decisions above. The required mappings are discussed next.

Mapping of 'DataTypes'. To determine the data types that are relevant for a mapping to the metamodel MM_{Add}, we first determine the set of all *DataTypes* contained in metamodel MM_{Domain}. Then, we remove all those data types that represent an 'Abstract Concept' or that shall become a part of the UML profile UP_{Add}. Finally, we map the remaining *DataTypes* one-to-one to corresponding items of metamodel MM_{Add}.

Mapping to 'additional' metaclasses. According to Design Decision 12, an MC_{AMC} metaclass of MM_{Domain} maps to an 'additional' metaclass of metamodel MM_{Add}. We employ the «ToMetaclass» stereotype to identify a metaclass of MM_{Domain} as an MC_{AMC} metaclass. However, not every metaclass that shall represent an MC_{AMC} metaclass must be qualified with this stereotype. Similarly to MC_{St} metaclasses which map to *Stereotypes*, it is sufficient that only the topmost MC_{AMC} metaclass within an inheritance hierarchy has the «ToMetaclass» stereotype applied, because we consider all subtypes of such a stereotyped metaclass to be MC_{AMC} metaclasses. For example, the ResetEvent metaclass shown in Fig. 4.17a inherits from the Event that has the «ToMetaclass» stereotype applied, so the ResetEvent metaclass is also considered to be an MC_{AMC} metaclass.

The mapping to 'additional' metaclasses is implemented by the mapping operation toAddMetaclass() in Listing 14. Only if the current context *Class* is an MC_{AMC} metaclass, it is mapped to an 'additional' metaclass in metamodel MM_{Add}. We verify this via the guard condition in Line 2. The mapping operation consists of the following parts:

Mapping of metaclass properties: In the first part, we map various properties of an MC_{AMC} metaclass one-to-one to their counterparts of an 'additional' metaclass.

Creation of inheritance relationships: The inheritance relationships of an 'additional' metaclass are created in the second part. Due to Design Decisions 22 and 23, a one-to-one mapping of these relationships is impossible. Therefore, we invoke the addMcSuperClasses() query to determine all super-types of an 'additional' metaclass. Because this query either returns UML metaclasses or MC_{AMC} metaclasses, we distinguish two cases in Lines 12 to 17.

```
Class ⇒ toAddMetaclass() : Class
when { self.mapsToAdditionalMetaclass() }
  // 1. Mapping of metaclass properties
  result.name := self.name;
  result.isAbstract := self.isAbstract;
  result.ownedComment := self.ownedComment−>map toComment();
  result.ownedRule := self.ownedRule−>map toConstraint();
  result.ownedAttribute := self.ownedAttribute−> map toProperty();
  result.ownedOperation := self.ownedOperation−>map toOperation();

  // 2. Creating inheritance realitionships
  self.addMcSuperClasses−>forEach(sc) {
    if sc.package.name = 'UML' then
      result.superClass += sc
    else
      result.superClass += sc.map toAddMetaclass() endif;
  };

  // 3. Adding additional OCL 'Constraints'
  result.ownedRule := self.ownedRule−>map toConstraint();
  self.genOclSpecForTypeConstraints()−>forEach(spec) {
    result.ownedRule += object Constraint {
      name := "Constraint_" + result.ownedRule−>size()+1.repr();
      language := "OCL";
      body := spec;
    };
  };

query Class::addMcSuperClasses() : Set(Class)
  // 1. Determining explicit specified UML super−classes
  if self.isStereotypedBy("ToMetaclass") then
    return self.getExplicitSuperClass() endif;

  // 2. Determing super−classes based on the 'superClass' property
  var res : Set(Class) := Set{};
  self.superClass−>forEach(sc) {
    if sc.isAbstractConcept() then
      res += getUmlTypeByName(sc.name).asType(Class)
    else
      res += sc endif;
  };
  return res;
```

Listing 14: Mapping operation toAddMetaclass() and the addMcSuperClasses() query

In the first case, we assign the returned UML metaclasses directly to the *superClass* property of an 'additional' metaclass; in the latter case, the returned MC_{AMC} metaclasses must be mapped first before their assignment is possible.

Creation of additional OCL constraints: In the last part, we introduce additional OCL *Constraints* as required by Design Decision 25. The OCL specifications for these constraints are generated by the query genOclSpecForTypeConstraints(). Based on each of these specifications, we create a *Constraint* element that is added to the existing items of the *ownedRule* property of an 'additional' metaclass. The details on creating these OCL specifications are treated later in this section.

We use the query addMcSuperClasses() shown in Listing 14 to determine the direct super-classes of an 'additional' metaclass. This is based either on the *superClasses* attribute of an applied «ToMetaclass» stereotype, or on the *superClass* property of an MC_{AMC} metaclass. In the first case, only UML metaclasses are returned, while the result of the second case can also contain MC_{AMC} metaclasses. For instance, when the addMcSuperClasses() query is invoked for the MC_{AMC} metaclass Event (see Fig. 4.17a), the UML metaclass Event is returned as result. By contrast, the MC_{AMC} metaclass Event is obtained for an application of the query to the MC_{AMC} metaclass ResetEvent.

Mapping to metaclass attributes. Because we map MC_{AMC} metaclasses to 'additional' metaclasses of the metamodel MM_{Add}, their attributes must also be mapped to corresponding counterparts. As the source and target elements of this mapping are of type *Property*, most of the properties of a metaclass attribute can be mapped one-to-one. However, the properties *type*, *subsettedProperty*, and *redefinedProperty* are exceptions.

As argued for Design Decision 23, the *subsettedProperty* and *redefinedProperty* of a metaclass attribute can refer to MC_{AC} metaclass attributes. However, because we assume that 'matching' UML counterparts exist for MC_{AC} metaclasses, we cannot preserve such relationships after the mapping. Hence, we recompute the *subsettedProperty* and *redefinedProperty* during the mapping, so they refer to UML metaclass attributes afterwards. For instance, the command attribute of the MC_{AMC} Statemachine shown in Fig. 4.17a is subsetting the nestedClassifier attribute of StateMachine. After applying our mapping, the corresponding command attribute shown in Fig. 4.17b is subsetting the nestedClassifier attribute of the UML metaclass StateMachine.

According to Design Decision 24, another kind of recomputation is required if the *type* property of an MC_{AMC} attribute refers to an MC_{St} metaclass. We recompute this property in such a way that it refers to a UML metaclass that is extended by a *Stereotype*. Thereby, we ensure that a derived metamodel MM_{Add} is compli-

```
Property ⇒ toProperty() : Property
  // 1. Mapping of metaclass properties
  result.name := self.name;
  result.aggregation := self.aggregation;
  result.lower := self.lower;
  result.upper := self.upper;
  result.isUnique := self.isUnique;
  result.isOrdered := self.isOrdered;
  result.ownedComment := self.ownedComment;
  result.isReadOnly := self.isReadOnly;
  result.isDerived := self.isDerived;
  result.isDerivedUnion := self.isDerivedUnion;

  // 2. Recomputing the 'type' property
  result.type := self.type.getNewType();

  // 3. Recomputing the 'redefinedProperty'
  self.redefinedProperty−>forEach(rp) {
    if rp._class.isAbstractConcept() then
      result.redefinedProperty += getUmlProperty()
    else
      result.redefinedProperty += rp.map toProperty() endif;
  };

  // 4. Recomputing the 'subsettedProperty'
  self.subsettedProperty−>forEach(sp) {
    if sp._class.isAbstractConcept() then
      result.subsettedProperty += getUmlProperty()
    else
      result.subsettedProperty += rp.map toProperty() endif;
  };

query Property::getUmlProperty() : Property
  var umlCls := getUmlTypeByName( self._class.name ).asType(Class);
  return umlCls.ownedAttribute−>any(a| a.name = self.name )
```

Listing 15: Mapping operation toProperty() and the getUmlProperty() query

ant to the MOF, because metamodels are not permitted to contain any kind of references to *Stereotypes*.

Due to Design Decision 23, a further recomputation of the *type* property of an MC_{AMC} attribute is required if it refers to an MC_{AC} metaclass. We perform this recomputation in such a manner that the *type* property of the mapped attribute refers to a UML metaclass that 'matches' the MC_{AMC} metaclass. An example of the recomputation of an attribute whose *type* property refers to an MC_{AC} metaclass is the event attribute of the MC_{AMC} metaclass AC_Statemachine shown in Fig. 4.17a. After the mapping, the attribute's *type* refers to the UML metaclass Event (see Fig. 4.17b).

We implement the mapping of MC_{AMC} attributes to corresponding counterparts of 'additional' metaclasses by the toProperty() mapping operation in Listing 15, which consists of four processing steps. In the first step, we process all those properties of an MC_{AMC} attribute that can be mapped one-to-one. Then, we recompute the value of the *type* property by invoking the getNewType() query in Line 15, so we can assign it to the mapping target afterwards.

In the third and fourth step (see Lines 18 – 31), we process the *redefinedProperty* and *subsettedProperty* properties of a metaclass attribute according to Design Decision 24. Hence, we verify whether an item of these attribute properties belongs to an MC_{AC} metaclass; if so, the getUmlProperty() query is invoked to determine the corresponding attribute of the 'matching' UML metaclass. Then, we assign this attribute to the corresponding property of the 'additional' metaclass attribute. In contrast, if an item of the *redefinedProperty* or *subsettedProperty* properties of an MC_{AMC} attribute belongs to an MC_{AMC} metaclass, we can directly map this item to an 'additional' metaclass attribute.

Mapping of 'Associations'. The specification of an *Association* between two metaclasses implies the creation of two *Property* elements that are referenced by the *memberEnd* property of an *Association*. Depending on the *Association* type, an introduced *Property* is owned by the *Association* itself or an involved metaclass. In the latter case, the *Property* becomes an item of the *ownedAttribute* property of a metaclass. Therefore, the mapping operation for metaclass attributes can also be used for items of the *ownedEnd* property of an *Association*.

All remaining properties of an *Association* can be mapped one-to-one, because no relevant design decisions are defined for them.

Mapping of 'Operations'. The *Operations* for 'additional' metaclasses can be mapped in a similar manner as *Stereotypes*, but the processing of the *redefinedOperation* property differs. As argued for the Design Decision 23, references to features of MC_{AC} metaclasses cannot be preserved after the mapping. Therefore, when mapping an MC_{AMC} *Operation* that is redefining an MC_{AC} *Operation*, we have to determine a corresponding UML operation, so we can assign it to the *redefinedOperation* property of the mapped *Operation* afterwards. Thus, a mapped *Operation* of an 'additional' metaclass only references features of UML metaclasses, instead of referring to those of MC_{AC} metaclasses.

As in the case of an MC_{AMC} attribute, the *type* property of an operation *Parameter* can refer to an MC_{AMC}, MC_{St}, or MC_{AC} metaclass of MM_{Domain}. Because of the same reasons as for MC_{AMC} attributes, we have to recompute the *type* property of an operation *Parameter* during the mapping, and therefore we can apply the same recomputation as for MC_{AMC} attributes.

The toOperation() mapping operation in Listing 16 implements the mapping of an *Operation*. In the first step, we map all properties of an *Operation* for which

```
1  Operation ⇒ toOperation() : Operation
2    // 1. Mapping of metaclass properties
3    result.name := self.name;
4    result.isQuery := self.isQuery;
5    result.ownedComment := self.ownedComment;
6    result.bodyCondition := self.bodyCondition.map toConstraint();
7    result.ownedParameter := self.ownedParameter−>map toParameter();
8
9    // 2. Recomputing the 'redefinedOperation'
10   self.redefinedOperation−>forEach(rop) {
11     if rop._class.isAbstractConcept() then
12       result.redefinedOperation += rop.getUmlOperation()
13     else
14       result.redefinedOperation += rop.map toOperation() endif;
15   };
16
17 query Operation::getUmlOperation() : Operation
18   var umlCls := getUmlTypeByName( self._class.name ).asType(Class);
19   return umlCls.ownedOperation−>any(op | op.name = self.name );
20
21 Parameter ⇒ toParameter() : Parameter
22   // 1. Mapping of metaclass properties
23   result.name := self.name;
24   result.direction := self.direction;
25   result.lower := self.lower;
26   result.upper := self.upper;
27   result.isOrdered := self.isOrdered;
28   result.isUnique := self.isUnique;
29
30   // 2. Recomputing the 'type' property
31   result.type := self.type.getNewType();
```

Listing 16: The mapping operations toOperation() and toParameter() for processing $\mathrm{MC}_{\mathrm{AMC}}$ operations

no recomputation is required; in the second, step the *redefinedOperation* property is mapped or recomputed.

If an item of the *redefinedOperation* property refers to a *MC_{AC} Operation*, we employ the getUmlOperation() query in Line 12 to obtain the corresponding operation of a UML metaclass. Then, we assign this *Operation* to the *redefinedOperation* property of the mapped *Operation*. In contrast, if an item of the *redefinedOperation* property refers to an *MC_{AMC} Operation*, we can directly map and assign this item (Line 14).

We employ the toParameter() mapping operation in Listing 16 to map *Parameters* of an *MC_{AMC} Operation*. The mapping is realized with two processing steps. In the first step, all properties of an *Operation* are mapped one-to-one, except for the *type* property. In the second step, we recompute and assign the *type* property by invoking query getNewType().

```
query Class::genOclSpecForTypeConstraints() : Set(String)
  var res : Set(String) := Set{};
  var attributes := self.ownedAttribute−>select(p|
    p.type.mapsToStereotype() );
  attributes−>forEach(a) {
    var stName := PROFILE_NAME + a.type.name;
    var oclString :=
      if a.upper = 1 then
          "self."+ a.name +"␣<>␣null␣implies\n"
        + "\tself."+ a.name + ".isStereotypedBy('"+ stName +"')"
      else
          "self."+ a.name + "->notEmpty()␣implies\n"
        + "self."+ a.name + "->forAll(isStereotypedBy('" + stName + "'))"
      endif;
    res := res−>including(oclString);
  };
  return res;
```

Listing 17: The genOclSpecForTypeConstraints() query

4.4.3 Additional OCL Constraints

Because we recompute the *type* property of an attribute of an 'additional' metaclass according to Design Decision 24, we introduce additional OCL *Constraints* to preserve the static semantics as defined by metamodel MM_{Domain}. As argued for Design Decision 25, OCL constraints have to be introduced only for those attributes of 'additional' metaclasses mapped from MC_{AMC} attributes that have a *type* property referring to an MC_{St} metaclass.

Assume we are given an MC_{AMC} attribute att, and its *type* property refers to an MC_{St} metaclass `A`. According to our derivation approach, this attribute is mapped to a corresponding attribute att', and a *Stereotype* `A'` is derived from `A`. In addition, this stereotype has an *Extension* to a UML metaclass MCB, and the *type* property of att' is recomputed so that it refers to the UML metaclass MCB. Due to this recomputation, any kind of MCB instance can be assigned to att', but the well-formedness of att' is only ensured if all elements assigned to it have the stereotype `A'` applied. To ensure this, we create an OCL constraint that consists of the following parts:

- Navigation to metaclass attribute att;
- Verification that all items of att have stereotype ST applied.

Furthermore, we must consider the cardinality of a metaclass attribute, because different kinds of constraints are required for single-valued and multi-valued attributes. Hence, we use the following patterns to generate OCL constraints:

```
(1) self.<att>.isStereotypedBy(<ST>)
(2) self.<att>−>forAll(isStereotypedBy(<ST>))
```

Pattern (1) is applied to single-valued stereotype attributes, whereas Pattern (2) is used in the case of a multi-valued attribute.

The OCL specifications for all additional *Constraints* of an 'additional' metaclass are generated by the genOclSpecForTypeConstraints() query shown in Listing 17. In the first part of this query, the set of all MC_{AMC} attributes with a *type* referring to an MC_{St} are collected. Then, we create an OCL specification for each collected attribute, by applying one of the generation patterns above. Finally, the set of all generated OCL specification is returned as result.

4.5 Update of Existing OCL Expressions

A metamodel includes OCL expressions for different purposes. On the one hand, OCL expressions can be used to define *Constraints* on metaclasses, so that the static semantics of a computer language or DSL is captured. On the other hand, OCL expressions can serve as the basis for computing values of metaclass attributes and operations at runtime.

The automatic transfer of the static semantics of a metamodel MM_{Domain} to a derived UML profile UP_{Domain} is a key feature of our approach. Because *Stereotypes* and their extended UML metaclasses exist as separate instances in a model, an automatic transfer of OCL expressions from a metamodel to a UML profile is impossible without updating them at the same time.

Before we can derive metamodel MM_{Domain}, we must update its OCL expressions. As argued in the Sec. 4.3.1, we do not consider the OCL update as a separate process but as an action of our UML profile derivation process. In contrast to the processing order, we first treated the derivation of UML profiles, so we can comprehensibly discuss the OCL update in the current section.

First, we give a brief overview of the OCL metamodel as a basis for specifying our design decisions for the OCL update. Then, we discuss the details of our approach for updating OCL expressions, and finally, we detail the most important aspects of our implementation.

4.5.1 The OCL Metamodel

We briefly introduce the relevant parts of OCL's abstract syntax, which is defined by a MOF-compliant metamodel [120]. The OCL metamodel is divided into several packages, and the most important of them are the following ones:

- The 'Expressions' package, which specifies the different OCL expression types;
- The 'Types' package defines OCL's type system, which includes concepts for using pre-defined types as well as user-defined types introduced by a metamodel.

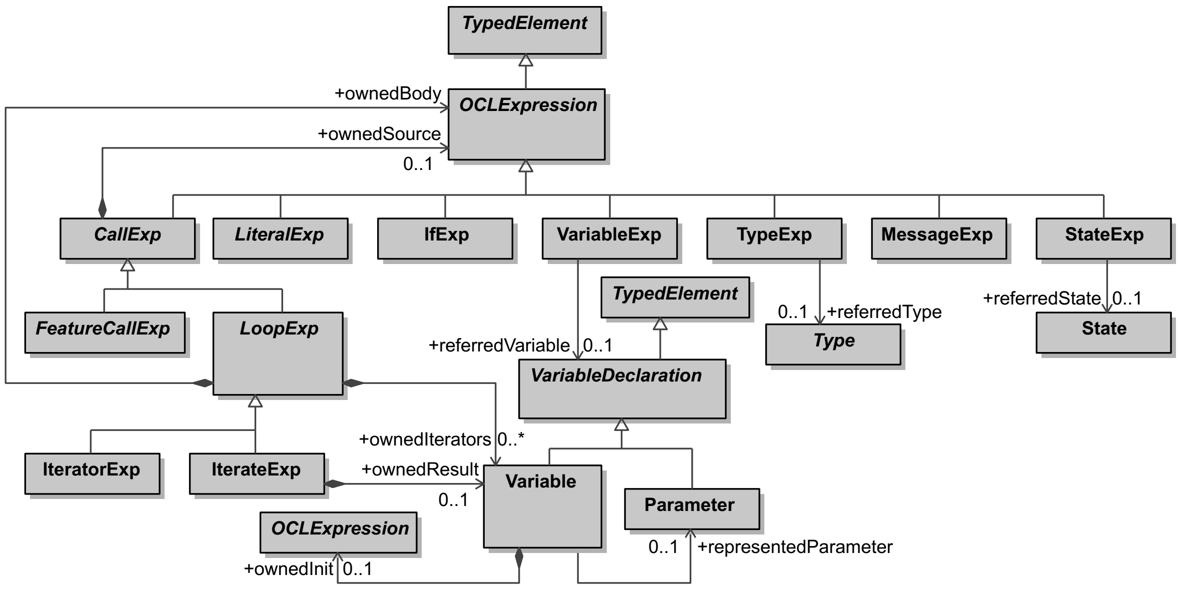


Figure 4.18: Basic structure of the abstract syntax of OCL[8]

Because only the OCL metaclasses that define the abstract syntax of OCL expressions are of interest for our OCL update, we only treat some metaclasses[7] of the 'Expressions' package. The basic structure of the abstract syntax of OCL[8] expressions is shown in Fig. 4.18.

OCL is a 'typed language' and, therefore, the most general metaclass *OCLExpression* inherits from the *TypedElement* so all OCL expressions own the *type* property. Typically, an abstract syntax tree (AST) of an analysed OCL expression consists of any number of nested OCL expression instances. Each of these instances has a static type specified, which can usually be determined automatically by a recursive analysis of all nested expressions. The result value of an expression is obtained by performing an evaluation, and the value thus determined must conform to the static type of the expression.

The objectives of the metaclasses shown in Fig. 4.18 are explained below. Only the metaclasses *MsgExp* and *StateExp* are excluded, because they cannot be used for OCL expressions contained in metamodels.

CallExp: A *CallExp* is employed to obtain the evaluation result of an operation or attribute of a classifier in a model. Moreover, it can be used to determine the evaluation result of a predefined iterator for a collection type. The exact purpose of a *CallExp* is defined by its concrete subtypes. A *CallExp* always has an *ownedSource* expression, whose static type determines which classifier or collection type shall be used for a *CallExp*.

[7] We refer to OCL metaclasses and their features by employing the same notation (i.e., names in italics) as for UML metaclasses and concepts of the MOF.

[8] Because the figures concerning the abstract syntax of OCL are derived from the Pivot OCL metamodel, minor differences compared to the standard OCL metamodel exist.

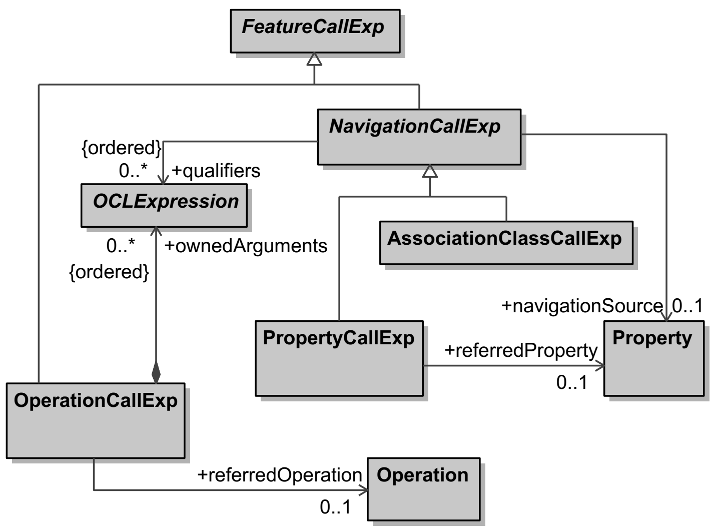


Figure 4.19: Abstract syntax of OCL's *FeatureCallExp*[8]

FeatureCallExp: The evaluation results of an attribute or operation of a classifier in a model are obtained by invoking a *FeatureCallExp*. This abstract OCL metaclass is more precisely defined by further subtypes; see below.

LiteralExp: A literal of a primitive type such as Integer or String is represented by a *LiteralExp*, and the specified literal is also returned as evaluation result.

TypeExp: A *TypeExp* refers to a classifier in a model or a predefined type. Typically, a *TypeExp* is passed as argument to invoke one of the predefined OCL operations oclAsType(), oclIsTypeOf(), or oclIsKindOf().

VariableDeclaration: The metaclass *VariableDeclaration* is a super-type of the two metaclasses *Variable* and *Parameter*. The metaclass *Variable* represents user-defined variables, whereas variables introduced by operation parameters are represented by the metaclass *Parameter*.

VariableExp: A *VariableExp* is a reference to an explicitly declared variable, an operation parameter, or the implicitly introduced variables 'self' and 'result'.

LoopExp: A *LoopExp* is used to iterate over all items of a collection type. The invocation of a predefined collection iterator is represented by *IteratorExp*, which is a concrete subtype of *LoopExp*. The *IteratorExp* is another concrete subtype of *LoopExp*, and it can be applied if the predefined collection iterators are not appropriate to define a specific loop construct.

IfExp: An *IfExp* always consists of two mandatory alternative expressions and a Boolean condition. Depending on the condition result, one of the alternative expressions is evaluated and the obtained result is returned.

The subtypes of the abstract OCL metaclass *FeatureCallExp* are shown in Fig. 4.19. Depending on the subtype, a *NavigationCallExp* represents a reference

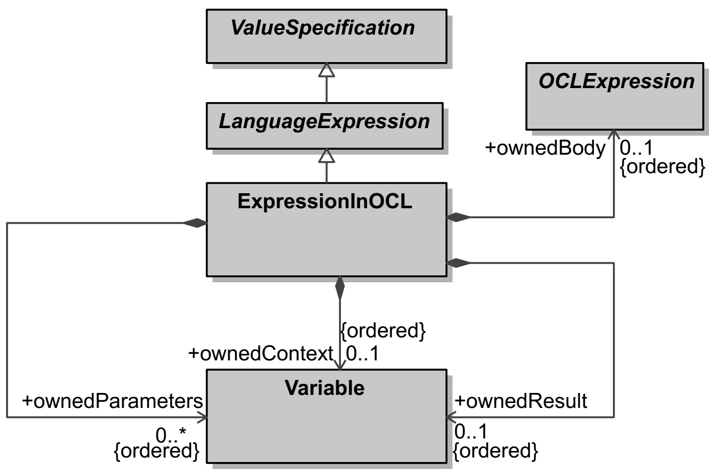


Figure 4.20: The OCL *ExpressionInOcl* metaclass[8]

to a classifier attribute or an association class. Because the latter element type is not applicable to OCL expressions of metamodels, we do not consider the metaclass *AssociationClassCallExp* further. The *PropertyCallExp* is used to evaluate the value of a classifier attribute, and the *referredProperty* property is the reference to this attribute.

Another subtype of the *FeatureCallExp* metaclass is the *OperationCallExp* that is used to evaluate the result of an operation invocation. The *referredOperation* property of this metaclass identifies the operation to be invoked, and the items of the *ownedArguments* property define the values that shall be passed to the operation's parameters.

Because the *OCLExpression* metaclass inherits from *TypedElement* instead of *ValueExpression*, a parsed OCL expressions cannot be part of a model. Therefore, the *ExpressionInOCL* serves as container for parsed OCL expressions, so the embedding in a model becomes possible. Thus, *Classifiers* defined by a metamodel can be accessed during the evaluation of an OCL expression.

As shown in Fig. 4.20, an *ExpressionInOCL* embraces an *ownedContext*, an *ownedResult* and an ordered set of *ownedParameters*, and the values of these properties are represented as *Variable* instances. The *ownedContext* property of an *ExpressionInOCL* always points to the *Classifier*, whereas the evaluation result of an OCL expression is referenced by the *ownedResult* property. While both properties are always present regardless of the expression context, the *ownedParameters* of an *ExpressionInOCL* are defined only when the *ownedContext* property refers to an *Operation*.

4.5.2 Design Decisions for Updating OCL Expressions

Based on our design decisions for deriving UML profiles specified in Sec. 4.3.2, we map MC_{St} metaclasses of the metamodel MM_{Domain} to corresponding *Stereotypes* of the UML profile UP_{Domain}, and MC_{St} attributes are mapped to stereotype attributes. Furthermore, we map *Operations* and *Constraints* of MC_{St} metaclasses to corresponding counterparts in *Stereotypes*. As the result of the mappings, all elements of an MC_{St} are owned by *Stereotypes*. The 'self' variable of an OCL expression contained in a *Stereotype* always refer to this *Stereotype* instead of a metaclass. If a feature of the extended UML metaclass shall be accessed based on the 'self' variable, the *'base_<metaclass>'* attribute must be accessed first when navigating to that metaclass feature.

Hence, we must update all OCL expressions that access an MC_{AC} attribute or MC_{AC} *Operation* based on a 'self' variable that refers to an MC_{St}. During this update, we introduce a *PropertyCallExp* that accesses the attribute *'base_<metaclass>'*.

Design Decision 26 *A PropertyCallExp that accesses the 'base_<metaclass>' attribute shall be introduced for OCL expressions that access an MC_{AC} attribute or MC_{AC} Operation via a 'self' variable that refers to an MC_{St}.*

The *type* properties of attributes are recomputed during the mapping, so that only *Classifiers* (*Classes* or *DataTypes*) of the MM_{UML} or MM_{Add} metamodel are referenced (see Design Decision 18). Moreover, the same recomputation is applied to the *type* property of an operation *Parameter*. Thus, neither the *type* properties of stereotype attributes nor those of operation parameters refer to *Stereotypes*. Due to Design Decision 24, this situation also applies to attributes and operation parameters of 'additional' metaclasses of MM_{Add}. Thus, we can conclude that accessing to mapped stereotype attributes or operation parameters always returns a metaclass instance.

If a *Property* or *Operation* of a *Stereotype* shall be accessed based on the result returned by an OCL expression, then the *'extension_<stereotype>'* attribute has to be employed for navigating to the stereotype feature. For this reason, we update all OCL expressions that contain a navigation from one MC_{St} feature to another MC_{St} feature, and during this update we introduce a *PropertyCallExp* that accesses the *'extension_<stereotype>'* attribute.

Design Decision 27 *A PropertyCallExp that accesses the 'extension_<stereotype>' attribute shall be introduced for OCL expressions containing a navigation from one MC_{St} feature to another MC_{St} feature.*

The predefined OCL operations oclIsTypeOf() and oclIsKindOf() are used to verify whether the evaluation result of an OCL expression is an instance of the *Classifier* defined as argument. The oclIsTypeOf() operation returns only value 'true', if the evaluation result exactly matches the type specified as argument, while the oclIsKindOf() operation returns 'true' also for subtypes.

Because the *type* property of stereotype attributes and operation parameters is recomputed during the mapping, OCL expressions that access these elements can never return a stereotype instance as result. Thus, we cannot utilize the aforementioned operations to verify whether a particular stereotype is applied. Instead, we employ the operations isStrictStereotypedBy() and isStereotypedBy(). We update an *OperationCallExp* that invokes the oclIsTypeOf() or oclIsKindOf() operation, passing an MC_{St} metaclass as argument. During this update, we introduce a new *OperationCallExp* that invokes the operation isStrictStereotypedBy() or isStereotypedBy().

Design Decision 28 *An OperationCallExp that invokes the* oclIsTypeOf() *or* oclIsKindOf() *operation, passing an* MC_{St} *as argument, shall be replaced by an OperationCallExp that invokes the operation* isStrictStereotypedBy() *or* isStereotypedBy().

The elements of metamodel MM_{Domain} are located in UP_{Domain}, MM_{Add}, and MM_{UML} after the mapping. Because a *TypeExp* refers to a particular *Classifier* of a metamodel, we have to update all OCL expressions of that type, so they refer to elements in MM_{Add} or MM_{UML}. The *Stereotypes* of UP_{Domain} are not considered for this update, because the *type* properties of stereotype attributes and parameters are recomputed during the mapping so that *Stereotypes* are not referenced.

Design Decision 29 *All TypeExp shall be updated so that they refer to Classifiers of* MM_{UML} *or* MM_{Add}.

4.5.3 The OCL Update in Detail

We now detail our approach for updating OCL expressions contained in the metamodel MM_{Domain}. In our discussion, we assume that a textually specified OCL expression is parsed and the resulting abstract syntax tree (AST) is present. This AST is an instantiation of the OCL metamodel discussed above.

FeatureCallExp. As we argued for Design Decisions 26 and 27, an additional *PropertyCallExp* that refers to the *'base_<metaclass>'* or *'extension_<stereotype>'* attribute must be introduced for an existing *FeatureCallExp* in certain situations. However, this applies only to instances of *PropertyCallExp* and *OperationCallExp*, because only these subtypes of *FeatureCallExp* are applicable in the context of metamodels.

Based on Design Decision 26, all following criteria must be met by an existing *FeatureCallExp*, so we create an additional *PropertyCallExp* to access an *'extension_<stereotype>'* attribute:

1. The *type* of the *ownendSource* expression of a *FeatureCallExp* refers to an MC_{St} metaclass;
2. The *referredProperty* of a *PropertyCallExp* is an MC_{St} attribute, or in case of an *OperationCallExp*, the *referredOperation* is an MC_{St} operation.

Assume that the *PropertyCallExp* shown below is used as input for the OCL update, and the above criteria are met. After applying the update, we obtain the result shown in the second line, where the <stereotype> placeholder represents the name of the *Stereotype* that owns the *referredProperty*.

```
input: source.referredProperty
result: source.extension_<stereotype>.referredProperty
```

Due to Design Decision 27, another kind of update of an existing *FeatureCallExp* is required to access a UML metaclass feature based on a 'self' variable that refers to a *Stereotype* instance. We create an additional *PropertyCallExp* that refers to the *'base_<metaclass>'* attribute, so we can access the UML metaclass feature. We only conduct this update if all following criteria are met by an existing *FeatureCallExp*:

1. The *ownendSource* expression of a *FeatureCallExp* is a *VariableExp* that refers to the 'self' variable, and the *type* of this *VariableExp* expression refers to an MC_{St} metaclass;
2. The *referredProperty* of a *PropertyCallExp* is an MC_{AC} attribute, or in case of an *OperationCallExp*, the *referredOperation* is an MC_{AC} operation.

Analogously to above, assume that the *PropertyCallExp* shown below is used as input for the OCL update, and the above criteria are met. The result of this update is shown in the second line. The <metaclass> placeholder corresponds to the name of the UML metaclass that owns the *referredProperty*:

```
input: self.referredProperty
result: self.base_<metaclass>.referredProperty
```

OperationCallExp. The predefined OCL operations oclIsTypeOf() and oclIsKindOf() can be employed to determine whether the result type of an expression matches the expected type. According to Design Decision 28, an update of an *OperationCallExp* is required if one of the aforementioned operations is applied to an MC_{St} metaclass. We conduct this update as follows:

- The referred operation oclIsTypeOf() of an *OperationCallExp* is replaced by the isStrictStereotypedBy() operation, if the passed argument is an MC_{St} metaclass.
- The operation oclIsKindOf() that is referred by an *OperationCallExp* is replaced by the isStereotypedBy() operation, if the passed argument is an MC_{St} metaclass.

For example, suppose the two OCL expressions below are used as input for the update, and both expressions meet the above criteria. In Case (A), we introduce the operation isStrictStereotypedBy(), whereas the operation isStereotypedBy() is employed in Case (B). The placeholder <stereotype> represents the name of the *Stereotype* that is derived from the MC_{St} shown in the input expressions:

```
input:
(A) source.oclIsTypeOf(MCSt)
(B) source.oclIsKindOf(MCSt)

result:
(A) source.isStrictStereotypedBy(<stereotype>)
(B) source.isStereotypedBy(<stereotype>)
```

TypeExp. According to Design Decision 29, we have to update all occurrences of *TypeExp* that refer to an MC_{AC} or an MC_{St} metaclass. A *TypeExp* that refers to an MC_{AC} metaclass is updated so that the 'matching' UML counterpart of the MC_{AC} is referenced. Furthermore, a *TypeExp* that refers to an MC_{AMC} metaclass is updated so that it refers to an 'additional' metaclass.

4.5.4 Implementation Aspects

Because OCL is specified by means of a metamodel, one could argue that the 'update' of OCL expressions can be realized by employing an M2M transformation. Before we implemented the current approach, we considered such a solution and evaluated it as infeasible. Due to an existing problem of OCL Pivot, parsed OCL expressions can be stored as an OCL model. But when this model is loaded afterwards, not all referenced classes and data types can be resolved; however, this is a prerequisite for processing models by an M2M transformation. Because of the mentioned reason, we implement the OCL update by using an OCL parser and a pretty printer.

Before the UML profile UP_{Domain} is derived from the metamodel MM_{Domain}, we generate an AST for each OCL expression in MM_{Domain}. This AST consists of various types of nested OCL expressions. Every OCL expression of an AST is then visited by the pretty printer in order to perform the update. During this visit, the AST is converted back to its textual notation and the update is conducted.

4.6 Derivation of M2M Transformations

As discussed in the introduction of our overall approach (see Sec. 4.1), we can derive two different M2M transformations based on a single metamodel in order to transform a DSL model into a UML model and vice versa. Because stereotypes and model elements exist as separated instances in a UML model, each transformation can only be employed for a distinct unidirectional transformation. Hence, the derived transformation $T_{DM\text{-}to\text{-}UML}$ transforms a DSL model into a corresponding UML model, while the transformation $T_{UML\text{-}to\text{-}DM}$ covers the opposite direction.

First, we introduce an MDE-based process that captures all steps required to derive both M2M transformations. Thereafter, we define the design decisions that establish the foundations of our approach, and we justify the reasons for selecting a specific transformation language and the use of a particular technology to implement the derivation. Then, we introduce notational conventions for the pseudocode that is employed to discuss our approach in more detail thereafter.

4.6.1 Process for Deriving Model Transformations

As in the case of UML profiles, we extend the development process proposed in [145] for deriving model transformations to achieve model interoperability between UML models with applied UML profile and corresponding DSL models. The activity that defines the actions involved is shown in Fig. 4.21, where the metamodel MM_{Domain} enriched with meta-information as discussed in Sec. 4.3 is used as input.

Based on the enriched metamodel MM_{Domain}, both M2M transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ are derived in Action (1), which implements Steps (D) and (E) of our derivation approach (see Sec. 4.1.1). Afterwards, the obtained M2M transformations must be manually revised in Action (2). This is necessary because the main objective of our approach is the automated derivation of UML profiles, and therefore, not all information required to derive model transformations is stored in an enriched metamodel MM_{Domain}.

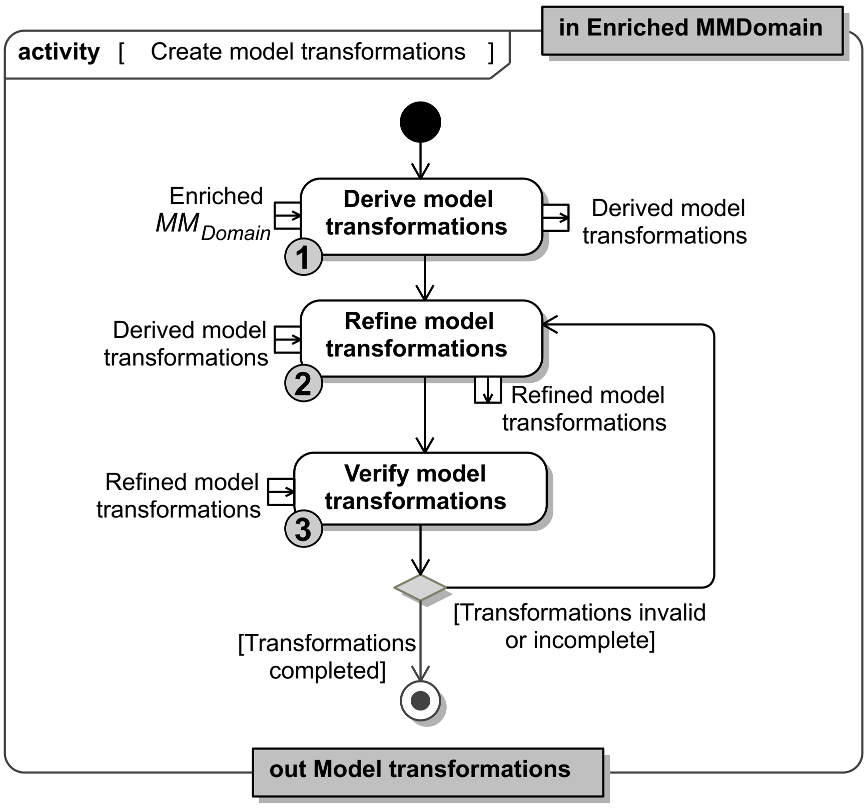

activity [Create model transformations]
in Enriched MMDomain
Enriched MMDomain
Derive model transformations
1
Derived model transformations
Derived model transformations
Refine model transformations
2
Refined model transformations
Refined model transformations
Verify model transformations
3
[Transformations invalid or incomplete]
[Transformations completed]
out Model transformations

Figure 4.21: Activity for deriving model transformations

The verification of the manually revised M2M transformations is conducted in Action (3), applying the approach discussed in Sec. 2.6.3. This is done in the next chapter, where we present two case studies.

4.6.2 Design Decisions for Deriving of Transformations

Because we can derive a UML profile based on a single metamodel, we assume that the same metamodel is employed to derive M2M transformations that transform a domain model to a UML model or vice versa. Due to the use of 'Abstract Concepts', which have 'matching' UML counterparts, we can derive the source elements and their corresponding mapping targets for an M2M transformation. However, this information is still insufficient for deriving M2M transformations based on a single metamodel.

In addition to a UML profile, we derive 'additional' metaclasses, but this requires additional meta-information in the metamodel MM_{Domain}. As argued in Sec. 4.3.2, we employ the UML profile 'MM2Profile' to enrich the metamodel MM_{Domain} with the required information. Such an enriched metamodel is required as input for deriving appropriate M2M transformations that support the mapping of UML *Stereotypes* and 'additional' metaclasses.

Design Decision 30 *A metamodel MM_{Domain} used as input for deriving M2M transformations must have the UML profile 'MM2Profile' applied.*

Stereotypes, UML metaclasses, and 'additional' metaclasses are contained in different *Packages*, resulting in different namespaces. Because the mappings of all elements are defined in a single M2M transformation, the elements to be mapped must be qualified by their namespaces, so that we can avoid name collisions and uniquely identify them.

Design Decision 31 *All elements that are to be mapped by a derived M2M transformation must be qualified by the use of namespaces.*

According to Design Decision 16, we map an MC_{St} attribute that is redefining or subsetting an MC_{AC} attribute to a 'derived' and 'read-only' stereotype attribute. The same applies to an MC_{St} attribute for which an alternative OCL specification has been defined manually (see Design Decision 19). Because of the semantics for metaclass attributes defined by the MOF, a value assignment for 'read-only' attributes is not supported. Moreover, the UML defines the same restriction for stereotype attributes. Thus, we cannot use value assignments for 'read-only' attributes in derived M2M transformations.

However, in the case of stereotype attributes that are derived from MC_{St} attributes that are subsetting or redefining MC_{AC} attributes, a value assignment to a UML metaclass attribute is sufficient. This is because, UML metaclass attributes are invoked by generated OCL expressions to compute the values of 'derived' and 'read-only' stereotype attributes at runtime according to our derivation approach. Therefore, the derived transformation $T_{DM\text{-}to\text{-}UML}$ shall map all MC_{AC} attributes that are redefined or subsetted by MC_{St} attributes to corresponding UML metaclass attributes.

Design Decision 32 *Transformation $T_{DM\text{-}to\text{-}UML}$ shall map all MC_{AC} attributes that are redefined or subsetted by MC_{St} attributes to corresponding UML metaclass attributes.*

This solution is not applicable to 'derived' and 'read-only' stereotype attributes that are derived from MC_{St} attributes having alternative OCL specifications defined. This type of MC_{St} attribute can be identified based on the applied «ToTaggedValue» stereotype, having defined a value for the oclSpecification attribute.

Because of the manually specified OCL expression, we cannot explicitly determine the UML metaclass attribute to be used for computing a stereotype attribute. Instead, the derivationSource attribute of the applied «ToTaggedValue» stereotype can be evaluated to obtain the required UML metaclass attribute. However, this information is not always sufficient, so we only propose a mapping when deriving $T_{DM\text{-}to\text{-}UML}$. This proposal has to be manually revised, if required.

Design Decision 33 *When deriving the transformation $T_{DM\text{-}to\text{-}UML}$, a mapping proposal shall be generated for MC_{St} attributes that have an alternative OCL specification.*

A UML model with an applied *Profile* consists of metaclass and *Stereotype* instances, and a *Stereotype* application is only possible for an already created model element. Hence, the transformation of a DSL model to a UML model has to be realized in two steps. In the first step, the elements of the UML model are created; in the second step stereotypes can be applied to these elements.

In contrast, the transformation of a UML model to a corresponding DSL model can be implemented in a single step, because no stereotypes have to be applied to elements of the DSL model.

Design Decision 34 *The transformation $T_{DM\text{-}to\text{-}UML}$ must consist of two processing steps. First, the elements of the UML model shall be created, so stereotypes can be applied to them afterwards.*

Even if a *DataType* and its subtypes *PrimitiveType* and *Enumeration* inherit from *Classifier*, they are represented differently at model level (M1). *DataTypes* of a metamodel are represented as dedicated instances, while literals are used for values of *PrimitiveTypes* and *Enumerations*. Therefore, both transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ have to transform *DataType* instances in a similar way as metaclass instances.

Design Decision 35 *The transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ shall transform DataType instances in a similar way as metaclass instances.*

Because, the values of *Enumerations* are represented in terms of literals, one could assume that the mapping of an attribute, whose *type* property refers to an *Enumeration*, could be realized by a simple assignment. However, this is infeasible as the source and target *Enumeration* are different entities. The *EnumerationLiteral* of an *Enumeration* can only be identified based on its *name*. Therefore, a model transformation must determine a target *EnumerationLiteral* that has the same *name* as the source element.

Design Decision 36 *The transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ shall determine a target EnumerationLiteral based on the name of the source EnumerationLiteral.*

Apart from the predefined *PrimitiveTypes* such as String or Integer, user-defined *PrimitiveTypes* can also be specified in metamodels. If the *type* property of a source attribute refers to one of the predefined types, the value of this attribute can be assigned directly to the corresponding target attribute.

In case of a user-defined *PrimitiveType*, the same type must be used in both the metamodel of the source model and that of the target model. For example, such a scenario can be realized by using a common model library. Because our UML profile derivation does not support model libraries, the derived model transformations can only support the predefined *PrimitiveTypes*.

Design Decision 37 *The transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ shall support the mapping of literal values of the predefined PrimitiveTypes.*

4.6.3 Target Language and Employed Technology

As the transformation languages of QVT [113] are standardized and supported by open source tools, we prefer them over other transformation approaches, as argued in Sec. 2.4.1. Because of the specific instantiation of stereotypes and metaclasses, we derive a dedicated M2M transformation for each transformation direction. Hence, the bidirectionality of QVTr is not advantageous for our transformations. In addition, Design Decision 34 requires that the transformation $T_{DM\text{-}to\text{-}UML}$ must be implemented as a two-staged transformation. Especially due to this reason, we assume that a fine-grained control on the workflow of a transformation is required, which can only be achieved by using QVTo. Hence, we employ this transformation language as target language for the M2M transformations generated by our derivation approach.

In principle, we can derive M2M transformations either using *higher-order transformations (HOT)* or M2T transformations. For instance, Tisi et al. [147] defines a HOT as follows:

> *"A higher-order transformation is a model transformation such that its input and/or output models are themselves transformation models."*

Typical application areas for HOTs are the synthesis, analysis, (de)composition, and modification of transformations [147]. However, only the transformation synthesis is of interest for the derivation of M2M transformations, because a HOT is employed to generate a transformation model based on a metamodel passed as input. Usually, a transformation model obtained by applying a HOT can be executed using a virtual machine or, if this is not applicable, it has to be serialized into the textual notation of a target language such as QVTo. For instance, we can implement such a serialization via a M2T transformation.

If we use a HOT to derive M2M transformations, the metamodels of UML and QVT, the metamodel MM_{Domain}, and the derived UML profile are required as input. In our opinion, this high number of required input models would lead to a rather complex HOT. Apart from this aspect, Design Decision 33 requires us to generate mapping 'proposals' that cannot be represented as a valid transformation model.

Due to the above drawbacks, HOTs are not applicable for implementing our derivation of M2M transformations; therefore, we use two M2T transformations instead of HOTs. We employ the standardized *MOF M2T Language (MTL)* [111] to implement both M2T transformations.

4.6.4 Common Concepts for Deriving the Transformations

Because of the design decisions specified above, the structure of both generated transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ differs. However, the key objective of both transformations is to map an entire input model to a corresponding output model. For this reason, we assume that a set of common concepts for generating both transformations can be identified. Apart from this aspect, QVTo offers various concepts for specifying transformations, so that their advantages and disadvantages should also be considered for defining our derivation approach.

In the following, we discuss and identify a number of general concepts that we wish to apply for deriving QVTo-based model transformations.

Discussion. According to the QVT specification [113], a transformation that is specified using the QVTo is referred to as *'operational transformation'*. Such a kind of transformation is formalized by *'operational mappings'* that define the mappings between source and target model elements. Typically, a mapping operation consists of a signature and an operation body that contains an ordered list of imperative expressions.

The reuse of existing mapping operations in an operational transformation is enabled by the merge and inheritance mechanisms of the QVTo. We use the latter mechanism to define that a mapping operation `B` inherits from another mapping operation `A`. When `B` is invoked, its body is processed after the body of `A`.

In contrast, we employ the merge mechanism of the QVTo to supplement a mapping operation with one or more other operations. When a supplemented mapping operation is invoked, its body is processed before the bodies of the mapping operations supplementing it. The QVT specification [113] argues that a high degree of modularization can be achieved using the merge mechanism. For example, the author of a model transformation can create a dedicated mapping operation for each atomic mapping rule specified in natural language. Then, these mapping operations can be merged to define a more complex one.

Because the manual creation of mapping operations for atomic mapping rules is not an objective of our derivation approach, we consider the use of the merge mechanism as inapplicable. In contrast, the reuse of mapping operations employing the inheritance mechanism could be of interest to our approach. In this case, we would generate a mapping operation for each metaclass of MM_{Domain}, where only the attributes of the associated metaclass are mapped, and the super-class attributes are processed by inherited mapping operations. Although the number of code lines of a transformation could be reduced using the inheritance mechanism, we cannot apply this mechanism for our model transformations because of Design Decisions 32 and 33. Due to the rules defined by these design decisions, inherited and owned attributes of a metaclass must always be processed by a single mapping operation.

Apart from the discussed mechanisms, a mapping operation can be specified in terms of a *'disjuncting mapping'*, which consists of an ordered list of references to mapping operations (also known as *'candidate mappings'*). This list is an optional part of the signature of a mapping operation. During the invocation of a disjuncting mapping, its list of candidate mappings is evaluated until one candidate mapping has been determined whose explicit and implicit constraints are met; then, the determined candidate mapping is invoked to compute the result of the disjuncting mapping. The explicit constraints of a mapping candidate are specified by its guard condition, while the implicit constraints are provided by its operation signature. In particular, the context element and all passed arguments must conform to the corresponding parameter types.

Provided that candidate mappings of disjuncting mappings have appropriate guard conditions defined, the execution order of mapping operations in an operational transformation can be controlled by disjuncting mappings. This is a prerequisite to implement the rules defined by Design Decision 34. Therefore, apart from mapping operations that have a non-empty operation body, our generated model transformations also use disjuncting mappings.

Common Concepts. The model transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ generated by us consist of two different types of mapping operations. We introduce mapping operations that have an operation body for defining the concrete mappings of metaclasses, while disjuncting mappings are created to control the execution order of the mapping operations contained in a derived transformation.

```
// (A) Mapping operation with a non−empty operation body
mapping <con−par>::<op−name>() : <res−par>
when { <guard−exp> }
{
  // Operation body with an ordered list of imperative expressions
}

// (B) Mapping operation that is a disjuncting mapping
mapping <con−par>::<op−name>() : <res−par>
disjuncts
  <candidate_A>,
  <candidate_B>,
  ...
  {}
```

Figure 4.22: Mapping operation signatures that are used for generated transformations

Operation signature: The signature of a mapping operation consists of several mandatory and optional parts, but only a few of these items are required for the mapping operations generated by our derivation approach. Based on the two examples given in Fig. 4.22, we briefly explain the parts of the operation signatures that are generated for both types of our mapping operations. Example (A) represents a mapping operation with a non-empty operation body, while a disjuncting mapping is shown in example (B). Regardless of the type of a mapping operation, the operation signature usually consists of a context parameter (<con-par>), the operation name (<op-name>), and a result parameter (<res-par>).

Two implicitly defined variables represent the current objects or values of the context and result parameters at runtime. In the operation body, the keyword **self** is used to access the context variable, and the result variable can be accessed via the keyword **result**. In addition to the parts of an operation signature employed by us, a signature can also define operation parameters; we do not require these because we determine all information for the mappings based on the context variables.

If not stated otherwise, we derive the various parts of a signature based on a metaclass of MM_{Domain} as follows. We use the pattern 'to<mc_name>' to derive the name of a mapping operation, where the placeholder <mc_name> is replaced by the *name* of a metaclass. According to Design Decision 31, all element types referenced in a mapping operation must be qualified. Thus, this also applies to the context and result parameters.

The placeholder <con-par> that represents the context parameter can be expanded as follows:

```
<par-dir> <namespace>::<type-name>
```

The <par-dir> defines the direction of a context parameter, which can be **in** or **inout**. A mapping operation with an **in** context parameter is applied out-place, i.e., a new model element is created based on the input model element. In contrast, a mapping operation with an **inout** context parameter is applied in-place, so that an input model element is only modified. Because most of our mapping operations map a model element to another, they have an **in** context parameter, unless stated otherwise.

A context parameter does not have a name, but its type must always be specified. The <namespace> placeholder represents the qualifier, and the <type-name> represents the name of the referenced element type. The value used to replace the <namespace> depends on the metamodel in which the referenced element type is contained. In case of the UML metamodel, we always use the statically defined String 'UML', whereas in all other cases the required value is determined based on the «MM2Profile» stereotype applied to MM_{Domain}. The value for placeholder <type-name> is derived from the name of a metaclass of MM_{Domain}. For example, we generate the String 'UML::StateMachine' to refer to the UML metaclass StateMachine.

Because a result parameter always has an **out** direction, it only consists of an element type reference. Hence, the value used for the <res-par> placeholder consists of a qualifier and the name of the element type. We determine both parts in the same way as for context parameters.

As argued earlier, we have to introduce guard conditions for all mapping operations with a non-empty operation body. The signature of mapping operation (A) defines such a guard condition, which is designated by the keyword **when**, followed by a Boolean expression (<guard-exp>) enclosed in curly brackets. The imperative expressions that define the operation body are all enclosed in curly brackets and must be defined directly below the guard condition.

In contrast, the operation body of the disjuncting mapping (B) is empty, which is specified by an opening and closing bracket without any enclosed content. The keyword **disjuncts** indicates the start of the candidate mapping list, where all items of this list are separated by commas.

Mapping of abstract metaclasses: Even if mapping operations with a non-empty operation body can be defined for abstract metaclasses, they must also be declared as abstract. Thus, an invocation of such a mapping operation is impossible; only non-abstract mapping operations inheriting from an abstract mapping operation can be invoked. Because we consider the inheritance mechanism as not applicable to our generated transformations, we do not create abstract mapping operations. Instead, we introduce a disjuncting mapping for each abstract metaclass of MM_{Domain}.

Assume that there is an abstract metaclass named AbsMC and a certain set of abstract and non-abstract subclasses. Based on metaclass AbsMC, we introduce a disjuncting mapping that is structured in the same way as mapping operation (B), see Fig. 4.22. The list of candidate mappings is derived from the set of all non-abstract subclasses of AbsMC, so that one candidate mapping is introduced for each of these metaclasses.

Provided the names of all candidate mappings are unambiguous, it is sufficient to specify only the names of the corresponding mapping operations; otherwise, the fully qualified identifiers have to be used. According to Design Decision 31, we employ the latter option, so we always introduce fully qualified identifiers for the mapping operations listed as candidate mappings. These identifiers consist of the following three parts:

```
<namespace>::<type−name>::<op−name>
```

We determine the values for the placeholders <namespace> and <type-name> in the same way as in the case of the context parameters. Furthermore, the <op-name> is derived according to the pattern that is applied for the generation of names for mapping operations.

Mapping of non-abstract metaclasses: Based on non-abstract metaclasses of MM_{Domain}, we generate mapping operations with operation bodies that contain imperative expressions for the mapping of metaclass attributes. Because the inheritance mechanism is not applied, a mapping operation for a given metaclass must map not only the attributes of that metaclass but also all attributes inherited from super-classes.

Assume that there is a non-abstract metaclass NAbsMC that has a certain set of super-classes. Based on metaclass NAbsMC, we introduce a mapping operation with a non-empty operation body (e.g., see mapping operation (A) in Fig. 4.22). Furthermore, we collect all attributes owned and inherited by a metaclass in order to generate assignment expressions and, if required, associated invocations of mapping operations. However, not all attributes so determined are mapped by a generated mapping operation, because this depends on the containing transformation ($T_{DM\text{-}to\text{-}UML}$ or $T_{UML\text{-}to\text{-}DM}$) and the kind of the metaclass (MC_{AC}, MC_{AMC}, or MC_{AC}) used to derive a mapping operation.

