Auto-tooling to Bridge the Concrete and Abstract Syntax of Complex Textual Modeling Languages

Adolfo Sánchez-Barbudo Herrera

Engineering Doctorate

University of York
Computer Science

May 2017

Abstract

This thesis contributes to improving support for complex textual modeling languages. This support refers to the automatic generation of tools for the end user – e.g. parsers, editors, views, etc. – and (parts of) the standard specifications defined to describe the languages – e.g. the Object Management Group open specifications for varied modeling languages.

The particular subset of languages considered in the thesis are textual, model-based and complex. They are considered textual when their concrete syntax is textual, in particular, defined by a grammar. They are considered model-based when their abstract syntax is defined by a meta-model. They are considered complex when there is a significant gap between the concrete and abstract syntax of the language; in other words, when the abstract syntax meta-model cannot directly be derived or inferred from the concrete syntax grammar.

The contributions of this thesis address the problem of bridging the concrete and abstract syntax of complex textual modeling languages. In particular, the contributions include (a) a gap analysis of the limitations of related work; (b) a domain-specific transformation language for defining and executing concrete syntax to abstract syntax bridges; (c) an experimental evaluation of the proposed solution including studies to compare with related work.

Existing related work presents different issues when working on complex textual modeling languages. Either sufficient automatic tooling generation is not provided (Gra2Mol), or model-based languages are not appropriately supported (Spoofax), or complex gaps between the concrete and abstract syntax cannot be bridged (Xtext).

This thesis identifies the different concerns that arise when bridging the concrete and abstract syntax of complex textual modeling languages. In addition, some limitations of relevant related work are shown. With the aim of addressing these identified concerns a new approach is proposed, showing how these concerns are particularly addressed. Specifically, the proposed approach consists of complementing relevant related work (Xtext) with a novel domain-specific transformation language to declare bridges between the concrete syntax and abstract syntax of complex textual modeling languages.

The domain-specific transformation language is the main contribution of this thesis and is evaluated by means of qualitative and quantitative studies. Subject to the presented examples, the conducted experiments show that the proposed approach brings measurable benefits – in terms of size of specification artefacts and execution time of the underlying implementation – when compared to the state-of-the-art.

Contents

Abstract iii

Contents v

List of Figures xiii

List of Tables xxi

Acknowledgements xxiii

Declaration of Authorship xxv

1 Introduction 1
- 1.1 Context 1
- 1.2 Motivation 2
  - 1.2.1 Industry 2
  - 1.2.2 Academia 3
    - Problem 4
    - Current Solution 5
- 1.3 Scope 6
- 1.4 Research Methodology and Questions 7
  - 1.4.1 Research Method 7
  - 1.4.2 Research Questions 7
    - Industrial Perspective 8
    - Academic Perspective 8
- 1.5 Thesis Contributions 8
  - 1.5.1 Proposed Solution 9
  - 1.5.2 Contributions 9
    - A DSTL to Specify CS2AS Bridges 10
    - A Prototype to Execute CS2AS Model Transformations 10
- 1.6 Intended Audience 10
- 1.7 Thesis Overview & Structure 11

2 Field Survey 13
- 2.1 Model-Driven Development 13
  - 2.1.1 Introduction 13
  - 2.1.2 What is a Model? 14
  - 2.1.3 Why are Models Used? 14
  - 2.1.4 What is a Meta-Model? 16
  - 2.1.5 What is *Model-Driven**? 17
  - 2.1.6 What is a Model Transformation? 19
- 2.2 From Grammarware to Modelware 21
  - 2.2.1 Introduction 21
  - 2.2.2 What are Technological Spaces? 22