Further differences are caused by the attribute cardinality, and the fact that an attribute referring to a *DataType* has to be mapped in another way as is the case for a metaclass. We treat the according details in the context of the discussion on the generation of the transformations.

Apart from the exceptions mentioned, we derive assignment expressions for the mapping of attributes as shown below. Assignment expression (A) is employed for the mapping of a single-valued attribute, while the mapping of a multi-valued attribute is realized with assignment expression (B):

```
(A) result.<att−name> := self.<att−name>.map <op−name>();
(B) result.<att−name> := self.<att−name>−>map <op−name>();
```

In both cases, we first invoke a mapping operation for an attribute of the input element, before the mapping result is assigned to an attribute of the created result element. The <att-name> placeholder represents the attribute name, which is derived from a metaclass attribute in MM_{Domain}. Furthermore, the mapping operation to be invoked is represented by the <op-name> placeholder, which we derive in the same way as is the case for an operation signature.

As justified earlier, we have to introduce guard conditions for mapping operations that have a non-empty operation body. This is also required because we cannot ensure an appropriate order of the candidate mappings of disjuncting mappings. Without a guard condition, a candidate mapping could also be invoked for a model element that is the instance of a subtype of the mapping operation's context type. Therefore, we introduce a guard condition that ensures that a mapping operation can only be invoked for direct instances of the specified context type. Unless specified otherwise, we create the guard condition of a mapping operation with an operation body as follows:

```
when { self.oclIsTypeOf( <namespace>::<type−name> ) }
```

The values used to replace both placeholders <namespace> and <type-name> are derived based on a metaclass of MM_{Domain}, as described for the context parameter of a mapping operation.

Mapping of DataTypes: According to Design Decision 35, *DataType* instances have to be mapped in a similar manner as metaclass instances. Therefore, we apply the same concept as for the mapping of metaclasses. For this reason, we introduce disjuncting mappings for abstract *DataTypes* and mapping operations with a non-empty body for non-abstract *DataTypes*. Because no particular requirement concerning the mapping of *DataTypes* exist, all attributes of a source *DataType* are mapped one-to-one to corresponding target attributes.

Mapping of Enumerations: As required by Design Decision 36, the target *EnumerationLiteral* has to be determined based on the *name* of a source *EnumerationLiteral*. In QVTo, this can only be implemented by iterating over all *EnumerationLiterals* defined for an *Enumeration*.

We introduce a QVT helper operation for each *Enumeration* of metamodel MM_{Domain}. Such an operation consists of a switch expression that contains one switch alternative for each *EnumerationLiteral*. Such a switch alternative checks whether the current value of an input *Enumeration* corresponds to a particular *EnumerationLiteral*, and if so, the corresponding output *EnumerationLiteral* is returned.

The general structure of such a helper operation is illustrated in Fig 4.23. The operation signature has the same parts as a mapping operation that is derived from a metaclass. Each switch alternative consists of a Boolean condition expression and a body. The condition expression verifies whether the input value of the operation is equal to a specific literal of the source *Enumeration*, and if so, the body returns the corresponding literal of the target *Enumeration*. Placeholders starting

```
helper <con−par>::<op−name>() : <res−par>
{
  switch {
    case (self = <src−literal−a>)
      return <tar−literal−a>;
    case (self = <src−literal−b>)
      return <tar−literal−b>;

    // Further cases...
  };
  return null;
}
```

Figure 4.23: Helper operation for determining an *EnumerationLiteral*

with the 'src-literal' represent fully qualified identifiers of the source literals, while those starting with 'tar-literal' represent target literals.

4.6.5 Derivation of the DM-to-UML Transformation

In the following, we introduce the details of how we derive the M2M transformation $T_{DM\text{-}to\text{-}UML}$ based on the metamodel MM_{Domain} in Step (D) of our overall approach, where the derivation is implemented by our M2T transformation $T_{GenDM\text{-}to\text{-}UML}$. The most important parts of this transformation are discussed based on pseudocode, which is specified using the notational conventions introduced in Sec. 2.4.3.

In general, we derive the transformation $T_{DM\text{-}to\text{-}UML}$ along the common concepts and design decisions discussed above. The generation of this transformation is based on one main template that creates the overall structure of the transformation. Further templates are invoked to generate the QVT code based on particular types of model elements contained in MM_{Domain}. The mapping operations produced by these templates are used to map metaclasses, metaclass attributes, and data types of MM_{Domain} to corresponding target elements.

The main structure. The general structure of the transformation $T_{DM\text{-}to\text{-}UML}$ is specified by the main template genDmToUML() in Listing 18. A generated transformation can be divided into the following segments: the transformation header, the 'main' operation, a section with mapping operations for metaclasses, and another section with mapping operations for data types.

The transformation header declares the metamodels MM_{UML}, MM_{Domain}, and MM_{Add} as well as the UML profile UP_{Domain} in Lines 3 – 7. Then, the metamodels thus defined can be referenced as parameter types in the transformation signature. Except for MM_{UML}, we determine the prefixes and the *Unified Resource Identifiers (URI)* of the metamodels and of the UML profile using global variables. However,

```
[template genDmToUml(p : Package)]
[** 1. The transformation header **/]
modeltype UML uses 'http://www.eclipse.org/uml2/5.0.0/UML';
modeltype [MM_PREFIX/] uses '[MM_NS_URI/]';
[if (p.allMcAdd()->notEmpty())]
modeltype [ADD_MM_PREFIX/] uses '[ADD_MM_NS_URI/]'; [/if]
modeltype [PROFILE_PREFIX/] uses '[PROFILE_NS_URI/]';

transformation [MM_PREFIX/]_to_UML(
  in [MM_NS_PREFIX/] : [MM_PREFIX/], in profile : UML,
  out [PROFILE_NS_PREFIX/] : UML
);
// A global list of all already applied Stereotypes
property allStInstances : List(Stdlib::Element) = List {};

[** 2. The transformation entry operation **/]
main() {
  [* a. Creating UML elements */]
  /* [MM_NS_PREFIX/].rootObjects()['['/]Package[']'/]->map xxx

  [* b. Applying Stereotypes */]
  [PROFILE_NS_PREFIX/].objectsOfKind(UML::Element)->map applyStereotypes();

  [* c.) Applying Stereotypes to attributes of Stereotypes */]
  while( allStInstances->size() <> 0) {
    var inst := allStInstances->first();
    var elements := inst.allSubobjects()['['/]Element[']'/];
    elements->forEach(elem) {
      elem.map applyStereotypes(); };
    allStInstances->removeFirst();
  };
}

[** 3. Mapping operations for metaclasses **/]
[for (c : Class | p.allMcAc())][c.mcAcMapping()/] [/for]
[for (c : Class | p.allMcSt())][c.mcStMapping()/] [/for]
[p.stApplMapping()/]
[for (c : Class | p.allMcAdd())][c.mcAddMapping()/] [/for]

[** 4. Mapping operations for data types **/]
[for (e : DataType| p.allAcDt())][e.acDtMapping()/] [/for]
[for (e : DataType| p.allUpDt())][e.upDtMapping()/] [/for]
[for (e : DataType| p.allMmAddDt())][e.addDtMapping()/] [/for]
[for (e : Enumeration| p.allAcDt())][e.acEnumHelperOp()/] [/for]
[for (e : Enumeration| p.allUpDt())][e.upEnumHelperOp()/] [/for]
[for (e : Enumeration| p.allMmAddDt())][e.addEnumHelperOp()/] [/for]
[/template]
```

Listing 18: Top-level template genDmToUml() of transformation $T_{GenDM\text{-}to\text{-}UML}$

we take MM_{Add} only into account if at least one metaclass of MM_{Domain} is defined as MC_{AMC}; recall that MM_{Add} is derived only in this case.

Because transformation $T_{DM\text{-}to\text{-}UML}$ shall be utilized to transform a DSL-model to a corresponding UML model, the transformation signature (Line 9) declares MM_{Domain} as input parameter, and MM_{UML} as output parameter. As the *Stereotypes* contained in the UML profile UP_{Domain} shall be applied to mapped UML elements, UP_{Domain} is specified as second input parameter. Finally, a list of references to already created stereotype instances is introduced as a global variable in Line 14, so that these instances can be accessed directly by mapping operations.

Each QVT transformation is started by invoking the main() operation. Typically, the root elements of an input model are selected, so they can serve as input for calling appropriate mapping operations. According to Design Decision 34, UML elements have to be created first, before stereotypes can be applied to them in a second pass. Therefore, transformation $T_{DM\text{-}to\text{-}UML}$ consists of different consecutive transformation passes, which are invoked in the main() operation.

In Pass (a), the root element of the DSL defined by MM_{Domain} is selected, and then it is mapped to corresponding UML element by employing an appropriate mapping operation. All elements enclosed by a root element are mapped in the context of the invoked mapping operation. Because a root element is specific to a DSL, the generated mapping call in Line 19 must be manually revised.

Once all UML elements are created, the stereotypes are applied in Pass (b). For this purpose, all UML *Elements* of the generated target model are selected in Line 22, and the mapping operation applyStereotypes() is applied to them in-place. Because stereotypes also have to be applied to items of stereotype attributes, we conduct this task once again in Pass (c), which is defined in Lines 24 – 30.

The remaining parts of the $T_{DM\text{-}to\text{-}UML}$ transformation consist of mapping operations for metaclasses and data types. These operations are generated by dedicated templates that are iteratively invoked for a given set of metaclasses or data types from MM_{Domain}. The required sets of elements are determined by employing particular queries. For instance, query allMcSt() determines all MC_{AC} metaclasses. An exception is the stApplMapping() template that is only invoked once, because the mapping operation generated by this template is used to apply all kinds of stereotypes.

Mapping operations for MC_{AC} metaclasses. As argued for the common transformation concepts, the derived transformation $T_{DM\text{-}to\text{-}UML}$ has to transform an entire DSL-specific model into a corresponding UML model. Thus, we derive disjuncting mappings for abstract metaclasses of MM_{Domain}, while mapping operations with an operation body are introduced for non-abstract metaclasses.

Because we assume that 'matching' UML metaclasses exist, we neither derive stereotypes nor metaclasses for 'Abstract Concept' metaclasses of MM_{Domain}. However, we have to derive mapping operations for such kinds of metaclasses, other-

```
[** Template (A) used for abstract $MC_{AC}$ **/]
[template mcAcMapping(c : Class) ? ( c.isAbstract )]
mapping [MM_PREFIX/]::[c.name/]::to[c.name/]() : UML::[c.name/]
disjuncts
  [for (sc : Class | c.allNonAbsSubClasses())]
  [if (sc <> c.allNonAbsSubClasses()->last())]
    [if (sc.mapsToStereotype())]
    [MM_PREFIX/]::[sc.name/]::to[sc.extTarget()/],
    [else]
    [MM_PREFIX/]::[sc.name/]::to[sc.name/],
    [/if]
  [else]
    [if (sc.mapsToStereotype())]
    [MM_PREFIX/]::[sc.name/]::to[sc.extTarget()/]
    [else]
    [MM_PREFIX/]::[sc.name/]::to[sc.name/]
    [/if]
  {}
  [/if]
  [/for]
[/template]

[** Template (B) used for non-abstract $MC_{AC}$ **/]
[template mcAcMapping(c : Class) ? (not c.isAbstract )]
mapping [MM_PREFIX/]::[c.name/]::to[c.name/]() : UML::[c.name/]
{
  [for (a : Property | c.mappableAtt())]
  [a.attMapping()/]
  [/for]
}
[/template]
```

```
query Class::allNonAbsSubClasses() : OrderedSet(Class)
  return Class.allInstances()
    ->select(cl | cl.allParents()->includes(self) ∧ !cl.isAbstract )->asOrderedSet();

query Class:mappableAtt() : Set(Property)
  return self.allParents()->including(self).attribute
    ->select(a | ( !a.isDerivedUnion ∨ !a.isReadOnly ) ∧ !a._class.mapsToStereotype() );
```

Listing 19: The mcAcMapping() templates and associated queries of transformation $T_{GenDM\text{-}to\text{-}UML}$

wise, a transformation of a DSL specific model into a UML model would be impossible. For this reason, we use the two templates named mcAcMapping() in Listing 19 to generate mapping operations for MC_{AC} metaclasses of MM_{Domain}. Template (A) is employed for abstract MC_{AC} metaclasses, while template (B) is employed for the non-abstract ones.

According to Design Decision 32, we qualify the element types referenced by derived mapping operations. For instance, UML::Class refers to the metaclass Class in MM_{UML}. Therefore, the types of context and result parameters of the generated mapping operations are qualified by namespaces (see Lines 3 and 25). In the case of UML elements, the namespace is specified statically, while the content of the global variable MM_PREFIX is used for elements of MM_{Domain}. The name of a derived mapping operation is always generated according to the same pattern, where the name of an MC_{AC} metaclass is prefixed by the letters 'to'.

Generation of disjuncting mappings: As mentioned above, the output of Template (A) is a disjuncting mapping. We determine the ordered list of mapping candidates for such mapping operations based on all non-abstract sub-classes of the current context MC_{AC} metaclass of the template. We obtain these metaclasses by invoking the allNonAbsSubClasses() query in Listing 19. As shown in Lines 5 to 20 of Template (A), the list of all metaclasses is iterated to generate invocations for corresponding mapping operations.

The outermost if-else-statement within the iteration loop is only required to generate the curly brackets at the end of the mapping operation. Apart from this detail, both alternatives of the if-else statement have the same structure. In the following, we do not distinguish between both alternatives.

The second-level if-else-statements within the iteration loop are used to generate a call to a mapping operation, depending on the metaclass currently identified by the loop variable sc. In the case of an MC_{St} metaclass, the name of the mapping operation to be invoked is determined based on the UML metaclass extended by a *Stereotype*. This is required because of our two-staged transformation approach as defined by Design Decision 34. In the case of MC_{AC} and MC_{AMC} metaclasses, the name of the mapping operation is derived directly from the *name* property of the metaclass.

Generation of mapping operations with operation body: We employ Template (B) in Listing 19 to generate mapping operations that have an operation body, where assignment statements are created in order to assign mapped attributes of the input element to corresponding attributes of the output element. Provided an owned or inherited attribute of an MC_{AC} metaclass is considered as 'mappable', we create an assignment statement for it according to our common transformation approach. This kind of attribute is obtained by invoking the mappableAtt() query. In Line 27, we iterate over the set of attributes returned by the query, so we can generate the assignment statements using the attMapping() template.

Using the set of all owned and inherited attributes of a metaclass, the mappableAtt() query in Listing 19 returns all those attributes that are not defined as 'read-only' or 'derivedUnion', and are not owned by an MC_{St} metaclass. Only if all these criteria are fulfilled, a value assignment to an attribute is feasible, so that we regard such an attribute as 'mappable'.

```
1  [** Template (A) used for abstract MC_St **/]
2  [template mcStMapping(c : Class) ? (c.isAbstract) ]
3  mapping [MM_PREFIX/]::[c.name/]::to[c.extTarget()/]() : UML::[c.extTarget()/]
4  disjuncts
5  [for (sc : Class | c.allNonAbsSubClasses())]
6     [if (sc <> c.allNonAbsSubClasses()−>last())]
7        [MM_PREFIX/]::[sc.name/]::to[sc.extTarget()/],
8     [else]
9        [MM_PREFIX/]::[sc.name/]::to[sc.extTarget()/] {}
10    [/if]
11 [/for]
12 [/template]
13
14 [** Template (B) used for non−abstract MC_St **/]
15 [template mcStMapping(c : Class) ? (not c.isAbstract)]
16 mapping [MM_PREFIX/]::[c.name/]::to[c.extTarget()/]() : UML::[c.extTarget()/]
17 when { self.oclIsTypeOf([MM_PREFIX/]::[c.name/]) }
18 {
19    [for (a : Property | c.mappableAtt())]
20    [a.attMapping()/]
21    [/for]
22    // Note: A refinement of the mappings below could be required!
23    [for (a : Property | c.attWithOclSpec())]
24    //result.[a.derivSrcAtt().name/] := self.[a.name/].map xxxxx;
25    [/for]
26 }
27 [c.stereotypeAppl()/]
28 [/template]
```

Listing 20: The $\mathrm{mcStMapping}()$ templates of transformation $T_{GenDM\text{-}to\text{-}UML}$

Mapping operations for MC_{St} metaclasses. As in the previous case, we use two templates shown in Listing 20 to generate mapping operations for MC_{St} metaclasses. Both templates are named $\mathrm{mcStMapping}()$. According to our common transformation approach, Template (A) generates disjuncting mappings from abstract MC_{St} metaclasses, while mapping operations having a body are created by applying Template (B) to non-abstract MC_{St} metaclasses.

The context and result parameter types of all generated mapping operations are qualified in the same manner as in the case of mapping operations created for MC_{AC} metaclasses. However, the operation names are not derived from the MC_{St} metaclasses but from the UML metaclasses extended by stereotypes. Therefore, we first determine an extended UML metaclass using the $\mathrm{extTarget}()$ query, as shown in Lines 3 and 16 of Listing 20. Then, the name of the returned metaclass is prefixed by the characters 'to', obtaining the operation name.

Generation of disjuncting mappings: The structure of Template (A) and that of the previously treated $\mathrm{mcAcMapping}()$ template are similar, because both templates are used to generate disjuncting mappings. According to our design decisions for the UML profile derivation (see Sec. 4.3.2), all metaclasses that inherit from an MC_{St} have to be of the same type. Thus, the ordered list of mappings can be generated only for MC_{St} metaclasses. Hence, we determine all subclasses of an MC_{St} by employing the allNonAbsSubClasses() query in Line 5 of Listing 20. Then, we iterate over all obtained metaclasses to generate the mapping invocations, as shown in Lines 5 – 11. Because mapping operations for MC_{St} metaclasses shall be invoked, the operation names can be derived according to the same pattern as discussed above.

Generation of mapping operations with operation body: Template (B) is applied to non-abstract MC_{St} metaclasses for generating mapping operations that have a body. To ensure that a mapping operation can be invoked for direct instances of an MC_{St} metaclass but not for instances of sub-classes, we introduce an appropriate guard condition directly after the operation header.

As shown by the pseudocode of Template (B) in Lines 15 – 28 of Listing 20, the assignment statements for the operation body are generated based on two different sets of attributes. The set of all 'mappable' attributes of an MC_{St} is obtained by invoking the mappableAtt() query specified in Listing 19. However, the returned attribute set only includes attributes that are inherited from MC_{AC} super-classes of an MC_{St}, because only this kind of attribute is mappable in the context of a UML metaclass. In contrast, all attributes owned by an MC_{St} or inherited from MC_{St} super-classes are mapped in the context of the stereotype, as discussed below.

Based on the set of obtained attributes, we invoke the attMapping() template in Line 20 of Listing 20 to iteratively generate the assignment statements. Because the set of 'mappable' attributes also includes MC_{AC} attributes that are redefined or subsetted by MC_{St} attributes, we have also met the requirements of Design Decision 32.

The second set of attributes processed by Template (B) consists of MC_{St} attributes for which an additional OCL specification is defined. As we justified for Design Decisions 33, a value assignment to such an attribute kind is impossible. However, we can make a mapping 'proposal' based on the mappingSource attribute of an applied «ToTaggedValue» stereotype. Hence, we generate assignment statements in terms of source code comments, which have to be revised manually, if required.

For example, the toStateMachine() mapping operation shown in Fig. 4.24 is derived from the metaclass Statemachine of our running example (see Fig. 4.7) by applying Template (B). Because the MC_{St} Statemachine does not own attributes with an alternative OCL specification, the operation body consists only of attribute mappings for inherited MC_{AC} attributes of the MC_{St}. Since only a small set of attributes is defined for the MC_{AC} metaclasses of the running example, the number of attribute mappings generated for the operation body is also low.

```
// 1. Mapping to a UML 'Statemachine' element
mapping SMDSL::Statemachine::toStateMachine():UML::StateMachine
when { self.oclIsTypeOf(SMDSL::Statemachine) }
{
  result.name := self.name;
  result.nestedClassifier += self.nestedClassifier−>map toClassifier();
  result.ownedBehavior += self.ownedBehavior−>map toBehavior();
  result.region += self.region−>map toRegion();
}

// 2. Mapping to the <<StateMachine>> stereotype
mapping inout UML::StateMachine::applyStereotypeForStatemachine()
when {
  self.invresolveone(SMP::Statemachine)−>notEmpty().oclIsTypeOf(SMP::Statemachine)
} {
  // 1. Stereotype application
  var stereotype := self.getApplicableStereotype('SMP::Statemachine');
  self.applyStereotype(stereotype);

  // 2. Attribute mapping
  var srcElement := self.invresolveone(SMDSL::Statemachine);

  // Storing stereotype instance for further stereotype applications
  allStInstances−>add(stInstance);
}
```

Figure 4.24: Derived mapping operations that map DSL model elements to UML elements with applied stereotypes

Creation of mapping operations for applying stereotypes: At the end of Template (B) in Listing 20, we invoke the stereotypeAppl() template of Listing 21 for generating an additional mapping operation that creates and applies a *Stereotype* to a UML element already present in the target model. At runtime, such a mapping operation is invoked in-place in the second pass of transformation $T_{DM\text{-}to\text{-}UML}$, as required by Design Decision 34. For this reason, we create the operation header without a result parameter and qualify the type of the context parameter by the namespace of UML, as shown in Line 2 in Listing 21. Because we have to apply a stereotype to an existing UML element, we derive the context parameter type using the extTarget() query, which returns the extended UML metaclass. The operation name is composed of the String 'applyStereotypeFor' and the name of an MC_{St} metaclass.

To ensure that a certain stereotype is only applied to a UML element mapped from a particular MC_{St}, we introduce a guard condition in Line 3 of Listing 21. First, we resolve the source MC_{St} from which the current UML element is mapped. We then apply a type check to ensure that the mapping operation is not invoked for a sub-classes instance of the source MC_{St}.

The body of the generated mapping operation consists of two parts. In the first part, two EMF-specific operations are invoked to apply the required *Stereotype* to

```
 1 [template stereotypeAppl(c : Class) ? (not c.isAbstract)]
 2 mapping inout UML::[c.extTarget()/]::applyStereotypeFor[c.name/]()
 3 when { self.invresolveone([MM_PREFIX/]::[c.name/])
 4   .oclIsTypeOf([MM_PREFIX/]::[c.name/]) }
 5 {
 6   // 1. Stereotype application
 7   var stereotype := self.getApplicableStereotype('[MM_PREFIX/]::[c.name/]');
 8   self.applyStereotype(stereotype);
 9   var stInstance := self.getStereotypeApplication(stereotype)
10     .oclAsType([PROFILE_PREFIX/]::[c.name/]);
11
12   // 2. Attribute mapping
13   var srcElement := self.invresolveone([MM_PREFIX/]::[c.name/]);
14   [for (a : Property | c.mappableStAtt()]
15   [a.stAttMapping()/]
16   [/for]
17   // Storing stereotype instance for further stereotype applications
18   allStInstances−>add(stInstance);
19 }
20 [/template]
21
22 [template stApplMapping(p : Package)
23 mapping inout UML::Element::applyStereotypes()
24 disjuncts
25 [for (c : Class | p.allMcSt()−>select(not isAbstract))]
26   [if (c <> p.allMcSt()−>select(not isAbstract)−>last())]
27   UML::[c.extTarget()/]::applyStereotypeFor[c.name/],
28   [else]
29   UML::[c.extTarget()/]::applyStereotypeFor[c.name/] {}
30   [/if]
31 [/for]
32 [/template]
```

```
33 query Class:mappableStAtt() : Set(Property)
34   return self.allParents()−>including(self).attribute−>select(a | !a.isDerivedProperty() );
```

Listing 21: The stereotypeAppl() and stApplMapping() templates and the mappableStAtt() query of transformation $T_{GenDM\text{-}to\text{-}UML}$

the current UML element (Lines 6 – 10). Then, the source MC_{St} from which the current UML element has been mapped is determined in the second part, and the owned or inherited MC_{St} attributes that are not considered as 'derived', are mapped to corresponding stereotype attributes (Lines 12 – 18).

Because of our derivation approach for UML profiles, we consider not only MC_{St} attributes whose *isDerived* property is 'true' as 'derived', but also other types of MC_{St} attributes (see Sec. 4.3.4). Hence, the mappableStAtt() query in Listing 21 employs the isDerivedProperty() query to verify whether an MC_{St} attribute is not specified as 'derived' according to our definition. All attributes that meet this condition are returned as a result of the mappableStAtt() query. Thus, we can generate the required assignment statements iteratively by calling the stAttMapping() template in Line 15 of Listing 21. Finally, the stereotype instance is added to the global variable allStInstances of the $T_{DM\text{-}to\text{-}UML}$ transformation.

For example, the mapping operation applyStereotypeForStatemachine() shown in Fig. 4.24 is derived from the MC_{St} Statemachine by applying the stereotype-Appl() template. The two imperative expressions in the first part of the operation body apply the «Statemachine» stereotype (see Fig. 4.12b). Because all attributes directly owned by this stereotype are 'read-only' and 'derived', the operation body contains no attribute mappings. However, the values for these attributes are computed by OCL expressions at runtime, where the computation is based on attributes of the extended UML metaclass.

Mapping operation for the second transformation pass. The mapping operation applyStereotypes(), which is applied in-place, is the entry point for the second transformation pass. Depending on the type of an element passed as context parameter, this mapping operation invokes the specific mapping operation that implements the creation and application of the required *Stereotype*.

We use the stApplMapping() template in Listing 21 to generate the mapping operation applyStereotypes(). Because all UML element types are processed by this mapping operation, we statically define the context parameter type as UML::Element. Based on the set of all non-abstract MC_{St} metaclasses of MM_{Domain}, the ordered list of mappings is iteratively generated in Lines 25 – 32. More precisely, we list all the mapping operations with a name starting with applyStereotypeFor. The purpose of the if-else statement within the iteration loop is only to enable the generation of the terminating curly brackets at the end of the mapping operation.

The disjuncting mapping shown in Fig. 4.25 is an example outcome of the stApplMapping() template. As justified above, such a mapping operation is only generated once for transformation $T_{DM\text{-}to\text{-}UML}$. Because the given example is based on our running example, which embraces seven MC_{St} metaclasses, an equal number of candidate mappings is created for the disjuncting mapping. Each of these

```
mapping inout UML::Element::applyStereotypes()
disjuncts
  UML::Behavior::applyStereotypeForAction,
  UML::Signal::applyStereotypeForCommand,
  UML::Event::applyStereotypeForEvent,
  UML::Event::applyStereotypeForResetEvent,
  UML::State::applyStereotypeForState,
  UML::StateMachine::applyStereotypeForStatemachine,
  UML::Transition::applyStereotypeForTransition
  {}
```

Figure 4.25: Disjuncting mapping applyStereotype() that is generated based on the running example

```
[template mcAddMapping(c : Class) ? (c.isAbstract)]
mapping [MM_PREFIX/]::[c.name/]::to[c.name/]() : [ADD_MM_PREFIX/]::[c.name/]
disjuncts
[for (sc : Class | c.subClasses())]
  [if (sc <> c.subClasses()->last())]
  [MM_PREFIX/]::[sc.name/]::to[sc.name/],
  [else]
  [MM_PREFIX/]::[sc.name/]::to[sc.name/] {} [/if]
[/for]
[/template]

[template mcAddMapping(c : Class) ? (not c.isAbstract)]
mapping [MM_PREFIX/]::[c.name/]::to[c.name/]() : [ADD_MM_PREFIX /]::[c.name/]
{
  [for (a : Property | c.mappableAtt())]
  [a.attMapping()/][/for]
}
[/template]
```

Listing 22: The mcAddMapping() templates of transformation $T_{GenDM\text{-}to\text{-}UML}$

candidate assignments is a mapping operation generated by the stereotypeAppl() template. Therefore, the mapping operation applyStereotypeForStatemachine() is also listed as one of the candidate mappings.

Mapping operations for MC_{AMC} metaclasses. According to our approach for deriving 'additional' metaclasses, we map MC_{AMC} metaclasses of MM_{Domain} one-to-one to corresponding metaclasses of metamodel MM_{Add}. However, an exception to this are MC_{AMC} attributes having a *type* property that refers to an MC_{St} metaclass. This is because we recompute the *type* properties to reference an extended UML metaclass instead of a *Stereotype*. As we regard this aspect when creating assignment statements for the mapping of attributes, we do not have to consider it again for the generation of mapping operations.

The mapping operations for MC_{AMC} metaclasses are generated by the two mcAddMapping() templates shown in Listing 22, which are similar to those employed for MC_{AC} metaclasses. The only difference is the namespace used for the result parameters of the mapping operations generated by the templates, because it is defined by the global variable ADD_MM_PREFIX.

Mapping operations for *DataTypes*. According to Design Decision 35 and our common concepts, we generate mapping operations for *DataTypes* in the same way as for metaclasses. Thus, disjuncting mappings are generated for abstract *DataTypes*, and mapping operations with a body for non-abstract *DataTypes*.

The mapping operations for mapping *DataTypes* of MM_{Domain} to corresponding types of MM_{Add}, MM_{UML} and UP_{Domain} are identical, except the namespaces of the

```
 1 [template acDtHelperOp(e : Enumeration)]
 2 helper [MM_PREFIX/]::[e.name/]::to[e.name/](): UML::[e.name/]
 3 {
 4   switch {
 5   [for (l : EnumerationLiteral | e.ownedLiteral)]
 6     case (self = [MM_PREFIX/]::[e.name/]::[l.name/])
 7       return UML::[e.name/]::[l.name/]; [/for]
 8   };
 9   return null;
10 } [/template]
```

Listing 23: The acDtHelperOp() template of transformation $T_{GenDM\text{-}to\text{-}UML}$

result parameter. Because *DataTypes* and MC_{AC} metaclasses are mapped one-to-one, the structure of their mapping operations is identical.

Helper operations for *Enumerations*. As we argued for Design Decision 35, the values of *Enumerations* are represented in terms of literals. Instead of a simple value assignment, the literal of a target *Enumeration* must be determined based on a given source literal. According to our common concepts, we generate a dedicated helper operation for each *Enumeration* of MM_{Domain}.

The acDtHelperOp() template in Listing 23 generates helper operations that determine the literal of a UML *Enumeration* for a given *Enumeration* literal of MM_{Domain}. The operation signature is compiled in the same way as for the mapping of MC_{AC} metaclasses. In Lines 5 – 7, we create the switch-alternatives in the operation body by iterating over the set of *EnumerationLiterals* owned by an *Enumeration* of MM_{Domain}.

The helper operation toTransitionKind() shown in Fig. 4.26 is an example output of the acDtHelperOp() template. This helper operation is derived from the AC_TransitionKind *Enumeration* that is a part of the metamodel of our running example. Assume that the internal literal of AC_TransitionKind is used as input for this operation. In consequence, the condition of the first switch alternative evaluates to 'true', so the internal literal of the UML Enumeration TransitionKind is returned as a result.

Mapping of attributes. As argued for the common concepts, the attribute mapping depends on several factors. Hence, the assignment expressions employed to map a source attribute to a corresponding target attribute differs. However, all assignment statements can be created based on similar patterns.

We introduce an assignment expression with a previous invocation of a mapping operation for those metaclass attributes in MM_{Domain}, whose *type* property refers to a metaclass or *DataType*. First, on the right-hand side of such an assignment expression, an attribute of the context element is accessed. Then, the returned object or value is mapped by invoking an appropriate mapping operation.

```
helper SMDSL::AC_TransitionKind::toTransitionKind() : UML::TransitionKind
{
  switch {
    case (self = SMDSL::AC_TransitionKind::internal)
      return UML::TransitionKind::internal;
    case (self = SMDSL::AC_TransitionKind::local)
      return UML::TransitionKind::local;
    case (self = SMDSL::AC_TransitionKind::external)
      return UML::TransitionKind::external;
  };
  return null;
}
```

Figure 4.26: Example of a generated helper operation for *Enumeration* data types

Finally, the mapped objects or values are assigned to an attribute of the result element specified on the left-hand side of the expression.

Depending on the cardinality of an attribute, a mapping operation has to be invoked in two different ways. Variant (1) is used to invoke a mapping operation for a single-valued attribute, while Variant (2) is applied for multi-valued attributes. In the latter case, the 'additive' assignment operator '**+=**' is employed to append new values to those ones already present:

```
(1) result.<att−name> := self.<att−name>.map <op−name>();
(2) result.<att−name> += self.<att−name>−>map <op−name>();
```

The assignment expressions for the mapping of metaclass attributes are generated by the attMappings() templates shown in Listing 24. Template (A) generates an assignment expression that corresponds to Variant (1), while Variant (2) is the outcome of Template (B).

The attribute names (placeholder <att-name>) are derived from the metaclass attribute that is the context element of the template. The name of the mapping operation (<op-name>) is determined using the targetName() query (see Listing 24) which is invoked based on the *type* property of a metaclass attribute. Provided the attribute *type* refers to an MC_{St} metaclass, this query determines the extended UML metaclass and returns the *name* of this metaclass. Otherwise, the *name* of the metaclass or the *DateType* passed as input is returned.

As argued for the derivation of mapping operations for data types, the value of a *PrimitiveType* or *Enumeration* is represented in terms of a literal, which has not to be mapped. For this reason and based on metaclass attributes with a *type* that refers to such data types, we generate assignment expressions that do not invoke mapping operations. Depending on the kind of a referenced data type and the attribute cardinality, three different assignment expressions are created:

```
(3) result.<att−name> := self.<att−name>;
(4) result.<att−name> := self.<att−name>.<op−name>();
(5) result.<att−name> := self.<att−name>−>collect( <op−name>() );
```

```
[** Mappings toward metaclass attributes **/]
[* Template (A): type == metaclass or DataType */]
[template attMapping(p : Property)
? ( p.upper = 1 and (p.type.oclIsTypeOf(Class) or p.type.oclIsTypeOf(DataType)) ]
  result.[p.name/] := self.[p.name/].map to[p.type.targetName()/]();
[/template]

[* Template (B): type == metaclass or DataType */]
[template attMapping(p : Property)
? ( p.upper > 1 and (p.type.oclIsTypeOf(Class) or p.type.oclIsTypeOf(DataType)) ]
  result.[p.name/] += self.[p.name/]—>map to[p.type.targetName()/]();
[/template]

[* Template (C): type == PrimitiveType */]
[template attMapping(p : Property)
? (p.type.oclIsKindOf(PrimitiveType))]
  result.[p.name/] := self.[p.name/];
[/template]

[* Template (D): type == Enumeration */]
[template attMapping(p : Property)
? (p.upper = 1 and p.type.oclIsKindOf(Enumeration))]
  result.[p.name/] := self.[p.name/].to[p.type.name/]();
[/template]

[* Template (E): type == Enumeration */]
[template attMapping(p : Property)
? (p.upper > 1 and p.type.oclIsKindOf(Enumeration))]
  result.[p.name/] := self.[p.name/]—>collect(to[p.type.name/]());
[/template]

[** Mappings to towards Stereotype attributes **/]
[template stAttMapping()(p : Property)
? ( p.upper = 1 and (p.type.oclIsTypeOf(Class) or p.type.oclIsTypeOf(DataType)) ]
  stInstance.[p.name/] := srcElement.[p.name/].map to[p.type.targetName()/]();
[/template]
[* Other variants analogously to the attMapping() templates! */]
```

```
query Type::targetName() : String
  return if self.isTypeOf(Class) ∧ self.mapsToStereotype() then
        self.asType(Class).extTarget()
    else
        self.name endif;
```

Listing 24: The various attMapping() templates and the targetName() query of transformation $T_{GenDM\text{-}to\text{-}UML}$

Variant (3) is generated by Template (C) for all metaclass attributes having a *type* property that refers to a *PrimitiveType*. We apply this template for single-valued as well as multi-valued attributes. By contrast, this is not the case for metaclass attributes with a *type* that references an *Enumeration*, because a specific helper operation must be invoked in order to determine a matching *EnumerationLiteral*.

Variant (4) that is generated by Template (D) is used for single-valued attributes, while the assignment of multi-valued attributes is realized with Variant (5) resulting from Template (E). Both assignment expressions invoke a helper operation that determines the *EnumerationLiteral* of a mapped *Enumeration*. This helper operation is derived as discussed above. In Variant (5), the operation call is embedded in a collect iterator expression in order to apply the operation to all values of the source attribute.

In the same way as we generate assignment expressions for mapping operations processing metaclasses, we create expressions for mapping operations processing stereotypes, which have a name starting with 'applyStereotypeFor'. The generated assignment expressions differ in the detail that mapped values are not assigned to attributes of the result variable but to attributes of a stereotype instance. Moreover, the source values are obtained from a variable named srcElement instead of the context variable of a mapping operation. For this reason, we employ five templates named stAttMapping() to generate mappings for stereotype attributes.

One variant of the attMapping() template is shown in Lines 33 – 36 of Listing 24. Because all template variants are similar to the attMapping() templates, we do not discuss any further details here.

4.6.6 Deriving the UML-to-DM Transformation

In the following, we detail our approach to derive a M2M transformation $T_{UML\text{-}to\text{-}DM}$ that transforms a UML model with an applied UML profile to a corresponding model that is an instance of metamodel MM_{Domain}. The derivation is based on the M2T transformation $T_{GenUML\text{-}to\text{-}DM}$ that implements Step (E) of our overall approach. As for transformation $T_{DM\text{-}to\text{-}UML}$, we respect the design decisions and the common concepts defined earlier in this chapter when deriving transformation $T_{UML\text{-}to\text{-}DM}$. Because only Design Decisions 30 and 31 are relevant for this derivation, the complexity of the generated transformation is lower when compared to transformation $T_{DM\text{-}to\text{-}UML}$; and thus, transformation $T_{GenUML\text{-}to\text{-}DM}$ is less complex than transformation $T_{GenDM\text{-}to\text{-}UML}$.

Just as for transformation $T_{DM\text{-}to\text{-}UML}$, transformation $T_{UML\text{-}to\text{-}DM}$ implements a rewrite system for an entire model. However, transformation $T_{UML\text{-}to\text{-}DM}$ is applied in the opposite direction when compared to transformation $T_{DM\text{-}to\text{-}UML}$, so it has not to create and apply *Stereotypes*. Therefore, transformation $T_{UML\text{-}to\text{-}DM}$ just requires one pass to transform a UML model into a model that is an instance of

```
[template genUml2Dm(p : Package)]
[** 1. The transformation header **/]
modeltype UML uses 'http://www.eclipse.org/uml2/5.0.0/UML';
modeltype [MM_PREFIX/] uses '[MM_NS_URI/]';
[if (p.additionalMetaclassSources()->notEmpty())]
modeltype [ADD_MM_PREFIX/] uses '[ADD_MM_NS_URI/]';[/if]
modeltype [p.profilePrefix()/] uses '[p.profileNsUri()/]';

transformation UML_to_[MM_PREFIX/](
  in uml : UML, in profile : UML, out [p.MM_NS_PREFIX/] : [MM_PREFIX/]);

[** 2. The transformation entry operation **/]
main() {
  // Invokation of the mapping of the topmost model element
  // uml.rootObjects()['['/]UML::Package[']'/]
  // ->map toPackageDefinition()
}
[** 3. Mapping operations for metaclasses **/]
[for (c : Class | p.allMcAC())][c.mcAcMapping()/] [/for]
[for (c : Class | p.allMcSt())][c.mcStMapping()/] [/for]
[for (c : Class | p.allMcAdd())][c.mcAddMapping()/] [/for]

[** 4. Mapping operations for data types **/]
[for (e : DataType| p.allAcDt())][e.acDtMapping()/] [/for]
[for (e : DataType| p.allUpDt())][e.upDtMapping()/] [/for]
[for (e : DataType| p.allMmAddDt())][e.addDtMapping()/] [/for]
[for (e : Enumeration | p.allAcDT())][e.acDtHelperOp()/] [/for]
[for (e : Enumeration | p.allProfileDT())][e.upDtHelperOp()/] [/for]
[for (e : Enumeration| p.allMmAddDT())]
  [e.addDtHelperOp()/] [/for]
[/template]
```

Listing 25: Top-level template genUmlToDm() of transformation $T_{GenUML\text{-}to\text{-}DM}$

MM_{Domain}. Moreover, only a single mapping operation is required for each UML element type, because the instances of applied stereotypes are directly accessible.

We now discuss the most important parts of transformation $T_{UML\text{-}to\text{-}DM}$, and of the templates of transformation $T_{GenUML\text{-}to\text{-}DM}$ generating these parts. These templates are specified in terms of pseudocode, which conforms to the conventions specified in Sec. 2.4.3. Because we name templates according to their task, some of them have identical names to those of transformation $T_{GenDM\text{-}to\text{-}UML}$. However, this does not apply for queries invoked by the templates below, i.e., if a query is not defined below, it is already introduced in the above paragraphs.

The main structure. Transformation $T_{UML\text{-}to\text{-}DM}$ consists of four sections that have exactly the same purposes as those of transformation $T_{DM\text{-}to\text{-}UML}$. The template genUml2Dm() in Listing 25 creates the overall structure of the $T_{UML\text{-}to\text{-}DM}$ transformation, and it is the entry point of the M2T transformation $T_{GenUML\text{-}to\text{-}DM}$.

As shown in Lines 3 – 7, we declare the four model types required by transformation $T_{UML\text{-}to\text{-}DM}$. However, the declaration of the metamodel for 'additional' metaclasses is only optional, because the derivation of this type of metaclasses is also optional. Then, the transformation signature with its parameters is defined in Line 9. Compared to $T_{DM\text{-}to\text{-}UML}$, the input and output parameters of $T_{UML\text{-}to\text{-}UML}$ are transposed because this transformation is applied in the opposite direction.

The $\mathrm{main()}$ operation of $T_{UML\text{-}to\text{-}DM}$ is generated in the second part of template $\mathrm{genUml2Dm()}$. As argued above, the transformation of a UML model can be realized in one pass, and therefore, the operation body of $\mathrm{main()}$ consists of only a single imperative expression. This expression is employed to call the mapping operation that transforms the outermost UML element of the input model. Because this UML element is DSL-specific, the imperative expression can only be generated as a comment (Line 15), which has to be refined manually.

In Lines 18 – 30 of template $\mathrm{genUml2Dm()}$, we generate mapping operations for UML metaclasses, 'additional' metaclasses and *DataTypes* by invoking further templates. Moreover, helper operations for *Enumerations* are created in the fourth section of the template.

Mapping operations for stereotyped UML elements. Regarding the design decisions and the common concepts that we introduced earlier, we generate mapping operations with non-empty operation body and disjuncting mappings for UML elements with an applied stereotype of the UML profile UP_{Domain}.

Stereotypes of UP_{Domain} are derived based on MC_{St} metaclasses contained in metamodel MM_{Domain}. Therefore, the result parameter types of the generated mapping operations refer to MC_{St} metaclasses. The disjuncting mappings are derived based on abstract MC_{St} metaclasses, while non-abstract MC_{St} metaclasses induce mapping operations with a non-empty operation body.

Both kinds of mapping operations are generated using the two $\mathrm{mcStMapping()}$ templates shown in Listing 26. According to Design Decision 31, all element types referenced within the body of mapping operations shall be qualified by namespaces. Because UML element types serve as input for the generated mapping operations, we use the static String 'UML' as namespace for all context parameter types. In contrast, the namespaces of the result types are determined dynamically, because the namespace of MM_{Domain} is user-defined employing the «$\mathrm{MM2Profile}$» stereotype. In both templates, this namespace is obtained by accessing the global variable $\mathrm{MM\_PREFIX}$.

Because we map UML elements with applied *Stereotypes*, the context parameter type is determined based on the extended UML metaclass. We obtain such a metaclass by invoking the $\mathrm{extTarget}$ query. In contrast, the type of a result parameter is directly determined from the *name* property of an MC_{St} metaclass. The names of all generated mapping operations are obtained by applying the name pattern

```
[** Template (A) used for abstract $MC_{St}$ **/]
[template mcStMapping(c : Class) ? (c.isAbstract)]
mapping UML::[c.extTarget().name/]::to[c.name/]() : [MM_PREFIX/]::[c.name/]
disjuncts
[for (sc : Class | c.c.allNonAbsSubClasses())]
  [if (sc <> c.allNonAbsSubClasses()->last())]
    UML::[sc.extTarget().name/]::to[sc.name/],
  [else]
    UML::[sc.extTarget().name/]::to[sc.name/] {} [/if]
[/for]
[/template]

[** Template (A) used for abstract $MC_{St}$ **/]
[template mcStMapping(c : Class) ? (not c.isAbstract)]
mapping UML::[c.extTarget().name/]::to[c.name/]() : [MM_PREFIX/]::[c.name/]
when { self.isStereotypedBy('[PROFILE_NAME/]::[c.name/]') }
{
  [* 1. Determining the stereotype instance */]
  var stereotype := self.getAppliedStereotype('[PROFILE_NAME/]::[c.name/]');
  var stInstance := self.getStereotypeApplication(stereotype)
    .oclAsType([PROFILE_NAME/]::[c.name/]);

  [* 2. Mapping of non-derived stereotype attributes */]
  [if (c.stAttForWriteableTar()->notEmpty())]
    // Mapping of stereotype attributes
  [for (a : Property | c.stAttForWriteableTar())]
    [a.stAttMapping()/]
  [/for]
  [/if]

  [* 3. Mapping of metaclass attributes */]
  [if (c.mappableUmlAtt()->notEmpty())]
    // Mapping of metaclass attributes
    [for (a : Property | c.mappableUmlAtt())]
    [a.attMapping()/]
  [/for]
  [/if]
}
[/template]
```

```
query Class::stAttForWriteableTar() : OrderedSet(Property)
  return self.allParents()->including(self)
    ->select( mapsToStereotype() ).attribute->select( !isReadOnly )

query Class::mappableUmlAtt() : OrderedSet(Property)
  return self.allParents()
    ->select(isAbstractConcept()).attribute->select(a | !a.isReadOnly )
```

Listing 26: The mcStMapping() templates and associated queries of transformation $T_{GenUML\text{-}to\text{-}DM}$

introduced for the common concepts. Thus, the operation name always consists of the prefix 'to', followed by the *name* of an MC_{St} metaclass.

Generation of disjuncting mappings: The disjuncting mappings are derived by applying Template (A) of Listing 26 to abstract MC_{St} metaclasses of MM_{Domain}. In Lines 5 – 11, the list of candidate mappings is generated by iterating over all non-abstract subclasses of an MC_{St} that are used as input for the template. An identifier representing a particular candidate mapping starts with the UML namespace, followed by the *name* of an extended UML metaclass and the name of a mapping operation. The latter two items are determined based on the MC_{St} metaclass currently referenced by the loop variable.

Because mapping operations for UML elements with an applied stereotype shall be referenced as candidate mappings, the operation names are determined as explained above. The if-else statement enclosed by the iteration loop is only required to append the opening and closing curly brackets after the last candidate mapping.

Generation of mapping operations with operation body: We derive mapping operations with a non-empty body based on non-abstract MC_{St} metaclasses of MM_{Domain} by employing Template (B) of Listing 26. As argued for the common concepts, we also have to introduce guard conditions for such kind of mapping operations. However, a simple type check of the input element of a mapping operation is not sufficient for this, because a particular UML element type may have applied different kinds of stereotypes. Hence, the guard condition introduced in Line 16 of Listing 26 verifies whether an input element has applied the expected stereotype. We generate such a guard condition according to the following pattern:

```
when { self.isStrictStereotypedBy('<namespace>::<type−name>') }
```

This guard condition pattern uses the isStrictStereotypedBy() query to verify the application of a specific stereotype. The fully qualified name of the stereotype must be defined in terms of a String value, which is compiled from the values of the placeholders <namespace> and <type-name>. As shown in the pseudocode of Template (B), the value for <namespace> is obtained from the global variable PROFILE_NAME, while the value for <type-name> is determined from the *name* of the current input MC_{St} of the template.

The operation body of a mapping operation generated by Template (B) can be divided into three parts. In the first part, we determine the instance of a stereotype that is applied to the context element by using EMF-specific operations. In Line 20 of Listing 26, we store the obtained instance in a variable named stInstance. Afterwards, we can use this variable to access the attributes of a stereotype.

The second part of the operation body consists of mappings for attributes of an applied stereotype to those MC_{St} attributes that are not defined as 'read-only', i.e., values can be assigned to them. In order to generate the mappings for such attributes, we determine the set of all 'read-only' attributes of an MC_{St} and of associated MC_{St} super-classes. We obtain this attribute set by invoking the stAttForWriteableTar() query in Line 24. Then, we iterate over the set of returned attributes to create the attribute mappings using the stAttMapping() template.

The attributes of a UML element are mapped in the third part of the operation body. Because metamodel MM_{Domain} contains MM_{AC} metaclasses instead of UML metaclasses, we determine the set of relevant UML attributes based on the attributes inherited from MC_{AC} super-classes. We only consider MC_{AC} attributes that are not defined as 'read-only'. In Template (B), we obtain the set of attributes that match the mentioned criteria by employing the mappableUmlAtt() query in Line 34 of Listing 26. Then, we iterate over the result set to generate the mappings for UML element attributes by invoking the attMapping() templates.

For instance, Fig. 4.27 illustrates the mapping operation that is generated for the MC_{St} Statemachine of our running example (see Fig. 4.6). Because the MC_{St} attribute command is the only attribute not defined as 'read-only', only a single assignment expression for the mapping of stereotype attributes is created. As all other attributes of the «Statemachine» stereotype are defined as 'read-only' and 'derived', their values are computed based on OCL expressions at runtime. These OCL expressions access attributes of the extended UML element so as to obtain the required values.

The assignment expressions for the mapping of UML metaclass attributes are contained in the second part of the operation body. Because the MC_{AC} metaclass AC_StateMachine contains only a small subset of those attributes defined by its 'matching' UML counterpart, a small number of attribute mappings is generated for the shown mapping operation.

```
mapping UML::StateMachine::toStatemachine() : SMDSL::Statemachine
when { self.isStrictStereotypedBy('SMP::Statemachine') }
{
  var stereotype := self.getAppliedStereotype('SMP::Statemachine');
  var stInstance := self.getStereotypeApplication(stereotype)
    .oclAsType(SMDSL::Statemachine);

  // Mapping of stereotype attributes
  result.command += stInstance.command->map toCommand();

  // Mapping of metaclass attributes
  result.name := self.name;
  result.nestedClassifier += self.nestedClassifier->map toAC_Classifier();
  result.ownedBehavior += self.ownedBehavior->map toAC_Behavior();
  result.region += self.region->map toAC_Region();
}
```

Figure 4.27: Generated mapping operation that maps a UML element with applied stereotype

Mapping operations for other element types. The mapping operations of transformation $T_{UML\text{-}to\text{-}UML}$ that map UML element types to corresponding MC_{AC} metaclasses, or 'additional' metaclasses to MC_{AMC} metaclasses, are almost identical to their counterparts of transformation $T_{DM\text{-}to\text{-}UML}$. They differ only in the detail that the types of context and result parameters are swapped. Moreover, mapping operations of this kind map the input elements one-to-one to corresponding output elements, so that their structure is simple. The same holds for mapping operations that map *DataTypes*, and for helper operations that resolve literals of *Enumerations*.

Mapping of attributes. Names for attributes and mapping operations are created in the same manner as in transformation $T_{DM\text{-}to\text{-}UML}$. Therefore, the assignment expressions for mapping operations of $T_{UML\text{-}to\text{-}DM}$ can be generated in the same way as for $T_{DM\text{-}to\text{-}UML}$. For this reason, no further details concerning this topic are discussed here.

4.7 Related work

After presenting the various aspects of our holistic approach to deriving DSL metamodels, corresponding UML profiles, and model transformations so as to achieve model interoperability, we are now in a position to properly discuss related work. We structure the related work analogously to the aforementioned aspects of our derivation approach.

Metamodel generation: In the past two decades, many language workbenches (e.g., [35, 78, 59] and [158, 159, 157, 155]) for DSLs have been developed in order to generate textual editors based on tool-specific notations for defining context-free grammars. Usually, these notations can be considered as variants of the Extended Backus-Naur Form (EBNF) as standardized in [63]. Depending on the employed language work bench, abstract syntax trees (AST) of generated editors can be represented as instances of metaclasses of an associated domain-specific metamodel. All the above-mentioned language work benches provide such a feature.

As a matter of principle, not only can textual notation editors be generated from production rules, but also required domain-specific metamodels can be derived from them by applying one of the algorithms proposed in [6, 153, 44, 94] or some other related works. However, only a small number of the language workbenches, e.g., xText [35], support this functionality.

If metamodels are derived from production rules that are not enriched with further information, these metamodels must be considered as primitive or initial, as discussed in [44] and [94]. This is because contained metaclasses can inherit only from other generated metaclasses, but not from already existing metaclasses that represent abstract modelling concepts. In addition, also advanced metamodelling techniques for attributes (e.g. subsetting and redefinition) as provided by the CMOF [121] cannot be used. Further limitations of such initial metamodels are discussed in [44]. Therefore, a manual refinement of generated metamodels as presented in [44] and [135] is required in order to obtain a metamodel that is comparable from a quality perspective to that one of the UML [115].

Apart from manual refinement, a tool-based approach is conceivable as proposed in [94]. In this paper, a two-staged derivation process is proposed, but only the algorithm for obtaining an initial metamodel (first step) is described in more detail, whereas the second step towards a final metamodel is missing. Unfortunately, this aspect is not covered in other papers of the author of [94], too.

In contrast to these related works, the approach presented by us enables the generation of more sophisticated metamodels. We achieve this by using annotations that are attached to production rules, which serve as input for deriving metamodels. The annotations can be employed to refer to 'Abstract Concepts' defined by an already existing metamodel, and to specify CMOF-provided language concepts, such as redefinition of attributes and subsetting. In such enriched production rules, sufficient information is available to derive CMOF-based instead of EMOF-based metamodels. In addition, the 'Abstract Concepts' enable the use of abstraction in metamodelling. Consequently, the metamodels derived by our approach have a higher degree of quality when compared to the 'initial' metamodels generated by other approaches. However, some manual refinements are still required. For example, the OCL-based static semantics of derived metamodels must be implemented manually.

UML Profile derivation: The most closely related works to ours are [49, 50, 125, 152], which can also by employed for deriving a UML profile from an existing metamodel for a DSL. Their commonality is that, in addition to the metamodel, mapping rules have to be provided as input for the profile derivation. Depending on the approach, this is realized in terms of so called 'Integration Metamodels' or 'Mapping Models'. In contrast, our approach expects CMOF-based metamodels as input for the derivation of UML profiles, which reuse 'Abstract Concepts' as proposed in [44, 135].

Integration Metamodels: Giachetti et al. propose an approach [49, 50] that can be applied to generate UML profiles and mapping rules for model transformations. To derive these artefacts, they employ a certain type of EMOF-based metamodel called 'Integration Metamodels' [52]. Such a metamodel is initially created as a copy of a metamodel for the DSL of interest. In the next step, the copy must be altered manually by adding additional metaclasses, until all rules defined for 'Integration Metamodels' are met. In addition, the structure and semantics of the UML metaclasses that are defined as a mapping target and the ones of the DSL metamodel have to be considered during the rework. Hence and because of the creation of new metaclasses, a revision of the OCL constraints contained in the 'Integration Metamodel' is required. Furthermore, the mappings to UML metaclasses also have to be specified in the 'Integration Metamodel'. Finally, a UML profile and the mapping rules can be generated based on the revised 'Integration Metamodel'.

Giachetti et al. state in [48, 49] that the OCL constraints contained in an 'Integration Metamodel' have to be included in a derived UML profile. Therefore, all references to elements of the 'Integration Metamodel' in OCL constraints are modified, so that corresponding UML metaclasses or stereotypes are referenced after UML profile generation.

In contrast to the proposal of Giachetti et al., our approach for deriving UML profiles can be applied to CMOF-based metamodels created by reusing 'Abstract Concepts'. Due to the correlation between 'Abstract Concepts' and UML metaclasses, we do not have to modify a metamodel so that its structure matches that of the UML metamodel, as is required for an 'Integration Metamodel'. In most cases, we can use these correlations to automatically determine the information required for deriving the different elements of a UML profile. The explicit definition of mapping information is only necessary if 'additional' metaclasses shall be derived, or if a derived *Stereotype* shall extend a UML metaclass that does not have a matching 'Abstract Concept' counterpart.

Another advantage of our approach is the possibility to define 'alternative' OCL expressions, because the value of a stereotype attribute can be computed at runtime without manual specification. This feature is also useful if the value of an attribute can only be determined based on UML model elements that do not have stereotypes applied, or if several model elements must be accessed to obtain the required values.

Equally important and in contrast to our work, the approach of Giachetti et al. does not treat the generation of OCL expressions and constraints for subsetting and redefining stereotype attributes. While, they discuss a transfer of OCL constraints of an 'Integration Metamodel' towards generated stereotypes by only adapting the referenced element types, this is insufficient to obtain valid OCL constraints for generated UML profiles. Because stereotypes and UML metaclasses are instantiated separately, additional attribute navigations must be inserted in OCL expressions, as supported by our approach.

In addition, the approach of Giachetti et al. does not treat the processing of multiple inheritance relationships of metaclasses, the mapping of attributes and operations defined via OCL expressions, and the derivation of 'additional' metaclasses. By contrast, our approach supports these features.

Other approaches: Another category of existing works covers the derivation of metamodels based on existing UML profiles, as proposed in [101, 109], which is exactly the opposite of our approach. The creation of a UML profile from scratch may be an option for a new DSL with low complexity, but for an existing DSL with higher complexity this can be difficult. This is because not only the static semantics of the DSL but also UML's static semantics have to be taken into account. Therefore, the manual creation of a metamodel followed by an automatic derivation of a UML profile should be preferred for more complex DSLs such as SDL [70]. This variant should also be chosen if the production rules for a DSL are given, because then existing tools [55] (e.g., EMFText [59] or xText [16, 35]) can be employed for automatically deriving metamodels.

Derivation of transformations: Our third contribution is an approach for deriving M2M transformations that transform a domain model to a corresponding UML model with an applied UML profile and vice versa. Related approaches can be split into two groups: approaches that expect two input sources, and others that require only a single input source.

Two derivation sources: A methodology for deriving abstract grammars for a UML profile and its corresponding domain-specific metamodel is proposed in [56]. But a formalized mapping between both grammars has to be specified manually. However, different approaches have been proposed to remedy the drawback of a manual specification of mapping rules, for instance in [41] and [98].

Even if the discussed concepts for deriving model transformations differ from each other, a common prerequisite of them is always that a source and a target metamodel have to be present. In contrast, our approach expects only a single metamodel as input for generating model transformations that can be employed to transform a UML model with applied UML profile to a corresponding DSL model, and vice versa.

Single input source: A second contribution of Giachetti et al. [49, 51] is an approach for deriving mapping rules for model transformations based on 'Integration Metamodels'. The authors use these mapping rules as input for a proprietary tool that converts UML models with an applied UML profile into a domain model, and vice versa. The transformation is always conducted in a two-staged process. First, the input model – a UML model or a domain model – is transformed into an intermediate model that is an instance of the DSL-specific 'Integration Metamodel'. Then, the intermediate model is transformed to the target model.

Giachetti et al. argue that the derived mapping rules can also serve as the starting point for implementing model transformations by means of QVT [113] or ATL [75], but this aspect is not treated in [49, 51]. In contrast, our approach directly generates QVT-based model transformations, which respect the peculiarities of the UML profiles derived by us.

4.8 Summary and Conclusions

The drawbacks of manually creating UML profiles identified by us during our work on a new edition of the UML profile for SDL motivated us to explore an automated derivation approach. The main goal of this approach is the automatic derivation of UML profiles from CMOF-based metamodels. In addition, this also includes an automatic update and transfer of the OCL-defined static semantics contained in such metamodels.

In the current chapter, we have presented the details of our derivation approach to meet this objective. To ensure the automatic interoperability between DSL and UML models with applied UML profile, another aspect of our approach captures the derivation of model transformations from metamodels, which are generated based on production rules of existing DSLs or, in case of new DSLs, are created from scratch. In addition, we have introduced an MDE-based development process that can be employed to create the various artefacts, in combination with our tool chain that implements our approach.

To illustrate the various artefacts that can be derived via our approach, we have used a small DSL example. However, this example is insufficient for verifying all aspects of our approach and to validate the usability of the created artefacts. Therefore, our verification and validation efforts shall be on a broader scale, using case studies for existing DSLs. In this context, we must consider that our derivation approach should not only be applicable for DSLs with existing production rules, but also for those defined by a metamodel.

As noted in Sec. 4.2, the generated metamodels still need to be revised manually. Furthermore, the metamodels must be enriched with meta-information so that UML profiles can be derived. We wish to analyse both aspects more closely in the case studies presented in the next chapter.

In contrast to the metamodel derivation, we have claimed that UML profiles derived via our approach do not have to be revised manually. The case studies presented in the next chapter shall demonstrate this aspect, as well as the usability of the UML profiles. In detail, our verification captures each derivation step for UML profiles as well as the automatic update and transfer of the OCL-defined static semantics.

Finally, we have to examine the model transformations that are derived to obtain model interoperability. As for the derived metamodels, these model transformations have to be completed manually, because we can only generate so-called ‘mapping proposals’ due to missing information in metamodels used as derivation source. In the case studies of the next chapter, we prove that our generated and manually completed transformations can indeed be employed to obtain model interoperability.

5 Case Study-Based Evaluation

In this chapter we evaluate the various artefacts generated by our automatic derivation approach presented in the previous chapter. We conduct our evaluation based on two case studies. For this purpose, we use two existing DSLs to compare the artefacts we have derived with the existing artefacts available for these DSLs.

In the first section, we define evaluation criteria that we use at the end of this chapter to evaluate the results obtained. Then, we briefly introduce the design of our evaluation. Based on this foundation, we present our two case studies in detail. In the last section, we discuss and evaluate the results achieved.

5.1 Evaluation Objectives and Requirements

We present the evaluation objectives we have defined, and the criteria used for this in the following. As already mentioned, we intend to conduct our evaluation using case studies, whose design considers the evaluation objectives outlined below.

Using our derivation approach, we can generate a UML profile, model transformations and, optionally, 'additional' metaclasses from a CMOF-based metamodel. During the derivation of a UML profile we also perform an update of the static semantics defined by OCL, so that this can be transferred from a metamodel to the UML profile. If production rules for a DSL are available, we can also generate the required metamodel. For the derivation of these artefacts, we have defined a set of design decisions that we employed as the basis for implementing our tool chain. To verify whether the derivation conforms to our design decisions, we wish to use the artefacts generated in the case studies.

Evaluation Objective 1 *It shall be proven that all artefacts generated using our derivation approach comply with the specified design decisions.*

Our derivation approach should be applicable to either an existing DSL or a new DSL to be created. In the first scenario, a metamodel is semi-automatically created based on production rules and, in the second scenario, manually. Regardless of how a metamodel is created, it must be appropriate as input for deriving a UML profile and model transformations. Therefore, it is necessary to evaluate the applicability of our derivation approach for the two mentioned scenarios separately.

Evaluation Objective 2 *Evaluation that the approach is applicable for an MDE-based development of new DSLs.*

Evaluation Objective 3 *Evaluation that the approach is applicable for existing grammar-based DSLs.*

Even if we verify the individual artefacts created with our derivation approach in Evaluation Objective 1, this does not yet prove that a generated UML profile and its OCL-based static semantics can be used without manual modifications. Therefore, this aspect must be evaluated separately.

Evaluation Objective 4 *Evaluation that a derived UML profile and its static semantics are usable without any manual modification.*

As already pointed out in the last chapter, we can only derive so-called 'mapping proposals' for model transformations based on a single metamodel. Therefore, a manual revision of the generated model transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ is required. The question arises whether the transformations obtained in this way are suitable for achieving model interoperability between UML models with applied UML profile and DSL models.

Evaluation Objective 5 *Evaluation that the semi-automatically derived and manually completed model transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ are applicable to obtain model interoperability.*

5.2 Evaluation Design

Based on case studies, we wish to demonstrate that we are able to derive various artefact types for DSLs by our derivation approach presented in Chap. 4. Due to the various artefact types and the two use cases 'existing DSL with available production rules' and 'newly developed DSLs', we presume that we cannot evaluate all aspects of our derivation approach with a single case study. Therefore, we conduct two case studies for two different DSLs.

The first case study evaluates the use case 'newly developed DSL', where a CMOF-based metamodel is first created manually using an MDE-based development approach, and then all further artefacts are generated based on it. This case study addresses Evaluation Objective 2, which requires an analysis of such a scenario.

We have selected the *Test Description Language (TDL)* [39] as an exemplary DSL to conduct our first case study. TDL is a new and standardized DSL [100, 148] for the design and specification of test descriptions. The development and standardization of the language is driven by the *European Telecommunications Standards Institute (ETSI)*.

Because a metamodel [39] and a corresponding UML profile are available for TDL, we can use the existing TDL metamodel to derive a UML profile and model transformations. To examine the usability of the UML profile created in this way, we compare it with the one standardized for TDL. By doing so, we also anticipate an answer to the question of whether UML profiles generated by us can be employed without manual modification (Evaluation Objective 4). Finally, we investigate the question raised by Evaluation Objective 5 regarding the usability of generated model transformations to achieve model interoperability. For this purpose, we examine the model transformations generated for the TDL case study.

We investigate the use case ‘existing DSL with available production rules’ with a second case study to address Evaluation Objective 3. We opted for the *Specification and Description Language (SDL)* [70] as an exemplary DSL, because this language motivated us to research on the automatic derivation of UML profiles. Furthermore, SDL fulfils the prerequisite of existing production rules and available UML profile [69]. According to the use case to be examined by the SDL case study, we focus on the derivation of CMOF-based metamodels from production rules. Furthermore, we examine the derivation of ‘additional’ metaclasses, because such metaclasses are also specified for the standardized UML profile, and thus a comparison with the ones generated by us becomes possible.

To address the question raised with Evaluation Objective 3, i.e., whether all artefact types are derived in compliance with the design decisions we specified, we first conduct a quantitative analysis in both case studies. This is intended to investigate whether the expected number of element features is derived for each artefact type. Based on the various artefact types generated in both case studies, we then verify in detail whether the generation has been carried out according to the relevant design decisions.

5.3 Case Study A: Test Description Language

The objective of the case study discussed in this section is to analyse and evaluate the applicability of our approach to a new DSL, where the metamodel is not derived based on syntax rules. Hence, we do not apply Step (A) of our overall approach. We employ the metamodel of TDL as starting point and enrich it with the meta-information discussed in Sec. 4.3.3. Then, we derive a UML profile and model transformations by applying Steps (B) – (E) of our approach.

All aspects concerning TDL are specified in the *ES 203 119* standard series. The metamodel of TDL and a corresponding UML profile are defined in document *ES 203 119 – Part 1* [39].

5.3.1 Background

According to Ulricht et al. [148], TDL shall close the gap between high-level test requirement specifications and executable test cases. Thus, TDL test descriptions can serve as the basis to create executable test cases in any kind of target language, such as the *Testing and Test Control Notation – Version 3 (TTCN-3)*. Because TDL is

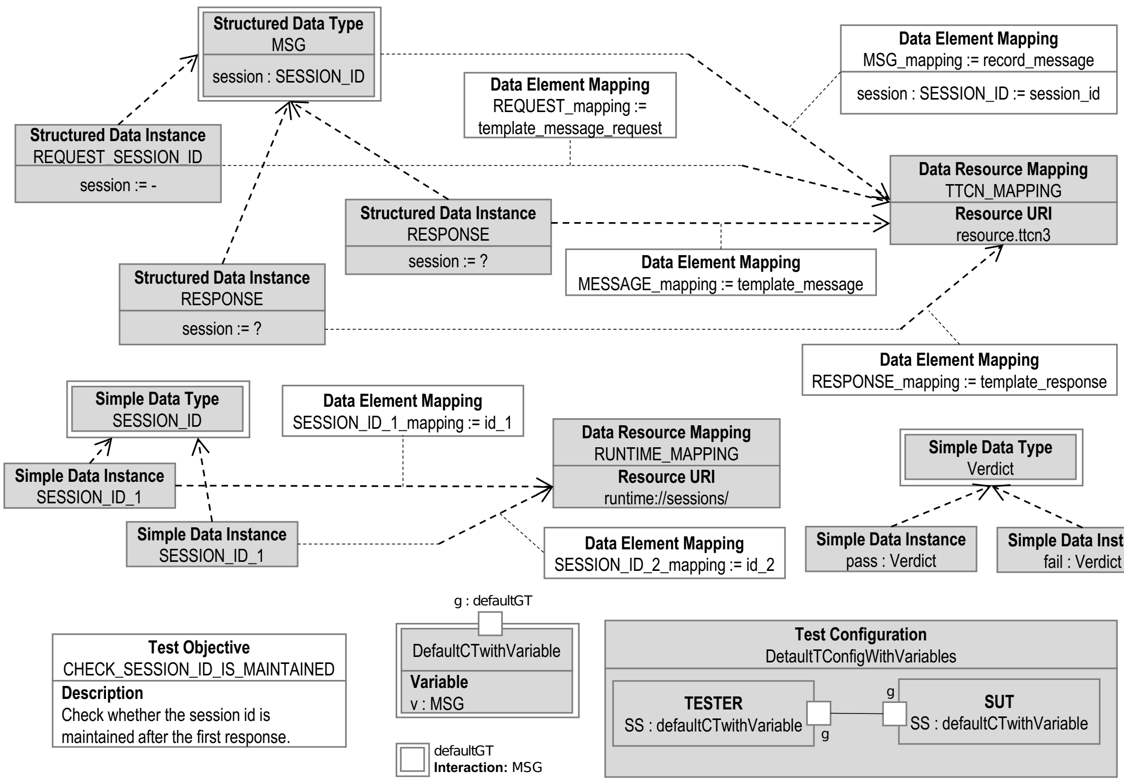


Figure 5.1: Data and test configuration definition example (according to [40] – Annex A.1) specified in TDL's graphical notation

a new language, the literature concerning successful applications of TDL is rather limited. Marroquin et al. [103] report about a successful application of TDL for creating test descriptions for telecommunication systems. Furthermore, the automatic derivation of TDL test descriptions from *Use Case Maps (UCM)* is discussed in [20].

A TDL test description consists of the following parts:

Test configuration: A test configuration defines the tester components and the components of a system under test (SUT) involved in a particular test scenario. All components communicate among each other via defined gates and associated connections.

Test descriptions: A particular test scenario is described in terms of a test description that defines the set of interactions between the test and SUT components of its associated test configuration. Various types of behavioural elements are available for specifying the control flow of a test description, e.g., sequential or parallel behaviour.

Data definitions: Data types can be specified as simple or structured data types, and their instances are used in interactions or can be passed as arguments for the invocation of parameterized test descriptions.

Behavioural elements: The various behavioural element types can be used to define the control flow, send messages, start and stop timers, or set the test result.

Both a graphical and a textual concrete syntax are defined for TDL (see [39, 40]), and thus a test description can be created by means of a graphical or textual notation. A test description example specified by employing TDL's graphical notation is shown in Figs. 5.1 and 5.2. The textual notation example given in Fig. 5.2b is equivalent to the graphical example illustrate in Fig. 5.2a.

5.3.2 Example Model used for the Evaluation

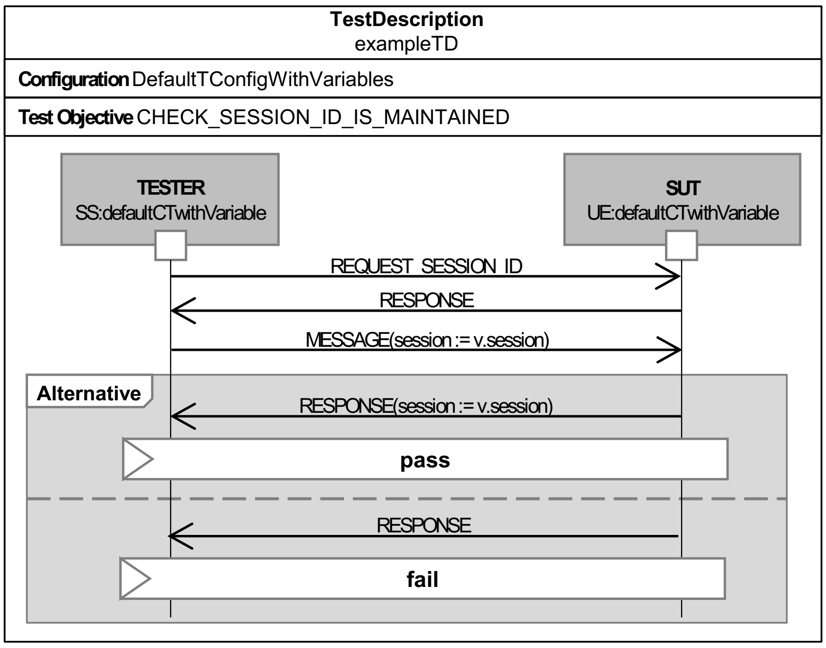
The diagram shown in Fig. 5.1 contains the definitions of data types and associated data instances. In addition, the 'DataResourceMapping' elements are used to specify a resource that contains an external representation for data types or their instances. Such an external representation is identified by a 'DataElementMapping'. For instance, the structured data type MSG is associated with the 'DataResourceMapping' TTCN_MAPPING.

Apart from data type definitions, the shown diagram also contains a *TestConfiguration* consisting of a tester and an SUT component. Because both components are instances of the *ComponentType* DefaultCTwithVariable, each of them owns a gate, which is of type defaultGT. The gates of the component instances are interconnected via a *Connection*.

Based on the type definitions and their instances, we can specify the behaviour of our TDL test description example. The graphical representation of the behaviour is illustrated in Fig. 5.2a, while the corresponding textual specification is shown in Fig. 5.2b. The behaviour specification is contained in a *TestDescription* element that has always to be associated with a *TestConfiguration*. In the case of our example, this is the DefaultTConfigWithVariables. The graphical notation of a *TestDescription* is similar to those notations used for UML sequence diagrams [115] or *Message Sequence Charts (MSC)* [68].

The *TestDescription* shown in Fig. 5.2a contains the tester and SUT component of the associated *TestConfiguration* and a lifeline[1] for each gate of these components is present in the diagram. The message exchange between the components is specified in terms of *InteractionInteraction* elements. For example, the RESPONSE interaction represents a message that is send from the tester to the SUT component. An alternative behaviour with two alternative areas is defined in the second part of the diagram. Finally, each area is terminated by setting a test verdict.

[1] According to [40] a lifeline is a *"vertical line [that] originates from a gate instance or a component instance, to which behavioural elements may be attached"*.


TestDescription
exampleTD
Configuration DefaultTConfigWithVariables
Test Objective CHECK_SESSION_ID_IS_MAINTAINED
TESTER
SS:defaultCTwithVariable
SUT
UE:defaultCTwithVariable
REQUEST_SESSION_ID
RESPONSE
MESSAGE(session := v.session)
Alternative
RESPONSE(session := v.session)
pass
RESPONSE
fail

(a) Test behaviour in TDL's graphical notation

```
Test Description exampleTD
uses configuration DefaultTConfigWithVariables
{
  SS.g sends REQUEST_SESSION_ID to UE.g;
  UE.g sends RESPONSE to SS.g where it is assigned to v;
  SS.g sends MESSAGE(session=SS−>v.session) to UE.g;
  alternatively {
    UE.g sends RESPONSE(session=SS−>v.session) to SS.g;
    set verdict to PASS;
  } or {
    UE.g sends RESPONSE to SS.g;
    set verdict to FAIL;
  } with {
    test objectives : CHECK_SESSION_ID_IS_MAINTAINED
  }
}
```

(b) Test behaviour in TDL's textual notation

Figure 5.2: TDL test behaviour example (according to [39] – Annex B.4)

5.3.3 Preparation of the Metamodel

According to our derivation approach, CMOF-based metamodels can either be semi-automatically generated from production rules of an existing DSL or manually created for a new DSL. We employ TDL as a case study for the latter scenario, where we would typically create a metamodel for this DSL from scratch. Because an EMOF-based metamodel is already provided by the TDL specification [39], we do not have to recreate this artefact. However, we have to make the adaptations and revisions explained below so that we can use this metamodel as input for our derivation approach.

Rework of the metamodel. To derive UML profiles and model transformations by employing our derivation approach, CMOF-based metamodels must reuse 'Abstract Concepts' by inheritance. As outlined in Sec. 4.1.2, these 'Abstract Concepts' are metaclasses taken from the UML metamodel [116] or the UML infrastructure library [114]. In the following we use the already introduced abbreviation MC_{AC} to refer to metaclasses representing 'Abstract Concepts'.

The MDE-based development process for creating metamodels that we have defined in Sec. 4.2.1 implies that the language concepts required for a DSL and MC_{AC} metaclasses representing these concepts must first be identified, before we can create a metamodel. The identified metaclasses are then copied from the UML metamodel or infrastructure library to the metamodel to be created. Then, non-required attributes and operations of the copied MC_{AC} metaclasses are removed, if any.

In the next step, we define the language concepts of a DSL as dedicated metaclasses and relate them to MC_{AC} metaclasses by means of *Generalizations*. Because a metamodel for TDL exists (referred to as MM_{std}), we deviated from our development process, so that we could conduct our case study in an appropriate manner.

TDL's standardized metamodel MM_{std} is divided into several *Packages*, one of which is the Foundation *Package*. The metaclasses contained in this *Package* define generic language concepts, such as *NamedElement* and *PackageableElement*. In a broader sense, these metaclasses are comparable to those contained in the UML 'Kernel' package. Thus, one can assume that these metaclasses of MM_{std} already correspond to the MC_{AC} metaclasses required by our approach. However, differences in syntax and semantics exist, which we analyse below.

All metaclasses of MM_{std} are specified using language concepts provided by EMOF, so their attributes do not use subsetting and redefinition. Furthermore, no 'derived' metaclass attributes are defined in MM_{std}. Consequently, elements of TDL do not support an automatic value computation for attributes at runtime. Another difference between TDL's metaclasses contained in the Foundation *Package* and the corresponding UML metaclasses is that the TDL metaclass attributes have different names when compared to the corresponding ones in UML.

The names of metaclass attributes contained in the Foundation *Package* of MM_{std} do not match those of UML metaclass attributes. However, identically named attributes of 'Abstract Concepts' and UML metaclasses are a prerequisite (see
Sec. 4.1.2) of our derivation approach. Therefore, we cannot use MM_{std} as input to derive a UML profile using our approach. Due to this reason, we have aligned MM_{std} to meet the requirements of our derivation approach. In the following, we refer to the revised metamodel of TDL using the acronym MM_{TDL}.

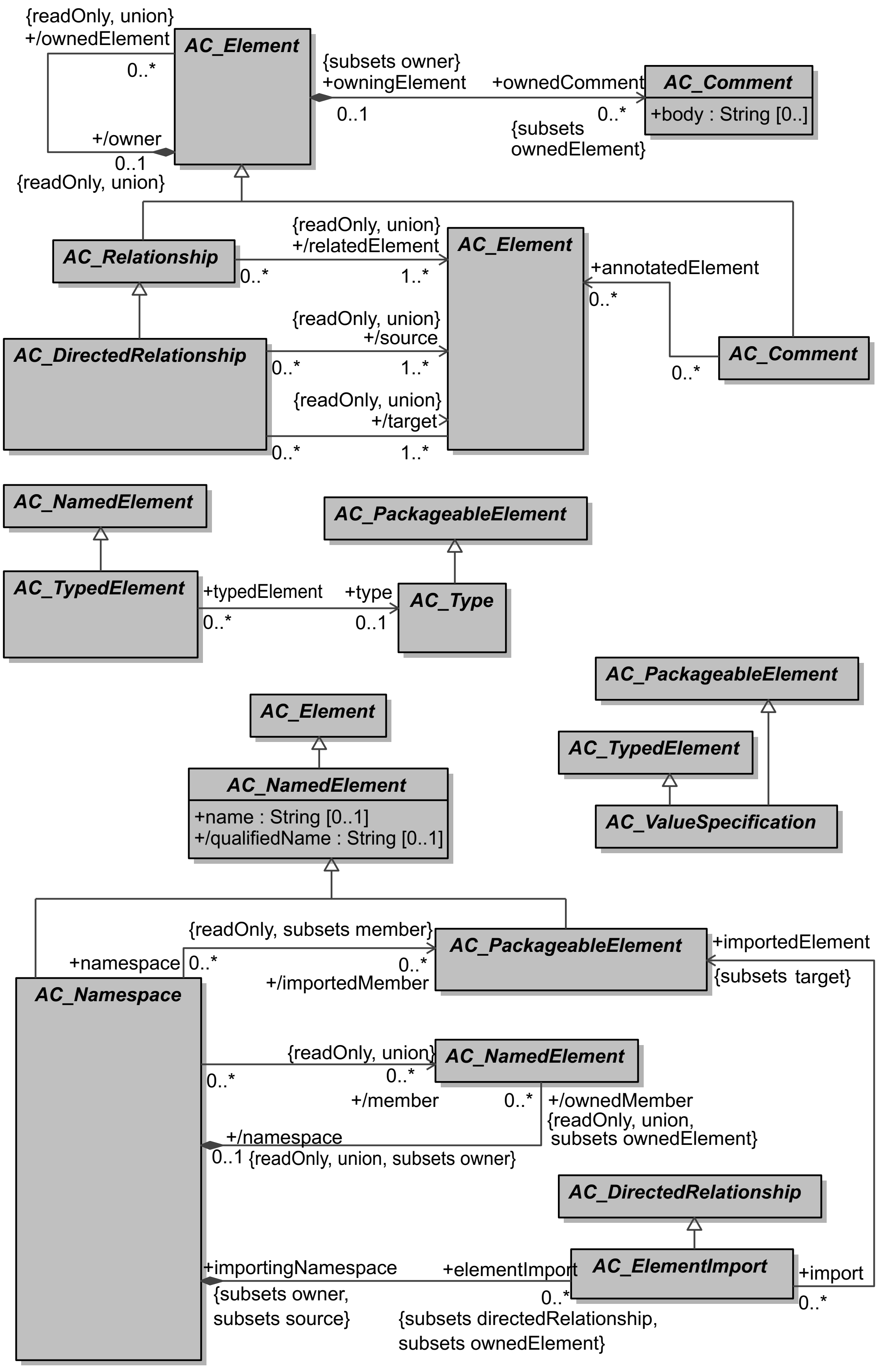


Figure 5.3: MC_{AC} metaclasses defined for creating the metamodel MM_{TDL}

In a first step, we copy the metaclasses from MM_{std} to MM_{TDL}. Then, we replace the metaclasses contained in TDL's Foundation *Package* with metaclasses of the same name from the UML Kernal *Package*. In addition to the replaced ones, we adopt further metaclasses from the UML Kernel *Package*. The most important of these are AC_TypedElement and AC_ValueSpecification, which are used in the context of data type and value specifications of TDL. The metaclasses shown in Fig. 5.3 represent all MC_{AC} metaclasses defined for our MM_{TDL} metamodel.

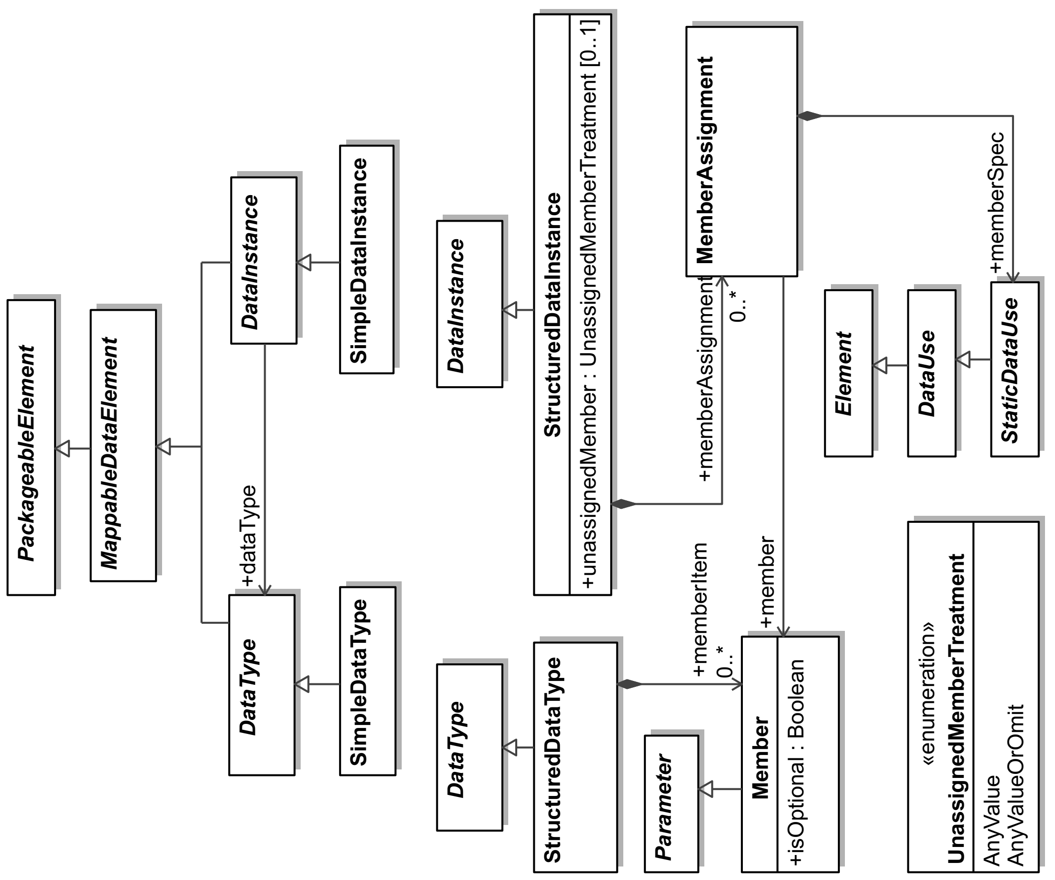
After preparing the MM_{AC} metaclasses as discussed, we put them in inheritance relationships to the TDL metaclasses by employing *Generalization* relationships. Additionally, we redefine or subset the properties inherited from MC_{AC} metaclasses where required. Our most important goal for the revision of MM_{TDL} is not to change the structure of the metaclasses adopted from MM_{std}.

An example of our modifications to metaclasses in MM_{TDL} is shown in Fig. 5.4b, where the metaclasses contained therein define the syntax for TDL data types and value specifications. The original metaclasses in MM_{std} are shown in Fig. 5.4a. If we compare both figures, we find that no other modifications are made to the TDL metaclasses than those described.

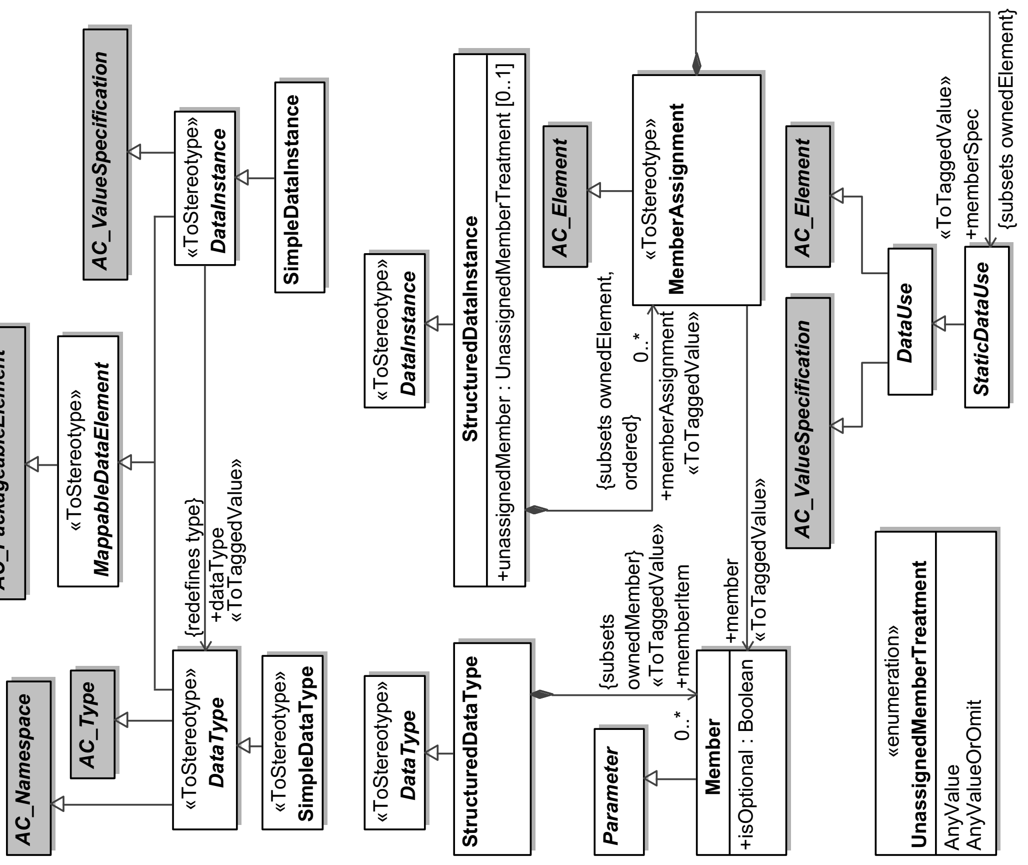
Specification of meta-information. Some metaclasses (e.g., MappableDataElement) and attributes shown in Fig. 5.4b have applied *Stereotypes* of our UML profile MM2Profile. These *Stereotypes* define meta-information required for deriving a UML profile, as explained in Sec. 4.3.3. By default, we assume that metaclasses whose names start without the prefix 'AC_' represent MC_{St} metaclasses that are to be mapped to *Stereotypes* of a UML profile.

In many cases, we can use the inheritance relationships between MC_{St} and MC_{AC} metaclasses to determine the UML metaclasses to be extended by created *Stereotypes*. However, this is infeasible if only MC_{AC} metaclasses with a high level of abstraction are used, as is the case of our TDL case study. For example, without additional meta-information, a *Stereotype* derived from the DataType metaclass shown in Fig. 5.4b would extend the UML metaclass *Type* or *Namespace*. Consequently, this *Stereotype* could then be applied to all UML elements that are instances of subclasses of these two UML metaclasses. However, this would not correspond to the syntax specified by the MM_{TDL} metamodel. Applying the «ToStereotype» stereotype on an MC_{St} metaclass remedies this issue, because its extendedMetaclass attribute can be employed to explicitly specify the UML metaclass to be extended by a *Stereotype*.

In addition to defining the UML metaclass to be extended, the noMapping attribute of the «ToStereotype» stereotype can be used to define that no *Stereotype* shall be derived from an MC_{St} metaclass marked in this way. However, this option should only be used if an MC_{St} metaclass does not have attributes, operations and OCL constraints. If a *Stereotype* were derived from such a metaclass, its use would be rather limited from a semantic and syntactical point of view. Because



(a) Metaclasses of MM_{std}



(b) Metaclasses of MM_{TDL}

Figure 5.4: Metaclasses that define TDL's data type concept

the MappableDataElement metaclass shown in Fig. 5.4b meets these criteria, we apply the aforementioned option to it.

If an MC_{St} attribute shall not be mapped according to the default rules of our derivation approach, then this can be specified using the «ToTaggedValue» stereotype. Consequently, an MC_{St} attribute marked in this way is mapped to a stereotype attribute that is defined as 'read-only' and 'derived'. The «ToTaggedValue» stereotype provides the attributes derivationSource and oclSpecification. The latter attribute can be used to define an OCL expression that specifies how the value of a stereotype attribute is to be computed at runtime. Because OCL expressions cannot be evaluated without parsing them, the derivationSource attribute of the «ToTaggedValue» stereotype can be employed to refer to a UML metaclass attribute that is used as the computation basis for the specified OCL expression. However, this information is only required for deriving model transformations but not for deriving UML profiles.

As Fig. 5.4b only shows the *Stereotypes* applied to elements of the metamodel, but not the values of the stereotype attributes, we summarize this information in Table 5.1 in relation to the associated metaclass.

Using the example of the MC_{St} metaclass DataType and its two subclasses, we briefly discuss the purpose of the items shown in Table 5.1. Based on the table entry for the MC_{St} metaclass DataType we find that the *Stereotype* derived from this metaclass shall extend the UML metaclass DataType. Although the metaclass StructuredDataType has not applied the «ToStereotype» stereotype because of its inheritance relationship to DataType, it is also automatically considered as MC_{St} metaclass. By contrast, we have specified that the *Stereotype* to be derived from the SimpleDataType metaclass shall extend the UML metaclass PrimitiveType.

As shown in Fig. 5.4b, the memberItem attribute of the MC_{St} metaclass StructuredDataType subsets the ownedMember attribute that is inherited from AC_Namespace. According to our derivation approach, the memberItem attribute would be mapped to a simple stereotype attribute to which values must be assigned manually at runtime. However, because the ownedAttribute property of the UML metaclass DataType contains exactly the required information, the values for the stereotype attribute can also be computed at runtime using OCL. Therefore, we applied the «ToTaggedValue» stereotype to the memberItem attribute of the MC_{St} StructuredDataType, so that we can define an appropriate OCL expression using the oclSpecification attributes as shown in Table 5.1.

Rework of OCL constraints. In addition to the operations provided by the OCL standard library, the TDL specification [120] introduces four OCL queries invoked by OCL constraints defined in the TDL metamodel. The operation signatures and a description (see Fig. 5.5a) in natural language are given for these queries, but a OCL specification is missing. Thus, when implementing the TDL metamodel, the

Table 5.1: Stereotypes of 'MM2Profile' applied to metaclasses and attributes of MM_{TDL} shown in Fig. 5.4b

Metaclass MappableDataElement
Attribute values of the «ToStereotype» stereotype: – extendedMetaclass: empty – noMapping: true
Metaclass DataType
Attribute values of the «ToStereotype» stereotype: – extendedMetaclass: UML::DataType – noMapping: false
Metaclass SimpleDataType
Attribute values of the «ToStereotype» stereotype: – extendedMetaclass: UML::PrimitiveType – noMapping: false
Metaclass DataInstance
Attribute values of the «ToStereotype» stereotype: – extendedMetaclass: UML::InstanceSpecification – noMapping: false
Metaclass attribute DataInstance::dataType
Attribute values of the «ToTaggedValue» stereotype: – derivationSource: UML::InstanceSpecification::classifier – oclSpecification: **if self**.base_InstanceSpecification.classifier   –>one(isStereotypedBy(`UP4TDL::DataType')) **then**   **self**.base_InstanceSpecification.classifier   –>any(isStereotypedBy(`UP4TDL::DataType')).oclAsType(UML::DataType) **else null endif**
Metaclass attribute StructuredDataType::memberItem
Attribute values of the «ToTaggedValue» stereotype: – derivationSource: UML::StructuredClassifier::ownedAttribute – oclSpecification: **self**.base_DataType.ownedAttribute–>asOrderedSet()
Metaclass attribute StructuredDataInstance::memberAssignment
Attribute values of the «ToTaggedValue» stereotype: – derivationSource: UML::InstanceSpecification::slot – oclSpecification: **self**.base_InstanceSpecification.slot–>asOrderedSet()

Table 5.1: Stereotypes of ‘MM2Profile’ applied to metaclasses and attributes of MM_{TDL} shown in Fig. 5.4b (continued)

Metaclass MemberAssignment
Attribute values of the «ToStereotype» stereotype: – extendedMetaclass: UML::Slot – noMapping: false
Metaclass attribute MemberAssignment::member
Attribute values of the «ToTaggedValue» stereotype: – derivationSource: UML::Slot::definingFeature – oclSpecification: **self**.base_Slot.definingFeature
Metaclass attribute MemberAssignment::memberSpec

Attribute values of the «ToTaggedValue» stereotype:
– derivationSource: UML::Slot::value
– oclSpecification:
if self.base_Slot.value−>notEmpty() **then**
  self.base_Slot.value−>first()
else null endif

OCL-defined queries must be created from scratch based on the textual description.

OCL-based implementation of TDL’s predefined queries: Because no standardized mechanism for extending the OCL standard library exists and OCL queries can be specified in the context of metaclasses, we opted for a metamodel-based implementation. Deviating from the TDL standard, we have implemented three of the above queries as ‘derived’ metaclass attributes (see Fig. 5.5b), as we consider the addressed aspects as properties of metaclasses.

Because the *owner* property of the *AC_Element* metaclass (see Fig. 5.3) always refers to an element that contains the current one, and because that each TDL metaclass inherits from this metaclass, we employ the *owner* property instead of the container() query.

According to the TDL specification [39], the getTestDescription() query shall be defined in the context of OCL’s predefined data type OclAny. Consequently, the applicability of this query is not limited to TDL elements; instead, it can be applied to any kind of metaclass or data type instance, e.g., UML metaclasses instances. However, the getTestDescription() query is called only by three OCL constraints contained in MM_{TDL}. Furthermore, these constraints are specified in the context of metaclasses inheriting from *AtomicBehaviour*. Hence, we define the query in the context of the *AtomicBehaviour* metaclass as shown in Lines 1 to 6 of Fig. 5.5b.

Instead of the getDataType() query specified by TDL, we introduce the data Type() query and the ‘derived’ attribute *determinedType* for the *DataUse* metaclass.

$\mathrm{OclAny::container()} : \mathrm{Element}$ – query that is applicable on any kind of TDL element to determine the element that directly contains the current context element of the query.

$\mathrm{OclAny::getTestDescription()} : \mathrm{TestDescription}$ – query that can be applied to any kind of TDL element to obtain the *TestDescription* that directly or indirectly contains the current context element of the query.

$\mathrm{DataUse::getDataType()} : \mathrm{DataType}$ – used to resolve the *DataType* of a *DataUse* item that is passed as context argument.

$\mathrm{Behaviour::isTesterInputEvent()} : \mathrm{Boolean}$ – employed to determine if the *Behaviour* item passed as argument is a tester-input event as defined in the TDL specification [39].

(a) The predefined queries that are standardized for TDL [39]

```
context AtomicBehaviour::getTestDescription() : TestDescription
body:
if self.allNamespaces()−>one(oclIsTypeOf(TestDescription)) then
  self.allNamespaces()−>any(oclIsTypeOf(TestDescription))
    .oclAsType(TestDescription)
else null endif

context DataUse::determinedType : DataType
derived: self.dataType()

context DataUse::dataType() : DataType
body: null

context FormalParameterUse::dataType() : DataType
body: if self.parameter <> null then
  self.parameter.dataType
else null endif

context Behaviour::isTesterInputEvent : Boolean
derived:
if ( self.oclIsTypeOf(Quiescence) or self.oclIsTypeOf(TimeOut)
      or self.oclIsTypeOf(Interaction) ) then
  true
else false endif
```

(b) The OCL-based implementation of the predefined queries for MM_{TDL}

Figure 5.5: TDL's predefined queries and their implementation for MM_{TDL}

Based on this query, we then compute the value of the *determinedType* attribute at runtime. Because of the various subclasses of the *DataUse* metaclass, we employ the operation redefinition feature of CMOF to define the dataType() query in a clear way. Therefore, subclasses inheriting from the *DataUse* metaclass owns specific queries that redefine the dataType() query of *DataUse*.

The definition of the dataType() query owned by *DataUse* and one of the redefining queries is shown in Lines 8 to 17. Because of the redefinition, one of the redefining queries is called instead of the *DataUse* query. Thus, we define the body expression of the latter as 'null' because it is never invoked.

Furthermore, we introduce the 'derived' attribute *isTesterInputEvent* for the *Behavior* metaclass instead of the corresponding query as is defined by the TDL specification. In contrast to the previous *determinedType* attribute, we do not employ a set of redefining operations for specifying the *isTesterInputEvent*. This is because we only have to take into account three cases where a *Behavior* element is considered to be a TDL tester-input event. Hence, we specified the *isTesterInputEvent* as shown in Lines 19 to 24 of Fig. 5.5b.

Revision of the standardized OCL constraints: After implementing the three 'derived' properties and the getTestDescription() query as discussed, we revise all OCL conditions defined for the TDL metamodel so as to invoke the properties and the query we have introduced, rather than the queries defined by TDL. For instance, consider the following OCL constraint for which we replaced the getDataType() query invocation with the determinedType property invocation.

```
// OCL constrained as defined in the TDL specification
self.memberSpec.getDataType() = self.member.dataType
// Reworked OCL constraint
self.memberSpec.determinedType = self.member.dataType
```

Apart from the modifications discussed above, further revisions are required due to errors of the OCL constraints contained in the TDL specification. In general, we have found two categories of errors when implementing the TDL metamodel.

The first category is caused by using a non-existing operator on collection types. As shown in the example below, the contains() operator is used instead of the includes() operator to determine whether a given element is member of a collection type. Therefore, we replace every invocation of the contains() operator with the includes() operator.

```
// OCL constrained as defined in the TDL specification
self.memberAssignment−>forAll(a | self.dataType.member−>contains(a.member))
// Reworked OCL constraint
self.memberAssignment−>forAll(a | self.dataType.member−>includes(a.member))
```

Table 5.2: Comparison between model elements of MM_{std} and MM_{TDL}

MM_{std}			MM_{TDL}
Metaclasses	**87**	**93**	**Metaclasses**
		13	MC_{AC} metaclasses
		80	MC_{St} metaclasses *(59 with* «ToStereotype» *stereotype)*
Metaclass attributes	**106**	**126**	**Metaclass attributes**
		27	**MC_{AC} attributes**
		99	**MC_{St} attributes** *(39 attributes with* «ToTaggedValue» *stereotypes)*
		3	MC_{St} attributes defined as 'derived' and 'read-only' *(1 attribute with* «ToTaggedValue» *stereotype)*
		9	Redefining MC_{St} attributes *(2 attributes with* «ToTaggedValue» *stereotype)*
		43	Subsetting MC_{St} attributes *(29 attributes with* «ToTaggedValue» *stereotype)*
		44	Other kind of MC_{St} attributes *(7 attributes with* «ToTaggedValue» *stereotype)*
Enumerations	3	3	*Enumerations*
OCL *Constraints*	67	67	OCL *Constraints*
OCL *Operations*	0	10	OCL *Operations*

The second error category we have identified concerns missing type-casts in OCL constraints defined by the TDL specification. To remedy this, we introduce the required type-casts using the the pre-defined oclAsType() operation.

Comparison between MM_{std} and MM_{TDL}. In Table 5.2 we compare the most essential elements of the standardized metamodel MM_{std} and our metamodel MM_{TDL}. This table shows that MM_{TDL} contains 93 metaclasses, whereas MM_{std} subsumes only 87 metaclasses. This difference is due to our discussed introduction of 13 MC_{AC} metaclasses to MM_{TDL}. On 59 metaclasses of MM_{TDL} we have applied the «ToStereotpye» stereotype, so that they and their inheriting subclasses are identifiable as MC_{St} metaclasses.

When deriving UML profiles, we distinguish between several types of metaclass attributes in a metamodel, which serves as input for the derivation. Table 5.2 summarizes the types of metaclass attributes relevant to our derivation approach. The 126 metaclass attributes contained in MM_{TDL} consist of 27 MC_{AC} attributes and 99 MC_{St} attributes. Because we do not derive model elements from MC_{AC} metaclasses, we do not further classify these attributes in Table 5.2.

We divide the 99 MC_{St} attributes in MM_{TDL} into four categories. Three MC_{St} attributes are specified as 'derived' and 'read-only'. Nine MC_{St} attributes redefine attributes of other metaclasses, and 43 MC_{St} attributes subset other metaclass attributes. Without employing the «ToTaggedValue» stereotype, all these MC_{St} attributes would be mapped to stereotype attributes specified as 'derived' and 'read-only', and OCL expressions would be generated to compute the attribute values at runtime.

As we did not intend to modify the metamodel standardized for TDL, we have only replaced existing metaclasses related to 'Abstract Concepts' with those of the UML 'Kernel' package, as described above. Because these metaclasses only represent very generic language concepts, e.g., name spaces or inheritance, an automatic generation of OCL expressions for 'derived' and 'read-only' stereotype attributes that are created based on subsetting/redefining metaclass attributes is infeasible in most cases. For this reason, we apply the «ToTaggedValue» stereotype to 39 MC_{St} attributes to specify OCL expressions that are used instead of the automatically generated ones.

Apart from metaclasses, MM_{std} and MM_{TDL} contain three *Enumerations* and 67 OCL *Constraints*, which are revised as explained above. Furthermore, we have implemented 10 OCL-defined *Operations* for MM_{TDL}, whereas MM_{std} does not contain such *Operations*. This difference is because we have introduced OCL-defined attributes and operations for metaclasses to allow the reuse of basic functionalities by the *Constraints* of the static semantics of MM_{TDL}. For instance, the type-conformance check is such a functionality.

5.3.4 Evaluation of the Derived UML Profile

In this section we verify the artefacts that we have derived based on the MM_{TDL} metamodel presented above. In detail, we verify the generation of a UML profile, the update and transfer of the OCL-defined static semantics, and the generation of model transformations based on our TDL case study. Because the derivation of a metamodel from production rules and the optional generation of 'additional' metaclasses are not part of this case study, we cannot conduct a verification for these artefact types. However, we verify them in the context of our SDL case study presented in a Sec. 5.4.

Evaluation Objective 1 requires the evidence that all artifacts generated by our approach are derived according to the defined design decisions. Therefore, we use these design decisions as evaluation criteria for verifying the created artifacts below.

Overview of the generated UML profile UP_{TDL}. In Table 5.3 we compare the elements contained in MM_{TDL} with those derived for the UML profile UP_{TDL}. According to Design Decision 10, MC_{AC} metaclasses are not mapped to any model element of a derived UML profile. Accordingly, the 13 MC_{AC} metaclasses in MM_{TDL}

have no counterparts in UP_{TDL}. If we subtract these 13 metaclasses from the 93 metaclasses contained in MM_{TDL}, 80 metaclasses remain. Because none of these metaclasses has applied the «ToMetaclass» stereotype, we consider them as MC_{St} metaclasses, which we map to *Stereotypes* of the UML profile to be generated.

The «ToStereotype» stereotype is applied to 59 of the 80 MC_{St} metaclasses in MM_{TDL}, mainly to define the UML metaclasses to be extended by generated *Stereotypes*, as defined by Design Decision 18. Based on the number of MC_{St} metaclasses in MM_{TDL}, one could assume that UP_{TDL} contains a total of 80 *Stereotypes*. As indicated in Table 5.3, only 79 *Stereotypes* are created for UP_{TDL}. This corresponds exactly to our desired result, because we have defined the MC_{St} MappableDataElement in MM_{TDL} as not to be mapped.

Stereotype extensions and generalizations: To explain more clearly for which kinds of MC_{St} metaclasses *Stereotypes* are introduced either with *Extensions* or with *Generalizations*, we distinguish the following two groups of MC_{St} metaclasses in Table 5.3:

- For MC_{St} metaclasses that inherit directly from MC_{AC} metaclasses, we introduce an *Extension* between generated *Stereotypes* and UML metaclasses identified based on the MC_{AC} metaclasses in accordance with Design Decision 13. Because one of the 33 MC_{St} metaclasses that inherit directly from MC_{AC} metaclasses is specified as not mappable, we derive only 32 *Stereotypes* with an *Extension* to a UML metaclass. 29 of these MC_{St} metaclasses have the «ToStereotype» stereotype applied, so that the UML metaclasses to be extended are explicitly specified (according to Design Decision 18) and not determined based on MC_{AC} metaclasses.
- All MC_{St} metaclasses that inherit from other MC_{St} metaclasses by means of *Generalizations* are summarized as a second group in Table 5.3. This group comprises 47 MC_{St} metaclasses of which 30 have the «ToStereotype» stereotype applied. For all *Stereotypes* that are generated based on these 47 MC_{St} metaclasses, we introduce *Generalizations* to other *Stereotypes* (Design Decision 14). In addition, we create 30 *Extension* relationships from *Stereotypes* to UML metaclasses, because the source MC_{St} metaclasses have applied the «ToStereotype» stereotype (Design Decision 18).

Stereotype attributes: Due to the higher number of relevant design decisions, the mapping of attributes is more complex than that of *Stereotypes*. Accordingly, several types of attribute mappings are regarded in Table 5.3. Furthermore, we distinguish the following kinds of stereotype attributes that are created as mapping result:

'Derived' stereotype attributes: In this group, we summarize all mapped stereotype attributes that are defined as 'derived' and 'read-only'.

Table 5.3: Comparison of metamodel MM_{TDL} and the UML profile UP_{TDL}

MM_{TDL}		UP_{TDL}	
Metaclasses			
MC_{AC} metaclasses	13	—	
MC_{St} metaclasses (59 with applied «ToStereotype» stereotype)	**80**	**79**	***Stereotypes***
• MC_{St} metaclasses inheriting from MC_{AC} metaclasses	33		
• 29 MC_{St} metaclasses with applied «ToStereotype» stereotype		32	*Stereotypes* with *Extensions* to UML metaclasses
• MC_{St} metaclasses inheriting from MC_{St} metaclasses	47	17	*Stereotypes* inheriting from other *Stereotypes*
• 30 MC_{St} metaclasses with applied «ToStereotype» stereotype		30	*Stereotypes* that inherit from other *Stereotypes* and have *Extensions* to UML metaclasses
		Stereotype attributes	
Metaclass attributes	**126**	‘non-derived’	‘derived’ & ‘read-only’
MC_{AC} attributes	27	—	—
MC_{St} attributes (39 with applied «ToTaggedValue» stereotypes)	**99**	**52**	**47**
• MC_{St} attributes defined as ‘derived’ and ‘read-only’	3	—	2
• 1 attribute has applied the «ToTaggedValue» stereotype		—	1
• Redefining MC_{St} attributes	9	1	6
• 2 attributes have applied the «ToTaggedValue» stereotype		—	2
• Subsetting MC_{St} attributes	43	14	—
• 29 attributes have applied the «ToTaggedValue» stereotype		—	29
• Other kind of MC_{St} attributes	44	37	—
• 7 attributes have applied the «ToTaggedValue» stereotype		—	7
Enumerations	3	3	***Enumerations***
OCL *Constraints* of metaclasses	67	113	**OCL *Constraints* of *Stereotypes***
		67	Copied and updated from MM_{TDL}
		40	Introduced for ‘non-derived’ stereotype attributes that refer to UML classes with applied stereotypes
		6	Created for redefining MC_{St} attributes
OCL *Operations*	10	10	OCL ***Operations***

'Non-derived' stereotype attributes: This group captures all stereotype attributes that do not belong to the first group.

As shown in Table 5.3, the MM_{TDL} metamodel contains a total of 126 metaclass attributes, of which 99 are considered as MC_{St} attributes that map to 52 'non-derived' and 47 'derived' stereotype attributes. The remaining 27 attributes are owned by MC_{AC} metaclasses and, therefore, are not considered when deriving the UML profile UP_{TDL}.

To discuss the mapping of MC_{St} attributes to 'derived' or 'non-derived' stereotype attributes, we group them according to their type in Table 5.3. In addition, we specify how many of the MC_{St} attributes of a group have applied the «ToTaggedValue» stereotype. This is because the application of this *Stereotype* overrides the default mapping for MC_{St} attributes, so that such kind of attributes are mapped to 'derived' stereotype attributes.

The four MC_{St} attribute groups considered in Table 5.3 are as follows:

1. The first group of MC_{St} attributes subsumes all MC_{St} attributes that are already defined as 'derived' and 'read-only' in MM_{TDL}. By default, MC_{St} attributes of this group are mapped to stereotype attributes also defined as 'derived' and 'read-only'. When mapping such attributes, we copy an OCL expression available as *defaultValue* and subjects it to the OCL update described in the following section.

   However, if an MC_{St} attribute of the first group of MC_{St} attributes has applied the «ToTaggedValue» stereotype, we use the OCL expression defined by this *Stereotype* instead of the existing one in MM_{TDL}.

   Overall, the MM_{TDL} metamodel contains three MC_{St} attributes defined as 'derived' and 'read-only', of which one has applied the «ToTaggedValue» stereotype. Accordingly, existing OCL expressions were transferred to UP_{TDL} for only two MC_{St} attributes. In the case of the MC_{St} attribute with applied «ToTaggedValue» stereotype, the OCL expression defined by this *Stereotype* was copied to UP_{TDL}.

2. The second MC_{St} attribute group contains nine MC_{St} attributes that redefine other metaclass attributes. Eight of these MC_{St} attributes are mapped to 'derived' and 'read-only' stereotype attributes, where six attributes are mapped based on Design Decision 16 and two others based on the applied «ToTaggedValue» stereotype, as required by Design Decision 19. One further MC_{St} attribute of the group is mapped to a 'non-derived' stereotype because it redefines an MC_{St} attribute and is therefore not captured by Design Decision 16.

3. All attributes that subset other metaclass attributes are captured by the third MC_{St} attribute group. In total, this group comprises 43 MC_{St} attributes,

of which 29 have the «ToTaggedValue» stereotype applied. Based on Design Decision 16 these 29 attributes are mapped to 'derived' and 'read-only' stereotype attributes. In addition, we introduce OCL expressions as *defaultValues*, which are used to compute the values of stereotype attributes at runtime.

One could now expect that all 14 remaining MC_{St} attributes would also be mapped to 'derived' and 'read-only' stereotype attributes according to the rules defined by Design Decision 16. However, this is not the case because we have supplemented these rules during the implementation of our derivation approach, as argued in Sec. 4.3.4.

Because of these more specific rules, only those MC_{St} attributes, which redefine or subset MC_{AC} attributes not declared as 'derivedUnion' or 'read-only', are mapped according to Design Decision 16. As the 14 remaining MC_{St} attributes do not meet this condition, they are mapped to 'non-derived' stereotype attributes.

4. The last MC_{St} attribute group comprises all attributes that are not captured by one of the three other groups. This applies to 44 MC_{St} attributes, of which seven have applied the «ToTaggedValue» stereotype, so they are mapped to seven stereotype attributes that are defined as 'derived' and 'read-only' (Design Decision 19). The 37 remaining metaclass attributes of this group are mapped one-to-one to 'non-derived' stereotype attributes according to Design Decision 15.

OCL-defined elements: The MM_{TDL} metamodel contains 67 OCL *Constraints* and ten *Operations* defined by OCL, which we map to corresponding elements in UP_{TDL} after a previous update. If we compare MM_{TDL} and UP_{TDL}, we find that UP_{TDL} comprises 113 OCL *Constraints* instead of 67. This difference is because we are introducing additional OCL *Constraints* for *Stereotypes* in UP_{TDL}, as required by Design Decisions 20 and 21.

According to the rules established by Design Decision 20, OCL *Constraints* must be created for all MC_{St} attributes mapped to 'non-derived' stereotype attributes in order to ensure that only UML elements with an appropriate *Stereotype* can be assigned to these attributes. Only MC_{St} attributes whose *type* property refers to an MC_{St} metaclass are considered. Table 5.3 shows that UP_{TDL} contains a total of 52 'non-derived' stereotype attributes, but only 40 OCL *Constraints* are created for these according to Design Decision 20. This is because only 40 of the 52 'non-derived' stereotype attributes are derived from MC_{St} attributes whose *type* properties refer to MC_{St} metaclasses. The remaining 12 MC_{St} attributes refer to *DataTypes* and are therefore not captured by Design Decision 20. Accordingly, we do not have to create any associated OCL *Constraints* for these 12 attributes.

Additional OCL *Constraints* are provided according to Design Decision 21 for 'derived' and 'read-only' stereotype attributes derived from subsetting or redefining MC_{St} attributes. These OCL *Constraints* ensure that only UML elements with matching *Stereotypes* can be assigned to the UML metaclass attributes that serve as computation source for the values of the stereotype attributes in question. In the case of UP_{TDL}, this only applies to six stereotype attributes derived from redefining MC_{St} attributes. Therefore, only six OCL constraints are introduced based on Design Decision 21.

Verification of the elements derived for UP_{TDL}. After we quantitatively analysed the generated UML profile UP_{TDL}, we verify whether the elements in UP_{TDL} are derived in compliance to Design Decisions 10 to 21. The verification is performed using the metaclasses of MM_{TDL} presented in Fig. 5.6a, which define the syntax for data type specifications of TDL. The *Stereotypes* derived based on these metaclasses are shown in Fig. 5.6b.

If we compare Figs. 5.6a and 5.6b one of the main differences is that Fig. 5.6b does not contain a stereotype for the MC_{St} metaclass MappableDataElement. This is because we have applied the «ToStereotype» stereotype to this metaclass and assigned the value 'true' to its noMapping attribute. Consequently, no stereotype is derived for this MC_{St}.

Next, we verify the derivation of *Stereotypes* and their associated *Extensions* and *Generalizations*. As specified by Design Decision 11, MC_{AC} metaclasses must not be mapped to *Stereotypes* or other model elements in UP_{TDL}. Our example in Fig. 5.6a contains seven MC_{AC} metaclasses. Compared with Fig. 5.6b, we find that there are no elements with the name prefix 'AC_'. Thus, we conclude that MC_{AC} metaclasses are not mapped to model elements in UP_{TDL}. Therefore, we consider Design Decision 11 as successfully verified.

All metaclasses in MM_{TDL} that are not identified as MC_{AC} metaclasses and not stereotyped with «ToMetaclass» are considered as MC_{St} metaclasses. Based on Design Decision 12 these metaclasses are mapped to *Stereotypes*. Apart from the MC_{St} MappableDataType that is not to be mapped, our example in Fig. 5.6a contains eleven further MC_{St} metaclasses that are mapped to exactly eleven *Stereotypes*, as shown in Fig. 5.6b. For this reason, we consider Design Decision 12 to be successfully verified.

An *Extension* between a *Stereotype* and a UML metaclass is introduced either implicitly based on Design Decision 13 or explicitly by applying the «ToStereotype» stereotype, which we use to implement Design Decision 18. For the implicit variant, we assume that *Extensions* are only introduced for *Stereotypes* derived from MC_{St} metaclasses that inherit directly from MC_{AC} metaclasses. In the case of our example, this variant does not apply because all relevant MC_{St} metaclasses in Fig. 5.6a have applied the «ToStereotype» stereotype; consequently, *Extensions* are introduced based on Design Decision 18. In addition, according to Table 5.3, the entire metamodel MM_{TDL} contains only three cases for which the implicit variant applies. Therefore, we verify the implementation of Design Decision 13 in our SDL case study.

In addition to the aforementioned MC_{St} metaclasses, the «ToStereotype» stereotype can also be applied to those inheriting from other MC_{St} metaclasses. In this case, too, an *Extension* to a UML metaclass is introduced based on a *Stereotype*. Apart from the MC_{St} MappableDataElement that is marked as not to be mapped, six other MC_{St} metaclasses in Fig. 5.6a have the «ToStereotype» stereotype applied, so that an *Extension* is introduced for six *Stereotypes* contained in Fig. 5.6b according to Design Decision 18, which corresponds exactly to the result we expected. Therefore, we also consider this design decision to be successfully verified.

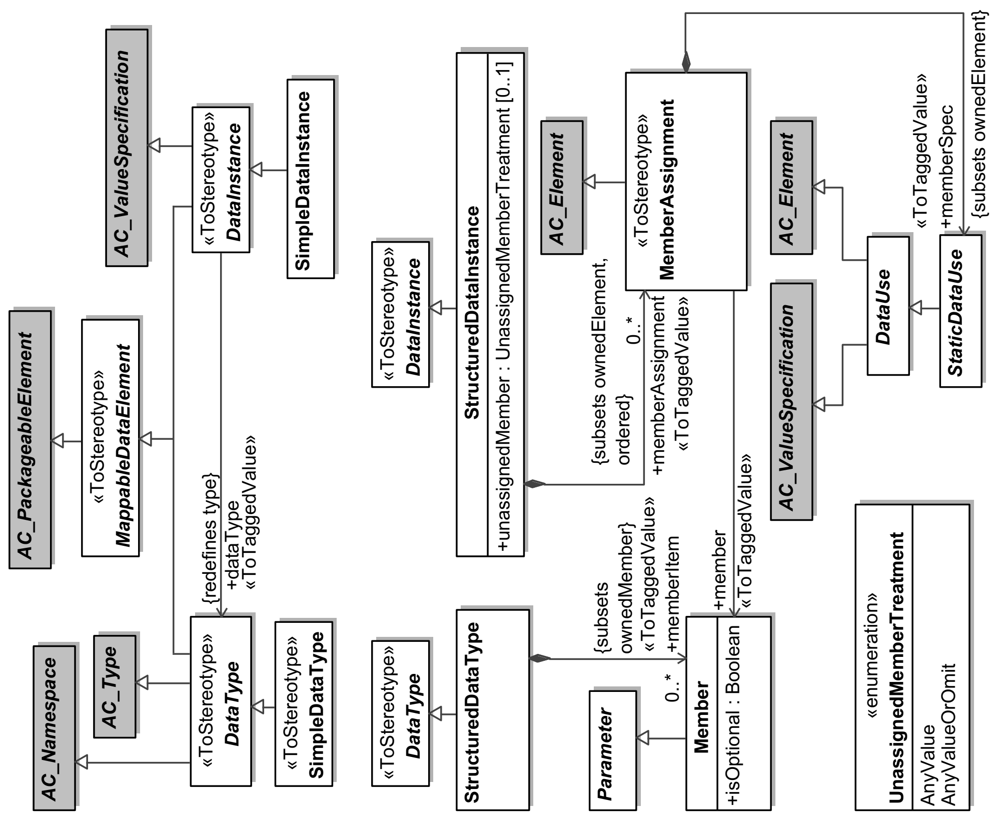
If a *Generalization* relationship between two MC_{St} metaclasses exists, Design Decision 14 requires the creation of a *Generalization* between the *Stereotypes* derived from these metaclasses. In general, the introduction of such a *Generalization* is independent of whether an MC_{St} has applied the «ToStereotype» stereotype. However, if an MC_{St} is defined as not to be mapped using the noMapping attribute of an applied «ToStereotype» stereotype, its incoming or outgoing *Generalizations* are not considered when deriving a UML profile.

Apart from the *Generalizations* of the MC_{St} MappableDataElement, Fig. 5.6a contains six *Generalizations* between MC_{St} metaclasses. Comparing Fig. 5.6a and Fig. 5.6b, we can observe the same number of *Generalizations* between MC_{St} metaclasses and the *Stereotypes* derived from them. Accordingly, we consider Design Decision 14 as successfully verified.

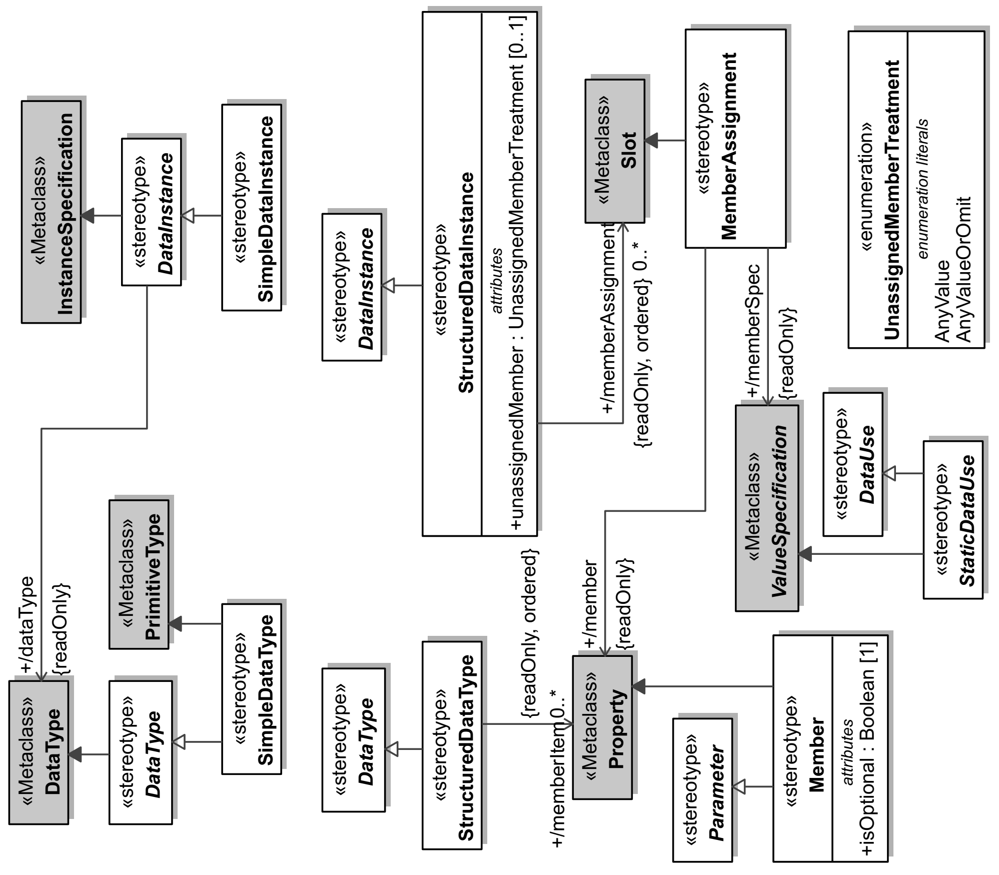
After analysing the derivation of *Stereotypes* from MC_{St} metaclasses, we review the mapping of MC_{St} attributes to corresponding attributes of *Stereotype* in accordance with Design Decisions 15 – 17 and 19. The general mapping to stereotype attributes is defined by Design Decision 15. However, it is important to note that MC_{St} attributes whose *type* properties refer to metaclasses are always mapped to non-composite stereotype attributes (*isComposite* = ‘false’). Consequently, these stereotype attributes represent references to metaclasses instances.

Considering the MC_{St} metaclasses in Fig. 5.6a, we find that two of their attributes are specified explicitly and five are implicitly introduced as *Associations*. Three of the MC_{St} attributes specified by *Associations* are defined as ‘composite’, which can be identified based on the filled diamonds at the opposite end of these *Associations*.

If we compare all MC_{St} attributes in Fig. 5.6a with the corresponding *Stereotype* attributes in Fig. 5.6b, we find that two stereotype attributes are specified directly and five using *Associations*. Unlike the MC_{St} attributes, all stereotype attributes refer to metaclasses or *DataTypes* and are defined as non-composite. Thus, we regard the mapping to stereotype attributes defined by Design Decision 15 as successfully verified.


AC_Namespace
AC_Type
«ToStereotype» DataType
«ToStereotype» SimpleDataType
AC_PackageableElement
«ToStereotype» MappableDataElement
AC_ValueSpecification
«ToStereotype» DataInstance
SimpleDataInstance
{redefines type} +dataType «ToTaggedValue»
«ToStereotype» DataType
StructuredDataType
«ToStereotype» DataInstance
StructuredDataInstance
+unassignedMember : UnassignedMemberTreatment [0..1]
{subsets ownedMember} «ToTaggedValue» +memberItem 0..*
Parameter
Member
+isOptional : Boolean
{subsets ownedElement, ordered} +memberAssignment «ToTaggedValue» 0..*
AC_Element
«ToStereotype» MemberAssignment
+member «ToTaggedValue»
AC_ValueSpecification
AC_Element
DataUse
StaticDataUse
«ToTaggedValue» +memberSpec {subsets ownedElement}
«enumeration» UnassignedMemberTreatment
AnyValue
AnyValueOrOmit

(a) Metaclasses of MM_{TDL}


«Metaclass» DataType
«stereotype» DataType
«stereotype» SimpleDataType
«Metaclass» PrimitiveType
+/dataType {readOnly}
«Metaclass» InstanceSpecification
«stereotype» DataInstance
«stereotype» SimpleDataInstance
«stereotype» DataType
«stereotype» StructuredDataType
«stereotype» DataInstance
«stereotype» StructuredDataInstance
attributes
+unassignedMember : UnassignedMemberTreatment [0..1]
{readOnly, ordered} 0..* +/memberItem
«Metaclass» Property
+/member {readOnly}
+/memberAssignment {readOnly, ordered} 0..*
«Metaclass» Slot
«stereotype» MemberAssignment
+/memberSpec {readOnly}
«Metaclass» ValueSpecification
«stereotype» DataUse
«stereotype» StaticDataUse
«stereotype» Parameter
«stereotype» Member
attributes
+isOptional : Boolean [1]
«enumeration» UnassignedMemberTreatment
enumeration literals
AnyValue
AnyValueOrOmit

(b) Stereotypes of UP_{std}

Figure 5.6: Metaclasses of MM_{TDL} and the *Stereotypes* derived from them for UP_{TDL}, which define TDL's data type concept

Furthermore, the *type* properties of stereotype attributes are recomputed according to Design Decision 17, so that they do not refer to *Stereotypes* but to the extended UML metaclasses. Therefore, the verification of this design decision can also be assessed as successful.

In addition to the general mapping of MC_{St} attributes according to Design Decision 15, a particular mapping variant is defined for MC_{St} attributes that are redefining or subsetting MC_{AC} attributes. According to the rules introduced by Design Decision 16, such MC_{St} attributes must be mapped to 'derived' and 'read-only' stereotype attributes with associated OCL expressions, which are employed to compute the attribute values at runtime.

As already highlighted, we have defined further criteria for this mapping variant when developing our derivation approach, so that not all MC_{St} attributes are mapped in this way. Table 5.3 shows that only six MC_{St} attributes that redefine MC_{AC} attributes are mapped according to the rules of Design Decision 16.

Because all attributes of the MC_{St} metaclasses we used as examples in Fig. 5.6a have applied the «ToTaggedValue» stereotype, we cannot verify a mapping of MC_{St} attributes according to the rules of Design Decision 16. For this reason, we employ the MC_{St} metaclasses shown in Fig. 5.7a, which define the syntax of TDL test descriptions.

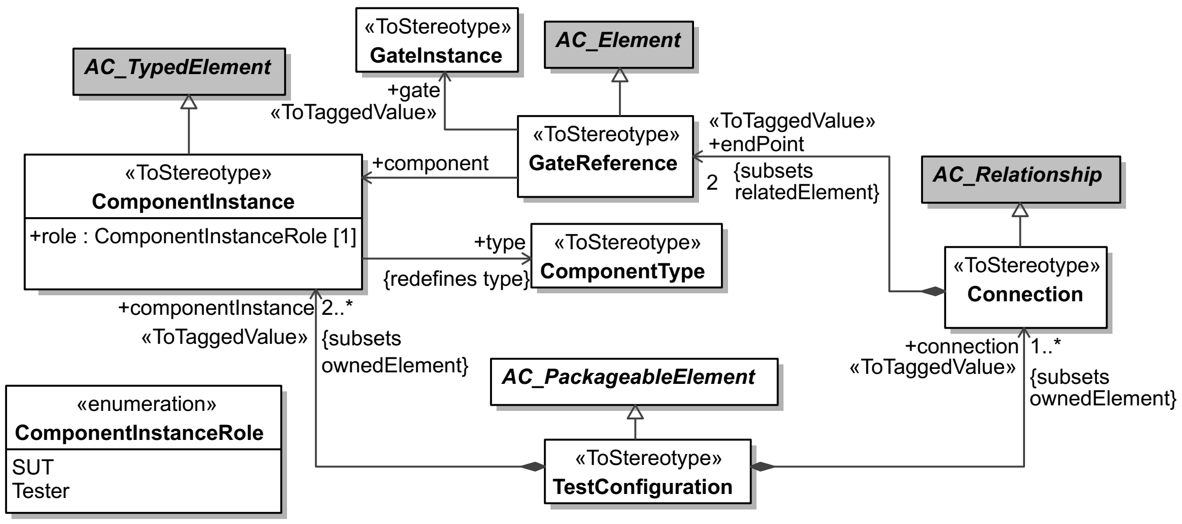
The type attribute of the MC_{St} metaclass ComponentInstance in Fig. 5.7a redefines an attribute of the same name inherited from MC_{AC} AC_TypedElement. As shown in Fig. 5.7b, this attribute is mapped to a 'derived' and 'read-only' attribute for the «ComponentInstance» stereotype according to Design Decision 16. In addition, the following OCL expression is assigned to the *defaultProperty* of this attribute:

```
if self.base_Property.type->one(isStereotypedBy('UP4TDL::ComponentType')) then
  self.base_Property.type->any(isStereotypedBy('UP4TDL::ComponentType'))
    .oclAsType(UML::Class)
else
  null
endif
```

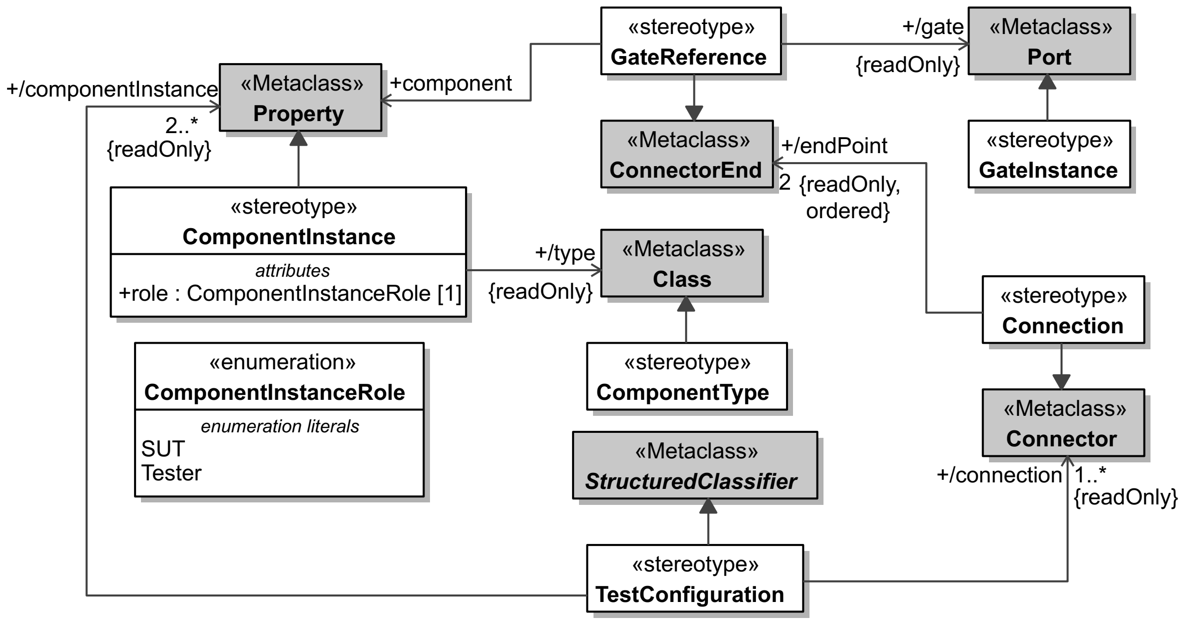
Because the type attribute of the «ComponentInstance» stereotype is only single-valued, we evaluate in the first line of the generated OCL expression whether an element with applied «ComponentType» stereotype is assigned to the type attribute of the UML metaclass TypedElement. However, to access this attribute, we must navigate from the «ComponentInstance» stereotype instance to the extended UML metaclass Property. We realize this with the following OCL expression:

```
self.base_Property
```

When deriving UP_{TDL}, we determine the type attribute based on its 'matching' attribute of the MC_{AC} AC_TypedElement, this is then used as source for the value computation in the generated OCL expression.



(a) Metaclasses of MM_{TDL}



(b) *Stereotypes* of UP_{TDL} that are derived from the metaclasses in Part (a)

Figure 5.7: Metaclasses of MM_{TDL} that define the syntax of TDL test descriptions and the *Stereotypes* derived from them for UP_{TDL}

If the above condition of the if-expression is met, we navigate to the type attribute of the UML metaclass Property in the second line of the OCL expression to determine the UML element provided by the «ComponentType» stereotype. To ensure that the returned result of the OCL expression is type-compliant with the *type* of the attribute, we then perform a type-cast. We must use the UML metaclass Class in the case of the type attribute of the «ComponentInstance» stereotype, as shown in Fig. 5.7b.

Based on the discussed example, we consider Design Decision 16 for MC_{St} attributes that redefine MC_{AC} attributes to be successfully verified. However, the verification for subsetting MC_{St} attributes is not yet addressed. We analyse this aspect in our SDL case study below, where subsetting MC_{St} attributes are widely used.

As shown Table 5.3, a total of 39 MC_{St} attributes are stereotyped with «ToTaggedValue». This *Stereotype* is employed to define the metadata required by Design Decision 19 for specifying alternative OCL expressions. All MC_{St} attributes marked in this way are mapped to 'derived' and 'read-only' stereotype attributes, and the OCL expressions specified via the «ToTaggedValue» stereotype are used as *defaultValue* of the generated stereotype attributes.

All five MC_{St} attributes introduced by *Association* ends shown in Fig. 5.6a have the «ToTaggedValue» stereotype applied. If we map these MC_{St} attributes according to Design Decision 19, we obtain the five 'derived' and 'read-only' stereotype attributes shown in Fig. 5.7b, which are also specified as *Association* ends. In addition, the OCL expressions listed in Table 5.1 are adopted as *defaultValue* of the stereotype attributes thus created. We regard Design Decision 19 as successfully verified, because we have shown that all MC_{St} attributes with applied «ToTaggedValue» stereotype were mapped according to this Design Decision.

The creation of additional OCL *Constraints* for *Stereotypes* in a derived UML profile is defined by the two Design Decisions 20 and 21. As we last discussed the mapping of redefined or subsetted metaclass attributes, we first consider Design Decision 21, which introduces OCL *Constraints* to constrain the UML metaclass attributes used as the computation basis in OCL expressions of 'derived' and 'read-only' stereotype attributes. Thus, we ensure that only UML elements with *Stereotypes* valid for the specific context can be assigned.

We verify Design Decision 21 using the type attribute of MC_{St} ComponentInstance shown in Fig. 5.7a, which redefines an attribute of the MC_{AC} AC_TypedElement. As only one attribute of the MC_{St} ComponentInstance redefines the type attribute of the MC_{AC} AC_TypedElement, only one attribute of the «ComponentInstance» stereotype has an OCL expression for a value computation based on the type attribute of UML metaclass TypedElement. Consequently, according to Design Decision 21, only elements with applied «ComponentType» stereotype shall be assignable to this UML metaclass attribute. To ensure this condition, the following OCL constraint is generated for the type attribute of the UML metaclass Property:

```
self.base_Property.type <> null implies
  self.base_Property.type.isStereotypedBy(`UP4TDL::ComponentType')
```

Because the type attribute of the UML metaclass TypedElement is only single-valued, Line 1 of the OCL *Constraint* tests whether a value exists at all. If this is the case, starting from an instance of the «ComponentInstance» stereotype, in Line 2, we navigate to the type attribute and invoke the isStereotypedBy() operation to evaluate whether the element assigned to this attribute has the «ComponentType» stereotype applied. Only if this applies, the evaluation of the OCL *Constraint* does return the Boolean value 'true' as result, while the value 'false' is returned in other cases, implying that the *Constraint* is violated. This exactly captures the condition defined by Design Decision 21. Therefore, we consider this design decision to be successfully verified for an application to redefined attributes. The application on subsetted attributes is verified in our subsequent SDL case study.

As we recompute the *type* property for stereotype attributes, Design Decision 20 requires that an OCL *Constraint* must be introduced for each 'non-derived' stereotype attribute to ensure that only UML elements with a specific *Stereotype* can be assigned to such an attribute. During the derivation of a UML profile, we determine the expected *Stereotype* based on the *type* property of an MC_{St} attribute.

We verify Design Decision 20 using the component attribute of the MC_{St} GateReference shown in Fig. 5.7a. The *type* property of this attribute refers to the MC_{St} ComponentInstance for which the «ComponentInstance» stereotype is derived in UP_{TDL}. As we recompute the *type* property for stereotype attributes, that of the component attribute of the «GateReference» stereotype refers to the UML metaclass Property. To ensure the syntactic correctness of this stereotype attribute, we therefore introduce the following OCL *Constraint* according to Design Decision 20:

```
self.component <> null implies
  self.component.isStereotypedBy(`UP4TDL::ComponentInstance')
```

Because the component attribute is single-valued, only a single item can be assigned to it. For this reason, we evaluate in Line 1 of the OCL *Constraint* whether a value was assigned to the stereotype attribute at all. If this applies, we employ the isStereotypedBy() operation in Line 2 to test whether the element assigned to the component attribute has applied the «ComponentInstance» stereotype. Only in this case is the condition implied by the *Constraint* met. Consequently, only elements with applied «ComponentInstance» stereotype can be assigned to the component attribute. Due to this reason, we consider Design Decision 20 to be successfully evaluated.

The standardized vs. the derived UML profile. After we have verified the correct implementation of the design decisions defined for the derivation of UML profiles,

Table 5.4: Comparison of the UML profiles UP_{std} and UP_{TDL}

	UP_{std}	UP_{TDL}
Mappings for UML elements without stereotypes	23	—
Stereotypes	56	79
Stereotype attributes	52	99
OCL *Constraints*	2	113
OCL *Operations*	0	10
Enumerations	3	3

we now compare our derived UML profile UP_{TDL} with the one standardized for TDL. In the following we refer to the latter UML profile by the acronym UP_{std}.

We wish to analyse whether a UML profile generated with our derivation approach is comparable to a manually created one. To answer this question, we first compare the number of model elements contained in UP_{std} and UP_{TDL}, so that we can draw initial conclusions. Afterwards, we review some *Stereotypes* of both UML profiles to obtain a detailed view on the quality of the UML profile UP_{TDL}.

Quantitative comparison: To compare the various element types contained in UP_{std} and UP_{TDL}, we summarize their quantities in Table 5.4. A specific feature of UP_{std} is shown in the first row. UP_{std} defines a mapping to TDL metaclasses for 23 UML element types without defining corresponding *Stereotypes*, i.e., the relevant UML model elements are transformed without TDL-specific *Stereotypes* applied to them. The disadvantage of this approach is that no OCL *Constraints* are defined for the affected UML element types due to the missing *Stereotypes*; thus, the static semantics of the TDL language concepts represented by these UML element types is not captured.

UP_{std} embraces a lower number of *Stereotypes* than UP_{TDL}, which is due to the discussed mappings of certain UML element types without existing *Stereotypes* in UP_{std}. In contrast and except of the $\mathrm{MappableDataElement}$ metaclass, UP_{TDL} provides a specific *Stereotype* for each MC_{St} metaclass of MM_{TDL}. In addition, the *Stereotypes* of UP_{TDL} have a larger number of attributes. One might assume that this is due to the larger number of *Stereotypes* in UP_{TDL}. However, the question arises whether the different numbers of *Stereotypes* is the only reason, because UP_{TDL} contains with 99 stereotype attributes almost twice as many as UP_{std}. Another reason might be that not each metaclass attribute has a corresponding *Stereotype* attribute in UP_{TDL}. We analyse this aspect in more detail at the end of this section.

As argued, the compliance of UML elements with TDL's static semantics can only be evaluated if *Stereotypes* with appropriate OCL *Constraints* exist. Because UP_{std} defines only two OCL *Constraints*, this statement also applies to those language constructs of TDL for which corresponding *Stereotypes* are specified in UP_{std}. In contrast, UP_{TDL} provides 113 OCL *Constraints* and OCL-defined *Operations*. These include not only the updated OCL *Constraints* transferred from MM_{TDL}, but also those introduced during the derivation of UP_{TDL}.

As UP_{std} contains only two OCL *Constraints*, we conclude that this UML profile cannot be employed to ensure TDL's static semantics for UML models with applied UP_{std}. This drawback could be remedied by using our UML profile UP_{TDL}.

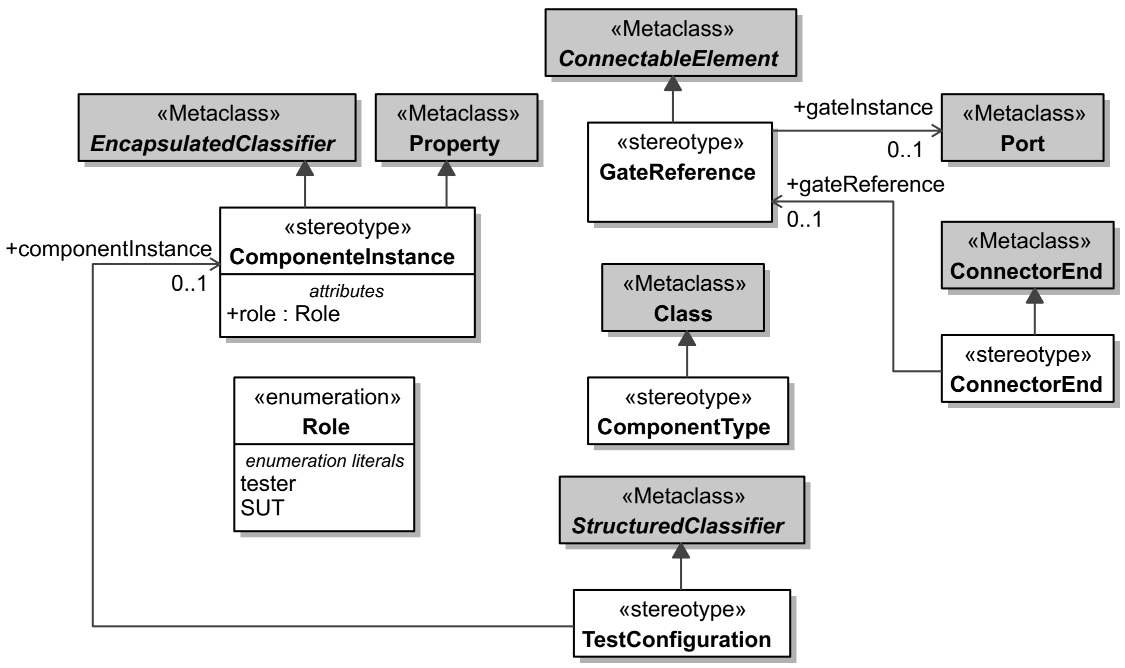
Comparison of the syntactic structure: After we determined that UP_{std} has some limitations compared to UP_{TDL}, we now compare the structure of the *Stereotypes* in both UML profiles. For this purpose, we examine the *Stereotypes* that define the abstract syntax for TDL test descriptions. The relevant Stereotypes of UP_{std} are shown in Fig. 5.8a, and those defined for UP_{TDL} can be found in Fig. 5.8b.

Considering the *Stereotypes* and the UML metaclasses they extend in Figs. 5.8a and 5.8b, we can identify two significant differences. The «ComponentInstance» stereotype in UP_{std} extends the two UML metaclasses EncapsulatedClassifier and Property, whereas the one in UP_{TDL} extends only the latter. Both UML metaclasses extended by this *Stereotype* of UP_{std} are fundamentally different language concepts. On the one hand, the UML metaclass Property represents a specialization of StructuralFeature. The EncapsulatedClassifier, on the other hand, is a specialization of Classifier, which can provide various kinds of Features, such as StructuralFeature and BehavioralFeature.

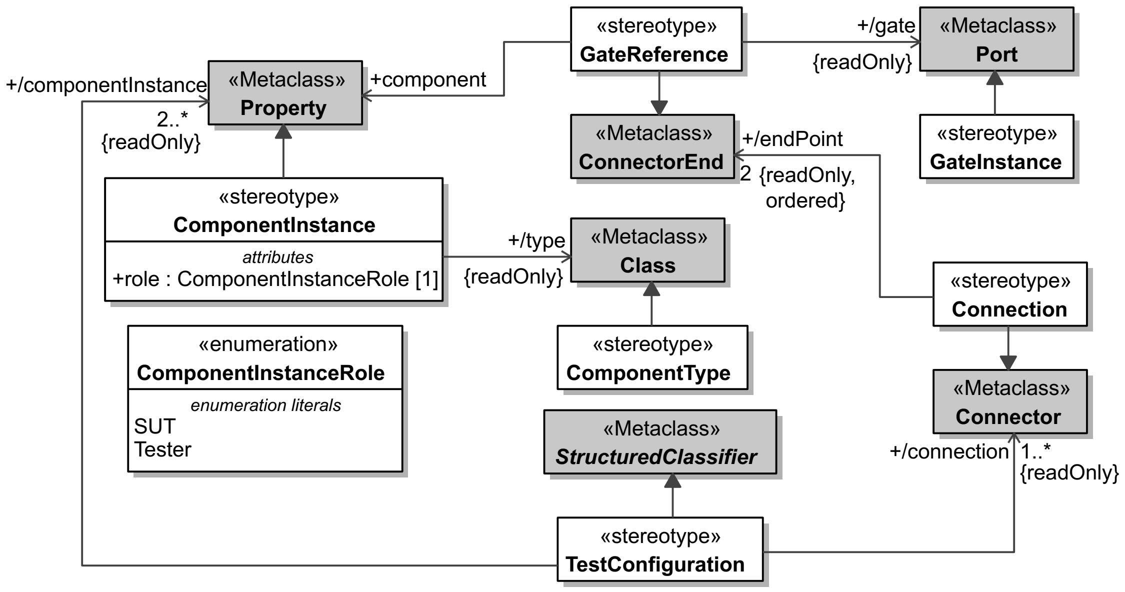
If we consider the TDL metaclasses shown in Fig. 5.7a, we see that the componentInstance attribute of the TestConfiguration metaclass is specified as a composition. That is, instances of ComponentInstance are parts of TestConfiguration instances. In addition, ComponentInstance provides a type attribute that is employed to refer to a ComponentType. Thus, from a semantic point of view, a TDL ComponentInstance can be compared with a UML Property rather than with a UML EncapsulatedClassifier. For this reason, we consider an extension of this UML metaclass by the «ComponentInstance» stereotype in UP_{std} as semantically incorrect, and therefore, this *Stereotype* should only extend the UML metaclass Property, as is the case in UP_{TDL}.

Another difference between the *Stereotypes* shown in Figs. 5.8a and 5.8b is that UP_{std}, unlike UP_{TDL}, provides a stereotype «ConnectorEnd», which extends the UML metaclass of the same name. In case of UP_{TDL}, the «Connection» stereotype is specified as an extension of the UML metaclass Connector, which is not present in UP_{std}. However, this *Stereotype* is not part of UP_{TDL}. From a semantic point of view, an extension of the UML metaclass Connector or ConnectorEnd by a *Stereotype* is possible, because instances of the latter metaclass represent the ends of Connectors.

After considering the stereotypes of both UML profiles, we analyse the potential differences of the specified stereotype attributes. Comparing Figs. 5.8a and 5.8b,



(a) *Stereotypes* of UP_{std} as defined in [39]



(b) *Stereotypes* of UP_{TDL} that are derived from the metaclasses shown in Fig. 5.7a

Figure 5.8: *Stereotypes* in UP_{std} and UP_{TDL} used to define the syntax of TDL test descriptions

we find that the stereotypes of UP_{TDL} have six and those of UP_{std} have only three attributes defined by *Association* ends. Because stereotypes of UP_{TDL} are automatically derived based on MC_{St} metaclasses of MM_{TDL}, they have the same number of attributes as the corresponding metaclasses. In contrast, stereotypes in UP_{std} have a lower number of attributes than TDL metaclasses.

However, instead of the non-existent stereotype attributes, a mapping of UML metaclass attributes to certain TDL metaclass attributes is defined in UP_{std}. This is a viable way, but due to the absence of OCL *Constraints* in UP_{std}, the values permitted for UML metaclass attributes are not constrained. Therefore, in contrast to our UP_{TDL}, UP_{std} allows the specification of models that are invalid in relation to the static semantics of TDL. Because UP_{std} does not specify a corresponding stereotype attribute for each metaclass attribute, this also explains the significant difference in the number of attributes between UP_{std} and UP_{TDL}, which has already been pointed out.

All stereotype attributes introduced by *Association* ends in UP_{TDL} refer to UML metaclasses extended by *Stereotypes*. In contrast, the attributes of the *Stereotypes* of UP_{std} shown in Fig. 5.8a always refer to *Stereotypes*. Because navigation from a *Stereotype* instance to the extended UML element or vice versa is possible, referencing *Stereotypes* or extended UML metaclasses is comparable from a syntactical point of view. However, the latter option requires the use of appropriate OCL *Constraints* to ensure that only UML elements with a specific applied *Stereotype* can be assigned to a stereotype attribute. As discussed in an earlier section, exactly such OCL *Constraints* are automatically introduced by our derivation approach.

Another difference between the stereotype attributes of UP_{TDL} and UP_{std} shown in Fig. 5.8 is their cardinality. If we compare the stereotype attributes in Fig. 5.8b with the corresponding metaclass attributes in Fig. 5.7a, we find that their cardinalities match. In contrast, the stereotype attributes shown in Fig. 5.8a have different cardinalities than their corresponding metaclass attributes of MM_{TDL}. Thus, a syntactical difference between UP_{std} and the TDL metamodel exists. The TDL standard [39] does not indicate that the deviating attribute cardinalities were an explicit design decision for the creation of UP_{std}. We therefore assume that these are issues that result from the manual creation of UP_{std}.

Finally, we wish to emphasize that all stereotype attributes of UP_{TDL} shown in Fig. 5.8b are defined as 'derived' and 'read-only', so that their values are computed at runtime based on OCL expressions. By contrast, the attributes of *Stereotypes* in UP_{std} must always be assigned manually.

Conclusions. After we quantitatively analysed the different elements in the UML profiles UP_{std} and UP_{TDL}, we compared the *Stereotypes* for a specific aspect in both UML profiles using an example.

In the quantitative analysis of both UML profiles we showed that a significant difference between UP_{std} and UP_{TDL} exists, because UP_{std} defines a mapping to TDL language concepts for certain UML element types but does not provide corresponding *Stereotypes* for these elements. Due to a lack of *Stereotypes*, no OCL *Constraints* are available for the UML element types concerned, and therefore, it is impossible to evaluate the static semantics as defined by TDL. Furthermore, we found that the *Stereotypes* defined in UP_{std} introduce only two OCL *Constraints*. Thus, we can state that for UML models that have applied UP_{std}, it is impossible to verify the static semantics defined by TDL. In contrast, such a verification is supported by our UP_{TDL}, because we derive *Stereotypes* for all metaclasses in MM_{TDL} and automatically transfer the OCL *Constraints* defined in MM_{TDL} to UP_{TDL}.

In the second part of our comparison, we found that our UP_{TDL} nearly extends the same UML metaclasses with *Stereotypes* as UP_{std}. However, one of the *Stereotypes* of UP_{std} we examined extends two different UML metaclasses, and we doubted whether this is incorrect from a semantic point of view. As another discrepancy, we found that the *Stereotypes* that we examined in UP_{std} have a lower number of attributes than the corresponding TDL metaclasses, which is not the case for our UP_{TDL}. A further specific aspect of UP_{std} is that the stereotype attributes considered have a different cardinality as the corresponding TDL metaclass attributes. In our opinion, this is an error that might be caused by the manual creation of UP_{std}.

Summarising, in the areas of UP_{std} and UP_{TDL} investigated, we determined a certain degree of correspondence between the *Stereotypes* of both UML profiles. However, we also found potential errors or syntactically incorrect implementations of stereotypes and their attributes contained in UP_{std}. Because of the automatic derivation, such errors are not present in our UP_{TDL}. Furthermore, static semantics is completely unconsidered in UP_{TDL}, which is not the case for UP_{TDL}. Hence, while UP_{std} and UP_{TDL} are structurally similar, the static semantics of TDL can only be verified with our UP_{TDL}.

5.3.5 Evaluation of the Updated OCL Constraints

Before we can use the revised metamodel MM_{TDL} as input for deriving our UML profile UP_{TDL}, the OCL expressions for *Constraints*, *Operations* and value computation of 'derived' attributes contained in UP_{TDL} must be updated. This is the only way to ensure that these OCL expressions can be used in automatically derived UML profiles without manual modification.

Table 5.5 lists the OCL *Constraints* of the metaclasses shown in Fig. 5.6a before and after conducting the OCL update. Based on these OCL *Constraints*, we investigate how we apply the rules introduced by Design Decisions 26 to 29 to our OCL update.

According to Design Decision 26, a *PropertyCallExp* that accesses the implicit base_<metaclass> attribute of a *Stereotype* must be introduced for all OCL expressions that invoke an attribute or *Operation* of an MC_{AC} metaclass from a **self** variable. We require this kind of update because after deriving a *Stereotype*, the attributes and *Operations* of an extended UML metaclass are only accessible via the implicit base_<metaclass> attribute. This is in contrast to the source MC_{St} metaclasses, where attributes and *Operations* inherited from MC_{AC} metaclasses are directly accessible for OCL expressions.

For example, the update defined by Design Decision 26 is applied to Constraint (1) of the MC_{St} metaclass StructuredDataType. The membersAreDistinguish-

Table 5.5: Source and updated *Constraints* of MC_{St} metaclasses in MM_{TDL}, which define the static semantics of TDL data type definitions

StructuredDataType
Constraint 1: All *Member* names of a *StructuredDataType* shall be distinguishable.
context TDL::StructuredDataType
inv: **self**.membersAreDistinguishable()
context UP4TDL::StructuredDataType
inv: **self**.base_DataType.membersAreDistinguishable()
SimpleDataInstance
Constraint 1: The inherited reference *dataType* from *DataInstance* shall refer to instances of *SimpleDataType* solely.
context TDL::SimpleDataInstance
inv: **self**.dataType.oclIsKindOf(SimpleDataType)
context UP4TDL::SimpleDataInstance
inv: **self**.dataType.isStereotypedBy(`` `UP4TDL::SimpleDataType' ``)
StructuredDataInstance
Constraint 1: The inherited reference *dataType* from *DataInstance* shall refer to instances of *StructuredDataType* solely.
context TDL::StructuredDataInstance
inv: **self**.dataType.oclIsKindOf(StructuredDataType)
context UP4TDL::StructuredDataInstance
inv: **self**.dataType.isStereotypedBy(`` `UP4TDL::StructuredDataType' ``)
Constraint 2: The referenced *Member* shall be contained in the *StructuredDataType* that the *StructuredDataInstance* that contains this *MemberAssignment*, refers to.
context TDL::StructuredDataInstance
inv: **self**.memberAssignment−>forAll(a | **self**.dataType.member−>includes(a.member))
context UP4TDL::StructuredDataInstance
inv: **self**.memberAssignment−>forAll(a | **self**.dataType.member−>includes(a.extension_MemberAssignment.member))
MemberAssignment
Constraint 1: The *determinedType* of the *StaticDataUse* of *memberSpec* shall coincide with the *DataType* of the *Member* of the *MemberAssignment*.
context TDL::MemberAssignment
inv: **self**.memberSpec.determinedType = **self**.member.dataType
context UP4TDL::MemberAssignment
inv: **self**.memberSpec.extension_StaticDataUse.determinedType = **self**.member.extension_Parameter.dataType
Constraint 2: If the *memberSpec* refers to a *StructuredDataInstance*, all of its *ParameterBindings* shall refer to *StaticDataUse*.
context TDL::MemberAssignment
inv: **self**.memberSpec.argument−>forAll(a | a.dataUse.oclIsKindOf(StaticDataUse))
context UP4TDL::MemberAssignment
inv: **self**.memberSpec.extension_StaticDataUse.argument−>forAll(a | a.extension_ParameterBinding.dataUse.isStereotypedBy(`` `UP4TDL::StaticDataUse' ``))

able() operation called in this constraint is inherited from the MC_{AC} metaclass AC_Namespace. This operation is not inherited by the *Stereotype* derived from the MC_{St} StructuredDataType, but we can invoke it via the extended UML metaclass. For this purpose, we must navigate from the *Stereotype* instance to that of the extended UML metaclass. Hence, we introduce an additional *PropertyCallExp* for the implicit attribute base_DataType based on Design Decision 26. Thus, we ensure that the updated Constraint (1) can access the membersAreDistinguishable() operation.

According to Design Decision 17, the *type* property of an MC_{St} attribute is recomputed during the derivation of a UML profile, so that the corresponding stereotype attribute always refers to a UML metaclass or an 'additional' metaclass, but never to a *Stereotype*. If a *Feature* of another MC_{St} is accessed from an MC_{St} attribute or operation in an OCL expression, we cannot transfer this expression one-to-one to UP_{TDL}. Because the *type* properties of stereotype attributes always refer to metaclasses, we must introduce a *PropertyCallExp* for such OCL expressions, so that the implicit extension_<stereotype> attribute navigates from a metaclass instance to the required stereotype instance. We capture this type of update with Design Decision 27.

We conduct the discussed update on Constraint (2) of the MC_{St} StructuredDataInstance and on Constraints (1) and (2) of the MC_{St} MemberAssignment in Table 5.5. For example, if we consider Constraint (2) of the latter MC_{St}, we find two OCL expressions to which the condition defined by Design Decision 27 applies. These two expressions are as follows:

```
1) self.memberSpec.determinedType
2) self.member.dataType
```

The underlined parts of both OCL expressions identify the *PropertyCallExp* for an MC_{St} attribute (e.g., memberSpec) from which a second MC_{St} attribute (e.g., determinedType) is accessed. As required by Design Decision 27, we introduce a *PropertyCallExp* for the extension_<stereotype> attribute in both OCL expressions. After the update, we obtain the following OCL expressions:

```
1) self.memberSpec.extension_StaticDataUse.determinedType
2) self.member.extension_Parameter.dataType
```

As highlighted in the context of Design Decision 28, the two predefined operations oclIsTypeOf() and oclIsKindOf() are used to determine whether the result of an OCL expression is type-compatible with the *Type* specified as argument (i.e., a metaclass or a *DataType*). Due to the recomputation of *type* properties for stereotype attributes, OCL expressions that access these attributes always return metaclass instances as result.

For the discussed reason, we must update OCL expressions that evaluate the type-compatibility of an MC_{St} instance to a particular MC_{St} metaclass using one of two predefined operations according to the rule defined by Design Decision 28.

Based on this rule, we replace all oclIsTypeOf(T) and oclIsKindOf(T) operation calls, where an MC_{St} is specified as argument T, with calls to operations oclIsStrictStereotypedBy(S) and oclIsStereotypedBy(S), respectively. As argument S for both operations, we specify the qualified name of the *Stereotype* derived from MC_{St} T.

The update according to Design Decision 28 is applied to Constraint (1) of the MC_{St} metaclasses SimpleDataInstance and StructuredDataInstance and to Constraint (2) of MC_{St} MemberAssignment for the OCL constraints listed in Table 5.5. If we consider the latter *Constraint* as example, the underlined part of the following fragment is updated:

```
a.dataUse.oclIsKindOf(StaticDataUse)
```

The underlined call of the oclIsKindOf() operation is replaced as described by an invocation of the oclIsStereotypedBy() operation. In addition, the MC_{St} StaticDataUse specified as argument is substituted with the qualified name of the corresponding *Stereotype*, so that we obtain the following OCL expression after the update:

```
a.extension_ParameterBinding.dataUse.isStereotypedBy('UP4TDL::StaticDataUse')
```

The last type of OCL update that we perform when deriving UML profiles is specified by Design Decision 29, which requires that all MC_{St} metaclasses referenced by an OCL *TypeExp* must refer to extended UML metaclasses instead of *Stereotypes*. Because the OCL *Constraints* shown in Table 5.5 do not include such a *TypeExp*, we explain the application of the update based on the getTestDescription() operation in Fig. 5.5b. Line 5 of this operation invokes the oclAsType(T) operation, where a *TypeExp* is passed as argument T, which refers to the MC_{St} TestDescription. After conducting our update, the UML metaclass BehavioredClassifier is passed as argument, as required by Design Decision 29.

```
if self.base_InteractionFragment.allNamespaces()->one(
 isStrictStereotypedBy(`UP4TDL::TestDescription'))
then
  self.base_InteractionFragment.allNamespaces()->any(
   isStrictStereotypedBy(`UP4TDL::TestDescription')
  ).oclAsType(UML::BehavioredClassifier)
else
  null
endif
```

5.3.6 Evaluation of the Derived Model Transformations

Model transformations are the remaining artefact type that we can derive from a metamodel MM_{Domain} using our derivation approach. As described in Sec. 4.6, we

Table 5.6: Comparison of MM_{TDL} and the transformation $T_{TDL\text{-}to\text{-}UML}$

MM_{TDL}		$T_{TDL\text{-}to\text{-}UML}$	
Abstract MC_{AC} metaclasses	13	13	'Disjuncting' mapping operations *(abstract MC_{AC} to MC_{UML})*
MC_{St} metaclasses	80	—	
Abstract MC_{St} metaclasses	16	16	'Disjuncting' mapping operations *(abstract MC_{St} to MC_{UML})*
Non-abstract MC_{St} metaclasses	64	64	Mapping operations with operation body *(non-abstract MC_{St} to non-abstract MC_{UML})*
		64	Mapping operations with operation body for applying *Stereotypes*
		1	'Disjuncting' mapping operation to invoke the specific ones for applying *Stereotypes*
Enumerations	3	3	Helper functions to determine *EnumerationLiterals* in UP_{TDL}

derive two model transformations $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ from a single metamodel MM_{Domain}. Both transformations are employed to achieve model interoperability. We use the abbreviation DM in the transformation names as placeholder for the DSL in question. When we replace this abbreviation for our current case study, we obtain $T_{TDL\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}TDL}$ as transformation names employed for TDL.

As in the previous sections, we use the relevant Design Decisions 30 – 37 from Sec. 4.6.2 as verification basis. As we formulated these design decisions without regarding the specific requirements of our chosen target language QVT Operational Mappings (QVTo) [113], we must also consider the transformation concept that we defined in Sec. 4.6.4.

Because we derive the model transformations based on a single metamodel, manual enhancements are required. Therefore, we can only verify the correct creation of the automatically generated transformation parts, but not the manually implemented ones. Furthermore, we validate the usability of the manually completed transformations using exemplary test models.

Verification of derived DM-to-UML transformations. In the following, we verify whether generated $T_{DM\text{-}to\text{-}UML}$ transformations conform to the relevant design decisions and the transformation concept based thereupon. We conduct the verification based on transformation $T_{TDL\text{-}to\text{-}UML}$. In a first step, we compare quantitatively the model elements in MM_{TDL} and the derived parts of $T_{TDL\text{-}to\text{-}UML}$. We then examine the derivation of the various transformation parts in detail.

Quantitative analysis: In Table 5.6, we compare the model elements contained in MM_{TDL} with the resulting mapping operations of $T_{TDL\text{-}to\text{-}UML}$. We distinguish two types of mapping operations. Mapping operations with operation body define the concrete mapping of metaclasses and *Stereotypes*, whereas 'disjuncting mappings'[2] are introduced to control the execution order of mapping operations in a transformation.

The first type of mapping operation is derived from non-abstract (*isAbstract* = false) metaclasses, whereas 'disjuncting mappings' are created based on abstract (*isAbstract* = true) metaclasses. Therefore, in the left column of Table 5.6 we distinguish between these two metaclass types. A further distinction is made between MC_{AC} and MC_{St} metaclasses. The former is only mapped to 'matching' counterparts in UML, whereas MC_{St} metaclasses are mapped to UML metaclasses and *Stereotypes* applied to them in a two-staged transformation according to Design Decision 34.

As shown in Table 5.6, we derive 13 'disjuncting mappings' from the 13 MC_{AC} metaclasses, which map MC_{AC} metaclasses to their 'matching' UML counterparts. 16 additional mapping operations of this type are generated based on the 16 abstract MC_{St} metaclasses.

Another situation arises when deriving mapping operations from non-abstract MC_{St} metaclasses. Here, we create an equal number of mapping operations with operation bodies from the 64 metaclasses of this category, which map non-abstract TDL metaclasses to non-abstract UML metaclasses. In addition, 64 additional mapping operations are generated from these metaclasses, which are intended for applying *Stereotypes* to the non-abstract UML metaclasses. Finally, a single 'disjuncting mapping' is created that calls an associated mapping operation based on the type of a TDL model element, in order to apply an appropriate *Stereotype*.

Apart from the metaclasses in MM_{TDL}, this metamodel also contains three *Enumeration* data types. Because *EnumerationLiterals* of *Enumerations* cannot be mapped in the same way as metaclasses, we create helper functions for *Enumerations* in metamodels according to Design Decision 36, which resolve the *EnumerationLiterals* in the target model. In our example, we create three helper functions for the transformation $T_{TDL\text{-}to\text{-}UML}$ based on the three *Enumerations* of MM_{TDL}.

Verification of the various transformation parts: After quantitatively analysing the components of transformation $T_{TDL\text{-}to\text{-}UML}$, we verify below the correct implementation of the design decisions defined for the derivation of $T_{DM\text{-}to\text{-}UML}$ transformations using the mapping operations listed in Fig. 5.9, which are derived from the MC_{St} metaclasses shown in Fig. 5.6a and Fig. 5.6b. In the following, we only capture those mapping operations that we require to conduct our verifications.

Design Decision 30 demands that a metamodel used as input for deriving model transformations must have the UML profile «MM2Profile» applied. This condition is a prerequisite for the correct application of our derivation approach for

[2] The term *'disjunctive mappings'* originates from the QVT standard [113]. A detailed explanation of this term can be found in Sec. 4.6.4.

```
mapping TDL::DataType::toDataType() : UML::DataType
disjuncts
  TDL::SimpleDataType::toPrimitiveType,
  TDL::StructuredDataType::toDataType
  {}

mapping TDL::StructuredDataType::toDataType() : UML::DataType
when { self.oclIsTypeOf(TDL::StructuredDataType) }
{
  result.elementImport += self.elementImport−>map toElementImport();
  result.name := self.name;
  result.ownedComment += self.ownedComment−>map toComment();
  result.package := self.package.map toPackage();
  result.packageImport += self.packageImport−>map toPackageImport();

  // Note: A manual refinement is required for the following attributes!
  // result.ownedAttribute := self.memberItem.map xxxxx;
}

mapping inout UML::DataType::applyStereotypeForStructuredDataType()
when {
  self.invresolveone(TDL::StructuredDataType).oclIsTypeOf(TDL::StructuredDataType)
} {
  var srcElement := self.invresolveone(TDL::StructuredDataType);
  var stereotype := self.getApplicableStereotype('UP4TDL::StructuredDataType');
  self.applyStereotype(stereotype);

  var stInstance :=
    self.getStereotypeApplication(stereotype).oclAsType(UP4TDL::StructuredDataType);

  // Storing stereotype instance for further stereotype applications
  allStInstances−>add(stInstance);
}

mapping TDL::ComponentInstance::toProperty() : UML::Property
when { self.oclIsTypeOf(TDL::ComponentInstance) }
{
  result.name := self.name;
  result.ownedComment += self.ownedComment−>map toComment();
  result.type := self.type.map toType();
}
```

Figure 5.9: Mapping operations of transformation $T_{TDL\text{-}to\text{-}UML}$ derived from meta-classes in MM_{TDL}

```
42 mapping inout UML::Property::applyStereotypeForComponentInstance()
43 when {
44   self.invresolveone(TDL::ComponentInstance).oclIsTypeOf(TDL::ComponentInstance)
45 } {
46   var srcElement := self.invresolveone(TDL::ComponentInstance);
47   var stereotype := self.getApplicableStereotype('UP4TDL::ComponentInstance');
48   self.applyStereotype(stereotype);
49
50   var stInstance :=
51     self.getStereotypeApplication(stereotype).oclAsType(UP4TDL::ComponentInstance);
52   stInstance.role := srcElement.role.toComponentInstanceRole();
53
54   // Storing stereotype instance for further stereotype applications
55   allStInstances->add(stInstance);
56 }
```

Figure 5.9: Mapping operations of transformation $T_{TDL\text{-}to\text{-}UML}$ derived from metaclasses in MM_{TDL} (continued)

model transformations based on a single metamodel, because the required information is specified using stereotypes of this UML profile.

For example, according to Design Decision 33, we create so-called 'Mapping Proposals' when an MC_{St} attribute has the stereotype «ToTaggedValue» applied. The mapping operation toDataType() shown in Fig. 5.9 contains such a mapping proposal in Line 17.

When creating model transformations, we also evaluate whether an MC_{St} metaclass has the «ToStereotype» stereotype applied, so as to explicitly specify the UML metaclass to be extended. If this is the case, we use this information to define the result element type of mapping operations in $T_{DM\text{-}to\text{-}UML}$ transformations. For instance, this is the case for all mapping operations of transformation $T_{TDL\text{-}to\text{-}UML}$ shown in Fig. 5.9.

As argued above, the application of the UML profile «MM2Profile» as required by Design Decision 30 is essential, but a direct verification of this aspect is infeasible, because no specific parts of a model transformation are generated from this.

Design Decision 31 defines that metaclasses, *DataTypes*, and *Stereotypes* referenced in model transformations must be qualified with namespaces in order to avoid name collisions. We have taken this requirement into account for our transformation concept, so that for each mapping operation to be created, its context and result parameters are qualified with a namespace. This is the case for all mapping operations listed in Fig. 5.9.

Because we transform $T_{TDL\text{-}to\text{-}UML}$ metaclasses from TDL to UML metaclasses and *Stereotypes* of the UML profile UP_{TDL}, the context parameters of the mapping operations are qualified with the namespace TDL and the result parameters with UML. As metaclasses, *DataTypes* and *Stereotypes* can be passed as parameters in operation calls, we also qualify them with namespaces, as is the case with the expressions in the **when** clauses of the mapping operations shown in Fig. 5.9. Based on the examples discussed, we consider Design Decision 31 to be successfully implemented.

As pointed out earlier, we create 'disjuncting mappings' from abstract MC_{AC} or MC_{St} metaclasses in MM_{TDL}. Instead of an operation body, a list of 'candidate mappings' is defined for such a mapping operation. Assuming we derive a 'disjunctive mapping' from an abstract metaclass A, then we create an entry in the 'candidate mappings' list for each metaclass inheriting directly or indirectly from A. Each of these entries represents a potential call for a specific mapping operation.

To verify that 'disjuncting mappings' conform to our transformation concept, we consider the mapping operation TDL::DataType::toDataType() shown in Lines 1 – 5 of Fig. 5.9 as an example. This 'disjuncting mapping' is derived from the abstract MC_{St} DataType displayed in Fig. 5.6a. Because the two metaclasses SimpleDataType and StructuredDataType inherit from this metaclass, the 'candidate mappings' list must contain two calls for mapping operations. As is evident from Fig. 5.9, mapping operation TDL::DataType::toDataType() contains the two expected entries. Thus, we consider the derivation of 'disjuncting mappings' from abstract metaclasses as successfully implemented.

We derive two types of mapping operation from non-abstract MC_{St} metaclasses, as emphasized in Sec. 4.6.5. The first type maps an MC_{St} metaclass in MM_{TDL} to a UML metaclass. With the second type of mapping operation we apply a stereotype of UP_{TDL} to a UML element and assign values to stereotype attributes.

The mapping operations for mapping MC_{St} to UML metaclasses have, in addition to their signature, an operation body that contains assignment expressions for mapping the metaclass attributes. Because we map metaclasses of TDL to those of UML, the context parameter type of such a mapping operation always corresponds to the MC_{St} metaclass from which this operation is derived. In contrast, the result parameter type always corresponds to the UML metaclass to be extended by a *Stereotype*, which we determine either based on the inheritance hierarchy of an MC_{St} or the «ToStereotype» stereotype applied to an MC_{St}. For example, the result parameter types of the two mapping operations TDL::StructuredDataType::toDataType() and TDL::ComponentInstance::toProperty() shown in Fig. 5.9 are determined based on the «ToStereotype» stereotype.

To ensure that a mapping operation generated from a non-abstract MC_{St} can only be invoked for instances of the metaclass specified as the context parameter type, but not for instances of subclasses, we introduce **when** clauses according to our transformation concept. These clauses evaluate the type equality by calling the oclIsTypOf() operation, and the same metaclass as defined as context parameter type of the current mapping operation is passed as argument.

The implementation according to our transformation concept can be verified based on the **when** clauses shown in Lines 8 and 36 of Fig. 5.9. As expected, the metaclasses passed as arguments to the oclIsTypeOf() operation in both **when** clauses exactly match the context parameter types of the corresponding mapping operation.

The body of a mapping operation, which we generate from a non-abstract MC_{St}, consists of two parts. In the first part, we map all inherited MC_{AC} attributes of an MC_{St} to UML metaclass attributes. As values can only be assigned to attributes that are not defined as 'read-only', we only consider this type of MC_{AC} attributes. In the second part, we create so-called 'Mapping Proposals', which must be completed manually.

To verify that generated assignment expressions for the mapping of inherited MC_{AC} attributes conform to our transformation concept, we examine two mapping operations shown Fig. 5.9. The MC_{St} StructuredDataType has a total of 12 inherited MC_{AC} attributes of which only five are not defined as 'read-only'. Based on these attributes, the five assignment expressions for mapping operation TDL::StructuredDataType::toDataType() shown in Lines 10 – 14 of Fig. 5.9 are generated. These assignment expressions invoke appropriate mapping operations to map the MC_{AC} attributes to attributes of the UML metaclass DataType.

As another example, we consider the MC_{St} StructuredDataType, which has seven inherited MC_{AC} attributes, three of which are not classified as 'read-only'. Based on these attributes, we derive three assignment expressions for mapping operation TDL::ComponentInstance::toProperty(), as shown in Lines 38 – 40 of Fig. 5.9.

According to the results shown, we consider the generation of assignment expressions for inherited MC_{AC} attributes as successfully verified. Because the generation of assignment expressions also captures those MC_{AC} attributes that are redefined or subsetted by MC_{St} attributes, we regard Design Decision 32 as fulfilled.

From MC_{St} attributes with applied «ToTaggedValue» stereotype, we derive stereotype attributes defined as 'read-only' and 'derived'. The values of these attributes are calculated at runtime via OCL expressions that access UML metaclass attributes. Hence, based on MC_{St} attributes with applied «ToTaggedValue» stereotype, we create so-called 'Mapping Proposals' in mapping operations that map TDL metaclasses to UML metaclasses, as required by Design Decision 33.

The MC_{St} StructuredDataType shown in Fig. 5.6a owns the attribute memberItem with applied «ToTaggedValue» stereotype. Consequently, we create a 'Mapping Proposal' for this attribute in the mapping operation TDL::StructuredDataType::toDataType(), as shown in Line 17 of Fig. 5.9. The right-hand side of the assignment expression is derived from the MC_{St} attribute, whereas the left-hand side is created based on the value of the derivationSource attribute of the applied «ToTaggedValue» stereotype.

Let us now consider the mapping operation TDL::ComponentInstance::toProperty(), which is derived from the MC_{St} ComponentInstance. Because no attribute of this MC_{St} has the «ToTaggedValue» stereotype applied, the mapping operation does not contain a mapping proposal. Based on the examples discussed, we consider Design Decision 33 to be successfully verified.

To meet the requirement of Design Decision 34 that *Stereotypes* shall only be applied to already created UML elements in a second transformation pass, we introduce in-place mapping operations according to our transformation concept. Using this type of mapping, we first create an instance of the required *Stereotype*, so we can assign it to the UML element to be extended. In a next step, all MC_{St} attributes classified as 'non-derived' (see Sec. 5.3.4) are mapped to corresponding stereotype attributes. We only consider this type of MC_{St} attributes, because only for them is a value assignment possible. In contrast, value assignments to stereotype attributes derived from 'derived' MC_{St} attributes are not required, because their values are computed at runtime via OCL expressions.

We verify the compliance to our transformation concept based on the two MC_{St} metaclasses StructuredDataType and ComponentInstance. The two mapping operations applyStereotypForStructuredDataType() and applyStereotypForComponentInstance() of transformation $T_{TDL\text{-}to\text{-}UML}$ shown in Fig. 5.9 are generated from both metaclasses.

As both mapping operations are applied in-place to existing UML elements, they have a context parameter but not a result parameter. Using the **when** clause, we first resolve the TDL source element from which the UML element passed as context parameter is created. Then, we evaluate whether the returned TDL element is a direct instance of the relevant TDL metaclasses. Furthermore, the expressions in the first part of the operating body serve to create and apply a *Stereotype* instance (see Lines 24 – 29 and 46 – 51).

In the second part of our two sample mapping operations, the assignment expressions for mapping the MC_{St} attributes classified as 'non-derived', if present, are treated. As neither the MC_{St} StructuredDataType nor any of its super-classes in Fig. 5.6a have 'non-derived' attributes, no corresponding assignment expressions for mapping operation applyStereotypForStructuredDataType() are generated in Fig. 5.9. Another situation exists for MC_{St} ComponentInstance (see Fig. 5.6b), where only the role attribute of this MC_{St} is classified as 'non-derived'. Consequently, only one assignment expression shown in Line 52 of Fig. 5.9 is generated. Thus, the number of assignment expressions in the mapping operation applyStereotypForComponentInstance() is equal to the number of 'non-derived' MC_{St} attributes of the ComponentInstance.

```
helper TDL::ComponentInstanceRole::toComponentInstanceRole()
  : UP4TDL::ComponentInstanceRole
{
  switch
  {
    case (self = TDL::ComponentInstanceRole::SUT)
      return UP4TDL::ComponentInstanceRole::SUT;

    case (self = TDL::ComponentInstanceRole::Tester)
      return UP4TDL::ComponentInstanceRole::Tester;
  };
  return null;
}
```

Figure 5.10: Helper function for determining an *EnumerationLiteral* in UP_{TDL}

Based on the analysis performed, we consider the derivation of mapping operations to apply *Stereotypes* to already generated UML elements according to our transformation concept and Design Decision 34 as successfully verified.

After analysing the mapping operations generated for transformation $T_{TDL\text{-}to\text{-}UML}$, we examine the generation of helper functions used to determine *EnumerationLiterals* for a distinct *Enumeration* according to Design Decision 36. As these helper functions differ only slightly in the $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ transformations regarding the employed namespaces, we only verify the conformity to our transformation concept using the $T_{TDL\text{-}to\text{-}UML}$ transformation. As shown in Table 5.6, the metamodel MM_{TDL} contains a total of three *Enumeration* data types for which three helper functions in transformation $T_{TDL\text{-}to\text{-}UML}$ are generated. We verify the generation of helper functions employing the *Enumeration* ComponentInstanceRole shown in Fig. 5.7a.

According to our transformation concept, exactly one **case** alternative in the **switch** expression of a helper function is created for each *EnumerationLiteral* of an *Enumeration*. Each of these **case** alternatives returns the corresponding *EnumerationLiteral* for the target model of the transformation. Because the ComponentInstanceRole *Enumeration* has two *EnumerationLiterals*, two corresponding **case** alternatives are created for the helper function toComponentInstanceRole() shown in Fig. 5.10. Therefore, we regard the generation of helper functions as conforming to our transformation concept and Design Decision 36.

When deriving UML profiles, we only support the five predefined *PrimitiveTypes*, such as String, but not the introduction of new *PrimitiveTypes*. For this reason and according to Design Decision 37, we only regard these predefined *PrimitiveTypes* for the derivation of model transformations. Our transformation concept expects that the five predefined *PrimitiveTypes* are imported by a metamodel MM_{Domain} and the derived UML profile UP_{Domain}. Therefore, the use of simple assignment expressions without calling mapping operations is sufficient in the transforma-

Table 5.7: Comparison of MM_{TDL} and the transformation $T_{UML\text{-}to\text{-}TDL}$

MM_{TDL}		$T_{UML\text{-}to\text{-}TDL}$	
Abstract MC_{AC} metaclasses	13	13	‘Disjuncting’ mapping operations *(MC_{UML} to abstract MC_{AC})*
MC_{St} metaclasses	80	—	
Abstract MC_{St} metaclasses	16	16	‘Disjuncting’ mapping operations *(MC_{UML} to abstract MC_{St})*
Non-abstract MC_{St} metaclasses	64	64	Mapping operations with operation body *(MC_{UML} to non-abstract MC_{St})*
Enumerations	3	3	Helper functions to determine *EnumerationLiterals* in MM_{TDL}

tions generated by us. Because we apply this approach to both $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ transformations, we verify the implementation only for the $T_{TDL\text{-}to\text{-}UML}$ transformation.

The generation of assignment instructions for *PrimitiveTypes* corresponding to our transformation concept can be verified based on the assignments for the name attributes of two TDL metaclasses as shown in Lines 11 and 38 of Fig. 5.9. As expected, the values are assigned in both cases without calling a mapping operation beforehand, which means that the values of the source elements are assigned directly to the target elements. Hence, we regard Design Decision 37 as successfully verified.

Verification of derived UML-to-DM transformations. Before we verify the derivation of $T_{UML\text{-}to\text{-}DM}$ transformations in detail using the $T_{UML\text{-}to\text{-}TDL}$ transformation as an example, we first analyse the parts of this transformation generated from the model elements of MM_{TDL} quantitatively.

Quantitative analysis: According to our transformation concept, we derive $T_{UML\text{-}to\text{-}DM}$ transformations analogously to the previously discussed $T_{DM\text{-}to\text{-}UML}$ transformations. Just as for $T_{DM\text{-}to\text{-}UML}$ transformations, we generate ‘disjuncting mappings’ for $T_{UML\text{-}to\text{-}DM}$ transformations from MC_{AC} and MC_{St} metaclasses defined as abstract, whereas we generate mapping operations with operation body based on non-abstract MC_{St} metaclasses. The main difference between the two mentioned transformation types is that we create only one mapping operation for each MC_{St} metaclasses, which maps both a UML metaclass and an associated *Stereotype* to a specific metaclass of an MM_{Domain} metamodel.

In Table 5.7, we compare the most essential model types of MM_{TDL} with the transformation parts generated from them. As our transformation concepts for deriving $T_{TDL\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}TDL}$ transformations are similar, the model elements listed in Table 5.6 and Table 5.7 and their number are identical. For the same reason, the transformation parts shown in Table 5.7 are close to those contained in Table 5.6. Because stereotype-specific mapping operations are not required for $T_{UML\text{-}to\text{-}DM}$ transformations, they are not regarded in Table 5.7.

As shown in Table 5.7, the same numbers of mapping operations and helper functions was generated as corresponding element types are available in MM_{TDL}. This corresponds exactly to what we expected, and we can therefore state that all parts of transformation $T_{UML\text{-}to\text{-}TDL}$ were generated in accordance with our transformation concept.

Verification of the various transformation parts: Some of the design decisions we defined for the derivation of model transformations are only relevant for the previously discussed $T_{DM\text{-}to\text{-}UML}$ transformations, but not for the creation of $T_{UML\text{-}to\text{-}DM}$ transformations. Therefore, we essentially consider our transformation concept when verifying $T_{UML\text{-}to\text{-}DM}$ transformations based on the example of the $T_{UML\text{-}to\text{-}TDL}$ transformation.

The previously discussed Design Decision 31 is also applicable for the derivation of $T_{UML\text{-}to\text{-}DM}$ transformations. Accordingly, we qualify the context and result parameters and all other type references in these transformations with the relevant *Namespace*. Because we map UML metaclasses to those of a metamodel MM_{Domain}, the types of context parameters of mapping operations must be qualified with the *Namespace* of UML, and the types of result parameters with the *Namespace* of MM_{Domain}. In case of the MM_{TDL}, the namespace to be used is TDL. As we see from the mapping operations shown in Fig. 5.11, their context and result parameters are qualified as expected. Thus, Design Decision 31 can be considered as successfully verified for $T_{UML\text{-}to\text{-}DM}$ transformations.

Just as for transformation $T_{TDL\text{-}to\text{-}UML}$ and according to our transformation concept $T_{UML\text{-}to\text{-}TDL}$, 'disjuncting mappings' must be generated from abstract MC_{St} metaclasses. We verify this aspect using the MC_{St} DataType shown in Fig. 5.6a as an example. Because two subclasses are specified for this metaclass in MM_{TDL}, the mapping candidates shown in Lines 3 and 4 of Figure 5.11 are created. This is exactly what we expect, and we consider the generation of 'disjuncting mappings' to be successfully verified.

Next, we shift our view to the creation of mapping operations with operation body based on non-abstract MC_{St} metaclasses. After creating the signature of these mapping operations and taking Design Decision 31 into account, we create a **when** clause that serves the same purpose as in $T_{DM\text{-}to\text{-}UML}$ transformations, i.e., to ensure that mapping operations can only be invoked for certain UML elements. However, in contrast to mapping operations of $T_{DM\text{-}to\text{-}UML}$ transformations, we must test whether a specific *Stereotype* is applied to the UML element passed as a context parameter. For this purpose, we invoke operation isStereotypedBy() in a **when** clause. As argument for this call, we pass the qualified stereotype name, which we determine based on the MC_{St} used to derive a mapping operation.

```
 1 mapping UML::DataType::toDataType() : TDL::DataType
 2 disjuncts
 3   UML::PrimitiveType::toSimpleDataType,
 4   UML::DataType::toStructuredDataType
 5   {}
 6
 7 mapping UML::DataType::toStructuredDataType() : TDL::StructuredDataType
 8 when { self.isStereotypedBy('UP4TDL::StructuredDataType') }
 9 {
10   // Mapping of 'taggedValues' of the applied stereotype 'UP4TDL::StructuredDataType'
11   var stereotype := self.getAppliedStereotype('UP4TDL::StructuredDataType');
12   var stInstance :=
13     self.getStereotypeApplication(stereotype).oclAsType(UP4TDL::StructuredDataType);
14   result.memberItem += stInstance.memberItem->map toMember();
15
16   // Mapping of metaclass attributes
17   result.elementImport += self.elementImport->map toAC_ElementImport();
18   result.name := self.name;
19   result.ownedComment += self.ownedComment->map toAC_Comment();
20   result.package := self.package.map toAC_Package();
21   result.packageImport += self.packageImport->map toAC_PackageImport();
22 }
23
24 mapping UML::Property::toComponentInstance() : TDL::ComponentInstance
25 when { self.isStereotypedBy('UP4TDL::ComponentInstance') }
26 {
27   // Mapping of 'taggedValues' of the applied stereotype 'UP4TDL::ComponentInstance'
28   var stereotype := self.getAppliedStereotype('UP4TDL::ComponentInstance');
29   var stInstance :=
30     self.getStereotypeApplication(stereotype).oclAsType(UP4TDL::ComponentInstance);
31   result.role := stInstance.role.toComponentInstanceRole();
32
33   // Mapping of metaclass attributes
34   result.name := self.name;
35   result.ownedComment += self.ownedComment->map toAC_Comment();
36   result.type := self.type.map toType();
37 }
```

Figure 5.11: Mapping operations of transformation $T_{UML\text{-}to\text{-}TDL}$, which are derived from metaclasses in MM_{TDL}

The generation of **when** clauses according to our transformation concept can be verified for the two MC_{St} metaclasses StructuredDataType and ComponentInstance. The mapping operations derived from these metaclasses are shown in Fig. 5.11. If we consider the **when** clauses of both mapping operations in Lines 8 and 25, we see that, for both clauses the expected stereotype name are generated as argument. Thus, we regard the generation of **when** clauses as successfully verified.

According to our transformation concept, each body of a mapping operation consists of three parts. In the first part, we determine the instance of the *Stereo-*

type applied to the UML element passed as a context parameter. Using the determined instance, the owned and inherited attributes of the *Stereotype* are mapped to corresponding metaclass attributes in the second part of the mapping operation. However, we only generate assignment expressions for those stereotype attributes for which the respective counterpart in MM_{Domain} is not defined as 'read-only'; otherwise, a value assignment is impossible. In the third part of a mapping operation generated from an MC_{St}, the attributes of the extended UML metaclass are mapped to corresponding MC_{AC} attributes. To create assignment expressions, we determine all MC_{AC} attributes inherited from an MC_{St}, considering only those that are not defined as 'read-only'.

We examine the creation of assignment expressions on the example of the MC_{St} StructuredDataType in Fig. 5.6a, which provides some inherited MC_{AC} attributes and one MC_{St} attribute. Because the only MC_{St} attribute memberItem is not defined as 'read-only', one assignment expression is created in Line 14 of Fig. 5.11. Furthermore, five of the inherited MC_{AC} attributes of the MC_{St} StructuredDataType are not specified as 'read-only. Hence, the same number of assignment expressions is created for these attributes in Lines 17 – 21. Another example is the MC_{St} ComponentInstance in Fig. 5.7a, which has two MC_{St} and six inherited MC_{AC} attributes, where one MC_{St} and three MC_{AC} attributes are marked as 'read-only'. According to our transformation concept, one assignment expression is introduced for the first attribute in Line 31 and for the latter three assignment expressions in Lines 34 – 36. Based on the results discussed, we consider the derivation of mapping operations with an operation body as successfully verified.

Required rework. Because we can only generate model transformations semi-automatically based on a single metamodel using our derivation approach, a manual revision of the two transformations $T_{TDL\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}TDL}$ is required. We have reworked the main() operation of both transformations, so that the outermost package of an input model is identified and an appropriate mapping operation can be invoked based on it. However, it was not required to revise the 'disjuncting mappings' in both transformations, so that we could use them as generated.

Another situation exists for mapping operations with an operation body in transformation $T_{TDL\text{-}to\text{-}UML}$. A revision of these mapping operations has been required due to the creation of 'mapping proposals'. For instance, we manually completed the mapping proposal contained in mapping operation TDL::StructuredDataType::toDataType() (see Line 17 in Fig. 5.9) as follows:

```
result.ownedAttribute += self.memberItem->map toProperty();
```

If we compare the original mapping proposal with the assignment expression above, we find that only the call of the mapping operation toProperty() is added. We proceeded in the same way for all other mapping proposals. Thanks to the auto-completion function of the QVTo editor, the effort to identify the mapping operations to be called was rather limited.

```
1  mapping TDL::TestDescription::toBehavioredClassifier() : UML::Interaction
2  when { self.oclIsTypeOf(TDL::TestDescription) }
3  {
4    result.name := self.name;
5    result.ownedComment += self.ownedComment−>map toComment();
6    result.ownedParameter := self.formalParameter.map toParameter();
7    result.fragment := self.behaviourDescription.map toInteractionFragment();
8    self.testConfiguration.componentInstance−>forEach(ci)
9    {
10     result.lifeline += object UML::Lifeline {
11       name := ci.name +"_Lifeline";
12       represents := ci.map toProperty();
13     };
14   };
15 }
```

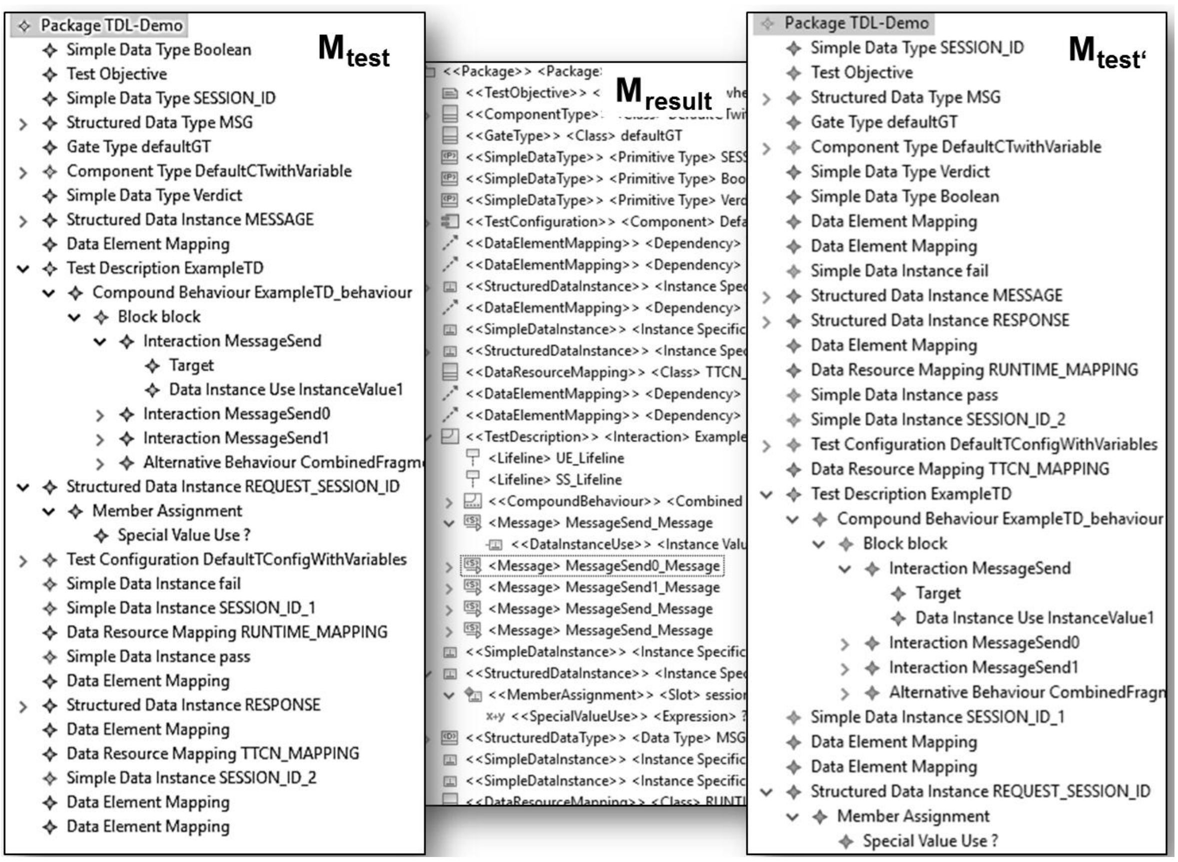
Figure 5.12: Reworked mapping operation of transformation $T_{TDL\text{-}to\text{-}UML}$ with creation of an additional UML element to bridge a syntactic gap

We have extended 14 mapping operations to create additional UML elements required to close various syntactic gaps between TDL and UML. Because these elements not represent TDL-specific language constructs, no *Stereotypes* of UP_{TDL} are applied to them. In Fig. 5.12 an example of such a mapping operation is shown, which was extended to create a UML Lifeline element in Lines 8 to 14. In TDL sequence diagrams, the message exchange is defined between ComponentInstances, whereas in corresponding UML diagrams, Lifelines must be employed as an additional element type. These elements then reference (see Line 12 in Fig. 5.12) an associated component instance in UML.

Applicability evaluation. To evaluate the applicability of $T_{DM\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}DM}$ transformations that are derived from a single metamodel, we apply the manual test methodology as presented in Sec. 2.6.3. A prerequisite is that in addition to the transformation T_{test} to be tested, an inverse transformation T_{inv} is present. Then, from a test model M_{test} that is used as input for T_{test}, a result model M_{result} is generated. Afterwards, M_{result} is transformed into a model $M_{test'}$ using T_{inv}. To evaluate a test as successfully passed, both models M_{test} and $M_{test'}$ must match. We employ the tool *EMFCompare* to automatically evaluate the correspondence of two models.

In addition to small model fragments, which we use to verify the generated artefacts of MM_{TDL} and UP_{TDL}, we create a complete model as an instance of MM_{TDL} and a corresponding UML model with applied UML profile UP_{TDL}. Both models serve as test models for evaluating the applicability of the transformations $T_{TDL\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}TDL}$.

Our general procedure for evaluating the applicability of the derived model transformations is explained below using the $T_{TDL\text{-}to\text{-}UML}$ transformation as exam-



M_{test}

```
Package TDL-Demo
  Simple Data Type Boolean
  Test Objective
  Simple Data Type SESSION_ID
> Structured Data Type MSG
  Gate Type defaultGT
> Component Type DefaultCTwithVariable
  Simple Data Type Verdict
> Structured Data Instance MESSAGE
  Data Element Mapping
v Test Description ExampleTD
  v Compound Behaviour ExampleTD_behaviour
    v Block block
      v Interaction MessageSend
          Target
          Data Instance Use InstanceValue1
      > Interaction MessageSend0
      > Interaction MessageSend1
      > Alternative Behaviour CombinedFragm
v Structured Data Instance REQUEST_SESSION_ID
  v Member Assignment
      Special Value Use ?
> Test Configuration DefaultTConfigWithVariables
  Simple Data Instance fail
  Simple Data Instance SESSION_ID_1
  Data Resource Mapping RUNTIME_MAPPING
  Simple Data Instance pass
  Data Element Mapping
> Structured Data Instance RESPONSE
  Data Element Mapping
  Data Resource Mapping TTCN_MAPPING
  Simple Data Instance SESSION_ID_2
  Data Element Mapping
  Data Element Mapping
```

M_{result}

```
<<Package>> <Package
<<TestObjective>> <
<<ComponentType>
<<GateType>> <Class> defaultGT
<<SimpleDataType>> <Primitive Type> SES
<<SimpleDataType>> <Primitive Type> Boo
<<SimpleDataType>> <Primitive Type> Verd
<<TestConfiguration>> <Component> Defa
<<DataElementMapping>> <Dependency>
<<DataElementMapping>> <Dependency>
<<StructuredDataInstance>> <Instance Spe
<<DataElementMapping>> <Dependency>
<<SimpleDataInstance>> <Instance Specific
<<StructuredDataInstance>> <Instance Spe
<<DataResourceMapping>> <Class> TTCN_
<<DataElementMapping>> <Dependency>
<<DataElementMapping>> <Dependency>
<<TestDescription>> <Interaction> Example
    <Lifeline> UE_Lifeline
    <Lifeline> SS_Lifeline
  > <<CompoundBehaviour>> <Combined
  v <Message> MessageSend_Message
      <<DataInstanceUse>> <Instance Valu
  > <Message> MessageSend0_Message
  > <Message> MessageSend1_Message
  > <Message> MessageSend_Message
  > <Message> MessageSend_Message
<<SimpleDataInstance>> <Instance Specific
<<StructuredDataInstance>> <Instance Spe
  v <<MemberAssignment>> <Slot> sessio
      x+y <<SpecialValueUse>> <Expression> ?
<<StructuredDataType>> <Data Type> MSG
<<SimpleDataInstance>> <Instance Specific
<<SimpleDataInstance>> <Instance Specific
<<DataResourceMapping>> <Class> RUNTI
```

$M_{test'}$

```
Package TDL-Demo
  Simple Data Type SESSION_ID
  Test Objective
> Structured Data Type MSG
  Gate Type defaultGT
> Component Type DefaultCTwithVariable
  Simple Data Type Verdict
  Simple Data Type Boolean
  Data Element Mapping
  Data Element Mapping
  Simple Data Instance fail
> Structured Data Instance MESSAGE
> Structured Data Instance RESPONSE
  Data Element Mapping
  Data Resource Mapping RUNTIME_MAPPING
  Simple Data Instance pass
  Simple Data Instance SESSION_ID_2
> Test Configuration DefaultTConfigWithVariables
  Data Resource Mapping TTCN_MAPPING
v Test Description ExampleTD
  v Compound Behaviour ExampleTD_behaviour
    v Block block
      v Interaction MessageSend
          Target
          Data Instance Use InstanceValue1
      > Interaction MessageSend0
      > Interaction MessageSend1
      > Alternative Behaviour CombinedFragn
  Simple Data Instance SESSION_ID_1
  Data Element Mapping
  Data Element Mapping
v Structured Data Instance REQUEST_SESSION_ID
  v Member Assignment
      Special Value Use ?
```

Figure 5.13: TDL example transformed to a UML model and the other way around

ple. As test model we use the TDL model shown in Fig. 5.6b, which we have adopted from the TDL specification [40]. To evaluate the transformation TDL-to-UML we use this model as M_{test}, which is shown as tree structure in Fig.5.13 and contains 63 model elements. On M_{test} we apply transformation $T_{TDL\text{-}to\text{-}UML}$ to obtain the UML model M_{result} as transformation result. This model has the UML profile UP_{TDL} applied and consists of 71 UML elements. However, only 63 of these model elements have *Stereotypes* of UP_{TDL} applied. For instance, UML Lifeline and Message are such elements without applied *Stereotypes*. These elements are created by transformation $T_{TDL\text{-}to\text{-}UML}$ in order to close syntactic gaps between TDL and UML.

We employ $T_{UML\text{-}to\text{-}TDL}$ as inverse transformation T_{inv} to transform M_{result} into the TDL model $M_{test'}$. Because both models M_{test} and $M_{test'}$ are instances of the TDL metamodel, a structural comparison is possible using EMFCompare.

Just as M_{test}, the model $M_{test'}$ shown in Fig.5.13 consists of 63 models. This proves that no additional elements are created by transformation $T_{TDL\text{-}to\text{-}UML}$ or that existing elements are omitted during the transformation. However, this purely quantitative comparison does not allow any conclusions to be drawn about the structural correlation between both models.

Using EMFCompare, we have detected deviations between $M_{test'}$ and M_{test} at certain points. The differences are observable at places where several elements are assigned to a multi-valued attribute. If we compare the child elements owned by TDL Package TDL-Demo in M_{test} and $M_{test'}$, we find a difference in the order of the child elements. One might argue that this is an error, but it is not. The packagedElement attribute of the TDL metaclass Package is defined as a Set. Consequently, the order of elements assigned to this attribute does not matter. Exactly this behaviour can be observed in model $M_{test'}$. The order of elements assigned to multi-valued attributes of type Set or Bag may differ from M_{test}, but this is correct.

Based on our evaluation on the example of the transformation $T_{TDL\text{-}to\text{-}UML}$, we conclude that the model transformations semi-automatically generated by our approach can be applied to transform domain models to UML models with applied UML profile and vice versa after manual completion. Even though we did not consider the transformation $T_{UML\text{-}to\text{-}TDL}$, we could at least show its usability in principle, because we used it as an inverse transformation for testing the transformation $T_{TDL\text{-}to\text{-}UML}$.

5.3.7 Summary of Evaluation Results.

We examined the applicability of the various aspects of our derivation approach using the TDL case study. We focussed on the automated derivation of a UML profile for TDL as well as the automatic update and transfer of the static semantics defined by OCL from the TDL metamodel to the derived UML profile. In addition, we investigated the semi-automatic derivation of model transformations for model interoperability.

The main objective of our TDL case study was to prove that our derivation approach is applicable not only to existing DSLs with available production rules but also to newly created DSLs (Evaluation Objective 2). Usually, we would first create a metamodel manually as part of an MDE-based development approach for DSLs, to develop further artefacts such as model transformations. Because a metamodel for TDL was present, we only had to enhance it. In a first step, we brought the TDL metaclasses in relation to required 'Abstract Concepts', so that our derivation approach was applicable. We then revised the OCL *Constraints* of the TDL specification [39] because they contained errors. The revised metamodel MM_{TDL} served us as a foundation for the derivation of all further artefacts.

Furthermore, our TDL case study addressed the verification required by Evaluation Objective 1 that all artefacts were derived according to the rules defined by our design decisions. Due to the design of the TDL case study, we were only able to successfully verify the design decisions from the areas of UML profile derivation, OCL update and derivation of model transformations. On the other hand, we could not examine the design decisions for deriving 'additional' metaclasses and for creating CMOF-based metamodels.

Another open issue is the verification of the design decisions concerning the derivation of stereotype attributes defined as 'read-only' and 'derived' from subsetting metaclass attributes. Because only a few attributes use this feature in MM_{TDL}, we could not find an appropriate example for the verification. For this reason, we examine all outstanding issues in the SDL case study below.

In the previous sections we could demonstrate that UML profiles derived using our automated derivation approach are comparable to manually created ones. Furthermore, we have shown that an update and transfer of the OCL-defined static semantics from the metamodel MM_{TDL} to the generated UML profile UP_{TDL} is possible. We carried out both steps automatically without the need to make any manual changes. Therefore, we consider Evaluation Objective 4 to be fulfilled.

As a last step in our TDL case study, we investigated the semi-automatic derivation of model transformations to achieve model interoperability between TDL and UML models with applied UML profile UP_{TDL}. Using a transformation example, we have shown that the manually added transformations $T_{TDL\text{-}to\text{-}UML}$ and $T_{UML\text{-}to\text{-}TDL}$ can be applied to transform a given model into both languages. Thus, we regard Evaluation Objective 5 to be fulfilled.

5.4 Case Study B: Specification and Description Language

In this section, we analyse the results of our case study on the *Specification and Description Language (SDL)* [70]. To address Evaluation Objective 3, the focus of this case study is to evaluate whether our derivation approach successfully implements the use case *'Existing DSL with existing production rules'*. We only consider those artefact types that were not captured by the previous TDL case study. Thus, we examine the generation of a metamodel MM_{Domain} based on production rules of an existing DSL and the derivation of 'additional' metaclasses, which extend the UML metamodel at MOF-level using inheritance. Because we could not fully investigate all aspects of the derivation of 'derived' and 'read-only' stereotype attributes in the TDL case study, our present case study also covers this issue.

Because the manual creation of a UML profile for SDL motivated us to develop an automated derivation approach, we finally compare our automatically derived language constructs with those in Chap. 3, where our manually created UML profile is presented. In this way, we intend to assess the applicability and quality of the UML profiles that our approach derives automatically.

5.4.1 Evaluation of the Derived Metamodel

In the following, we evaluate the metamodel MM_{Domain} created in Step (A) of our derivation approach (see Sec. 4.1.1) using the production rules specified for the abstract syntax of SDL [70]. To refer to the metamodel thus obtained, we use the acronym MM_{SDL}. Before this metamodel is quantitatively analysed by us, we first introduce the 'Abstract Concepts' employed for its creation. Then, we verify the various types of model elements contained in MM_{SDL}, in the light of the relevant Design Decisions 1 to 9.

The 'Abstract Concepts' used for SDL. To derive the metamodel MM_{SDL} using our approach, we reuse some metaclasses of UML [116] as 'Abstract Concepts'. We copy these UML metaclasses to a metamodel MM_{AC}, which we use in addition to SDL's production rules as input for generating MM_{SDL}.

Compared to TDL, SDL provides significantly more language constructs, particularly for specifying behaviour. Because the metaclasses contained in UML's 'Kernel' package only provide concepts for structural aspects, e.g., classes and data types, they are not sufficient to define the 'Abstract Concepts' required for MM_{SDL}. Due to SDL's different types of state machines, we also require 'Abstract Concepts' for the behaviour specification. For this reason, our MM_{AC} for SDL also reuses metaclasses contained in the 'State Machines', 'Activities' and 'Common Behavior' packages of UML.

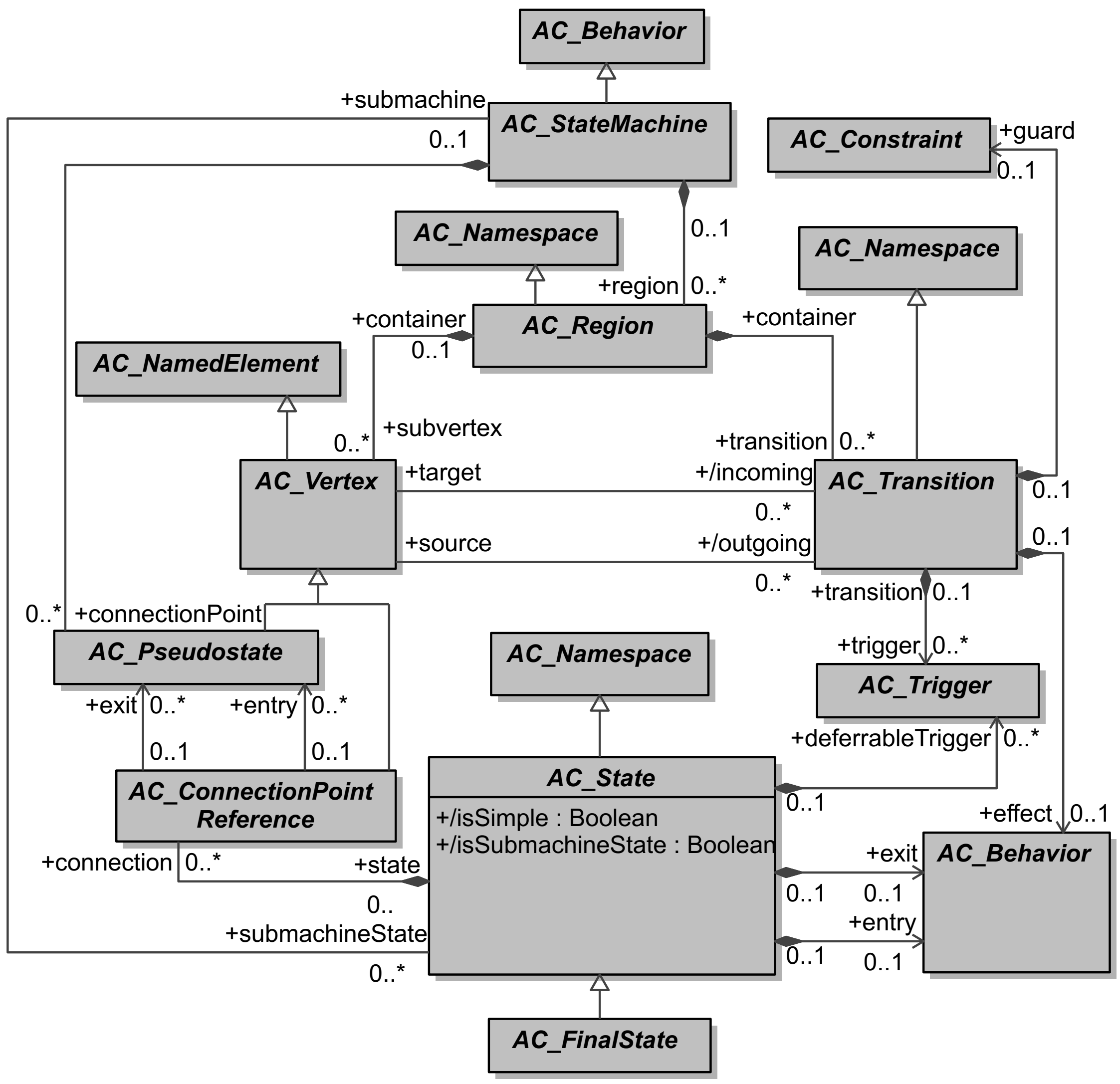


Figure 5.14: The 'Abstract Concepts' for a state machine in SDL

We have adopted a total of 43 UML metaclasses to capture the 'Abstract Concepts' for SDL. An example of the 'Abstract Concepts' used is given in Fig. 5.14, which contains a modified UML StateMachine. The 'Abstract Concepts' shown in this figure serve us to define the SDL language concepts ProcedureDefinition and CompositeStatetypeDefinition.

To obtain the final 'Abstract Concepts', we have removed all non-required features from the copied UML metaclasses. An example of such a metaclass is AC_State, which provides a lower number of features than its UML counterpart. This is because an SDL state only supports the invocation of an activity when it is entered or left, but not when the system remains in it. Therefore, we removed the doActivity property of the metaclass AC_State.

Overview of the derived metamodel. In Table 5.8, we quantitatively compare the production rules of SDL's abstract syntax and the derived model elements for the MM_{SDL} metamodel. In a first step, we analyse whether the correct number of expected model elements is obtained.

Creation of metaclasses: Based on 180 production rules of SDL's abstract syntax, we create seven abstract and 94 non-abstract metaclasses listed in Table 5.8, and two *Enumeration* data types for MM_{SDL} according to the mapping rules defined by Design Decision 1. The lower number of metaclasses compared to the production rules is due to the fact that 77 *EquivalenceRules* with a *NonTerminalDomain* expression are not mapped to metaclasses in MM_{SDL} (Design Decision 4). This is because such production rules refer to other production rules and do not introduce new symbols.

If we compare the *Rules* that have an *Alternative* expression, with the abstract metaclasses created from them, we notice a difference in their quantities. Therefore, one might expect five abstract metaclasses to be created, but seven are present in MM_{SDL}. The two additional metaclasses are introduced because of two production rules for which an **abstract class** annotation is defined according to Design Decision 3.

In addition to the inheritance relationships, which we automatically determine based on *Rules* with *Alternative* expressions, it is also possible to explicitly define an inheritance relationship using the **generalization** annotation (Design Decision 2). We can employ this annotation type to put the metaclasses that are derived from production rules in inheritance relations to 'Abstract Concepts'. In total, 69 SDL production rules have a **generalization** annotation attached, so that an equal number of *Generalizations* between generated and MC_{AC} metaclasses is present in MM_{SDL}.

Our annotation supports the definition of constraints for metaclasses. For this purpose, we provide the **constraint** annotation that has to be attached to a *Rule* node. As shown in Table 5.8, we have defined 169 annotations of this type for *Rule* nodes. Consequently, an equal number of *Constraints* for metaclasses is created in

Table 5.8: SDL's production rules compared with the derived metamodel MM_{SDL}

SDL Production Rules		MM_{SDL}	
Production rules	**180**	**103**	**Model Elements**
Rules with *Alternative* expression	5	7	Abstract metaclasses
Rules with 'abstract class' annotation	2		
EquivalenceRules with *NonTerminalDomain* expression	77	—	
Rules with other expression types	94	94	Non-abstract metaclasses
Rules with *EnumerationDomain* expression	2	2	*Enumerations*
Rules with 'generalization' annotation pointing to an 'Abstract Concept'	69	69	Metaclasses inheriting from 'Abstract Concepts'
Rules with 'constraint' annotation	196	196	*Constraints*
Rule expressions	**265**	**265**	**Metaclass attributes**
Expressions referring to an *ElementaryDomain* or *EnumerationDomain*	179	145	Metaclass attributes defined as reference to metaclasses or *DataTypes*
Expressions referring to a *NonTerminalDomain*	86	120	Metaclass attributes defined as composite property
Expression annotations to support MOF Features		**Effect to metaclass attributes**	
Expressions with 'derived union' annotation	1	1	Attributes defined as 'derived union'
Expressions with 'subsetted property' annotation	118	118	Attributes subsetting other attributes
Expressions with 'redefined property' annotation	42	42	Attributes redefining other attributes
Expressions with 'derived property' annotation	42	42	Attributes specified as 'derived'
Expressions with 'read only' annotation	3	3	Attributes defined as 'read-only'

MM_{SDL}. Because we only specify the textual description of SDL's well-formedness rules [72] using **constraint** annotations, we have to manually translate the created *Constraints* to OCL. The details are discussed in a later section.

Creation of metaclass attributes: Apart from *Alternative* expressions, we map 265 *Expressions* of *Rules* to an equal number of metaclass attributes as required by Design Decision 1. The metaclass or *DataType* to be referenced by the *type* property of a metaclass attribute thus created is determined using the algorithm defined by Design Decision 2. If a metaclass is returned as result of this algorithm, a metaclass attribute is declared as composite feature (*aggregation* = *composite*); otherwise, it represents a reference (*aggregation* = *none*) to the determined *DataType*. Because of these two mapping variants, we distinguish between two groups of *Ex-*

pressions in Table 5.8. The first comprises 179 *Expressions* that have to be mapped to metaclass attributes that represent references, while the second group captures 86 *Expressions* that have to be mapped to composite metaclass attributes.

One could assume that based on the *Expressions* of the two groups, an equal number of metaclass attributes is created in MM_{SDL}. As shown in Table 5.8, our assumption cannot be true because the number of attributes created differs from the number of expressions present in both groups. This is due to the fact that our default mapping of *Expressions* can be skipped using the annotations **referenced rule**, **referenced type** and **composite type**, which we introduced to meet Design Decision 5. These annotations allow us to manually specify not only the *aggregation* kind of metaclass attributes, but also the metaclass or data type to be referenced by the attribute's *type* property.

Although 179 *Expressions* referring to an *ElementaryDomain* or *EnumerationDomain* exist, only 145 metaclass attributes defined as references are created in MM_{SDL}. The remaining 34 *Expressions* of this group are mapped to metaclass attributes defined as composite property, because they have a **referenced type** or **referenced rule** annotation attached. Including the 86 *Expressions* that are mapped in the default manner, we obtain a total of 120 metaclass attributes that represent composite properties.

Effect of production rule annotations: In addition to the annotation types discussed above, we offer additional ones that do not affect the default mapping of *Expressions* to metaclass attributes. These are intended to support those MOF language concepts that cannot be represented by means of language constructs of an EBNF notation, such as attribute redefinition. In Table 5.8, we summarize all these annotation types in the section *'Expression annotations to support MOF features'*.

To meet the rules of Design Decision 6, we provide the three annotations **derived union**, **redefined property** and **subsetted property**, which support the equally named CMOF features related to the definition of metaclass attributes. As shown in Table 5.8, the number of *Expressions* that have a particular type of annotation attached is equal to the number of metaclass attributes for which the CMOF feature associated with this annotation type is defined. For instance, exactly one metaclass attribute, whose *isDerivedUnion* property has the value 'true' assigned, is introduced for an *Expression* with an attached **derived union** annotation.

We provide the two annotations **derived property** and **read only** to fulfil Design Decision 7. Each of these annotation types represents an EMOF feature of the same name, which is applicable to metaclass attributes. Again, the number of annotated *Expressions* is equal to the number of metaclass attributes affected.

Because the information provided by means of **documentation** annotations is intended for documentational purposes only, we do not include this annotation type in Table 5.8. In total, 293 *Rules* and *Expressions* have a **documentation** annotation attached. Therefore, an equal number of *Comment* elements is created according to Design Decision 9.

Verifying the elements derived for MM_{SDL}. In the following, we verify whether the various types of model elements in MM_{SDL} are created according to the relevant Design Decisions 1 to 9. We conduct these verifications on the example of two production rules from SDL, which allow us to examine all aspects of the mentioned design decisions.

Creation of metaclasses: We first verify whether model elements of MM_{SDL} are derived from production rules according to the rules defined by Design Decision 1. For this purpose, we use the production rule Procedure-definition and the metaclass derived from it, which are shown in Fig. 5.15. Thereafter, we consider the various annotation types in detail.

As pointed out earlier, we map a *DefinitionRule* whose *Expression* is not of type *Alternative* to a non-abstract metaclass. Because our Procedure-definition example is such a *DefinitionRule*, it is mapped to the non-abstract metaclass shown in Fig. 5.15b.

Our considered production rule has a *Composition* as *Expression*, which is defined in Lines 10 to 38 of Fig. 5.15a. A *Composition* enables us to specify a production rule consisting of several symbols that are mapped to an equal number of metaclass attributes. Based on the ten *Expressions* contained in our *Composition* example, we thus derive ten metaclass attributes in Fig. 5.15b, where the explicitly specified procedureName attribute and nine attributes defined via *Association* ends are shown. The *type* properties of the explicitly defined attributes refer to the *PrimitiveType* String, whereas that of the implicitly defined ones refers to different metaclasses.

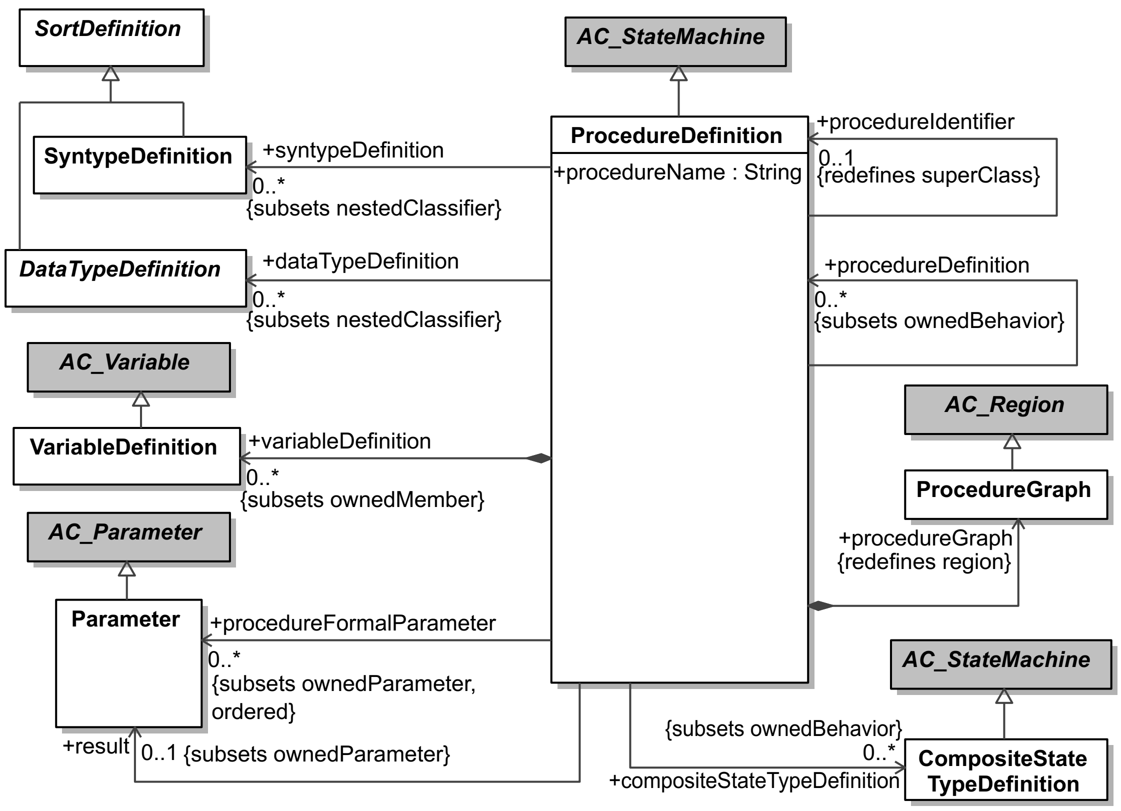
Algorithm for resolving attribute types: The *type* properties of the metaclass attributes procedureName, variableDefinition and procedureGraph are determined from the source expressions in Lines 11, 28 and 36 of Fig. 5.15a using the algorithm introduced by Design Decision 4. This algorithm resolves *EquivalenceRules* that are linked via *NonTerminalDomain* expressions until either a *DefinitionRule* or an *EquivalenceRule* with an expression other than *NonTerminalDomain* is reached. Based on the *Rule* resolved in this way, we determine the metaclass or data type to be used as the *type* of a metaclass attribute.

If we apply our algorithm to the *NonTerminalDomain* expression shown in Line 36 of Fig. 5.15a, we expect the *DefinitionRule* Procedure-graph to be returned as result. This is because the algorithm has to terminate exactly for this type of production rule. If we consider the attribute procedureGraph in Fig. 5.15b, we find that its *type* is specified as an *Association* end referring to the metaclass Procedure-Graph, which is derived from the production rule determined by our algorithm.

```
Procedure−definition :: annotations {
  generalized class ”AC_StateMachine”;
  documentation: ”This metaclass represents ‘Procedure−definition’ as specified in Z.102 /
    9.4 Procedure.”;
  constraint name ”Constraint_1”:
    body ”In an ‘SdlSpecification’, all potentially instantiated procedures shall have a
      ‘ProcedureStartNode’.”;
  constraint name ”Constraint_2”:
    body ”The ‘ProcedureGraph’ of the ‘ProcedureDefinition’ of an operation shall not
      contain a ‘StateNode’.”;
  constraint name ”Constraint_3”:
    body ”The ‘procedureDefinition’ property of the ‘ProcedureDefinition’ of an operation
      shall be empty.”;
{
  Procedure−name annotations {
    redefined property ”AC_NamedElement::name” }
  Procedure−formal−parameter* annotations {
    derived property = ”All ‘ownedAttribute’ items with a direction not equal to ‘return’.”;
    referenced type ”SDL−MM::Parameter” }
  [Result] annotations {
    derived property = ”An ‘ownedAttributes’ with a direction of ‘return’.”;
    referenced type ”SDL−MM::Parameter” }
  [Procedure−identifier] annotations {
    referenced rule Procedure−definition;
    redefined property ”AC_Class::superClass” }
  Data−type−definition −set annotations {
    subsetted property ”AC_Class::nestedClassifier”;
    referenced rule Data−type−definition }
  Syntype−definition −set annotations {
    subsetted property ”AC_Class::nestedClassifier”;
    referenced rule Syntype−definition }
  Variable−definition −set annotations {
    subsetted property ”AC_Namespace::ownedMember” }
  Composite−state−type−definition −set annotations {
    subsetted property ”AC_BehavioredClassifier::ownedBehavior”;
    referenced rule Composite−state−type−definition }
  Procedure−definition −set annotations {
    subsetted property ”AC_BehavioredClassifier::ownedBehavior”;
    referenced rule Procedure−definition }
  Procedure−graph annotations {
    derived property = ”This is derived from the ‘region’ property.”; }
}
```

(a) Annotated production rule of SDL’s Procedure-definition

Figure 5.15: Production rule of SDL Procedure Definition and the metaclass derived from it



(b) Metaclass of MM_{SDL} that is derived from the production rule in Part (a)

Figure 5.15: The production rule of SDL Procedure Definition and the metaclass derived from it (continued)

As another example for the verification of the algorithm defined by Design Decision 4, we now consider the procedureName attribute in Fig. 5.15b, which is derived from the *NonTerminalDomain* expression in Line 11 of Fig. 5.15a. This expression refers to the *ElementaryDomain* token via the following two *EquivalenceRules*:

```
Procedure−name = Name
Name = Token
```

The type resolution algorithm resolves the above *EquivalenceRules* until it reaches an expression that is not a *NonTerminalDomain*. This is the *ElementaryDomain* Token in the case of our example. As we always map this expression type to the *PrimitiveType* String, the *type* property of the procedureName attribute in Fig. 5.15b must reference exactly this data type. This meets our expectations and, therefore, we regard Design Decision 4 as successfully verified.

Explicit attribute type specification: Apart from the type resolution based on our algorithm, we can also explicitly define the attribute type using the **referenced rule** and **referenced type** annotations (Design Decision 5). In addition, both annotation types imply the mapping of an *Expression* to a metaclass attribute that represents a reference to a metaclass or a data type. Seven *Expressions* shown in Fig. 5.15a have attached one of the two annotations types. To be able to consider Design Decision 5 as successfully verified, we expect that the data types or metaclasses that are explicitly specified by the annotations are referenced by the *type* properties of the derived metaclass attributes.

The two *Expressions* listed in Lines 13 and 16 of Fig. 5.15a have **referenced type** annotations, each of which references the metaclass SDL-MM::Parameter. We expect the metaclass attributes procedureFormalParameter and result, which are derived from the mentioned *Expressions*, to refer to the specified metaclass. If we consider these two attributes in Fig. 5.15b, we find that our expectations regarding their *type* properties are met.

In contrast to the **referenced type** annotation, we employ the **referenced rule** annotation to refer to a production rule. However, the effect of both annotations is the same; the metaclass or data type to which the *type* property of an attribute shall refer is explicitly specified. For example, the **referenced rule** annotation of the *Expression* in Line 33 of Fig. 5.15a refers to the production rule Procedure-definition from which the ProcedureDefinition metaclass in MM_{SDL} is derived. As we expect, the *type* property of the procedureDefinition attribute derived from the *Expression* in question refers to the metaclass of the same name (see Fig. 5.15b).

Because seven *Expressions* in Fig. 5.15a are provided with a **referenced type** or **referenced rule** annotation, we expect that the metaclass attributes derived from them represent references. If we check Fig. 5.15b, we find that all seven attributes derived from the *Expressions* in question are defined as references. Consequently, we consider Design Decision 5 as successfully verified.

Attribute collections and cardinalities: *Set* expressions introduces a set of symbols that we map to multi-valued metaclass attributes with a lower cardinality 0 and an upper cardinality infinite. As multi-valued metaclass attributes represent a set by default (*isOrdered = false*), we do not have to specify this explicitly. A *Set* expression can be identified by the keyword **set** at its end. To verify the mapping of *Set* expressions according to Design Decision 1, we use the expression shown in Line 22 of Fig. 5.15a from which we derive the attribute dataTypeDefinition in Fig. 5.15b. If we examine this attribute, we notice that the values for its lower and upper cardinality are set as anticipated by us.

All expressions not specified as *Option, List* or *Set* are mapped to single-valued attributes with a lower and upper cardinality equal to 1, according to the rules of Design Decision 1. In the case of the production rules shown in Fig. 5.15a, the aforementioned prerequisite applies to the two expressions defined in Lines 11 and 36, from which we derive the two attributes procedureName and procedure-Graph in Fig. 5.15b. If we examine both attributes, we find that the values of their lower and upper cardinalities are equal to 1, which is the value we expect.

Processing of annotations: In addition to the *Generalizations* that we introduce between metaclasses for *Alternative* expressions, we can also explicitly define inheritance relationships using the **generalized class** annotation according to Design Decision 2. Essentially, we use this annotation to relate the generated metaclasses via inheritance to the MM_{AC} metaclasses that represent the 'Abstract Concepts' defined for a DSL. For instance, the Procedure-definition production rule in Line 2 of Fig. 5.15a has such an annotation attached, which defines that the metaclass AC_StateMachine shall be generalized. For this reason, we expect a *Generalization* relationship to be introduced between the metaclasses ProcedureDefinition and AC_StateMachine. If we consider both metaclasses shown in Fig. 5.15b, we find that this *Generalization* is present. Therefore, we consider Design Decision 2 as successfully verified.

Employing the **abstract class** annotation we can specify that a metaclass to be derived from a production rule shall be defined as abstract (*isAbstract* = *'true'*), as required by Design Decision 3. The production rule Sdl-expression in Fig. 5.16a has such an annotation attached. Therefore, we expect that an abstract metaclass is derived from this production rule. If we consult the metaclass SdlExpression shown in Fig. 5.16b, we notice that its name is displayed in italics, which is the defined representation for abstract metaclasses in class diagrams. Hence, we consider the verification of Design Decision 3 as successfully.

Design Decision 6 requires that expressions in production rules can be annotated to enable the use of attribute subsetting and redefinition. In addition, an annotation must be provided to specify that an *Expression* shall be mapped to a metaclass attribute defined as 'derived union'. For these reasons, we provide the three annotations **redefined property**, **subsetted property** and **derived union**.

The expressions of the production rule Sdl-expression shown in Fig. 5.16a have all three mentioned annotation types applied. The expression in Line 10 has the **derived union** and **subsetted property** annotations attached, and the **redefined property** annotation is present for the expression shown in Line 17.

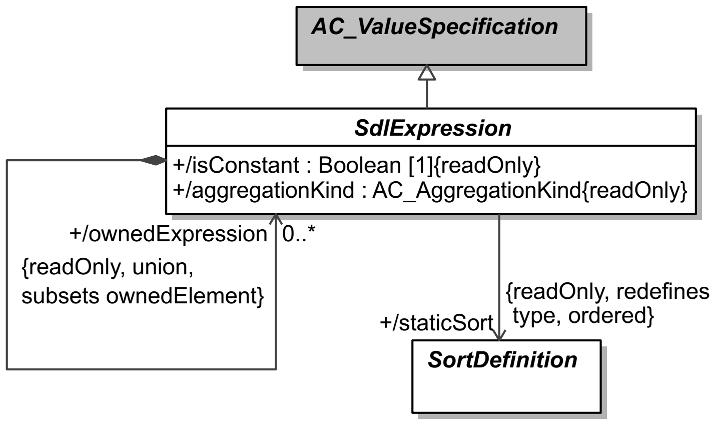
The ownedExpresssion attribute in Fig. 5.16b derived from the *Expression* in Line 10 is labelled with union and subsets. The latter label is followed by the attribute name ownedElement, which represents the attribute to be subsetted. If we compare the properties of the metaclass attribute in question with the annotations of the source expression in Line 10 of Fig. 5.16a, we find a match. Thus, we consider the annotations **derived union** and **subsetted property** to be implemented in accordance with Design Decision 6.

The metaclass attribute staticSort in Fig. 5.16b has the redefines label attached, which is followed by the attribute name type. This indicates that the latter attribute is redefined by the staticSort attribute. If we consider the **redefined property** annotation for the source expression in Line 17 of Fig. 5.16a, we notice that exactly the attribute specified there is redefined. Therefore, we evaluate the **redefined property** annotation as conforming to Design Decision 6. Based on the results we achieved in the current and previous paragraph, we consider the design decision in question to be successfully verified.

```
Sdl−expression :: annotations {
  generalized class ”AC_ValueSpecification”;
  abstract class;
  documentation: ”This metaclass represents ‘Expression’ as specified in Z.107 /
    12.2.1 Expressions.”;

  constraint name ”Constraint_1”:
    body ”The ‘aggregationKind’ property shall not have the value of ‘shared’.” }
{
  Owned−expression −set annotations {
    composite type ”SDL−MM::Expression”;
    derived union; read only;
    subsetted property ”AC_Element::ownedElement”; }
  [CONSTANT] annotations {
    derived property = ”False, if a ‘VariableAccess’ or an ‘ImperativeExpression’ or a
      ‘ValueReturningCallNode’ is contained.” }
  Static−sort annotations {
    redefined property ”AC_TypedElement::type”;
    referenced rule Sort−definition;
    derived property = ”This additional property returns the static sort of an SDL
      expression.” }
  Aggregation−kind annotations {
    referenced type ”AC_AggregationKind”;
    derived property = ”This additional property returns aggregation kind of an SDL
      expression.” }
}
```

(a) Annotated SDL-expression production rule


AC_ValueSpecification
SdlExpression
+/isConstant : Boolean [1]{readOnly}
+/aggregationKind : AC_AggregationKind{readOnly}
+/ownedExpression
0..*
{readOnly, union, subsets ownedElement}
{readOnly, redefines type, ordered}
+/staticSort
SortDefinition

(b) Metaclass of MM_{SDL} that is derived from the production rule in Part (a)

Figure 5.16: Production rule of SDL Expression and the metaclass of MM_{SDL} derived from it

According to Design Decision 7, an *Expression* shall be allowed to be annotated with meta-information so that the metaclass attribute derived from it is defined as 'derived' and 'read-only'. To enable this, we provide the two annotations **read only** and **derived property**. In addition, an optional textual description can be provided in the context of the latter annotation.

We verify the two annotations in question using the two *Expressions* in Lines 10 and 22 of Fig. 5.16a. The expression in Line 10 has the **read-only** annotation attached and is mapped to the metaclass attribute ownedElement in Fig. 5.16b. According to the specified annotation, this metaclass attribute is expected to be defined as 'read-only'. Because the readOnly label is present for the ownedElement metaclass attribute in Fig. 5.16b, we consider our expectation as confirmed.

We use the *Expression* shown in Line 22 as example to verify the **derived property** annotation. According to our expectations, the metaclass attribute aggregationKind derived from this expression must be specified as 'derived' and 'read-only'. In class diagrams, attributes defined as derived are displayed with character '/' before the attribute name. If we analyse Fig. 5.16b, we notice that the attribute name of aggregationKind has exactly such a prefix.

We do not evaluate the creation of *Constraints* for metaclasses from **constraint** annotations according to Design Decision 8, because we simply copy the textual specification of the annotation when deriving a metamodel. The same applies to *Comment* elements created from **documentation** annotations according to Design Decision 9.

Manual revision of MM_{SDL}. When we introduced our approach to deriving metamodels in Sec. 4.2, we noted that we can only derive initial metamodels from the production rules of a DSL. Therefore, a manual revision and enhancement of a derived metamodel is required. Typically, the textual descriptions of *Constraints* have to be translated manually into OCL specifications, metaclasses have to be deleted or modified, and the *opposite* ends of *Associations* might need to be redefined or subsetted. Even if we could derive the latter from production rules using an additional annotation type, we opt against such a feature. In our point of, these revisions should be performed directly in a UML model editor, which can evaluate the conformity of metamodels to the static semantics of the MOF. As our editor for production rules presented in Sec. 4.1.4 is implemented using grammar-ware, a validation of the static semantics of the MOF would only be feasible with great effort.

We deleted one metaclass of the 103 metaclasses initially derived from SDL's production rules, because it had only one attribute. Instead, we moved this attribute to another metaclass. Thus, the final metamodel MM_{SDL} consists of a total of 145 metaclasses, 102 of which are created from production rules and 43 of which are copied from MM_{AC}.

Table 5.9: Model elements of the reworked metamodel MM_{SDL}

Model elements of MM_{SDL}	
Metaclasses in total	**145**
MC_{AC} metaclasses	43
MC_{AMC} metaclasses *(7 with «ToMetaclass» stereotype)*	31
MC_{St} metaclasses *(21 with «ToStereotype» stereotype)*	71
Enumerations	**6**
OCL *Constraints* in total	**204**
Constraints owned by MC_{AMC}	44
Constraints owned by MC_{St}	160
OCL *Operations*	**63**
Operations owned by MC_{AMC}	35
Operations owned by MC_{St}	28

In the revised metamodel MM_{SDL}, we stereotyped seven metaclasses with the «ToMetaclass» stereotype, while 21 metaclasses have the «ToStereotype» stereotype applied. As the application of these stereotypes also affects subclasses of the stereotyped metaclasses, we obtain the 31 MC_{AMC} and 71 MC_{St} metaclasses in MM_{SDL} listed in Table 5.9. The former metaclasses are mapped to 'additional' metaclasses, whereas *Stereotypes* are derived from the latter ones.

To manually create OCL *Constraints* for MM_{SDL}, we used the same procedure as for the manually created UML profile for SDL published in the latest version of ITU-T Rec Z.109 [69]; i.e., we applied our approach presented in Sec. 3.4.3 for MM_{SDL}. To translate the complex conditions of SDL's static semantics [72] into simpler constructs, we split some of the 196 *Constraints* in our initial metamodel and introduced OCL-defined *Operations*. Thus, the final metamodel MM_{SDL} contains a total of 204 *Constraints*, 160 of which belong to MC_{St} metaclasses and 44 belong to MC_{AMC} metaclasses. Furthermore, it embraces 63 OCL-defined *Operations* that are invoked by the OCL *Constraints*.

We validated the correctness of the OCL *Constraints* and *Operations* we translated manually using the procedure described in Sec. 3.5, which we successfully applied to the UML profile we created manually for SDL. For this reason, we have decided not to discuss the procedure again at this point.

Conclusions. We quantitatively compared the production rules of SDL's abstract syntax with the resulting model elements in MM_{SDL}, and were able to prove that the number of model elements expected by us corresponds to the number actually generated. However, this analysis does not allow conclusions to be drawn regarding the question whether the generated model elements conform to the relevant Design Decisions 1 to 9. Therefore, we also verified the various types of model elements in detail and were able to successfully verify all design decisions mentioned. For this reason, we consider our derivation approach to be appropriate for deriving initial metamodels from annotated production rules.

As highlighted earlier, a manual revision of a metamodel derived from production rules is always required. This apply especially to the static semantics, which has to be implemented in OCL. However, the classification of metaclasses into MC_{St} and MC_{AMC} metaclasses, as described for the MM_{SDL} metamodel, is only necessary if a UML profile and model transformations are to be generated automatically from a metamodel using our derivation approach.

5.4.2 Evaluation of the Derived UML Profile

In the following, we examine the UML profile for SDL (referred to as UP_{SDL}) obtained by applying our derivation approach to the revised metamodel MM_{SDL}. As with our TDL case study, we wish to demonstrate the applicability and correct implementation of our derivation approach. Using our SDL case study, we address Evaluation Objective 3 that requires the investigation of the use case *'Existing grammarware-based DSL'*, which differs from the one we investigated in the TDL case study. For this reason, we use the metamodel MM_{SDL} generated from the production rules of SDL's abstract syntax as the starting point for deriving a UML profile.

Because we already verified most of the Design Decisions 10 – 20 relevant for the derivation of UML profiles, below we do only consider those not yet addressed. Moreover, we do not investigate the update of OCL constraints, as we already evaluated this aspect comprehensively in our TDL case study.

Overview of the derived UML profile UP_{SDL}. Below we quantitatively compare the element types available in the metamodel to the artifacts generated from them for the UML profile UP_{SDL}. To better illustrate this aspect, we compare in Table 5.10 the number of elements of a certain type in MM_{SDL} to the corresponding elements in UP_{SDL}. Because we map only MC_{St} metaclasses in MM_{SDL} to *Stereotypes* in UP_{SDL} according to Design Decision 10, we regard only this metaclass type in Table 5.10.

Derivation of stereotypes: Of the 71 MC_{St} metaclasses available in MM_{SDL}, 21 are explicitly defined as MC_{St} (Design Decision 17) using the «ToStereotype» stereotype. The remaining 50 MC_{St} metaclasses are implicitly considered as MC_{St} metaclasses, because they inherit either from one of the aforementioned metaclasses or from an MC_{AC} metaclass (Design Decision 11).

All 71 MC_{St} metaclasses of MM_{SDL} are mapped to a corresponding number of *Stereotypes* in UP_{SDL}. In addition, we introduce either *Extensions* or *Generalizations*

Table 5.10: Comparison of metamodel MM_{SDL} and UML profile UP_{SDL}

MM_{SDL}		UP_{SDL}			
Metaclasses		**Stereotypes**			
MC_{St} metaclasses (21 with applied «ToStereotype» stereotype)	**71**	**79**	***Stereotypes***		
• MC_{St} metaclasses inheriting from MC_{AC} metaclasses	57				
• 15 MC_{St} metaclasses with applied «ToStereotype» stereotype		57	*Stereotypes* with *Extensions* to UML metaclasses		
• MC_{St} metaclasses inheriting from MC_{St} metaclasses	14	8	*Stereotypes* inheriting from other *Stereotypes*		
• 6 MC_{St} metaclasses with applied «ToStereotype» stereotype		6	*Stereotypes* that inherit from other *Stereotypes* and have *Extensions* to UML metaclasses		
Metaclass attributes				**Stereotype attributes**	
				‘non-derived’	‘derived’ & ‘read-only’
MC_{St} attributes (27 with applied «ToTaggedValue» stereotypes)	**207**			**65**	**142**
• MC_{St} attributes defined as ‘derived’ and ‘read-only’	56			—	51
• 5 attribute have applied the «ToTaggedValue» stereotype				—	5
• Redefining MC_{St} attributes	38			—	38
• No attribute has applied the «ToTaggedValue» stereotype				—	—
• Subsetting MC_{St} attributes	65			25	26
• 14 attributes have applied the «ToTaggedValue» stereotype				—	14
• Other kind of MC_{St} attributes	48			40	—
• 8 attributes have applied the «ToTaggedValue» stereotype				—	8
Enumerations	6	1	***Enumeration***		
OCL *Constraints* of metaclasses	**160**	**253**	**OCL *Constraints* of *Stereotypes***		
		160	Copied and updated from MM_{SDL}		
		23	Introduced for ‘non-derived’ stereotype attributes that refer to UML classes with applied stereotypes		
		52	Created for subsetting/redefining MC_{St} attributes		
OCL *Operations*	**28**	**28**	**OCL *Operations***		

for the created *Stereotypes*. For better traceability, we differentiate between these two groups of *Stereotypes* in Table 5.10.

As required by Design Decision 12, we introduce *Extensions* to UML metaclasses for *Stereotypes* created from those MC_{St} metaclasses that directly inherited from MC_{AC} metaclasses. As shown in Table 5.10, 57 MC_{St} metaclasses in MM_{SDL} extend UML metaclasses, where 15 of them have the «ToStereotype» stereotype applied. In the case of *Stereotypes* that are derived from such MC_{St} metaclasses, we determine the UML metaclasses to be extended based on the «ToStereotype» stereotype. In contrast, we ascertain the UML metaclass to be extended from the inheritance relationships between MC_{St} and MC_{AC} metaclasses for all remaining *Stereotypes*.

We introduce *Generalizations* according to Design Decision 13 for all *Stereotypes* derived from those MC_{St} metaclasses that inherit from other MC_{St} metaclasses. This applies to 14 MC_{St} metaclasses in MM_{SDL}, however, six of which have the «ToStereotype» applied. Consequently, eight *Stereotypes* that inherit from other *Stereotypes* are created, as well as six further ones that have both an *Extension* to a UML metaclass and a *Generalization* to a *Stereotype*. Because of Design Decision 17, we introduce such *Extensions* based on the «ToStereotype» stereotype applied to the source MC_{St} metaclasses.

Derivation of stereotype attributes: Due to the higher number of relevant design decisions, mapping MC_{St} attributes to stereotype attributes is more complex than creating *Stereotypes*. For this reason, our derivation approach must consider different kinds of attribute mappings. To capture all these aspects in Table 5.10, we classify the MC_{St} attributes and the stereotype attributes derived from them into several groups.

We distinguish between two groups of generated stereotype attributes. One group contains all stereotype attributes that are defined as 'derived' and 'read-only', whereas the other group captures all 'non-derived' stereotype attributes.

Furthermore, we divide the MC_{St} attributes into four groups in Table 5.10. For each of these groups we specify how many of the contained MC_{St} attributes have the «ToTaggedValue» stereotype applied. The default mapping is skipped for the MC_{St} attributes marked in this way; instead, these attributes are always mapped to 'derived' stereotype attributes.

1. The first group of MC_{St} attributes in Table 5.10 includes all MC_{St} attributes that are already defined in MM_{SDL} as 'derived' and 'read-only'. By default, we map attributes of this group to stereotype attributes, which are also defined as 'derived' and 'read-only'.

   In total, 56 of such MC_{St} attributes exist in MM_{SDL} and five of which have applied the «ToTaggedValue» stereotype. Accordingly, we map 51 MC_{St} attributes including their OCL expressions directly to stereotype attributes in UP_{SDL}. We also map the five MC_{St} attributes with applied «ToTaggedValue» stereotype in this way but, according to Design Decision 18, their existing OCL expressions are replaced by those defined by their «ToTaggedValue» stereotype instances.

2. The second MC_{St} attribute group in Table 5.10 captures 38 MC_{St} attributes that redefine other attributes. According to Design Decision 15 we map all attributes of this group to 'derived' and 'read-only' stereotype attributes and introduce new OCL expressions to compute their values at runtime.

3. The third attribute group comprises 56 MC_{St} attributes that subset MC_{AC} attributes, where 14 of these MC_{St} attributes have the «ToTaggedValue» stereotype applied. Based on the rules defined by Design Decision 15, we map 26 MC_{St} attributes of the group to stereotype attributes specified as 'derived' and 'read-only', whereas 'non-derived' stereotype attributes are generated from 25 MC_{St} attributes that do not meet the criteria of Design Decision 15. Because we already outlined the details of the concrete mapping criteria in our TDL case study, we refer to the explanations there. As the result, the 14 MC_{St} attributes with present «ToTaggedValue» stereotype are mapped to 'derived' and 'read-only' stereotype attributes according to Design Decision 18.

4. The last MC_{St} attribute group in Table 5.10 embraces all attributes that do not belong to one of the previous groups. This applies to a total of 48 MC_{St} attributes, eight of which have the «ToTaggedValue» stereotype applied. From the latter MC_{St} attributes, we derive eight 'derived' and 'read-only' stereotype attributes, and the remaining MC_{St} attributes are mapped one-to-one to 'non-derived' attributes according to Design Decision 14.

Other model elements: As shown in Table 5.10, six *Enumeration* data types are contained in the MM_{SDL} metamodel, but only one of them is mapped to an *Enumeration* in UP_{SDL}. This is because the remaining five *Enumeration* are used either in the context of the 'Abstract Concepts' or 'additional' metaclasses.

Furthermore, the MM_{SDL} metamodel contains 160 OCL *Constraints* and 28 OCL-defined *Operations*. After a previous update, these elements are mapped to corresponding ones in UP_{SDL}. If we compare MM_{SDL} and UP_{SDL}, we find that UP_{SDL} comprises 253 OCL *Constraints* instead of 160. This is because we create additional OCL *Constraints* for *Stereotypes* in UP_{SDL} according to Design Decisions 20 and 21.

Verification of the elements derived for *UP_{SDL}*. In our TDL case study, we successfully verified the design decisions relevant for the derivation of UML profiles from metamodels, except of Design Decisions 15 and 20. Now we consider these design decisions, which define the creation of 'derived' and 'read-only' stereotype attributes from subsetting or redefining MC_{St} metaclass attributes and the introduction of additional OCL *Constraints* to ensure the static semantics of *Stereotypes*.

In the TDL case study, we only examined the redefinition of metaclass attributes and the creation of OCL *Constraints* for single-valued metaclass attributes. Therefore, we verify below the derivation of 'derived' and 'read-only' stereotype attributes from subsetting multi-valued MC_{St} attributes. We also consider the creation of OCL *Constraints* from such kind of attributes. We conduct our verification using the «ProcedureDefinition» stereotype in UP_{SDL} shown in Fig. 5.17b, which is derived from the MC_{St} metaclass of the same name in MM_{SDL} shown in Fig. 5.17a.

Mapping of subsetting/redefining attributes: According to the rules defined by Design Decision 15, we must map all attributes of MC_{St} metaclasses that redefine or subset attributes of MC_{AC} metaclasses to 'derived' and 'read-only' stereotype attributes. In addition, we have to introduce OCL expressions that compute the values of the stereotype attributes at runtime. We verify the derivation of such stereotype attributes using as example the two metaclass attributes syntypeDefinition and dataTypeDefinition shown in Fig. 5.17a. We employ these two MC_{St} attributes because they subset the same MC_{AC} attribute nestedClassifier. This aspect is important later for our evaluation of the OCL *Constraints* that are introduced according to Design Decision 20.

When considering the stereotype attributes in Fig. 5.17b, which are generated from our exemplary MC_{St} attributes, we find that our expectations are met, because both attributes in question are marked as 'derived' and 'read-only'. However, this is insufficient to assess Design Decision 15 as successfully verified. We must also evaluate the OCL expressions introduced as *defaultValue* for both stereotype attributes, which are specified as follows:

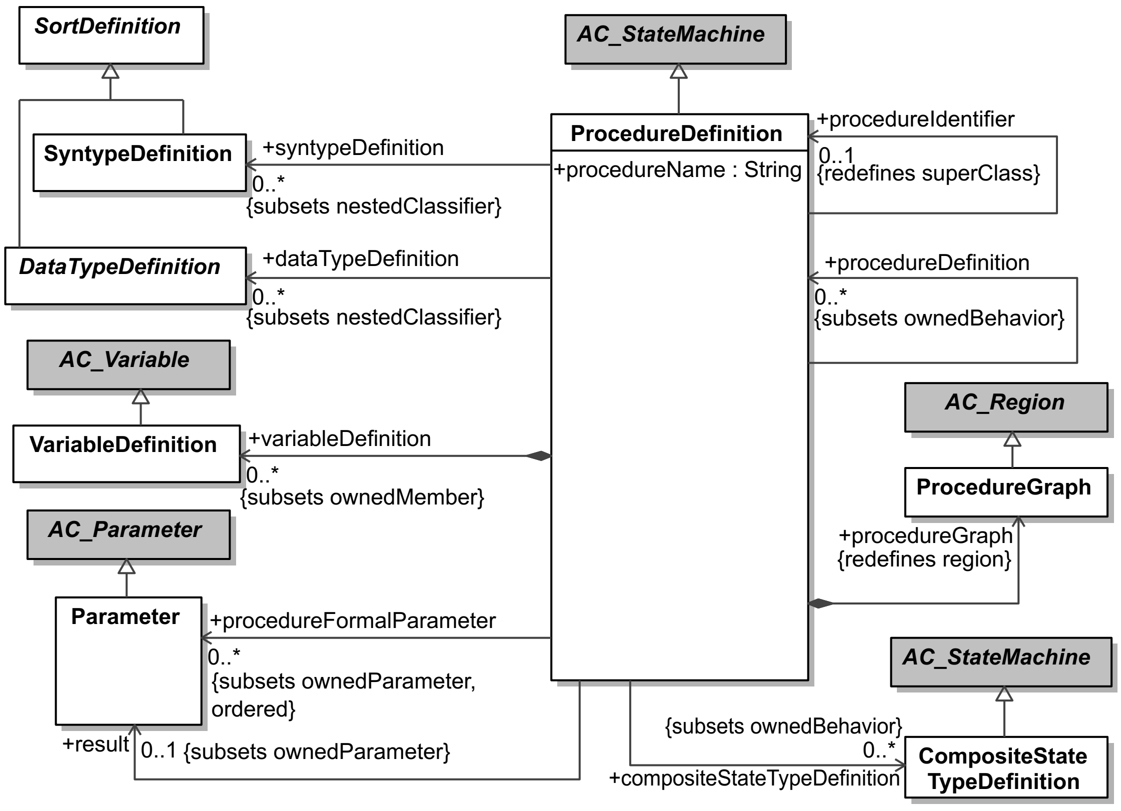
1. OCL expression of the dataTypeDefinition attribute:

```
self.base_StateMachine.nestedClassifier->select(
  isStereotypedBy(`SDLUML::DataTypeDefinition')
).oclAsType(UML::Classifier)->asSet()
```

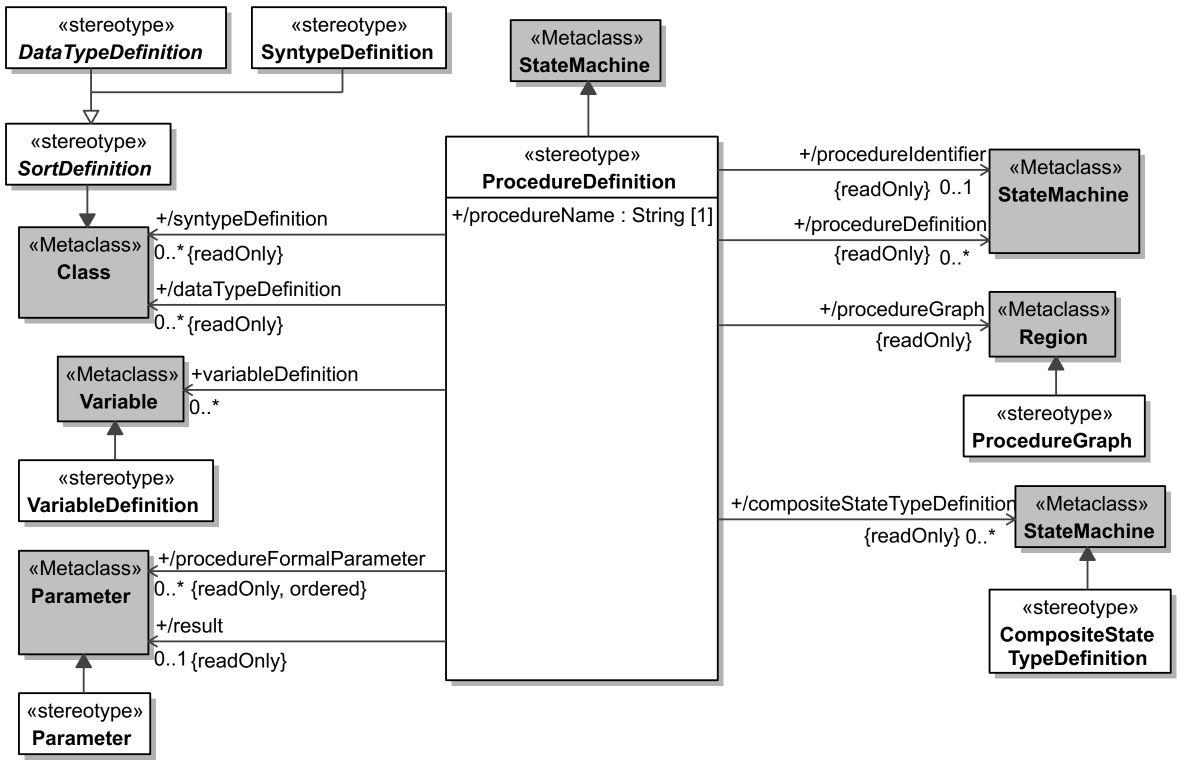
2. OCL expression of the syntypeDefinition attribute:

```
self.base_StateMachine.nestedClassifier->select(
  isStereotypedBy(`SDLUML::SyntypeDefinition')
).oclAsType(UML::Class)->asSet()
```

The first OCL expression is introduced for the dataTypeDefinition attribute and the second for the syntypeDefinition attribute of the «ProcedureDefinition» stereotype. Both OCL expressions compute the value of the concerned stereotype attribute, based on the UML metaclass attribute nestedClassifier of a UML StateMachine that is extended by the «ProcedureDefinition» stereotype. We determine this UML metaclass attribute based on the MC_{AC} attribute nestedClassifier subsetted in MM_{SDL}. Then, we create the following partial expression:



(a) The MC_{St} metaclass ProcedureDefinition in MM_{SDL}



(b) The «ProcedureDefinition» stereotype of UP_{SDL}

Figure 5.17: The «ProcedureDefinition» stereotype and the MC_{St} metaclass from which it is derived

```
self.base_StateMachine.nestedClassifier
```

This expression fragment serves us in both OCL expressions to navigate from the stereotype instance to the extended UML StateMachine and then to its nestedClassifier attribute. Because this attribute is a multi-valued attribute, we use the predefined select operator to identify the items required for the value computation.

We select the items in question according to their applied stereotype by employing the isStereotypedBy(S) operation. When calling this operation, we pass the fully qualified stereotype name as argument for parameter S. We determine this name from the MC_{St} metaclasses used to derive the *Stereotype* S. For example, we use the qualified stereotype name SDLUML::DataTypeDefinition to select all UML elements with applied «DataTypeDefinition» stereotype in OCL expres-linebreak sion (1).

In the last part of each created OCL expression, we perform a type-cast to adapt the selected UML elements to the required type and the collection kind of a multi-valued stereotype attribute. For this purpose, we first call the predefined operation oclAsType() and then a suitable collection operator, such as asSet(), to adjust the collection kind.

Because we successfully verified the derivation of stereotype attributes from redefining MC_{St} attributes in our TDL case study and could now demonstrate this also for subsetting MC_{St} attributes, we consider Design Decision 15 as successfully verified for all aspects.

Introduction of additional OCL constraints: Just as for 'derived' and 'read-only' stereotype attributes, which we derive from redefining MC_{St} attributes, we must also introduce additional OCL *Constraints* for those stereotype attributes that are derived from subsetting MC_{St} attributes according to Design Decision 20. These OCL *Constraints* ensure that the UML metaclass attributes, which are used as the computational basis in OCL expressions of 'derived' and 'read-only' stereotype attributes, only refer to UML elements with valid stereotypes. In our TDL case study, we examined the generation of OCL *Constraints* based on redefined MC_{AC} attributes. Below, we verify the introduction of such OCL *Constraints* based on subsetted MC_{AC} attributes.

We evaluate the introduction of OCL constraints based on the MC_{AC} attribute nestedClassifier shown in Fig. 5.17a, which is subsetted by the two MC_{St} attributes syntypeDefinition and dataTypeDefinition. Our derivation approach substitutes subsetting relationships between MC_{St} and MC_{AC} attributes with the previously discussed OCL expressions for 'derived' and 'read-only' stereotype attributes. These OCL expressions use the UML metaclass attribute determined for a subsetted MC_{AC} attribute as computational base. In our given example, this is the nestedClassifier attribute of the UML metaclass StateMachine.

According to the rules of Design Decision 20 we have to ensure that only UML elements with certain *Stereotypes* can be assigned to such a UML metaclass attribute. We determine the required *Stereotypes* based on the MC_{St} metaclasses referenced by the *type* properties of MC_{St} attributes. In case of the MC_{St} attributes syntypeDefinition and dataTypeDefinition, these are the two MC_{St} metaclasses SyntypeDefinition and DataTypeDefinition. Consequently, only elements with applied «SyntypeDefinition» or «DataTypeDefinition» stereotype may be assigned to the UML metaclass attribute nestedClassifier. To ensure this condition, we create the following OCL *Constraint* in the context of the «ProcedureDefinition» stereotype:

```
self.base_StateMachine.nestedClassifier−>notEmpty() implies
  self.base_StateMachine.nestedClassifier−>forAll(
    isStereotypedBy(`SDLUML::SyntypeDefinition')
    or isStereotypedBy(`SDLUML::DataTypeDefinition')
  )
```

In the first line of the above OCL *Constraint* we navigate from the «Procedure-Definition» stereotype instance to the StateMachine element and then to the nestedClassifier attribute. Because this is a multi-valued attribute, we first use the predefined collection operator notEmpty() to check whether this attribute has assigned elements. If so, we check in Lines 2 to 4 whether all elements assigned to the nestedClassifier attribute have the mentioned *Stereotypes* applied. Hence, we consider Design Decision 15 as successfully verified in all aspects.

Comparison of the standardized with the derived UML profile. After examining the still open design decisions for deriving UML profiles on the UP_{SDL} example, we compare this UML profile with the one standardized in ITU-T Rec. Z.109 of 2013 [69]. To refer to the latter UML profile, we use the acronym UP_{std}.

As in our TDL case study, we intend to answer the question whether a UML profile generated via our derivation approach is comparable to a manually created one. To gain initial insights, we first compare the model elements contained in UP_{std} and UP_{SDL}. Afterwards, we consider a stereotype contained in both UML profiles, so that we obtain a detailed conclusion on the quality of the UML profile UP_{SDL} derived by us.

In our TDL case study, we already proved that our derivation approach can be applied to generate a UML profile whose syntactic structure is comparable to that of a manually created one. Therefore, we now examine in particular the static semantics defined using OCL. In the case of SDL, this is a promising option because OCL constraints are available for both UP_{std} and UP_{SDL}.

Quantitative comparison: To compare the model elements of both UML profiles quantitatively, we correlate the number of their contained model elements in Table 5.11. A first essential difference between UP_{std} and UP_{SDL} is shown in the first line of the table. UP_{SDL} contains 71 *Stereotypes*, whereas UP_{std} only contains 39

Table 5.11: Comparison of the standardized UP_{std} and the derived UP_{SDL}

	UP_{std}	UP_{SDL}
Stereotypes	39	71
Stereotype attributes (total)	37	207
OCL-defined attributes	0	142
OCL *Constraints*	215	235
OCL *Operations*	33	28
Enumerations	1	1

Stereotypes. This is due to the different representation of SDL language constructs in both UML profiles. Our derivation approach creates exactly one *Stereotype* for each MC_{St} metaclass, i.e., each language construct of some DSL defined by means of a metaclass is mapped one-to-one to a corresponding stereotype. In contrast, a *Stereotype* in UP_{std} can represent not only one but several SDL language constructs. We already highlighted this issue in Chap. 3 and pointed out that such an approach implies a higher complexity for mapping rules and constraints.

The next difference between the UML profiles UP_{std} and UP_{SDL} concerns the number of stereotype attributes, because UP_{std} contains significantly less attributes than UP_{SDL}. Thus, each *Stereotype* in UP_{std} has approximately one attribute only. This makes necessary that the model elements to be mapped to syntactic constructs of SDL have to be determined based on the attributes of stereotyped UML elements.

In contrast to UP_{std}, a one-to-one relationship between the metaclass attributes in MM_{SDL} and the stereotype attributes in UP_{SDL} exists, i.e., each of the 207 stereotype attributes in UP_{SDL} has a corresponding counterpart in MM_{SDL}. Thus, when transforming a UML model with applied UP_{SDL} to a corresponding SDL model, we can determine the model elements to be mapped directly based on the existing stereotype attributes. Consequently, the mapping rules for UP_{SDL} are simpler than for UP_{std}.

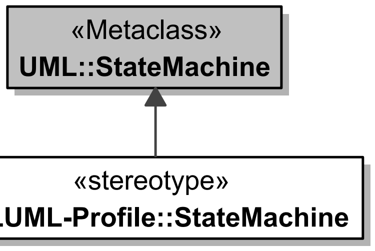
Because of the greater number of stereotype attributes present in UP_{SDL} when compared to UP_{std}, someone might argue that building a UML model with the former UML profile requires much more modelling effort than the latter. As Table 5.11 shows, 142 of the 207 stereotype attributes available in UP_{SDL} are specified as 'derived' and 'read-only' attributes with a OCL-defined *defaultValue*. Consequently, values do not have to be assigned manually to these stereotype attributes at runtime, because they are computed at runtime using OCL expressions. Thus, only 65 attributes remain for which manual value assignment is required.

In addition to the model elements for specifying the syntactic structure, both UML profiles also contain *Constraints* and *Operations* that define the static semantics. If we compare these model elements in UP_{std} and UP_{SDL}, we find that a comparable number of both model element types contained in both UML profiles. Therefore, our first conclusion is that a comparable number of well-formedness rules are defined for both UML profiles.

Detailed comparison: Because our quantitative comparison shows that the two UML profiles UP_{std} and UP_{SDL} differ regarding their stereotypes and associated attributes, we investigate this issue on the example of the SDL language concept Procedure Definition. For this purpose, we consider the specific stereotypes that implement this SDL language concept in both UML profiles. The stereotype «StateMachine» represents this language concept in the case of UP_{std}, whereas the «ProcedureDefinition» stereotype is used for this purpose in UP_{SDL}.

As mentioned earlier, UP_{std} uses some stereotypes to represent not just one but several SDL language concepts. Our «StateMachine» is exactly such a stereotype, as it can represent an SDL Procedure Definition or an SDL Composite State Type Definition. The specification attribute of a UML StateMachine extended with the «StateMachine» stereotype controls which one of the two variants applies. If this attribute refers to a UML Operation, a stereotyped UML StateMachine represents an SDL Procedure Definition. If no value is assigned to the specification attribute, the UML StateMachine in question represents an SDL Composite State Type Definition. The abstract syntax and the static semantics of the «StateMachine» stereotype are shown in Fig. 5.18.

As we can see in Fig. 5.18a, the «StateMachine» stereotype does not own an attribute. Therefore, all model elements to be mapped to SDL constructs must be determined based on the attributes of a stereotyped UML StateMachine. In contrast, the «ProcedureDefinition» stereotype of UP_{SDL} shown in Fig. 5.17b provides ten attributes, which corresponds exactly to the number of symbols of the SDL production rule Procedure-definition and, consequently, also to the number of attributes of the MC_{St} metaclass ProcedureDefinition. Thus, there is a one-to-one relation between the attributes of the stereotype and those of the corresponding metaclass in MM_{SDL}. Based on these relationships, we can simply map the «ProcedureDefinition» stereotype to the ProcedureDefinition metaclass, which is in contrast to its equivalent in UP_{std}.



(a) The stereotype's abstract syntax

Figure 5.18: The «StateMachine» stereotype and its OCL *Constraints* as defined in ITU-T Rec. Z.109 [69]

Constraint 1: The *ownedConnector* shall be empty.

```
base_StateMachine.ownedConnector−>isEmpty()
```

Constraint 2: If a «StateMachine» maps to a Composite-state-type-definition, no *ownedParameter* property shall have a *direction* = *return* (so that the «StateMachine» does not return a result).

```
self.isCompositeStateTypeDefinition() implies
  base_StateMachine.ownedParameter−>forAll(
    direction <> uml::ParameterDirectionKind::return )
```

Constraint 3: If a «StateMachine» maps to a Composite-state-type-definition, the *specification* property shall be empty.

```
self.isCompositeStateTypeDefinition() implies
  base_StateMachine.specification−>isEmpty()
```

Constraint 4: If a «StateMachine» maps to a Procedure-definition, the *connectionPoint* property shall be empty.

```
self.isProcedureDefinition() implies base_StateMachine.connectionPoint−>isEmpty()
```

Constraint 5: If a «StateMachine» maps to a Procedure-definition, the *classifierBehavior* of a «StateMachine» shall be empty.

```
self.isProcedureDefinition() implies base_StateMachine.classifierBehavior−>isEmpty()
```

Constraint 6: If a «StateMachine» maps to a Procedure-definition, the *ownedPort* shall be empty.

```
self.isProcedureDefinition() implies base_StateMachine.ownedPort−>isEmpty()
```

Constraint 7: If a «StateMachine» maps to a Procedure-definition, the *specification* shall not be an *Operation* contained in an «Interface».

```
self.isProcedureDefinition() implies
  not base_StateMachine.specification.owner.oclIsTypeOf(uml::Interface)
```

Constraint 8: If a «StateMachine» maps to a Procedure-definition, the *specification* property shall be an *Operation*.

```
self.isProcedureDefinition() implies
  base_StateMachine.specification.oclIsTypeOf(uml::Operation)
```

Constraint 9: If a «StateMachine» maps to a Procedure-definition, there shall only be one *Region*.

```
self.isProcedureDefinition() implies base_StateMachine.region−>size()=1
```

(b) The OCL *Constraints* of the «StateMachine» stereotype

Figure 5.18: The «StateMachine» stereotype and its OCL *Constraints* as defined in ITU-T Rec. Z.109 [69] (continued)

Because of the high number of attributes owned by the «ProcedureDefinition» stereotype, someone might argue that a much higher modelling effort is required when compared to the equivalent in UP_{std}. However, because the values for nine of the ten attributes of the «ProcedureDefinition» stereotype are computed at run-time using OCL expressions, values must only be assigned manually to one attribute; three of these OCL expressions are shown as examples at the bottom of Fig. 5.19. We consider the modelling effort required for both stereotypes to be similar.

After evaluating the syntactic structure of the two stereotypes in UP_{std} and UP_{SDL}, we analyse their static semantics defined by OCL. First, we consider the nine OCL *Constraints* shown in Fig. 5.18b, which define the static semantics of the «StateMachine» stereotype. Then, we compare them with the *Constraints* defined for the «ProcedureDefinition» stereotype.

Because the «StateMachine» stereotype represents either an SDL Procedure Definition or an SDL Composite State Type Definition, its OCL *Constraints* must also regard the static semantics defined for both SDL language concepts. If we consider the *Constraints* listed in Fig. 5.18b, we find that *Constraint* (2) and (3) only apply if an SDL Composite State Type Definition is represented, whereas *Constraint* (4) – (9) apply in the case of an SDL Procedure Definition. *Constraint* (1) is an exception, because its condition must always be met. In the following, we only examine the representation of an SDL Procedure Definition and, therefore, we discuss only *Constraints* (4) – (9).

To check the static semantics specified for an SDL Procedure Definition, we expect the OCL *Constraints* of the «StateMachine» stereotype in UP_{std} to capture the relevant well-formedness rules of SDL. Three **constraint** annotations are defined for the production rule Procedure-definition shown in Fig. 5.15a, which capture all relevant well-formedness rules of SDL's static semantics [72]. Exactly these rules should also be present for the «StateMachine» stereotype.

However, if we consult *Constraints* (4) – (9) in Fig. 5.18b, we cannot find any match with the mentioned SDL rules. For this reason, we conclude that SDL's well-formedness rules are not captured by the «StateMachine» stereotype. Thus, the question arises what the considered OCL *Constraints* ensure instead of SDL's well-formedness rules. A closer look at *Constraints* (4) – (9) in Fig. 5.18b shows that they only assure structural aspects.

For example, *Constraints* (4) and (5) evaluate whether no values are assigned to a particular attribute of a UML StateMachine stereotyped by «StateMachine». Moreover, *Constraint* (9) specifies that exactly one element must be assigned to the region attribute of a StateMachine. All restrictions on attribute values imposed by *Constraints* (4) to (9) in Fig. 5.18b are derived from the syntactic structure of SDL's abstract syntax and, thus, they ensure that only a modelling that conforms to this syntactic structure is possible in UML models with applied UP_{std}.

Constraint 1: In an «SdlSpecification», all potentially instantiated procedures shall have a «ProcedureStartNode».

```
not self.base_StateMachine.isAbstract implies
  self.base_StateMachine.allOwnedElements()
  −>union( self.base_StateMachine.superClass.allOwnedElements() )
  −>select( isStrictStereotypedBy('SDLUML::ProcedureStartNode') )−>notEmpty()
```

Constraint 2: The *procedureGraph* of a «ProcedureDefinition» for an operation shall not contain a «StateNode» (explicitly or implicitly).

```
self.base_StateMachine.specification <> null implies
  self.base_StateMachine.allOwnedElements()
  −>select( isStereotypedBy(`SDLUML::StateNode') )−>isEmpty()
```

Constraint 3: The *procedureDefinition* property of the «ProcedureDefinition» for an operation shall be empty.

```
self.base_StateMachine.specification <> null implies self.procedureDefinition−>isEmpty()
```

Constraint 4: The following properties of a «ProcedureDefinition» shall be empty: *connectionPoint*, *subMachine*, *classifierBehavior*, *ownedConnector* and *ownedPort*.

```
self.base_StateMachine.connectionPoint−>isEmpty()
  and self.base_StateMachine.submachineState−>isEmpty()
  and self.base_StateMachine.classifierBehavior−>isEmpty()
  and self.base_StateMachine.ownedConnector−>isEmpty()
  and self.base_StateMachine.ownedPort−>isEmpty()
```

Constraint 5: The *region* of a «ProcedureDefinition» shall contain exactly one item.

```
self.base_StateMachine.region−>notEmpty() implies self.base_StateMachine.region−>size()
  = 1
```

Constraint 6: The *nestedClassifier* property shall only contain elements stereotyped by «SyntypeDefinition», «DataTypeDefinition», «CompositeStateTypeDefinition» and «ProcedureDefinition».

```
self.base_StateMachine.nestedClassifier−>notEmpty() implies
  self.base_StateMachine.nestedClassifier−>forAll(
    isStereotypedBy(`SDLUML::SyntypeDefinition')
    or isStereotypedBy(`SDLUML::DataTypeDefinition')
    or isStereotypedBy(`SDLUML::CompositeStateTypeDefinition')
    or isStereotypedBy(`SDLUML::ProcedureDefinition'))
```

Figure 5.19: OCL *Constraints* and OCL-defined attributes of the «ProcedureDefinition» stereotype of the derived UP_{SDL}

Constraint 7: The *ownedBehavior* property shall only contain «Composite-StateTypeDefinition» and «ProcedureDefinition».

```
self.base_StateMachine.ownedBehavior−>notEmpty() implies
  self.base_StateMachine.ownedBehavior−>forAll(
    isStereotypedBy(`SDLUML::CompositeStateTypeDefinition')
    or isStereotypedBy(`SDLUML::ProcedureDefinition'))
```

Constraint 8: The *superClass* property shall only contain elements stereotyped by «ProcedureDefinition».

```
self.base_StateMachine.superClass−>notEmpty() implies
  self.base_StateMachine.superClass
  −>forAll( isStereotypedBy(`SDLUML::ProcedureDefinition'))
```

Constraint 9: The *variableDefinition* property shall only contain elements stereotyped by «VariableDefinition».

```
self.variableDefinition−>notEmpty() implies
  self.variableDefinition
  −>forAll(isStereotypedBy(`SDLUML::VariableDefinition'))
```

Derived attribute *compositeStateTypeDefinition*:

```
self.base_StateMachine.ownedBehavior−>select(
  isStereotypedBy(`SDLUML::CompositeStateTypeDefinition')
).oclAsType(UML::StateMachine)−>asSet()
```

Derived attribute *procedureDefinition*:

```
self.base_StateMachine.ownedBehavior−>select(
  isStereotypedBy(`SDLUML::ProcedureDefinition')
).oclAsType(UML::StateMachine)−>asSet()
```

Derived attribute *dataTypeDefinition*:

```
self.base_StateMachine.nestedClassifier−>select(
  isStereotypedBy(`SDLUML::DataTypeDefinition')
).oclAsType(UML::Classifier)−>asSet()
```

Figure 5.19: OCL *Constraints* and OCL-defined attributes of the «ProcedureDefinition» stereotype of the derived *UP_{SDL}* (continued)

For the same reason as for «StateMachine» in *UP_{std}*, the three well-formedness rules of the SDL Procedure Definition should also be captured by the OCL *Constraints* of the «ProcedureDefinition» stereotype in *UP_{SDL}*. If we compare the textual descriptions of the well-formedness rules in Fig. 5.15a with *Constraints* (1) – (3) listed in Fig. 5.19, we find a close match. The only difference is that names of stereotypes are used in the descriptions of the mentioned *Constraints*.

Just as with the «StateMachine» stereotype, the OCL *Constraints* (4) – (9) of the «ProcedureDefinition» in Fig. 5.19 serve to ensure the syntactic structure as specified by the MM_{SDL}. However, we must distinguish between constraints that are adopted from the MM_{SDL} and those that are introduced during the derivation of a UML profile. *Constraints* (4) and (5) belong to the first group and are subjected to our OCL update before they are transferred from MM_{SDL} to UP_{SDL}. We already use these constraints in MM_{SDL} to ensure syntactic correctness, and they serve the same purpose in UP_{SDL}.

Constraints (6) – (8) in Fig. 5.19 ensure that attributes of a UML StateMachine stereotyped with «ProcedureDefinition» can only have assigned elements with certain stereotypes. As argued earlier, the attributes restricted in this way serve us to compute the values for the nine 'derived' and 'read-only' attributes of the «ProcedureDefinition» stereotype. In contrast, *Constraint* (9) restricts the variableDefinition attribute in such a way that only elements with applied «VariableDefinition» stereotype can be referenced.

Above we considered the OCL *Constraint* of the two stereotypes «StateMachine» in UP_{std} and «ProcedureDefinition» in UP_{SDL} separately. To identify any commonalities and differences, we compare the OCL *Constraints* of both stereotypes below. An essential point is that the three well-formedness rules of SDL's static semantics [72] are only captured by the «ProcedureDefinition» stereotype in UP_{SDL}, but not by the «StateMachine» stereotype in UP_{std}. Thus, the compliance to SDL's static semantics cannot be verified for the latter.

Furthermore, both stereotypes provide OCL *Constraints* that ensure the syntactic correctness regarding the syntax of the SDL Procedure Definition. However, because both stereotypes are specified in different ways, their OCL *Constraints* for preserving syntactic aspects differ. The *Constraints* of the «StateMachine» stereotype exclusively constrain attributes of the stereotyped UML StateMachine, whereas those of the «ProcedureDefinition» stereotype also capture stereotype attributes.

When considering the constraints of both stereotypes introduced to ensure syntactic aspects, we find that *Constraint* (1) of the «StateMachine» stereotype is comparable to *Constraint* (5) of the «ProcedureDefinition» stereotype. Furthermore, all restrictions defined by *Constraints* (1), (4) – (6) of the «StateMachine» stereotype are captured by *Constraint* (4) of the «ProcedureDefinition» stereotype. In addition, the latter *Constraint* also restricts the subMachine attribute of a UML StateMachine with applied «ProcedureDefinition», while a comparable restriction is missing in the case of the «StateMachine» stereotype. Thus, the constraints of this stereotype do not prevent the creation of invalid models that involve the subMachine attribute.

All remaining *Constraints* of the «ProcedureDefinition» stereotype restrict the attribute values of this stereotype and a stereotyped UML StateMachine so that only elements with certain stereotypes can be assigned. This aspect is not covered by the OCL *Constraints* of the «StateMachine» stereotype, where any model elements can be assigned to the attributes of a UML StateMachine extended with this stereotype. Thus, the syntactically correct modelling with the «StateMachine» stereotype of the UP_{std} is not enforced.

Conclusions. In the first part of this section, we could show that the expected number of different types of model elements for the automatically derived UML profile UP_{SDL} are generated based on the metamodel MM_{SDL}. We successfully verified those design decisions for the derivation of a UML profile, which we could not completely examine in the TDL case study. Because we could successfully prove the conformity to the relevant Design Decisions 10 – 20, we also consider our derivation approach for UML profiles as successfully implemented. However, this does not yet answer the question whether the UML profiles created with this approach are comparable to those created manually. For this purpose, we compared our generated UP_{SDL} to UP_{std} that is standardized in ITU-T Rec. Z.109 [69] of 2013.

Using a quantitative comparison, we found that UP_{SDL} has almost twice as many *Stereotypes* as UP_{std}. The reason for this is that some stereotypes of UP_{std} may represent different SDL language concepts depending on their context. However, this results in higher complexity of mapping rules and OCL *Constraints* in UP_{std}. Furthermore, the comparison of both UML profiles showed that UP_{SDL} contains a significantly higher number of stereotype attributes than UP_{std}. This fact is due to the different approaches used to create the two UML profiles. A disadvantage of the UML profiles derived via our approach could be that manual value assignments for the various stereotype attributes could result in a higher modelling effort. However, this is not the case due to the high number of 'derived' and 'read-only' stereotype attributes, whose values are automatically computed at runtime using OCL expressions. Thus, only few stereotype attributes remain to which values have to be assigned manually.

Our comparison of existing OCL *Constraint* showed a comparable number in both UML profiles. Thus, our first conclusion was that a comparably high coverage of the well-formedness rules of SDL's static semantics is achieved by both UML profiles. However, a detailed comparison of the stereotypes «StateMachine» in UP_{std} and «ProcedureDefinition» in UP_{SDL} showed that our expectation is not fulfilled. Only the latter stereotype in UP_{SDL} captures the well-formedness rules of the SDL Procedure Definition. Furthermore, we found that the OCL *Constraints* of the «StateMachine» have gaps regarding syntactic aspects, which are not present in the «ProcedureDefiniton» stereotype in UP_{SDL}.

Due to the results of our quantitative and detailed comparison of the stereotypes in UP_{std} and UP_{SDL}, we could not determine a direct structural comparability of both UML profiles. This fact is due to the different specification approaches used to create both UML profiles. However, the stereotypes we generated for UP_{SDL} have a one-to-one relationship with SDL's language concepts, which is not the case for UP_{std}. We also proved that both the well-formedness rules of the static semantics of SDL and the syntactic structure are captured by OCL constraints of the stereotypes in UP_{SDL}, which is only partially the case for UP_{std}.

5.4.3 Evaluation of Derived 'Additional' Metaclasses

In addition to UML profiles, we derive 'additional' metaclasses from a metamodel MM_{Domain}, which extend UML metaclasses at MOF-level by inheritance. As argued in Chap. 1, this way of customizing the UML can be used as an alternative to the profile-based variant. Our derivation approach supports such an extension especially for adapting the UML to SDL language concepts, which cannot be represented by *Stereotypes* or only to a limited extent.

As pointed out in Chap. 3, the language concepts provided by SDL for representing expressions and values can be mapped to *Stereotypes* only with some restrictions. Therefore, these SDL language concepts are already represented in the latest version of the UML profile for SDL-2010 [70] using metaclasses. In the following, we evaluate the derivation of 'additional' metaclasses using as example the metamodel MM_{SDL}. In particular, we verify whether the derivation of 'additional' metaclasses conforms to the relevant Design Decisions 22 – 25 in Sec. 4.4.

Overview of the 'additional' metaclasses derived from MM_{SDL}. The metamodel MM_{SDL} contains seven metaclasses with applied «ToMetaclass» stereotypes, which represent MC_{AMC} metaclasses including their inheriting metaclasses, from which we derive 'additional' metaclasses according to Design Decision 11. As shown in Table 5.12, 31 MC_{AMC} metaclasses are contained in MM_{SDL}. We derive an equal number of 'additional' metaclasses for the MM_{VS} metamodel. 'VS' is employed as an acronym for 'Value Specification' to indicate that the metaclasses contained in MM_{VS} represent SDL's language concepts for specifying values and expressions. For the same reason, we use the term MC_{VS} to refer to the 'additional' metaclasses contained in MM_{VS}.

Because of Design Decision 23, we must replace all references to MM_{AC} metaclasses with corresponding to UML metaclasses when mapping MC_{AMC} to MC_{VS} metaclasses. This also applies to the inheritance relationships specified by means of *Generalizations* between seven MC_{AMC} and MC_{AC} metaclasses in MM_{SDL}. Therefore, as shown in Table 5.12, an equal number of MC_{VS} metaclasses inheriting from UML metaclasses are present in MM_{VS}. All MC_{VS} metaclasses created by us inherit directly or indirectly from the two UML metaclasses Element or ValueSpecification. As required by Design Decision 22, we import these UML metaclasses into MM_{VS} using two *ImportElement* constructs.

If we subtract the discussed seven MC_{AMC} metaclasses from the 31 existing ones in MM_{SDL}, 24 remain that inherit from other MC_{AMC} metaclasses. Because the rules of Design Decision 23 do not apply for the latter metaclasses, we map them one-to-one to 24 MC_{VS} metaclasses, which do not inherit from UML metaclasses.

Table 5.12: Comparison of the metamodels MM_{SDL} and MM_{VS}

MM_{SDL}			MM_{VS}
Metaclasses			**Metaclasses**
MC_{AMC} metaclasses (7 with applied «ToMetaclass» stereotype)	**31**	**31**	**MC_{VS} metaclasses**
MC_{AMC} metaclasses with «ToMetaclass» stereotype inheriting from MC_{AC} metaclasses	7	7	MC_{VS} inheriting from UML metaclasses
MC_{AMC} metaclasses inheriting from other MC_{AMC} metaclasses	24	24	MC_{VS} metaclasses inheriting from other MC_{VS} metaclasses
Metaclass attributes			**Metaclass attributes**
MC_{AMC} attributes	**60**	**60**	**MC_{VS} attributes**
MC_{AMC} attributes subsetting other MC_{AMC} attributes	12	12	MC_{VS} attributes subsetting other MC_{VS} attributes
MC_{AMC} attributes subsetting MC_{AC} attributes	14	14	MC_{VS} attributes subsetting UML metaclass attributes
MC_{AMC} attributes redefining MC_{AC} attributes	2	2	MC_{VS} attributes redefining UML metaclass attributes
Other MC_{AMC} attribute types	32	32	Other MC_{VS} attribute types
Enumerations	**6**	**1**	***Enumeration***
OCL *Constraints* of MC_{AMC} metaclasses	**44**	**66**	**OCL *Constraints* of MC_{VS} metaclasses**
		44	Copied and updated from MM_{SDL}
		22	Introduced for 'non-derived' stereotype attributes that refer to UML elements with applied stereotypes
OCL *Operations*	**35**	**35**	**OCL *Operations***

Metaclass attributes: 60 of the 372 metaclass attributes contained in MM_{SDL} belong to MC_{AMC} metaclasses. As far as these MC_{AMC} attributes do not redefine or subset MC_{AC} attributes, we map them one-to-one to MC_{VS} attributes in MM_{VS}. In this way we map 32 MC_{AMC} attributes, which are summarized in the category 'Other MC_{AMC} attribute types' in Table 5.12.

As all remaining MC_{AMC} attributes either redefine or subset MC_{AC} attributes, we have to consider Design Decision 23 when mapping them. Based on the MC_{AC} attributes involved, we determine their counterparts of UML metaclasses. The UML metaclass attributes determined in this way are then subsetted or redefined by the MC_{VS} attributes generated from MC_{AMC} attributes. This type of mapping affects 12 subsetting and two redefining MC_{AMC} attributes. Another 14 MC_{AMC} attributes subset other attributes of MC_{AMC} metaclasses, so we map them one-to-one to a corresponding number of MC_{VS} attributes. Thus, the created MC_{VS} attributes are not subsetting attributes of UML metaclasses but of MC_{VS} metaclasses.

Enumeration data types: In addition to metaclasses, MM_{SDL} also contains six *Enumeration* data types, which we map one-to-one to corresponding counterparts either in UP_{SDL} or in MM_{VS}. If such a data type is referenced exclusively by MC_{St} attributes, it becomes part of UP_{SDL}. However, if one of these *Enumerations* is referenced by MC_{St} and MC_{AMC} attributes, we create a corresponding counterpart as part of MM_{VS}. This is necessary because data types of a metamodel can also be referenced by stereotype attributes, but data types in UML profiles cannot be referenced by metaclass attributes. The reason is that UML profiles are not a language concept of the MOF [121].

OCL constraints and operations: MC_{AMC} metaclasses in MM_{SDL} have 44 OCL *Constraints* and 35 *Operations* defined via OCL. As for OCL artefacts of MC_{St} metaclasses, we perform an OCL update for those of MC_{AMC} metaclasses before copying them to MM_{VS}. Because we do not distinguish between MC_{St} and MC_{AMC} metaclasses for this update and we already analysed this issue in our TDL case study, we not examine this aspect once again.

Due to the transfer of OCL constraints from MC_{AMC} metaclasses to MM_{SDL}, one could assume that the metaclasses of MM_{VS} should have a total of 44 OCL *Constraints*. However, as shown in Table 5.12, 66 OCL *Constraints* exist for MC_{VS} metaclasses. This difference is caused by 22 additional OCL *Constraints* that we create to ensure the static semantics of MC_{VS} attributes according to Design Decision 25. These additional *Constraints* are required because we map MC_{AMC} attributes with a *type* property that refers to an MC_{St} metaclass to MC_{VS} attributes whose *type* properties refer to UML metaclasses (see Design Decision 24). Because we map 22 MC_{AMC} attributes in this way, we have to create an equal number of OCL *Constraints* in MM_{VS}, as required by Design Decision 25.

Summary: The quantitative comparison of model elements in MM_{SDL} and MM_{VS} showed that the number of source elements and the resulting target elements match. Only the OCL *Constraints* available in MM_{VS} are an exception, because we require additional OCL *Constraints* in MM_{VS} to ensure the static semantics. But the number of these *Constraints* also meets our expectations. Therefore, as a first conclusion we can state that the number of model elements created for MM_{VS} meets our expectation. However, this does not answer the question whether the different model element types in MM_{VS} are created according to the rules defined by the Design Decisions 22 – 25.

Verification of the elements derived for MM_{VS}. In contrast to the derivation of UML profiles from metamodels, the creation of 'additional' metaclasses is simple because of the frequent possible one-to-one mapping. Exceptions exist only for MC_{AMC} metaclasses in MM_{SDL}, which have references or dependencies to MC_{AC}

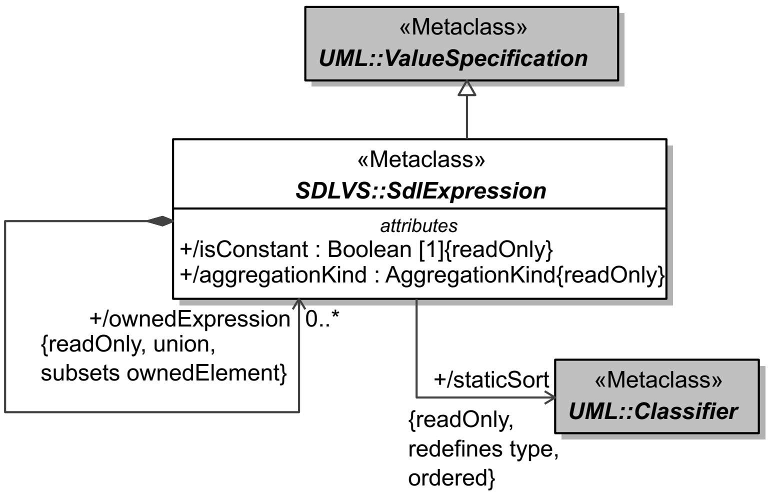


Figure 5.20: MC_{VS} SdlExpression that is derived from the metaclass in Fig. 5.16b

metaclasses. The mappings to be applied for these exceptions are defined by Design Decisions 22 – 25, which we verify below. We conduct our analysis on the MC_{AMC} metaclass SdlExpression in Fig. 5.16b and the derived counterpart in Fig. 5.20 as an example.

Metaclasses: As shown in Fig. 5.16b, the MC_{AMC} SdlExpression in MM_{SDL} inherits directly from the MC_{AC} AC_ValueSpecification. Because this involves an MC_{AC} metaclass, we must substitute this inheritance relationship when mapping the MC_{AMC} SdlExpression to an MC_{VS} metaclass according to Design Decison 23. Therefore, we determine the 'matching' UML metaclass ValueSpecification, which then serves as a new superclass for the derived MC_{VS} SdlExpression as illustrated in Fig. 5.20. To enable this kind of inheritance relationship, we import the required UML metaclass into the metamodel MM_{VS} using *ElementImport* (see Design Decision 22).

Redefining/subsetting metaclass attributes: In addition to inheritance relationships between MC_{AMC} and MC_{AC} metaclasses, we must consider possible dependencies between the attributes of these metaclasses. If MC_{AC} attributes are subsetted or redefined by MC_{AMC} attributes, we must also substitute these dependencies when creating MC_{VS} attributes, taking Design Decision 23 into account. The aim of this substitution is to redefine or subset UML metaclass attributes instead of MC_{AC} attributes.

To verify the mapping of subsetting or redefining MC_{AMC} attributes to corresponding MC_{VS} attributes in MM_{VS}, we consider the ownedExpression attribute of the MC_{AMC} SdlExpression in Fig. 5.16b This attribute subsets the ownedElement attribute of the MC_{AC} metaclass AC_ValueSpecification. After the mapping is conducted, the newly created MC_{VS} attribute ownedExpression must subset the identically-named attribute of the UML metaclass ValueSpecification instead of the mentioned MC_{AC} attribute. If we consider the MC_{VS} attribute ownedExpression in Fig. 5.20, we find that the ownedElement attribute of the UML metaclass ValueSpecification is subsetted as expected by us. For this reason, we consider Design Decision 23 as successfully verified.

Recomputation of attribute types: As highlighted earlier, when mapping MC_{AMC} attributes to MC_{VS} attributes according to Design Decision 24, we must recompute their *type* properties if MC_{St} metaclasses are referred by these properties. In this case, we determine the UML metaclass that is extended by a *Stereotype* derived from a referenced MC_{St}. Then, we use the determined UML metaclass as *type* of the generated MC_{VS} attribute.

We verify the recomputation of MC_{AMC} attributes according to Design Decision 24 on the example of the MC_{AMC} attribute staticSort, whose *type* property refers to the MC_{St} metaclass SortDefinition in MM_{SDL}, shown in Figure 5. Based on this MC_{St} metaclass, we derive the «SortDefinition» stereotype for UP_{SDL}, which extends the UML metaclass Classifier. After the MC_{AMC} attribute is mapped to the MC_{VS} attribute staticSort in MM_{VS}, the *type* property of this attribute is expected to reference the UML metaclass Classifier in accordance with Design Decision 24. If we inspect the mapped attribute in Fig. 5.20, we notice that the expected UML metaclass is referenced. Thus, we consider the mapping of MC_{AMC} attributes to be successfully verified.

Because of the recomputations of the *type* properties of MC_{VS} attributes, we introduce additional OCL *Constraints* according to Design Decison 25, which ensure the static semantics as specified by the MM_{SDL} metamodel. For all MC_{VS} attributes affected by the type recomputation, we create an OCL *Constraint* that verifies whether the UML elements assigned to an attribute have the expected *Stereotype* of UP_{SDL} applied. This *Stereotype* is determined based on the MC_{St} referenced by the *type* property of the source MC_{AMC} attribute.

Additional OCL constraints: The *type* property of our previously examined MC_{VS} attribute staticSort in Fig. 5.20 refers to the UML metaclass Classifier, whereas the MC_{St} SortDefinition is employed for the original MC_{AMC} attribute, from which we derive the «SyntypeDefinition» stereotype. To ensure the static semantics of the MC_{VS} SdlExpression, we must ensure that a UML Classifier instance assigned to the staticSort attribute has the «SyntypeDefinition» stereotype applied. For this reason, we introduce the following OCL *Constraint* for the MC_{VS} SdlExpression:

```
self.staticSort <> null implies
  self.staticSort.isStereotypedBy(`SDLUML::SortDefinition')
```

In the first line of the above *Constraint*, we evaluate whether an element is assigned to the staticSort attribute. Only if this provided, we check whether the assigned element has the «SyntypeDefinition» stereotype applied. For this purpose, we invoke the isStereotypedBy() operation in the second line of the *Constraint*. Using this example, we could demonstrate the introduction of OCL *Constraints*

Table 5.13: The standardized metaclasses that represent SDL expressions vs. the automatically derived ones in MM_{VS}

	MM_{VStd}	MM_{VS}
Metaclasses	23	25
Metaclass attributes	34	51
OCL *Constraints*	55	58
OCL *Operations*	2	35
Enumerations	1	1

that ensure the presence of matching stereotypes. Therefore, we regard Design Decision 25 to be successfully verified.

The standardized and the derived 'additional' metaclasses compared. To obtain a conclusion on the usability of the metaclasses of the metamodel MM_{VS}, which we automatically derived from the MM_{SDL}, we compare them with those specified for the UML profile UP_{std}. Below we use the abbreviation MM_{VSstd} to refer to the metamodel that contains the 'additional' metaclasses specified in ITU-T Rec. Z.109 of 2013 [69].

Quantitative comparison: The metamodel MM_{VS} contains 31 metaclasses, whereas MM_{VSstd} embraces 23. This difference is due to the fact that MM_{VS} contains 25 metaclasses for representing SDL expressions and six others for realizing SDL Connection Definitions. This SDL language concept is required in the context of SDL Aggregation States in state machines, but the UML metamodel provides no comparable counterpart. For this reason, we represent the SDL Connection Definition as a metaclass that extends the UML. To compare MM_{VSstd} and the automatically generated MM_{VS}, we only consider the metaclasses involved in the representation of SDL expressions. Therefore, Table 5.13 only contains the model elements relevant for SDL Expressions.

The first row in Table 5.13 shows that MM_{VSstd} provides 23 and MM_{VS} 25 metaclasses for capturing SDL Expressions. This difference is due to the fact that MM_{VSstd} is based on SDL-2000 [64], whereas MM_{VS} is automatically derived from the production rules of SDL-2010 [70]. The latter edition of SDL provides two additional expression types, resulting in a total of 25 language constructs related to SDL Expressions. Because the most recent version of the UML profile for SDL as defined in ITU-T Rec. Z.109 of 2013 [69] is based on SDL-2000, it does not capture the new SDL expressions introduced for SDL-2010. Therefore, it only provides 23 metaclasses related to SDL Expressions, which are contained in MM_{VSstd}.

Fig. 5.21 shows the 25 metaclasses contained in MM_{VS}, where the metaclasses EncodingExpression and DecodingExpression correspond to the newly introduced SDL Expressions. Because all other metaclasses of MM_{VS} displayed in the figure

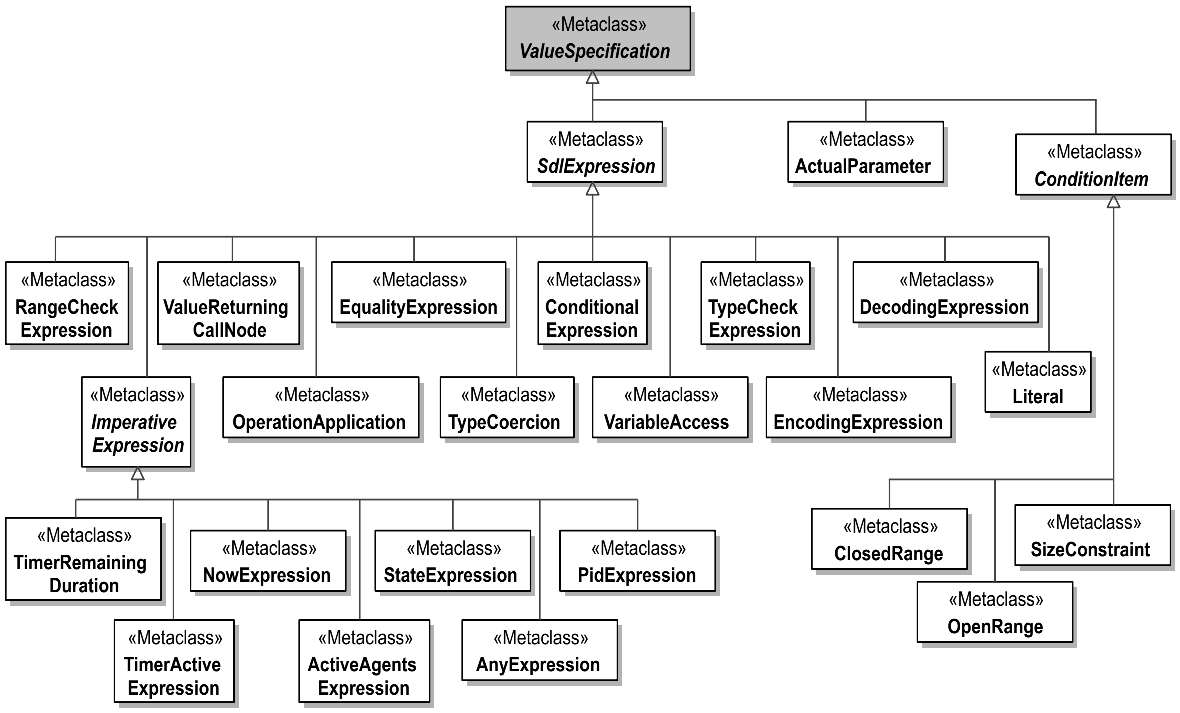

«Metaclass» ValueSpecification
«Metaclass» SdlExpression
«Metaclass» ActualParameter
«Metaclass» ConditionItem
«Metaclass» RangeCheck Expression
«Metaclass» ValueReturning CallNode
«Metaclass» EqualityExpression
«Metaclass» Conditional Expression
«Metaclass» TypeCheck Expression
«Metaclass» DecodingExpression
«Metaclass» Literal
«Metaclass» Imperative Expression
«Metaclass» OperationApplication
«Metaclass» TypeCoercion
«Metaclass» VariableAccess
«Metaclass» EncodingExpression
«Metaclass» TimerRemaining Duration
«Metaclass» NowExpression
«Metaclass» StateExpression
«Metaclass» PidExpression
«Metaclass» TimerActive Expression
«Metaclass» ActiveAgents Expression
«Metaclass» AnyExpression
«Metaclass» ClosedRange
«Metaclass» SizeConstraint
«Metaclass» OpenRange

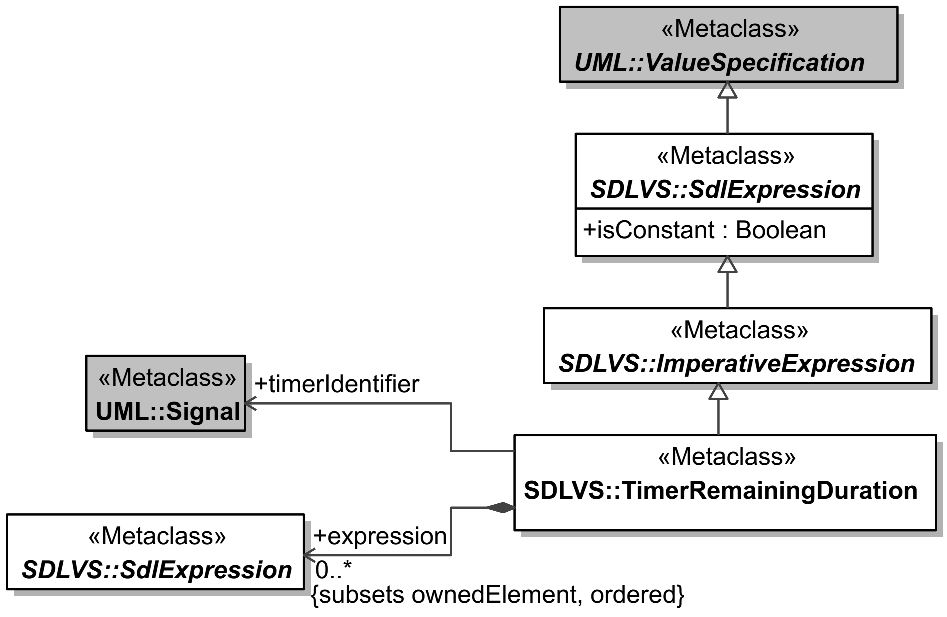
Figure 5.21: Derived metaclasses of MM_{VS} that represent SDL expressions

have corresponding counterparts in MM_{VSstd}, we do not illustrate the metaclasses of this metamodel separately.

Considering Table 5.13, we find that the number of metaclass attributes differs in MM_{VSstd} and MM_{VS}. On the one hand, this is because of the two additional metaclasses in MM_{VS}, which introduce a total of nine attributes. On the other hand, due to our one-to-one mapping of symbols in production rules to metaclass attributes, the metaclasses in MM_{VS} have a higher number of attributes than their counterparts in MM_{VSstd}. Apart from metaclasses, both metamodels MM_{VSstd} and MM_{VS} contain one *Enumeration* data type.

As Table 5.13 shows, both the metaclasses in MM_{VSstd} and those in MM_{VS} have OCL *Constraint* and *Operations* specified via OCL. Although the number of *Constraints* in both metamodels is comparable, there is a difference in the number of *Operations* defined by OCL – MM_{VS} contains a significantly higher number of *Operations*. This results from the modelling technique that we use to specify the static semantics of the source metaclasses in MM_{SDL}; because of our one-to-one derivation, this also affects the number of *Constraints* in MM_{VS}.

Detailed comparison: In the following, we use the SDL Timer Remaining Duration expression as example to analyse how metaclasses in MM_{VSstd} and MM_{VS} are defined. Furthermore, we investigate how the static semantics of this SDL expression type is represented in both metamodels. For this purpose, Fig. 5.22 shows the abstract syntax and static semantics of the TimerRemainingDuration metaclass in MM_{VSstd}, whereas Fig. 5.23 captures both aspects for MM_{VS}.



(a) The abstract syntax of the TimerRemainingDuration metaclass

Constraint 1: The *type* property shall be of predefined Duration type.

```
self.type <> null implies
  self.type.qualifiedName = `Predefined::Duration'
```

Constraint 2: The *type*, the order and the number of items in the *expression* list shall match with the *ownedAttribute* items of the associated timer.

```
let tPar:Sequence(uml::Type) = self.timerIdentifier.ownedAttribute.type,
    ePar:Sequence(uml::Type) = self.expression.type
in tPar−>size() = ePar−>size() and tPar = ePar
```

(b) The OCL *Constraints* of the TimerRemainingDuration metaclass

Figure 5.22: The TimerRemainingDuration metaclass and its *Constraints* in MM_{VSstd}, as defined in ITU-T Rec. [69]

The metaclass TimerRemainingDuration of MM_{VSstd} displayed in Fig. 5.22a indirectly inherits from SdlExpression and owns two attributes. By the timerIdentifier attribute we refer to the relevant timer whose current value is to be determined. A timer is represented as a UML Signal with applied «Timer» stereotype and can have optional parameters. However, this identifier by itself is insufficient to identify a started timer. In addition, the arguments passed during the timer instantiation must be used. Therefore, any SDL expression referring to a timer must also specify the required arguments. For this reason, the TimerRemainingDuration in MM_{VSstd} has the multi-valued expression attribute.

The two *Constraint* shown in Fig. 5.22b correspond to the well-formedness rules defined for SDL Timer Remaining Duration. *Constraint* (1) ensures that the type property of a TimerRemainingDuration always refers to the predefined value Duration. Furthermore, *Constraint* (2) specifies that a list of arguments must be provided if a referenced timer has parameters. If this is provided, the number of arguments must match that of the timer parameter, and the type of each argument must match that of the associated timer parameter.

Before we evaluate the details of the TimerRemainingDuration of the MM_{VS} metamodel, we first examine the differences of the SdlExpression metaclass of MM_{VS} shown in Fig. 5.23a and its equivalent in MM_{VSstd} shown in Fig. 5.22a. The SdlExpression metaclass of the MM_{VS} metamodel has four 'derived' and 'read-only' attributes and five *Operations* defined via OCL. In MM_{VSstd}, on the other hand, the SdlExpression metaclass only has a single attribute to which a value must be assigned manually.

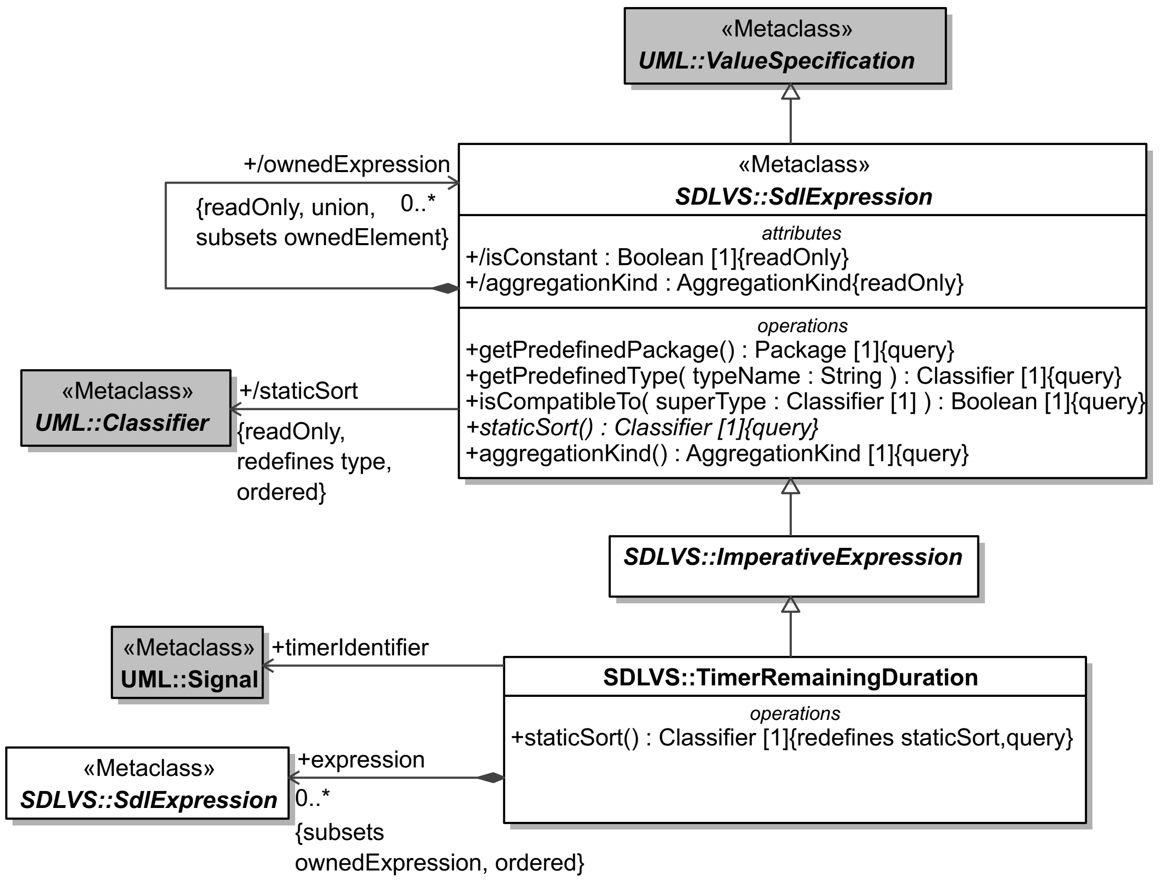
The isConstant attribute of both metaclasses specifies whether an SDL Passive Expression or an Active Expression is represented. In the case of the SdlExpression metaclass in MM_{VS}, the value of this attribute is computed automatically via an OCL expression. The values of the aggregationKind and staticSort attributes of this metaclass are also determined automatically in this way. To implement this, we invoke OCL-defined operations that are redefined in subclasses of the SdlExpression metaclass. For instance, this applies to the TimerRemainingDuration metaclass in Fig. 5.23a. Thus, we always compute the context-dependent values of both attributes in the relevant metaclass and do not specify this in the SdlExpression metaclass. This modelling approach is also the reason for the significantly higher number of *Operations* in the metamodel MM_{VS} when compared to MM_{VSstd}.

The aggregationKind attribute available for the SdlExpression metaclass in MM_{VS} is unavailable in MM_{VSstd}, because this feature has been introduced with the SDL version of 2016 [70]. Furthermore, the SdlExpression metaclass in MM_{VSstd} uses the type attribute inherited from the UML metaclass ValueSpecification, instead of the staticSort attribute to define the result type of an expression.

As the static sort must be assigned manually to the type attribute of the TimerRemainingDuration metaclass in MM_{VSstd}, *Constraint* (1) in Fig. 5.22b constrains this attribute so that it must always refer to the predefined data type Duration. A comparable *Constraint* does not exist in the case of MM_{VS}, because the static sort is always determined context-specifically using the staticSort() operation.

In the case of MM_{VS}, *Constraint* (2) in Fig. 5.22b is captured by the two *Constraints* (1) and (2) shown in Fig. 5.23b. The condition to be ensured is divided in this way, because the number of elements of the attribute expression and the type match of the argument values can be evaluated separately.

Constraint (3) displayed in Fig. 5.23b is introduced automatically during the derivation of MM_{VS} from MM_{SDL}. This constraint ensures that the timerIdentifier attribute is syntactically correct. It constrains this attribute in such a way that a referenced UML Signal must always have the stereotype «TimerDefinition» applied. We would also expect such a constraint for the SdlExpression metaclass in MM_{VSstd}. However, as is apparent from Fig. 5.22b, no corresponding *Constraint*


«Metaclass»
UML::ValueSpecification
+/ownedExpression
{readOnly, union, subsets ownedElement}
0..*
«Metaclass»
SDLVS::SdlExpression
attributes
+/isConstant : Boolean [1]{readOnly}
+/aggregationKind : AggregationKind{readOnly}
operations
+getPredefinedPackage() : Package [1]{query}
+getPredefinedType(typeName : String) : Classifier [1]{query}
+isCompatibleTo(superType : Classifier [1]) : Boolean [1]{query}
+staticSort() : Classifier [1]{query}
+aggregationKind() : AggregationKind [1]{query}
«Metaclass»
UML::Classifier
+/staticSort
{readOnly, redefines type, ordered}
SDLVS::ImperativeExpression
«Metaclass»
UML::Signal
+timerIdentifier
SDLVS::TimerRemainingDuration
operations
+staticSort() : Classifier [1]{redefines staticSort,query}
«Metaclass»
SDLVS::SdlExpression
+expression
0..*
{subsets ownedExpression, ordered}

(a) The abstract syntax of the TimerRemainingDuration metaclass

Constraint 1: The sorts of the *expression* list shall be equal to the *sortReferenceIdentifier* list of the associated *TimerDefinition*.

```
self.expression−>notEmpty() and self.timerIdentifier <> null implies
  self.expression−>size() =
    self.timerIdentifier.extension_TimerDefinition.sortReferenceIdentifier−>size()
  and self.expression.staticSort =
    self.timerIdentifier.extension_TimerDefinition.sortReferenceIdentifier
```

Constraint 2: The *expression* list shall have the same size as the *sortReferenceIdentifier* list of the referenced *TimerDefinition*.

```
self.timerIdentifier <> null implies
  self.expression−>size() =
    self.timerIdentifier.extension_TimerDefinition.sortReferenceIdentifier−>size()
```

Constraint 3: The *timerIdentifier* property shall only refer to elements stereotyped by «TimerDefinition».

```
self.timerIdentifier <> null implies
  self.timerIdentifier.isStereotypedBy(`SDLUML::TimerDefinition')
```

(b) The OCL *Constraints* of the TimerRemainingDuration metaclass

Figure 5.23: The TimerRemainingDuration metaclass and its *Constraints* in MM_{VS}

is present. Consequently, a syntactically correct modelling cannot be ensured in the case of MM_{VSstd}.

Conclusions. In the first part of this section we showed that the expected number of 'additional' metaclasses that extend UML metaclasses by inheritance can be created based on the MM_{SDL} metamodel. Then, we successfully verified the relevant Design Decisions 22 – 13 for this derivation. Because 'additional' metaclasses are the last artefact type to be verified for our derivation approach, we can also consider Evaluation Objective 1 to be completely fulfilled.

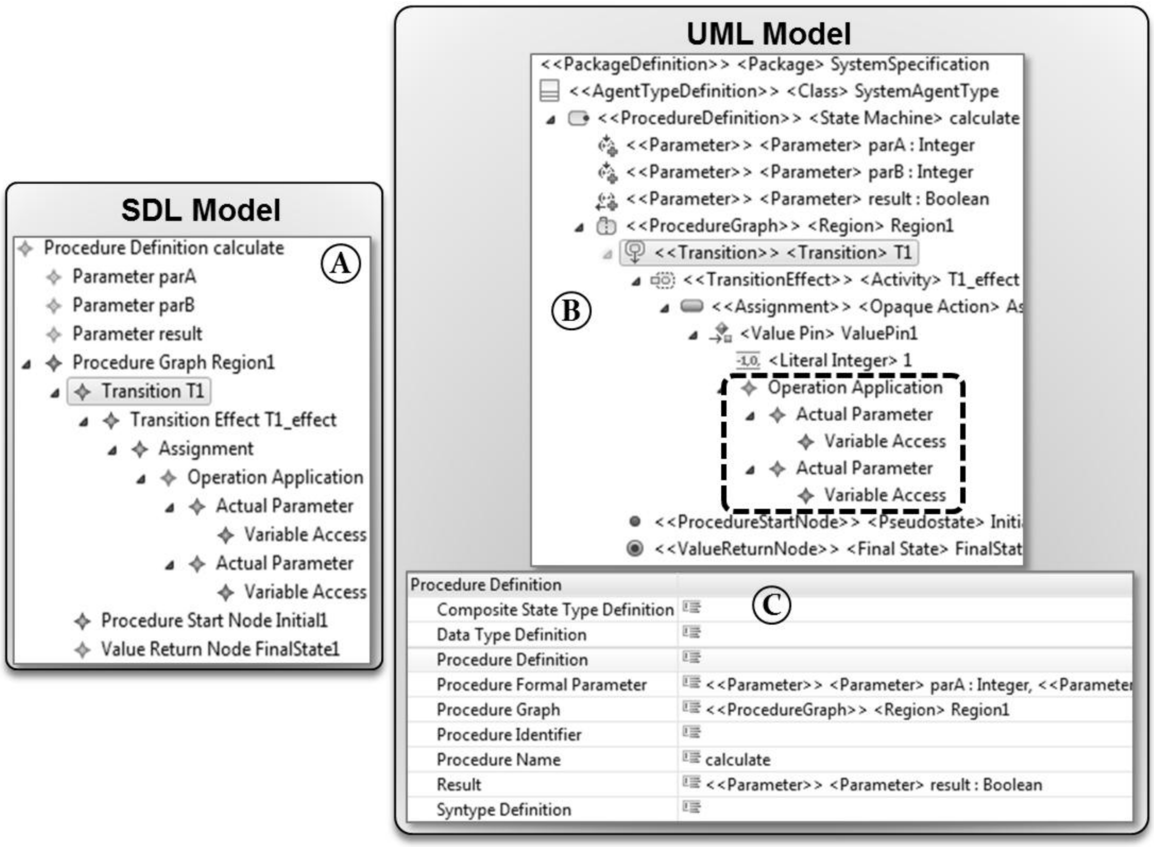
Finally, in the last part of this section, we compared the 'additional' metaclasses derived from MM_{SDL} first quantitatively and then with the metaclasses standardized in ITU-T Rec. Z.109 of 2013 [69] as an example. We were able to show that our derivation approach allows us to automatically generate an approximately comparable number of metaclasses to represent SDL expressions. It should be noted that the MM_{VS} metamodel derived by us, which comprises the 'additional' metaclasses, contains two additional metaclasses when compared to the standardized version. This is due to a change made to the SDL standard in 2016 [70], which introduces two additional types of SDL expressions.

When we assessed an exemplary metaclass, we found that both the standardized metaclass and the one we automatically derived capture the relevant well-formedness rules of SDL's static semantics [72]. In the case of the standardized metaclass, this is realized exclusively via OCL *Constraints*. In contrast, the metaclass we automatically created does not require the explicit specification of an OCL *Constraint* for one of the well-formedness rules, because the value for the attribute in question is computed at runtime based on OCL. This eliminates the demand for a *Constraint* in this case.

In addition to SDL's well-formedness rules, the metaclasses we derive also have automatically generated OCL *Constraint* that ensure various syntactic aspects. For a metaclass attribute to be constrained in this way, we determined that no corresponding OCL *Constraint* for the standardized metaclass under investigation exists. This shows another major advantage of an automated derivation approach over manual creation: the automated variant is less error-prone.

5.4.4 Evaluation and Validation of the Approach's Applicability

In the previous sections, we successfully verified the generation of the UML profile UP_{SDL} and the 'additional' metaclasses contained in the metamodel MM_{VS}. In addition, we demonstrated that the artefacts generated in this way, are to a certain extent, identical to their counterparts defined in ITU-T Rec. Z.109 [69]. However, these examinations do not allow any conclusions to be drawn regarding the usability of the generated artefacts. To address this, the artefacts mentioned must not only be instantiated, but their static semantics defined via the OCL must also be successfully validated.


SDL Model
Procedure Definition calculate
Parameter parA
Parameter parB
Parameter result
Procedure Graph Region1
Transition T1
Transition Effect T1_effect
Assignment
Operation Application
Actual Parameter
Variable Access
Actual Parameter
Variable Access
Procedure Start Node Initial1
Value Return Node FinalState1
A
UML Model
<<PackageDefinition>> <Package> SystemSpecification
<<AgentTypeDefinition>> <Class> SystemAgentType
<<ProcedureDefinition>> <State Machine> calculate
<<Parameter>> <Parameter> parA : Integer
<<Parameter>> <Parameter> parB : Integer
<<Parameter>> <Parameter> result : Boolean
<<ProcedureGraph>> <Region> Region1
<<Transition>> <Transition> T1
<<TransitionEffect>> <Activity> T1_effect
<<Assignment>> <Opaque Action> As
<Value Pin> ValuePin1
<Literal Integer> 1
Operation Application
Actual Parameter
Variable Access
Actual Parameter
Variable Access
<<ProcedureStartNode>> <Pseudostate> Initi
<<ValueReturnNode>> <Final State> FinalStat
B
C

Procedure Definition	
Composite State Type Definition	
Data Type Definition	
Procedure Definition	
Procedure Formal Parameter	<<Parameter>> <Parameter> parA : Integer, <<Parameter
Procedure Graph	<<ProcedureGraph>> <Region> Region1
Procedure Identifier	
Procedure Name	calculate
Result	<<Parameter>> <Parameter> result : Boolean
Syntype Definition	

Figure 5.24: Example of an SDL Procedure Definition represented in terms of UP_{SDL} and MM_{SDL} models

Before considering UP_{SDL} and MM_{VS}, we first validate the instantiation and static semantics of metamodel MM_{SDL}. For this purpose, we create so-called test models, which we use to validate the instantiation of the various metaclasses in MM_{SDL} and to validate the OCL-defined *Constraints*, *Operations* and metaclass attributes. We employ the same approach as we did for the validation of our manually created UML profile for SDL (see Sec. 3.5).

As in the case of TDL, we also semi-automatically generate and manually complete two model transformations for SDL in order to achieve model interoperability. Hence, we are able to reuse the test models already created for MM_{SDL} to validate UP_{SDL}. Using these transformations, we convert our test models for MM_{SDL} into corresponding UML models with applied UP_{SDL}. The test models created in this way enable us to validate the instantiations of UP_{SDL} and MM_{VS}, and to test the OCL-defined *Constraints*, *Operations* and attributes of the *Stereotypes* in UP_{SDL} and the metaclasses in MM_{VS}.

In the following, we illustrate our validation approach on the example of the SDL Procedure Definition already used in the previous sections. An exemplary instantiation of this SDL language construct is shown in Part (A) of Fig. 5.24, and the corresponding UML model is contained in Part (B) of the figure. Both models represent test models for positive tests, i.e., all elements of these models must be instantiable for both MM_{SDL} and UML, and all *Constraints* must return a positive evaluation result.

Both example models represent a simple operation to accumulate two Integer values. Thus, the procedure graph of the procedure definition consists of only one procedure start node with a transition to a terminating value return node. The effect of the transition is an assignment statement that assigns the computed value to a result variable. Apart from UML elements with applied stereotypes of UP_{SDL}, the UML model shown in Part (B) of Fig. 5.24 includes instances of 'additional' metaclasses (shown in a dashed box) of MM_{VS}.

To check the instantiation of the *Stereotypes* of UP_{SDL} and the 'additional' metaclasses in MM_{VS}, we transform the existing test models for MM_{SDL} into corresponding UML models using the transformation $T_{SDL\text{-}to\text{-}UML}$. Subsequently, the inverse transformation $T_{UML\text{-}to\text{-}SDL}$ is employed to retransform the models into SDL models, which we compare structurally with the original test models. For an automated comparison of two models, we use the tool *EMFCompare* introduced in Sec. 2.6. We consider a test case as successfully passed, if no structural difference is detected between the original test model and the SDL model obtained as the transformation result.

For instance, if we would find that there are fewer model elements in the result model when compared to the test model, this may indicate a missing *Stereotype* in the UML model. Typically, this is caused by the fact that the guard conditions of all mapping operations in transformation $T_{UML\text{-}to\text{-}SDL}$ check whether a particular *Stereotype* of UP_{SDL} is applied to a UML element used as input. If this is not the case, no corresponding counterpart for the involved UML element is created in the SDL model.

We also use the described validation approach to check the stereotype attributes defined as 'derived' and 'read-only', whose values are computed at runtime using OCL expressions. If these OCL expressions contain errors, they return no or incorrect values for stereotype attributes. As we employ the transformation $T_{UML\text{-}to\text{-}SDL}$ to map the stereotype attributes to corresponding metaclass attributes in MM_{SDL}, any incorrect values also exist in the result model and can be identified by comparison with the test model.

Nine out of ten attributes of the «ProcedureDefinition» stereotype are defined as 'derived' and 'read-only' and are computed at runtime via OCL expressions. Part (C) of Fig 5.24 shows the values of these stereotype attributes for our example. For instance, if the value of the result attribute would not be present, we could determine this by comparing the test model and the model obtained as transformation result.

When validating the structural aspects, we have not identified any errors in the automatically derived OCL expressions for stereotype attributes that we created based on subsetting or redefining MC_{St} attributes in MM_{SDL}. In contrast, we have found a few errors such as missing type-casts in the OCL expressions that we manually specified using the «ToTaggedValue» stereotype. Instead of fixing these errors in UP_{SDL}, we only modify the OCL expressions defined via the «ToTagged-Value» stereotype in MM_{SDL} and then regenerate UP_{SDL}. This ensures that our objective – the automatic generation of UML profiles – can be met.

In addition to validating the structural aspects of UP_{SDL}, we transform our SDL test models to UML models in order to validate the OCL *Constraints* in UP_{SDL} and MM_{VS}. During our validation activities for the SDL case study, we have not encountered any erroneous *Constraints*. This proves that the OCL *Constraints* that are automatically updated and adopted by MM_{SDL} can also be used in UP_{SDL} without any manual revision.

5.4.5 Summary of the Evaluation Results

We examined the applicability of the different aspects of our derivation approach using a case study for SDL, where the focus is on the semi-automated creation of an SDL metamodel based on production rules and on the automatic derivation of a UML profile and additional metaclasses. Because we have already evaluated most aspects of the derivation of UML profiles using TDL as example, our current case study only captures the remaining points. This involves the creation of 'derived' and 'read-only' stereotype attributes from subsetting or redefining metaclass attributes, where we also consider the OCL expressions and *Constraints* generated for these stereotype attributes.

The main objective of the SDL case study is to demonstrate that our derivation approach is applicable to an existing DSL with available production rules (Evaluation Objective 3). For this purpose, we semi-automatically derive a metamodel from SDL's production rules. After the initial creation of the metamodel and its manual refinement, we manually specify OCL *Constraints* that capture SDL's static semantics [72]. The resulting metamodel MM_{SDL} serves as input for deriving the UML profile UP_{SDL} and 'additional' metaclasses in MM_{VS}.

Because the creation of 'derived' and 'read-only' stereotype attributes from redefining or subsetting metaclass attributes is not captured in the TDL case study, we evaluate this aspect in more detail on the example of SDL. We have been able to successfully verify the derivation of such stereotype attributes. In addition, we have successfully verified the design decisions related to the derivation of CMOF-based metamodels and 'additional' metaclasses, which have been addressed in the TDL case study. As we have successfully verified all defined design decisions in our two case studies for SDL and TDL, we regard Evaluation Objective 1 as fulfilled.

In addition to the detailed verification, we have also analysed the usability of the artefacts generated with our derivation approach for SDL using an exemplary model. Our main goal has been to address the question postulated by Evaluation Objective 4, whether a derived UML profile and its static semantics specified via OCL can be used without manual modifications. In general, we can confirm this based on the test results obtained for UP_{SDL} and the associated 'additional' metaclasses. Only the OCL expressions for 'derived' and 'read-only' stereotype attributes in UP_{SDL}, which we have defined manually by means of «ToTagged-Value» stereotypes, were an exception. The test cases we have conducted for such stereotype attributes detected some errors due to the manual specification of OCL expressions. We have then fixed these bugs not in UP_{SDL} but in MM_{SDL}, and re-generated UP_{SDL}. This means that no manual modifications were made to the automatically derived UML profile to correct the errors. For this reason, we also consider Evaluation Objective 4 as fulfilled.

By comparing the automatically generated UML profile UP_{SDL} with the one standardized for SDL [69], we observe that the *Stereotypes* in the latter have a much smaller number of attributes. Furthermore, we notice that many *Stereotypes* of the standardized UML profile represent not just one but several SDL language concepts. Both identified differences result in more complex mapping rules and OCL *Constraints* in the standardized variant. In contrast, the UP_{SDL} we have generated has a one-to-one relationship between *Stereotypes* and the language concepts of SDL that they represent. Consequently, the complexity of mapping rules and OCL *Constraints* has decreased.

When comparing the OCL *Constraints* that capture SDL's well-formed rules in UP_{std} and UP_{SDL}, we find that some rules are not captured in UP_{std}, whereas all rules are considered in UP_{SDL}. Furthermore, we identify differences regarding OCL *Constraints* that ensure the syntactic structure: OCL *Constraints* of the standardized UML profile capture less syntactic aspects than those of the derived UP_{SDL}. This is because our derivation approach automatically introduces OCL *Constraints* in order to ensure syntactic aspects, e.g., the application of specific *Stereotypes* to attribute items. In contrast, this kind of OCL *Constraint* may be ignored when manually creating a UML profile, because such OCL *Constraint* are often the result of implicit requirements imposed by structural aspects. Hence, we consider the automated derivation of UML profiles to be less error-prone when compared to a manual creation.

5.5 Discussion of the Evaluation Results

In the present chapter we evaluated our derivation approach by means of two case studies on existing DSLs and examined the applicability of the created artefacts. We pursued five evaluation objectives as specified in Sec. 5.1. In the first case study, we used the *Test Description Language (TDL)* [39] to investigate the applicability of our derivation approach to new DSLs. In contrast, the *Specification and Description Language (SDL)* [70] was employed in the second case study to evaluate the applicability for existing DSLs with available production rules.

Evaluation Objective 1 required a verification that all types of artefacts generated by our derivation approach, such as metamodels and UML profiles, conform to the design decisions that are defined for them. In the TDL case study, we proved this for a CMOF-based metamodel that was generated semi-automatically from production rules and for a UML profile derived from it. We also successfully verified the automatic transfer of the OCL-defined static semantics from the metamodel to the generated UML profile. In addition, we demonstrated the semi-automatic derivation of model transformations for model interoperability in conformance with the specified design decisions. Moreover, in the SDL case study, we evaluated the derivation of 'additional' metaclasses and the creation of 'derived' and 'read-only' stereotype attributes from redefining or subsetting metaclass attributes. As we could successfully verify the conformance of our derivation approach to all specified design decisions by both case studies, we consider Evaluation Objective 1 to be met.

In our TDL case study, we investigated the applicability of our derivation approach to new DSLs as required by Evaluation Objective 2. To evaluate the usability of a UML profile that is derived by our approach, we transformed a TDL model to a UML model with applied UML profile for TDL and vice versa. Furthermore, we evaluated the compliance of both models to TDL's static semantics using the OCL Constraints specified in the TDL metamodel and the UML profile. As a result, we found that the static semantics can be successfully evaluated in both cases. Therefore, we consider Evaluation Objective 2 as fulfilled.

We used the SDL case study to assess the application of our derivation approach to an existing DSL. In this case study, we instantiated an SDL model and successfully transformed it to a corresponding UML model with applied UML profile for SDL, and evaluated the OCL-defined static semantics. Consequently, we consider the applicability of our derivation approach for existing DSLs as proved, and therefore, we have met Evaluation Objective 3.

The instantiation of automatically derived UML profiles using test models enabled us in both case studies not only to prove the applicability of the artefacts derived with our approach, but also to verify that no demand for a manual modification (Evaluation Objective 4) of these artefacts exists. In detail, we checked the instantiation of the *Stereotypes* contained in the UML profiles. Furthermore, we used the existing and additionally created test models to validate the OCL-defined attributes, *Operations* and *Constraints* of the *Stereotypes* of the automatically derived UML profiles. Both case studies showed that no manual revision to the automatically generated Stereotypes and their OCL Constraints are required. Therefore, our two case studies also fulfilled the objective of Evaluation Objective 4.

The required evidence for the usability of the semi-automatically generated model transformations according to Evaluation Objective 5 was provided in the TDL case study. For this purpose, we also used test models, which we successfully transformed from TDL to UML and vice versa with our derived and manually completed model transformations. Then, we compared the transformation results with the source test models to evaluate the validations performed. A comparison of the transformation results with the test models showed only the deviations that we expected in the element order of the attributes defined as Set or Bag. However, because such attribute types do not provide a specific order for their elements, we do not assess these deviations as errors. Thus, we also consider Evaluation Objective 5 as fulfilled.

Apart from the discussed evaluation objectives, our two case studies were intended to answer the question to what extent the UML profiles that we automatically derived are comparable to manually created ones, and whether these can be replaced with the generated ones. Therefore, the existence of a UML profile for the DSLs we used in both case studies was an important selection criterion, because this was the only way we could compare the UML profiles we automatically derived with existing ones. Thus, we obtained a statement as to whether manually and automatically generated UML profiles are comparable from a syntactic and semantic point of view.

In both case studies, we compared the standardized UML profiles with our automatically derived ones quantitatively and qualitatively in order to draw the following conclusions about the syntactic structure and static semantics.

In the case of TDL, the comparison showed that some TDL language concepts in the standardized UML profile are not represented as *Stereotypes* but as UML elements, for which mapping rules but no well-formedness rules defined via OCL exist. Concerning the standardized UML profile, we also noticed that the contained *Stereotypes* have fewer attributes than the respective TDL metaclasses. Furthermore, the cardinalities of some stereotype attributes did not match those of the corresponding TDL metaclass attributes.

When compared to the standardized UML profile for TDL, the one we automatically derived showed a certain correlation regarding the *Stereotypes* and extended UML metaclasses. Due to the one-to-one derivation of *Stereotypes* from TDL metaclasses, the highlighted syntactic drawbacks of the standardized UML profile are not present in the automatically derived one. Because of our automatic transfer of OCL *Constraints*, the derived UML profile enables an evaluation of the static semantics of TDL, which is infeasible with the standardized UML profile due to the absence of OCL *Constraints*.

The comparison of the standardized UML profile for SDL with the one we automatically derived indicated that the latter contains almost twice as many *Stereotypes*. This is because some *Stereotypes* of the standardized UML profile represent different SDL language concepts depending on the context of use. This causes a higher complexity in mapping rules and OCL *Constraints*. We also found that the UML profile we derived contains a higher number of stereotype attributes than the standardized one. As in the case of TDL, this is due to the one-to-one derivation of stereotypes from SDL metaclasses. Only a few of these stereotype attributes require a manual value assignment at runtime, because the remaining ones are defined as 'derived' and 'read-only', so their values are automatically computed at runtime via OCL expressions.

Furthermore, the comparison of the standardized UML profile for SDL with the one that we generated showed that a comparable number of OCL *Constraints* exists in both. However, a detailed analysis confirmed that only our derived UML profile captured all SDL well-formedness rules in terms of OCL *Constraints*. In addition, we identified that the OCL *Constraints* contained in the standardized UML profile have gaps regarding syntactic aspects, while such gaps are not present in the UML profile derived by us.

As a conclusion for the comparison of the profiles standardized for SDL and TDL with the automatically derived ones, we can state that these UML profiles are only comparable to a limited extent from a structural point of view.

In the case of TDL, only our automatically derived UML profile offers *Stereotypes* for all TDL language concepts. In contrast, only certain language concepts are represented as *Stereotypes* of the standardized UML profile for TDL, and the remaining ones are captured by UML metaclasses.

Another situation exists for SDL. All language concepts of SDL are represented by *Stereotypes* in both UML profiles, the standardized one and the one we derived. However, only the latter offers a specific *Stereotype* per language concept, while often a single *Stereotype* captures several language concepts in the standardized UML profile, implying the drawbacks discussed earlier.

Moreover, we also found differences in the static semantics of the analysed UML profiles. The most serious drawback concerns the standardized UML profile for TDL, which does not take TDL's static semantics into account. In contrast, the UML profile we derived embraces all TDL well-formedness rules because of our automatic OCL update and transfer. In the case of SDL, both the standardized and our derived UML profile capture SDL's static semantics, where the former has the disadvantages argued earlier.

The above comparison of the UML profiles clearly shows that the UML profiles generated with our derivation approach captures all language concepts of a DSL as dedicated *Stereotypes*. This is the essential prerequisite for an automatic transfer of the OCL-defined static semantics from a metamodel to a UML profile. This enables the evaluation of the static semantics for UML models with applied UML profile. For manually created UML profiles, an OCL-defined static semantics is either not available or not all well-formedness rules are captured. Furthermore, our automatic derivation prevents manual modelling errors such as false cardinalities of stereotype attributes.

6 Conclusions and Future Work

This chapter first summarizes our results and compares them with the posed research questions, in order to assess whether our presented derivation approach meets the research objective defined for this dissertation. Thereafter, we summarize the current limitations of our derivation approach and present our conclusions and directions for future research.

6.1 Summary

In Chap. 1, we examined various standardized DSLs and noticed that, for some of them, a UML profile was defined in addition to a metamodel. Furthermore, we found that the well-formedness rules of the DSL's static semantics are rarely defined via OCL, while natural language was used in all other cases. Because the latter alternative often leads to ambiguities, the OCL-based implementation of well-formedness rules by vendors of modelling tools can be error-prone. To avoid such pitfalls, it would be more appropriate if the OCL-defined static semantics would be made available by standardization organizations.

During our work on a new version of a UML profile [69] for the *Specification and Description Language (SDL)* [70], which we presented in Chap. 3, we found various shortcomings regarding the syntax and static semantics of the UML Profile as published in 2007 [66]. For example, the syntax defined by *Stereotypes* of this UML profile is incomplete or differs from the abstract syntax [64] defined for SDL. In addition, we noticed that the well-formedness rules specified for this UML profile were incomplete and partly ambiguous due to the use of natural language. In our point of view, these drawbacks can be avoided by an automated generation. This inspired us to conduct research on an automatic derivation of UML profiles from metamodels.

Because of the discussed challenges regarding the MDE-based development of DSLs, we defined the following overall research objective:

> *"Establishing a holistic MDE-based approach for the automatic derivation of UML profiles from CMOF-based metamodels of existing or new DSLs, while considering static semantics and model interoperability."*

Metamodel derivation. To meet this research goal, we proposed in Chap. 4 a novel approach for the automatic derivation of UML profiles from CMOF-based metamodels [121]. Even if existing approaches could be used for this purpose, only our approach enables the processing of CMOF language concepts. Especially noteworthy is our support of *subsetting* and *redefining* metaclass attributes, which we map to OCL-defined 'read-only' and 'derived' stereotype attributes, where the values of such attributes are automatically computed at runtime. Furthermore, we restrict the values that are assignable to stereotype attributes by introducing additional OCL constraints, so that we can ensure the static semantics as defined by the metamodel of a DSL. As a further innovation when compared to existing works, our derivation approach enables an automatic update and transfer of the OCL-defined static semantics from a metamodel to a derived UML profile.

UML profile derivation. To create UML profiles for existing DSLs that are based on grammar-ware, our derivation approach supports the semi-automatic derivation of metamodels from production rules. In contrast to existing approaches (e.g., [35, 67, 59]), the annotation we propose for production rules enables the generation of CMOF-based metamodels. We provide this feature because CMOF's language concepts are especially valuable when creating large metamodels, as highlighted in [8]. According to [134], this is due to the fact that the language concepts of CMOF enable a higher degree of modularity, abstraction and reuse of existing metamodels.

Generation of model transformations. Furthermore, our approach facilitates the semi-automatic derivation of transformations for obtaining model interoperability from a single metamodel. On the one hand, this enables the exchange of created models between DSL-specific model editors and UML modelling tools. On the other hand, we can employ the generated model transformations to validate and verify a derived UML profile.

Toolchain. All aspects of our holistic derivation approach are implemented by a toolchain named *DSL Metamodelling and Derivation Toolchain (DSL-MeDeTo)*. Both case studies and the toolchain are available as open source software and can be downloaded from our homepage [156].

Case studies. In Chap. 5, we successfully demonstrated the applicability of all aspects of our derivation approach on the *Specification and Description Language (SDL)* [70] and the *Test Description Language (TDL)* [39]. In addition to verifying the design decisions we defined, we compared the UML profiles generated by us to the respective standardized UML profiles in both case studies.

In the case of standardized UML profiles, we noticed that not every language concept of a DSL was captured by a corresponding *Stereotype*. In addition, we observed that the standardized UML profiles define incorrect mappings of stereotype attributes to corresponding metaclass attributes or expressions of production rules, whereas such errors are not present in our automatically derived UML profiles.

Regarding the static semantics, we found that the standardized UML profile for TDL did not contain any OCL *Constraints*, whereas constraints in natural language were defined for the standardized UML profile for SDL, but some well-formedness rules of SDL's static semantics were not captured. The two case studies showed that the OCL-defined static semantics of a metamodel can be automatically updated and transferred to a UML profile. Because of our automated approach, we avoid the issues identified for the standardized UML profiles right from the start.

6.2 Results

In Chap. 1, we have postulated four research questions that we employ below to assess whether the derivation approach presented by us captures all aspects of the defined goal.

A prerequisite for the automatic generation of a UML profile using our approach is that a CMOF-based metamodel is available for the DSL of interest. In addition to creating a metamodel from scratch, it can also be created automatically if production rules are available. Although EMOF-based metamodels can be generated with existing tools (e.g., [158, 159, 157, 155]), this is impossible for those metamodels that are based on CMOF. Therefore, desired CMOF language concepts (e.g., attribute redefinition) must be manually applied to generated metamodels. To remedy this drawback, we have postulated the following research question:

> ***RQ 1:*** *"Is it possible to support CMOF language concepts when generating metamodels from grammar production rules?"*

In Sec. 4.2 we presented our approach for deriving CMOF-based metamodels from production rules. We noticed that EBNF-based grammars only provide language constructs comparable to those supported by EMOF. Based on this correlation, an EBNF-defined DSL can be mapped to an EMOF-based metamodel and vice versa. However, EBNF-based grammars feature no language constructs that are comparable to those of the CMOF. To support CMOF language constructs and to relate production rules to 'Abstract Concepts', our derivation approach provides an appropriate annotation for production rules. In the first step, we derive a metamodel from production rules. Then, we process the annotations and apply the associated language concepts of the CMOF to elements of the metamodel created in the first step. As result, we automatically obtain a CMOF-based metamodel.

As highlighted in Sec. 4.2 and confirmed in our SDL case study, metamodels that are generated via our derivation approach can use the language concepts of CMOF. However, these metamodels need to be revised manually, as is the case for those created using existing approaches (see [44, 135]). For example, redefinition or subsetting relations of opposite ends of *Associations* have to be specified manually, if required. Furthermore, the static semantics defined via OCL needs to be implemented manually because, to the best of our knowledge, no approach is available that automatically creates OCL *Constraints* from natural language descriptions.

Even if the metamodels generated by our derivation approach from annotated production rules have to be revised manually, we support the automatic application of CMOF language concepts while deriving a metamodel. For this reason, we regard Research Question 1 as fulfilled.

Existing approaches [49, 50, 125, 152] for the automatic creation of UML profiles from metamodels do not support CMOF language concepts, e.g., subsetting or redefining metaclass attributes cannot be mapped to stereotype attributes in an appropriate manner. To overcome this shortcoming, we posed the following research question:

> ***RQ 2:*** *"Is the automatic derivation of a UML profile from a CMOF-based metamodel possible when taking into account the language concepts provided by the CMOF?"*

We are able to automatically generate UML profiles from CMOF-based metamodels, either derived from production rules or created manually, using our approach explained in Sec. 4.3. To enable this, the metamodels used as input for the derivation of UML profiles must be supplemented with specific meta-information. The two case studies we conducted on TDL and SDL in Chap. 5 demonstrated the successful application of our derivation approach. In this context, we evaluated and verified the mapping of metaclasses to *Stereotypes* and 'additional' metaclasses as defined by our design decisions. In both case studies, we paid particular attention to the creation of OCL-defined 'read-only' and 'derived' stereotype attributes whose values are automatically computed at runtime. These stereotype attributes substitute the redefining or subsetting metaclass attributes.

Apart from the language concepts mentioned above, CMOF offers the possibility to define 'derived' attributes, *Operations* and *Constraints* using OCL. Thus, the static semantics of a metamodel can be specified via OCL. As highlighted in Sec. 4.5, OCL constructs cannot be transferred from a metamodel to a derived UML profile without appropriate modification. Because we address this aspect with a dedicated research question, we answer Research Question 2 together with the following one:

> ***RQ 3:*** *"Can the static semantics of a DSL-specific metamodel be transferred automatically to a derived UML profile?"*

According to our approach for deriving UML profiles, the static semantics of an input metamodel must be subjected to the OCL update presented in Sec. 4.5. Afterwards, the derivation of a UML profile can take place. Because the OCL update is conducted before a UML profile is derived, we can map the OCL-defined elements of a metamodel in the same way as all other element types.

Based on the UML profiles generated in the case studies, we successfully verified that the OCL update complies with the defined design decisions. In addition, we validated the OCL constructs of both UML profiles using test models for positive and negative tests. We obtained the same test results as for the corresponding tests that we conducted for the metamodels employed for deriving the UML profiles. Thus, we proved that no manual modifications to the automatically updated and transferred OCL constructs in the generated UML profiles are required. Therefore, we consider Research Question 3 to be confirmed. Because we thus also demonstrated that we can map OCL-defined 'derived' attributes, *Operations* and *Constraints* to corresponding elements in UML profiles, we consider Research Question 2 to be fulfilled, too.

As discussed in Chap. 1, model transformations for obtaining model interoperability can be (semi-)automatically generated from metamodels using existing approaches [2, 51, 80, 101]. Because these approaches do not support CMOF-based metamodels, we have addressed the following research question:

> ***RQ 4:*** *"To what extent is it possible to automatically derive model transformations from a single CMOF-based metamodel to obtain model interoperability?"*

Our derivation approach for UML profiles supports the definition of 'alternative' OCL expressions for meta class attributes using meta-information. Instead of mapping the meta class attributes provided with this meta information to corresponding stereotype attributes in the default manner, we always map them to stereotype attributes specified as 'derived' and 'read-only'. In addition, the 'alternative' OCL expressions are used as *defaultValues* of such attributes. Thus, the attribute values can be computed at runtime. Because we cannot parse and process the 'alternative' OCL expressions when deriving model transformations, no mapping instructions are generated for the concerned metaclass attributes. Instead, we introduce so-called 'mapping proposals', i.e., incomplete assignment instructions that are marked as comments and have to be completed manually.

As argued, we can derive model transformations for model interoperability from a CMOF-based metamodel, but manual revision of the created 'mapping proposals' is necessary. Therefore, our proposed derivation approach can be considered as semi-automatic. However, in our two case studies we could demonstrate that the model transformations generated by us could be completed with reasonable effort and could be used for transforming UML models with applied UML profiles to corresponding DSL models and vice versa. Based on these results, we argue wrt. Research Question 4 that model transformations can be derived semi-automatically from CMOF-based metamodels via our derivation approach.

6.3 Limitations

Although we achieved our main goal – the fully automated derivation of UML profiles and their static semantics – using our derivation approach, some limitations regarding the derivation of CMOF-based metamodels and transformations for model interoperability exist. This is due to the fact that we cannot fully automatically generate the artefact types mentioned, but only semi-automatically. Therefore, the question arises whether a higher degree of automation can be achieved.

To derive CMOF-based metamodels, we use annotated production rules. We enable the creation of subsetting and redefining metaclass attributes from annotated expressions of production rules using two particular annotation types. If the *type* property of a created metaclass attribute refers to a metaclass, we also introduce an *Association* to the metaclass in question. However, in Sec. 4.2.5 we highlighted that the *opposite* ends of such *Associations* must also redefine or subset other attributes. As the CMOF [121] defines several well-formedness rules that have to be met for this type of subsetting or redefinition, we opted against an automatic support for this feature. Instead, we expect that this aspect is modelled manually using a UML tool, which has the advantage that the well-formedness rules can be evaluated by employing build-in features of the tool.

Supporting the redefinition and subsetting of *opposite* ends of *Associations* with our production rule annotation is conceivable, but can only be implemented with greater effort. In addition to an extension of the production rule annotations, the relevant well-formedness rules have to be implemented in our editor for production rules, and the metaclass attributes that are subject of redefinition or subsetting have to be made accessible to the editor.

We support the specification of 'alternative' OCL expressions using the «ToTaggedValue» stereotype in order to bridge syntactic or semantic gaps between a DSL and the UML. As we map the OCL expressions specified in this way to a derived UML profile without further processing, they must always be created for the context of the respective UML profile, i.e., *Extension* relationships between *Stereotypes* and the UML metaclasses must be considered. As observed in our two case studies, errors may occur due to the manual specification of the 'alternative' OCL expressions. To avoid changes to an automatically generated UML profile, we expect that incorrect OCL expressions are revised in the source metamodel and the UML profile is regenerated afterwards.

A syntactic check of the 'alternative' OCL expressions at modelling time could help one to avoid at least errors related to the static semantics of OCL. However, this would require that the *Stereotypes* and UML metaclasses captured by 'alternative' OCL expressions are already available at the time of specification, which is difficult to realize from a technical point of view. To achieve this, our approach presented would have to be modified in such a way that UML profiles would be derived 'online' during the editing of the source metamodel. However, this would result in a higher complexity of the derivation approach and a lower performance of the used modelling tool.

Another restriction, which we have consciously accepted, concerns the definition of *Stereotypes* as 'required'. A *Stereotype* marked in this way is automatically applied to all instances of the extended UML metaclass and all instances of inheriting metaclasses. Because no standardized possibility to prevent or restrict the automatic application of *Stereotypes* to instances of particular metaclasses exists, we do not support the definition of 'required' *Stereotypes*.

Our derivation approach supports the creation of stereotype attributes defined as 'derived' and 'read-only' from subsetting or redefining metaclass attributes, but this does not apply to metaclass attributes that redefine or subset multiple attributes. Because our analysis of the UML metamodel in Chap. 1 showed that subsetting and redefinition of multiple attributes is rarely used, we do not support an automatic derivation of OCL expressions that substitute such metaclass attributes in UML profiles. However, the «ToTaggedValue» stereotype can be employed to manually specify an 'alternative' OCL expression for the affected metaclass attributes, if required.

Because we generate model transformations for model interoperability based on a single metamodel, we cannot gain information about how syntactic or semantic gaps between a DSL and UML can be bridged. To remedy this issue for stereotype attributes that are derived from metaclass attributes, we can use 'alternative' OCL expressions. We first considered to employ a similar approach for deriving model transformations. However, we opted against this, because not only small code fragments but also entire mapping statements would have to be specified. Instead, we generate so-called 'mapping proposals' in terms of source code comments that have to be revised manually. In our view, fully automatic code generation for transformations is only possible by means of an alternative approach, where information is extracted from both the source and target metamodels. But also in this case, additional information is required to close syntactic and semantic gaps.

6.4 Conclusions

While only syntactic constructs for UML profiles can be automatically derived from EMOF-based metamodels using existing works [49, 50, 125, 152], the derivation approach presented by us also supports CMOF-based metamodels. Particularly noteworthy is the substitution of redefining or subsetting metaclass attributes by 'derived' and 'read-only' stereotype attributes, whose values are computed at runtime using the automatically generated OCL expressions. The automated update and transfer of OCL-defined attributes, *Operations* and *Constraints* to a UML profile is a further innovation of our approach compared to existing ones. This enables us to fully automatically transfer the static semantics of a DSL metamodel to a derived UML profile.

In our two case studies, we identified that the static semantics of standardized UML profiles is either not defined or only defined in natural language. Therefore, no possibility exists to validate the static semantics of a UML model that has one of these UML profiles applied. Exactly this problem is remedied with our derivation approach, because we enable the automatic transfer of static semantics to a UML profile. A prerequisite for this is, of course, that the static semantics of a metamodel used as input for deriving a UML profile is specified via OCL. In addition, our two case studies showed that the automatic generation of UML profiles can prevent errors (e.g., missing stereotypes or wrong attribute cardinalities) that occur when manually creating UML profiles. Based on the achieved results, we consider the use of our derivation approach for the fully automated generation of UML profiles to be advantageous over to a manual creation.

As we argued in the earlier section, we could successfully answer all research questions defined for this dissertation. In particular, we addressed the fully automatic derivation of a UML profile from a CMOF-based metamodel, while taking the OCL-defined static semantics into account. For this reason, we consider the research objective of this dissertation as accomplished.

6.5 Directions for Future Research

Because the derivation approach presented by us closed essential shortcomings of existing approaches for the derivation of UML profiles – especially the support of CMOF language concepts and the automatic transfer of static semantics, we anticipate that there is no demand for substantial future work in this area. Furthermore, we expect that most of the mentioned limitations of our approach concerning the semi-automatic derivation of metamodels can be remedied without intensive research.

In contrast, we anticipate that further research in generating model transformations for model interoperability is required. As we have already clarified, these transformations can only be generated semi-automatically via our derivation approach. However, a fully-automatic generation of these transformations would be desirable in order to avoid manual efforts for completion and to enable regeneration. One might argue that an existing approach (e.g., [41, 98]) could be used for this purpose. However, this is not straightforward because of our approach-specific mapping of subsetting and redefining metaclass attributes to 'derived' and 'read-only' stereotype attributes. For instance, when transforming a UML model into a corresponding DSL model, it must be ensured that first the derived' and 'read-only' stereotype attributes are mapped to their corresponding DSL counterparts, and thereafter the UML metaclass attributes are mapped.

A further issue to be solved is how concrete mapping statements can be generated automatically instead of the 'Mapping Proposals' currently used by us. As we employ 'Mapping Proposals' in the case of syntactic or semantic gaps between a DSL and the UML, the information required to generate such mapping statements cannot be extracted from the source metamodel alone. Additional information that can be used to generate mapping instructions is required. In this context, the question arises whether a source and target metamodel are sufficient to extract this information, and if not, how the missing data can be specified in a suitable way.

Apart from model transformations, our two case studies showed that great effort was required to manually specify positive and negative tests to validate the generated UML profiles, the CMOF-based metamodels, and the static semantics specified via OCL. We opted to apply this manual test method because existing approaches for the automated verification discussed in Sec. 2.6 have various shortcomings regarding the support of CMOF and OCL, and therefore, we found no added value in employing them. Nevertheless, in the future, we see benefits in an automated verification of all artefacts that are created in the context of an MDE-based development of DSLs, as this not only reduces effort but could also improve the quality of the created artefacts.

Bibliography

[1] Ab. Rahim, L., Whittle, J.: A survey of approaches for verifying model transformations. Software & Systems Modeling **14**(2), 1003–1028 (2015)

[2] Abouzahra, A., Bézivin, J., Del Fabro, M.D., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. In: Proceedings of Best Practices for Model Driven Software Development at OOPSLA (2005). URL http://www.softmetaware.com/oopsla2005/bezivin1.pdf

[3] Adrion, W.R., Branstad, M.A., Cherniavsky, J.C.: Validation, verification, and testing of computer software. ACM Computing Surveys (CSUR) **14**(2), 159–192 (1982)

[4] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison Wesley (1986)

[5] Akehurst, D., Howells, G., McDonald-Maier, K.: Implementing associations: UML 2.0 to Java 5. Software & Systems Modeling **6**(1), 3–35 (2007)

[6] Alanen, M., Porres, I.: A relation between context-free grammars and meta object facility metamodels. Tech. Rep. 606, Turku Centre for Computer Science, Finnland (2004). URL https://www.researchgate.net/publication/2896192_A_Relation_Between_Context-Free_Grammars_and_Meta_Object_Facility_Metamodels

[7] Alanen, M., Porres, I.: Subset and union properties in modeling languages. Tech. Rep. 731, Turku Centre for Computer Science, Finnland (2005). URL https://www.researchgate.net/publication/31596705_Subset_and_Union_Properties_in_Modeling_Languages

[8] Alanen, M., Porres, I.: A metamodeling language supporting subset and union properties. Software & Systems Modeling **7**(1), 103–124 (2008)

[9] Amelunxen, C., Schürr, A., Bichler, L.: Codegenerierung für Assoziationen in MOF 2.0. In: Proceedings of Modellierung 2004, Lecture Notes in Informatics, vol. 45, pp. 149–168. Gesellschaft für Informatik (2004)

[10] Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software & Systems Modeling **9**(1), 69 (2008)

[11] Andrews, A., France, R., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Software Testing, Verification and Reliability **13**(2), 95–127 (2003)

[12] Baier, C., Katoen, J.P., Larsen, K.G.: Principles of model checking. MIT press (2008)

[13] Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: ECMDA WS on Integration of Model Driven Development and Model Driven Testing (2006). URL https://hal.inria.fr/inria-00542781

[14] Baudry, B., Ghosh, S., Fleurey, F., France, R., Le Traon, Y., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM **53**(6), 139–143 (2010)

[15] Beckert, B., Keller, U., Schmitt, P.H.: Translating the Object Constraint Language into first-order predicate logic. In: Proceedings of the VERIFY Workshop at Federated Logic Conference (FLoC), pp. 113–123. Datalogisk Institut, University of Copenhagen (2002)

[16] Bergmayr, A., Wimmer, M.: Generating metamodels from grammars by chaining translational and by-example techniques. In: Proceedings of the 1st International Workshop on Model-driven Engineering By Example, CEUR Workshop Proceedings, vol. 1104, pp. 22–31. CEUR-WS.org (2013). URL http://ceur-ws.org/Vol-1104/

[17] Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Hardware and Software: Verification and Testing, Lecture Notes in Computer Science, vol. 10629. Springer (2017)

[18] Boehm, B.W., et al.: Software engineering economics, vol. 197. Prentice-hall Englewood Cliffs (NJ) (1981)

[19] Borger, E., Stark, R.F.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer (2003)

[20] Boulet, P., Amyot, D., Stepien, B.: Towards the generation of tests in the test description language from use case map models. In: SDL 2015: Model-Driven Engineering for Smart Cities, 17th International SDL Forum, Lecture Notes in Computer Science, vol. 9369, pp. 193–201. Springer (2015)

[21] Bourque, P., Fairley, R.E., et al.: Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0. IEEE Computer Society Press (2014)

[22] Bruck, J., Hussey, K.: Customizing UML: Which technique is right for you? (2008). White paper, Eclipse UML Project

[23] Brun, C., Pierantonio, A.: Model differences in the Eclipse modelling framework. European Journal for the Informatics Professional **9**(1), 29–34 (2008)

[24] Bu, L., et al.: Model-based construction and verification of cyber-physical systems. SIGSOFT Software Engineering Notes **43**(3), 6–10 (2018)

[25] Chang, C.L., Lee, R.C.T.: Symbolic logic and mechanical theorem proving. Academic Press (2014)

[26] Clark, T., Sammut, P., Willans, J.S.: Applied metamodelling: A foundation for language driven development (3rd edition). Computing Research Repository (CoRR) **abs/1505.00149** (2015). URL http://arxiv.org/abs/1505.00149

[27] Clarke, E.M., Grumberg, O., Peled, D.: Model checking. MIT Press (1999)

[28] Crocker, D., Overell, P.: Augmented BNF for syntax specifications: ABNF, RFC 5234. Internet Engineering Task Force (IETF) (2008)

[29] Cunha, A., Garis, A., Riesco, D.: Translating between Alloy specifications and UML class diagrams annotated with OCL. Software & Systems Modeling **14**(1), 5–25 (2015)

[30] Czarnecki, K., Helsen, S.: Classification of model transformation approaches. In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture (2003). URL http://www.softmetaware.com/oopsla2003/czarnecki.pdf

[31] Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3), 621–645 (2006)

[32] van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. SIGPLAN Notices **35**(6), 26–36 (2000). URL http://doi.acm.org/10.1145/352029.352035

[33] Di Ruscio, D., Eramo, R., Pierantonio, A.: Model transformations. In: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM 2012), Lecture Notes in Computer Science, vol. 7320, pp. 91–136. Springer (2012)

[34] D'Souza, D., Sane, A., Birchenough, A.: First-class extensibility for UML — Packaging of profiles, stereotypes, patterns, Lecture Notes in Computer Science, vol. 1723, pp. 265–277. Springer, Berlin, Heidelberg (1999)

[35] Efftinge, S., Völter, M.: oAW xText: A framework for textual DSLs. In: Modeling Symposium at Eclipse Summit, vol. 32, pp. 118–121. eclipsecon.org (2006)

[36] Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as Eclipse plug-ins. In: Proceedings of 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05), pp. 134–143. ACM (2005)

[37] Eschbach, R., Glässer, U., Gotzhein, R., Prinz, A.: On the formal semantics of SDL-2000: a compilation approach based on an abstract SDL machine, Lecture Notes in Computer Science, vol. 1912, pp. 242–265. Springer (2000)

[38] ETSI: The UML Profile for Communicating Systems, Draft. European Telecommunications Standards Institute (2004). URL https://docbox.etsi.org/mts/mts/05-CONTRIBUTIONS/2004/200410-MTS39/. Accessed: 02.01.2019

[39] ETSI: ES 203 119-1: The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics, V1.3.1. European Telecommunications Standards Institute (2016)

[40] ETSI: ES 203 119-2: The Test Description Language (TDL); Part 2: Graphical Syntax, V1.2.1. European Telecommunications Standards Institute (2016)

[41] Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In: Model Driven Engineering Languages and Systems, 11th International Conference (MoDELS), Lecture Notes in Computer Science, vol. 5301, pp. 326–340. Springer (2008)

[42] Favre, J.M.: Towards a basic theory to model model driven engineering. In: Proceedings of the 3rd Workshop in Software Model Engineering (WiSME), pp. 262–271 (2004). URL https://www.researchgate.net/publication/239016408_Towards_a_Basic_Theory_to_Model_Model_Driven_Engineering

[43] Fischer, J., Holz, E., Löwis, M.V., Prinz, A.: SDL-2000: A language with a formal semantics. In: Proceedings of the 2000 International Conference on Rigorous Object-Oriented Methods, ROOM'00, pp. 1–15. BCS Learning & Development Ltd. (2000)

[44] Fischer, J., Piefel, M., Scheidgen, M.: A metamodel for SDL-2000 in the context of metamodelling ULF. In: System Analysis and Modeling, Lecture Notes in Computer Science, vol. 3319, pp. 208–223. Springer (2004)

[45] Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Qualifying input test data for model transformations. Software & Systems Modeling **8**(2), 185–203 (2009)

[46] Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to UML profiles. The European Journal for the Informatics Professional **5**(2) (2004)

[47] Gerber, A., Raymond, K.: MOF to EMF: There and back again. In: Proceedings of 2003 OOPSLA Workshop on Eclipse Technology eXchange, Eclipse '03, pp. 60–64. ACM (2003)

[48] Giachetti, G., Albert, M., Marín, B., Pastor, O.: Linking UML and MDD through UML profiles: a practical approach based on the UML association. Journal of Universal Computer Science **16**(17), 2353–2373 (2010)

[49] Giachetti, G., Marín, B., Pastor, O.: Integration of domain-specific modelling languages and UML through UML profile extension mechanism. International Journal of Computers and Applications **6**(5), 145–174 (2009)

[50] Giachetti, G., Marín, B., Pastor, O.: Using UML as a domain-specific modeling language: A proposal for automatic generation of UML profiles. In: Advanced Information Systems Engineering (CAiSE 2009), Lecture Notes in Computer Science, vol. 5565, pp. 110–124. Springer (2009)

[51] Giachetti, G., Marín, B., Pastor, O.: Using UML profiles to interchange DSML and UML models. In: 3rd IEEE Int. Conf. on Research Challenges in Information Science, RCIS 2009, pp. 385–394. IEEE (2009)

[52] Giachetti, G., Valverde, F., Pastor, O.: Improving automatic UML2 profile generation for MDA industrial development. In: Advances in Conceptual Modeling –- Challenges and Opportunities, Lecture Notes in Computer Science, vol. 5232, pp. 113–122. Springer (2008)

[53] Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming **69**(1), 27 – 34 (2007). Special issue on Experimental Software and Toolkits

[54] Gogolla, M., Hilken, F.: Model validation and verification options in a contemporary UML and OCL analysis tool. In: Proceedings of Modellierung 2016, Lecture Notes in Informatics, vol. 254, pp. 205–220. GI (2016)

[55] Goldschmidt, T., Becker, S., Uhl, A.: Classification of concrete textual syntax mapping approaches. In: Model Driven Architecture – Foundations and Applications (ECMDA-FA), Lecture Notes in Computer Science, vol. 5095, pp. 169–184. Springer (2008)

[56] Grammes, R.: Formalisation of the UML profile for SDL – a case study. Tech. Rep. 352/06, Department of Computer Science, University of Kaiserslautern, Germany (2006). URL https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/1792

[57] Grammes, R.: Syntactic and semantic modularisation of modelling languages. Ph.D. thesis, Department of Computer Science, University of Kaiserslautern, Germany (2007). URL https://kluedo.ub.uni-kl.de/frontdoor/index/index/year/2007/docId/1897

[58] Grangel, R., Bigand, M., Bourey, J.P.: A UML profile as support for transformation of business process models at enterprise level. In: Proceedings of the 1st International Workshop on Model Driven Interoperability for Sustainable Information Systems (MDISIS'08), CEUR Workshop Proceedings, vol. 340, pp. 73–87. CEUR-WS.org (2008). URL http://ceur-ws.org/Vol-340/

[59] Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-based language engineering with EMFtext. In: Generative and Transformational Techniques in Software Engineering IV, Lecture Notes in Computer Science, vol. 7680, pp. 322–345. Springer (2013)

[60] Hilken, F., Gogolla, M., Burgueño, L., Vallecillo, A.: Testing models and model transformations using classifying terms. Software & Systems Modeling pp. 1–28 (2016)

[61] IEEE: IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990). IEEE Computer Society (1990)

[62] IEEE: IEEE Guide-Adoption of the Project Management Institute (PMI®) Standard – A Guide to the Project Management Body of Knowledge (PMBOK® Guide), 4th edition. IEEE Computer Society (2011)

[63] ISO: Information technology – Syntactic metalanguage – Extended BNF, ISO/IEC 14977. International Standardization Organisation (1996)

[64] ITU-T: Recommendation Z.100: Specification and Description Language. International Telecommunication Union (1999)

[65] ITU-T: Recommendation Z.109: Specification and Description Language – SDL combined with UML. International Telecommunication Union (1999)

[66] ITU-T: Recommendation Z.109: Specification and Description Language – SDL-2000 combined with UML. International Telecommunication Union (2007)

[67] ITU-T: Recommendation Z.111: Notations and guidelines for the definition of ITU-T languages. International Telecommunication Union (2008)

[68] ITU-T: Recommendation Z.120: Message Sequence Chart (MSC). International Telecommunication Union (2011)

[69] ITU-T: Recommendation Z.109: Specification and Description Language – Unified Modeling Language profile for SDL-2010. International Telecommunication Union (2013)

[70] ITU-T: Recommendation Z.100: Specification and Description Language – Overview of SDL-2010. International Telecommunication Union (2016)

[71] ITU-T: Recommendation Z.100: Specification and Description Language – Overview of SDL-2010, Annex F1: SDL-2010 formal definition: General overview. International Telecommunication Union (2016)

[72] ITU-T: Recommendation Z.100: Specification and Description Language – Overview of SDL-2010, Annex F2: SDL-2010 formal definition: Static semantics. International Telecommunication Union (2016)

[73] ITU-T: Recommendation Z.100: Specification and Description Language – Overview of SDL-2010, Annex F3: SDL-2010 formal definition: Dynamic semantics. International Telecommunication Union (2016)

[74] Jackson, D.: Alloy: A lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology **11**(2), 256–290 (2002)

[75] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming **72**(1), 31 – 39 (2008). Special Issue on Second issue of experimental software and toolkits (EST)

[76] Karanam, M., Gottemukkala, L.: Model transformation languages: State–of–the-art. International Journal on Computer Science and Engineering (IJCSE) **9**, 404–408 (2017). Special Issue on Second issue of experimental software and toolkits (EST)

[77] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., Völkel, S.: Design guidelines for domain specific languages. In: Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM' 09). arxiv.org (2009). URL http://arxiv.org/abs/1409.2378

[78] Kats, L.C., Visser, E.: The spoofax language workbench: Rules for declarative specification of languages and ides. SIGPLAN Not. **45**(10), 444–463 (2010)

[79] Kent, S.: Model Driven Engineering, Lecture Notes in Computer Science, vol. 2335, pp. 286–298. Springer (2002)

[80] Kern, H.: Model interoperability between meta-modeling environments by using M3-level-based bridges. Ph.D. thesis, Faculty of Mathematics and Computer Science, Leipzig University, Germany (2016). URL https://www.researchgate.net/publication/308502200_Model_Interoperability_between_Meta-Modeling_Environments_by_using_M3-Level-Based_Bridges

[81] Khan, A.H., Porres, I.: Consistency of UML class, object and statechart diagrams using ontology reasoners. Journal of Visual Languages & Computing **26**(Supplement C), 42 – 65 (2015)

[82] Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering and Methodology **14**(3), 331–380 (2005)

[83] Kobryn, C.: UML 2001: A standardization odyssey. Communications of the ACM **42**(10), 29–37 (1999)

[84] Kolovos, D.S.: Establishing correspondences between models with the Epsilon Comparison Language, Lecture Notes in Computer Science, vol. 5562, pp. 146–157. Springer (2009)

[85] Kraas, A.: Open issues of the SDL-UML profile. Tech. Rep. N-106519, Fraunhofer ESK, Munich, Germany (2009). URL http://publica.fraunhofer.de/documents/N-106519.html

[86] Kraas, A.: A model-based formalization of the textual notation for SDL-UML. In: SDL 2011: Integrating System and Software Modeling - 15th International SDL Forum, Lecture Notes in Computer Science, vol. 7083, pp. 218–232. Springer (2011)

[87] Kraas, A.: The SDL-UML profile revisited. In: System Analysis and Modeling: About Models, 6th International Workshop (SAM 2010), Lecture Notes in Computer Science, vol. 6598, pp. 108–123. Springer (2011)

[88] Kraas, A.: Realizing model simplifications with QVT operational mappings. In: Proceedings of the 14th International Workshop on OCL and Textual Modelling (OCL@ MoDELS), CEUR Workshop Proceedings, vol. 1285, pp. 53–62. CEUR-WS.org (2014). URL http://ceur-ws.org/Vol-1285/

[89] Kraas, A.: Towards an extensible modeling and validation framework for SDL-UML. In: System Analysis and Modeling: Models and Reusability, 8th International Conference (SAM 2014), Lecture Notes in Computer Science, vol. 8769, pp. 255–270. Springer (2014)

[90] Kraas, A.: Automated tooling for the evolving SDL standard: From metamodels to UML profiles. In: SDL 2017: Model-Driven Engineering for Future Internet, 18th International SDL Forum, Lecture Notes in Computer Science, vol. 10567, pp. 1–21. Springer (2017)

[91] Kraas, A.: On the automated derivation of domain-specific UML profiles. In: 13th European Conference on Modelling Foundations and Applications (ECMFA 2017), Lecture Notes in Computer Science, vol. 10376, pp. 3–19. Springer (2017)

[92] Kraas, A., Rehm, P.: Results in using the new version of the SDL-UML profile. In: Joint ITU-T and SDL Forum Society Workshop on ITU System Design Languages, Geneva, Switzerland. International Telecommunication Union (2008). URL https://www.itu.int/dms_pub/itu-t/oth/06/18/t06180000010041pdfe.pdf

[93] Kühne, T.: Matters of (meta-) modeling. Software & Systems Modeling **5**(4), 369–385 (2006)

[94] Kunert, A.: Semi-automatic generation of metamodels and models from grammars and programs. Electronic Notes in Theoretical Computer Science **211**, 111–119 (2008)

[95] Kurtev, I.: State of the art of QVT: A model transformation language standard. In: Applications of Graph Transformations with Industrial Relevance: 3rd International Symposium (AGTIVE 2007), Lecture Notes in Computer Science, vol. 5088, pp. 377–393. Springer (2009)

[96] Küster, J.M., Abd-El-Razik, M.: Validation of model transformations – first experiences using a white box approach, Lecture Notes in Computer Science, vol. 4364, pp. 193–204. Springer (2007)

[97] Lagarde, F., Espinoza, H., Terrier, F., Gérard, S.: Improving UML profile design practices by leveraging conceptual domain models. In: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ASE '07, pp. 445–448. ACM (2007)

[98] Langer, P., Wimmer, M., Kappel, G.: Model-to-model transformations by demonstration. In: Theory and Practice of Model Transformations (ICMT 2010), Lecture Notes in Computer Science, vol. 6142, pp. 153–167. Springer (2010)

[99] López-Fernández, J.J., Cuadrado, J.S., Guerra, E., de Lara, J.: Example-driven meta-model development. Software & Systems Modeling **14**(4), 1323–1347 (2015)

[100] Makedonski, P., Adamis, G., Käärik, M., Kristoffersen, F., Zeitoun, X.: Evolving the ETSI test description language. In: System Analysis and Modeling: Technology-Specific Aspects of Models, 9th International Conference (SAM 2016), Lecture Notes in Computer Science, vol. 9959, pp. 116–131. Springer (2016)

[101] Malavolta, I., Muccini, H., Sebastiani, M.: Automatically bridging UML profiles to MOF metamodels. In: 41st Euromicro Conference on Software Engineering and Advanced Applications, pp. 259–266. IEEE (2015)

[102] Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class diagrams analysis using Alloy revisited. In: Model Driven Engineering Languages and Systems: 14th International Conference (MoDELS 2011), Lecture Notes in Computer Science, vol. 6981, pp. 592–607. Springer (2011)

[103] Marroquin, A., Gonzalez, D., Maag, S.: A novel distributed testing approach based on test cases dependencies for communication protocols. In: Proceedings of the 2015 Conference on Research in Adaptive and Convergent Systems (RACS '15), pp. 497–504. ACM (2015)

[104] Mens, T., Gorp, P.V.: A taxonomy of model transformation. In: Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005), Electronic Notes in Theoretical Computer Science, vol. 152, pp. 125–142 (2006)

[105] Merilinna, J., Pärssinen, J.: Verification and validation in the context of domain-specific modelling. In: Proceedings of the 10th Workshop on Domain-Specific Modeling (DSM '10), pp. 9:1–9:6. ACM (2010)

[106] Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Computing Surveys **37**(4), 316–344 (2005)

[107] Motik, B., Patel-Schneider, P.F., Parsia, B., Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., et al.: OWL 2 web ontology language: Structural specification and functional-style syntax. W3C recommendation **27**(65), 159 (2009)

[108] Mottu, J.M., Baudry, B., Le Traon, Y.: Model transformation testing: oracle issue. In: IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW '08), pp. 105–112. IEEE (2008)

[109] Noyrit, F., Gérard, S., Selic, B.: FacadeMetamodel: Masking UML. In: Model Driven Engineering Languages and Systems, 15th International Conference (MoDELS 2012), Lecture Notes in Computer Science, vol. 7590, pp. 20–35. Springer (2012)

[110] Oberkampf, W.L., Trucano, T.G., Hirsch, C.: Verification, validation, and predictive capability in computational engineering and physics. Applied Mechanics Reviews **57**(5), 345–384 (2004)

[111] OMG: MOF Model to Text Transformation Language – Version 1.0. Object Management Group (2008)

[112] OMG: Software Systems Process Engineering Metamodel (SPEM) – Version 2.0. Object Management Group (2008)

[113] OMG: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification – Version 1.1. Object Management Group (2011)

[114] OMG: OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1. Object Management Group (2011)

[115] OMG: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1. Object Management Group (2011)

[116] OMG: OMG Unified Modeling Language (OMG UML), Version 2.5. Object Management Group (2011)

[117] OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1. Object Management Group (2011)

[118] OMG: Service oriented architecture Modeling Language (SoaML) – Version 1.0.1. Object Management Group (2012)

[119] OMG: Business Process Model and Notation (BPMN) – Version 2.0.2. Object Management Group (2013)

[120] OMG: Object Constraint Language – Version 2.4. Object Management Group (2014)

[121] OMG: OMG Meta Object Facility (MOF) Core Specification – Version 2.5. Object Management Group (2015)

[122] OMG: XML Metadata Interchange (XMI) Specification – Version 2.5.1. Object Management Group (2015)

[123] Ostrand, T.J., Balcer, M.J.: The category-partition method for specifying and generating fuctional tests. Communications of the ACM **31**(6), 676–686 (1988)

[124] Pardillo, J., Mazón, J.N., Trujillo, J.: Bridging the semantic gap in olap models: Platform-independent queries. In: Proceedings of the ACM 11th International Workshop on Data Warehousing and OLAP (DOLAP '08), pp. 89–96. ACM (2008)

[125] Pastor, O., Giachetti, G., Marín, B., Valverde, F.: Automating the interoperability of conceptual models in specific development domains. In: Domain Engineering: Product Lines, Languages, and Conceptual Models, pp. 349–373. Springer (2013)

[126] Perry, W.E.: Effective methods for software testing: Includes complete guidelines, checklists, and templates. John Wiley & Sons (2007)

[127] Petriu, D.C.: Software model-based performance analysis, pp. 139–166. John Wiley & Sons, Inc. (2013)

[128] Prinz, A., Scheidgen, M., Tveit, M.S.: A model-based standard for SDL. In: SDL 2007: Design for Dependable Systems, 13th International SDL Forum, Lecture Notes in Computer Science, vol. 4745, pp. 1–18. Springer (2007)

[129] Reed, R.: SDL-2010: Background, rationale, and survey. In: SDL 2011: Integrating System and Software Modeling, Lecture Notes in Computer Science, vol. 7083, pp. 4–25. Springer (2011)

[130] Rehm, P.: Analyse zur Modellierung mit dem SDL-UML-Profil. Diploma's thesis, Department of Electrical and Computer Engineering, Technical University of Munich, Germany (2008)

[131] Riccobene, E., Scandurra, P.: An executable semantics of the systemc uml profile. In: Proceedings of the 2nd International Conference on Abstract State Machines, Alloy, B and Z, Lecture Notes in Computer Science, vol. 5977, pp. 75–90. Springer (2010)

[132] Robinson, A.J., Voronkov, A.: Handbook of automated reasoning, vol. 1. Elsevier (2001)

[133] Sadilek, D.A., Weißleder, S.: Testing metamodels. In: Model Driven Architecture – Foundations and Applications: 4th European Conference (ECMDA-FA 2008), Lecture Notes in Computer Science, vol. 5095, pp. 294–309. Springer (2008)

[134] Scheidgen, M.: CMOF-model semantics and language mapping for MOF 2.0 implementations. In: 4th Workshop on Model-Based Development of Computer-Based Systems and 3rd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MBD-MOMPES '06), pp. 10–20. IEEE (2006)

[135] Scheidgen, M.: Description of languages based on object-oriented metamodelling. Ph.D. thesis, Math.-Natural Sci. Dept. II, Humboldt-University, Berlin, Germany (2009). URL https://edoc.hu-berlin.de/handle/18452/16565

[136] Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2), 25–31 (2006)

[137] Schmitt, M.: The development of a parser for SDL – 2000. In: Formale Beschreibungstechniken für verteilte Systeme, 10. GI/ITG-Fachgespräch, Lübeck, pp. 131–142. Shaker Verlag (2000)

[138] Selic, B.: A systematic approach to domain-specific language design using UML. In: 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC '07), pp. 2–9. IEEE Computer Society (2007)

[139] Selim, G.M.K., Cordy, J.R., Dingel, J.: Model transformation testing: The state of the art. In: Proceedings of the 1st Workshop on the Analysis of Model Transformations (AMT '12), pp. 21–26. ACM (2012)

[140] da Silva, A.R.: Model-driven engineering: A survey supported by the unified conceptual model. Computer Languages, Systems & Structures **43**(Supplement C), 139 – 155 (2015)

[141] Staab, S., et al.: Model driven engineering with ontology technologies. In: U. Aßmann, A. Bartho, C. Wende (eds.) Reasoning Web. Semantic Technologies for Software Engineering: 6th International Summer School 2010, Tutorial Lectures, pp. 62–98. Springer (2010)

[142] Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)

[143] Stephan, M., Cordy, J.R.: Application of model comparison techniques to model transformation testing. In: Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD) 2013, pp. 307–311. SciTePress (2013)

[144] Stephan, M., Cordy, J.R.: A survey of model comparison approaches and applications. In: Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2013), pp. 265–277. SciTePress (2013)

[145] Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Software: Practice and Experience **39**(15), 1253–1292 (2009)

[146] Tenbergen, B., Bohn, P., Weyer, T.: Ein strukturierter Ansatz zur Ableitung methodenspezifischer UML/SysML-Profile am Beispiel des SPES 2020 Requirements Viewpoints. In: Software Engineering 2013 - Workshopband

(inkl. Doktorandensymposium), Lecture Notes in Informatics, vol. 215, pp. 235–244. GI (2013)

[147] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. In: Model Driven Architecture - Foundations and Applications, 5th European Conference (ECMDA-FA 2009), Lecture Notes in Computer Science, vol. 5562, pp. 18–33. Springer (2009)

[148] Ulrich, A., Jell, S., Votintseva, A., Kull, A.: The ETSI Test Description Language TDL and its application. In: Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2014), pp. 601–608. SCITEPRESS-Science and Technology Publications (2014)

[149] Walderhaug, S., Stav, E., Mikalsen, M.: Experiences from model-driven development of homecare services: UML profiles and domain models, Lecture Notes in Computer Science, vol. 5421, pp. 199–212. Springer (2009)

[150] Werner, C.: UML profile for communicating systems: A new UML profile for the specification and description of internet communication and signaling protocols. Ph.D. thesis, Faculty of Mathematics and Computer Science, University of Göttingen, Germany (2007). URL https://ediss.uni-goettingen.de/handle/11858/00-1735-0000-0006-B38D-3

[151] Werner, C., Kraatz, S., Hogrefe, D.: A UML profile for communicating systems. In: System Analysis and Modeling: Language Profiles, 5th International Workshop (SAM 2006), Lecture Notes in Computer Science, vol. 4320, pp. 1–18. Springer (2006)

[152] Wimmer, M.: A semi-automatic approach for bridging DSMLs with UML. International Journal of Web Information Systems **5**(3), 372–404 (2009)

[153] Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Satellite Events at the MoDELS 2005 Conference, Lecture Notes in Computer Science, vol. 3844, pp. 159–168. Springer (2006)

[154] Wu, H.: MaxUSE: A tool for finding achievable constraints and conflicts for inconsistent UML class diagrams, Lecture Notes in Computer Science, vol. 10510, pp. 348–356. Springer (2017)

[155] EMFText concrete syntax mapper homepage. https://marketplace.eclipse.org/content/emftext (Accessed: 02.01.2019)

[156] Homepage of the SU-MoVal and DSL-MeDeTo toolchains. http://www.su-moval.org (Accessed: 02.01.2019)

[157] Spoofax language workbench homepage. https://www.metaborg.org/ (Accessed: 02.01.2019)

[158] Textual Editing Framework (TEF) homepage. https://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html/ (Accessed: 02.01.2019)

[159] xText homepage. http://www.eclipse.org/Xtext/ (Accessed: 02.01.2019)

Index

Abstract Concepts, 118, 119
abstract grammar, 64

development process, 15, 120, 124, 142, 198
DSL and DSML, 1

Ecore, 3
ETSI, 3
evaluation objectives, 233

first-order predicate logic, 64

GPL, 1
grammarware, 1

ITU, 3

MDE, 1, 15
metaclasses
- MC_{AC}, 144
- MC_{AMC}, 144
- MC_{St}, 144
- additional, 119, 181
- matching, 119

metamodel derivation, 124
metamodelling, see MOF
- meta-metamodel, 3
- static semantics, 5

metamodels
- MM_{AC}, 118, 119, 239, 285
- MM_{Add}, 119, 181
- MM_{Domain}, 118
- MM_{SDL}, 285
- MM_{Syntax}, 133
- MM_{TDL}, 239
- MM_{UML}, 119
- MM_{VS}, 313
- MM_{std}, 239
- OCL metamodel, see OCL

model interoperability, 6
model transformations, 31
- $T_{DM\text{-}to\text{-}UML}$, 119, 197, 208
- $T_{GenDM\text{-}to\text{-}UML}$, 124, 208
- $T_{GenUML\text{-}to\text{-}DM}$, 124, 222
- $T_{MM\text{-}to\text{-}AddMC}$, 123, 181
- $T_{MM\text{-}to\text{-}Profile}$, 123, 155
- $T_{SR\text{-}to\text{-}MM}$, 123, 134
- $T_{UML\text{-}to\text{-}DM}$, 119, 197
- common concept, 202
- endogenous, 32
- exogenous, 32
- higher-order transformations, 201
- in-place, 32
- M2M, 32
- M2T, 32
- mapping proposal, 200
- MTL, see MTL
- out-place, 32
- QVT, see QVT

MOF, 17
- *Association*, 22
- *Classifier*, 19
- *Class*, 19
- *Constraint*, 28
- *DataType*, 19
- *ElementImport*, 24
- *Operation*, 22, 27
- *PackageImport*, 24
- *PackageMerge*, 24
- *Package*, 19

Property, 21, 24
CMOF, 3, 23
collection types, 21
derived union, 27
EMOF, 3, 19
hierarchy, 18
multi-valued properties, 21
property cardinality, 21
redefinition, 27
single-valued properties, 21
static semantics, see OCL
subsetting, 26
MTL, 37
comments, 38
for block, 37
if block, 38
query, 38
template, 37
template expression, 37

notational conventions, 39

OCL, 28, 190
self, 30
CallExp, 191
FeatureCallExp, 192
IfExp, 192
LiteralExp, 192
LoopExp, 192
OCLExpression, 191
TypeExp, 192
VariableDeclaration, 192
VariableExp, 192
collection operators, 30
context, 30
default value, 31
expression, 28
infix operators, 29
initial value, 31
invariant constraints, 30
metamodel, 190
post-condition, 31
pre-condition, 31
predefined operations, 29
query operation, 31
standard library, 29
type system, 28
OCL update, 190
OMG, 3

production rules
annotations, 130
Definition-rule, 131
Equivalence-rule, 131
expressions, 126
intermediate representation, 133
mapping rules, 126
non-terminal symbols, 126
terminal symbols, 126

QVT, 33
main(), 35
candidate mappings, 203
disjuncting mappings, 203
guard condition, 36
helper operation, 36
mapping operation, 35
operation body, 36
operation parameters, 35
operational language, 33
operational mappings, 33
query, 36
QVTo, 33
QVTr, 33
transformation parameters, 34
transformation signature, 34

running example, 133

SDL, 56
action language, 63
activity diagram, 70
agent type, 60
behavioural aspects, 61, 69
class diagram, 67
composite structure diagram, 69
concrete grammar, 64

data type definition, 61
disambiguation, 109
dynamic semantics, 65
evolution, 56
expressions, 64, 71
procedure definition, 61
SDL-UML, 57
specification, 59
state machine, 61
statemachine diagram, 70
static semantics, 64
structural aspects, 60, 67
type definitions, 60
UML profile, 56
Spoofax workbench, 108, 123
stereotypes
«MM2Profile» stereotype, 151
«OmitGeneralization» stereotype, 154
«ToMetaclass» stereotype, 153
«ToStereotype» stereotype, 152
«ToTaggedValue» stereotype, 154

TDL, 236
example model, 237
OCL constraints, 245
toolchain
DSL-MeDeTo, 122
SU-MoVal, 58, 107
tools
Acceleo, 124
Eclipse MDT, 122
Eclipse QVTo, 123
Papyrus, 124

UML, 40
activity diagram, 42
extension mechanisms, 44
profiles, see UML profiles
state machine diagram, 41
UML profiles, 5, 45
Extension, 46
Profile, 45
Stereotype, 45
UP_{Domain}, 119, 155
UP_{SDL}, 297
UP_{TDL}, 250
UP_{std}, 261
application, 47
MM2Profile, 123, 150
stereotype instances, 47

validation, 47
verification, 47

University
of Bamberg
Press

The model-driven engineering (MDE) of domain-specific languages (DSLs) becomes increasingly important for standardization. Both metamodels and profiles for the Unified Modeling Language (UML) are often provided for standardized DSLs. Usually, the mappings of metamodels to UML profiles are only specified in some abstract manner by such standards. Consequently, not all details required to create executable model transformations are defined. Moreover, the static semantics of metamodels and/or UML profiles are often specified only in natural language. Thus, on the one hand, ambiguities can occur, and on the other hand, the soundness of the static semantics cannot be verified and validated without a manual translation into a machine-processable language.

The main goal of this dissertation is to remedy the identified weaknesses in developing DSLs, for which a metamodel and a UML profile shall be created. To achieve this goal, we propose a holistic MDE-based approach for automatically deriving UML profiles and model transformations based on CMOF metamodels. This approach enables an automatic transfer of DSL's static semantics to UML profiles, so that the well-formedness of UML models with applied UML profile of a DSL can be verified automatically. In addition, the interoperability between UML and DSL models can already be validated in the development phase using the derived model transformations. Apart from new DSLs that are created from scratch, our approach also supports a migration of existing grammar-based DSLs towards metamodels, provided syntax rules exist.

eISBN: 978-3-86309-671-7

9 783863 096717

www.uni-bamberg.de/ubp