2.2.3 Why Bridge *Grammarware* and *Modelware*? 24
2.2.4 Concrete Syntax vs Abstract Syntax 25
Terminology in *Grammarware* 25
Terminology in *Modelware* 26
2.2.5 Tools to Bridge Grammarware and Modelware 27
2.3 Language Workbenches 29
2.3.1 Introduction 29
2.3.2 What are Language Workbenches? 29
2.3.3 Why Use Language Workbenches? 32
2.3.4 What is Xtext? 32
2.3.5 Why Use Xtext? 33
2.4 Related Work of Interest 34
2.4.1 Xtext 34
2.4.2 Spoofax 35
2.4.3 Gra2Mol 35
2.5 Summary 35

3 Problem Analysis **37**
3.1 Summary of Required Concepts 37
3.2 Running Example: Mini OCL 39
3.2.1 Why Mini OCL? 39
3.2.2 Mini OCL Description 39
Meta-model Import 40
Meta-model Definition 40
Constraint Definition 40
Types 41
Literal Expressions 41
Let Expressions 42
Variable Expressions 42
Call Expressions 42
Equality Expressions 43
3.3 Bridging Concrete and Abstract Syntax: Coarse Grained Analysis 43
3.3.1 1-to-1 Mappings 44
3.3.2 N-to-1 Mappings 45
3.3.3 1-to-N Mappings 47
3.3.4 N-to-M Mappings 48
3.4 Bridging Concrete and Abstract Syntax: Fine Grained Analysis 50
3.4.1 Concern 1: Mapping an LHS non-terminal to a meta-class 50
LHS non-terminal mappings that create new AS elements 50
LHS non-terminal mappings that refer to existing AS elements 50
3.4.2 Concern 2: Mapping an RHS non-terminal to a reference 51
3.4.3 Concern 3: Mapping a terminal to an attribute 51
3.4.4 Recap: Simple vs. Complex CS2AS Bridges 52
3.4.5 Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes 52
3.4.6 Concern 5: Mapping an LHS non-terminal to many meta-classes . . . 53
3.4.7 Concern 6: Multi-way mappings from LHS non-terminals to meta-classes 53
3.4.8 Concern 7: Mapping properties from non grammar terms 54
3.5 Concern 8: Name Resolution 55
3.5.1 Overview 56

3.5.2 Targets, Inputs, Providers and Consumers . . . 57
3.5.3 Lookup Scopes . . . 58
Scope propagation . . . 59
Nested scopes . . . 61
3.5.4 Name-based Lookups . . . 62
Ambiguous Names . . . 63
Nested Scopes . . . 63
3.5.5 Qualified Name-based Lookups . . . 63
3.5.6 Name-based External Lookups . . . 65
3.5.7 Additional Lookup Criteria . . . 66
3.5.8 Looking Up into External Models . . . 66
3.6 How Does Xtext Address These Concerns? . . . 67
3.6.1 Xtext Grammar Introduction . . . 67
3.6.2 Concern 1: Mapping an LHS non-terminal to a meta-class . . . 68
3.6.3 Concern 2: Mapping an RHS non-terminal to a reference . . . 68
3.6.4 Concern 3: Mapping a terminal to an attribute . . . 69
3.6.5 Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes . . . 69
3.6.6 Concern 5: Mapping an LHS non-terminal to many meta-classes . . . 69
3.6.7 Concern 6: Multi-way mappings from LHS non-terminals to meta-classes . . . 69
3.6.8 Concern 7: Mapping properties from non grammar terms . . . 69
3.6.9 Concern 8: Name Resolution . . . 69
3.7 Summary . . . 71

4 Solution Design & Implementation **73**
4.1 Solution Overview . . . 73
4.1.1 Overall Approach . . . 73
4.1.2 Approach Discussion . . . 75
Grammarware-to-*Modelware* vs CS2AS . . . 75
Reformulating the CS2AS Fine Grained Concerns . . . 75
CS2AS As a *Modelware* Activity . . . 76
Testing the Approach . . . 76
4.1.3 Related Work Discussion . . . 77
CS Meta-Model . . . 77
Grammarware-to-*Modelware* vs CS2AS . . . 78
4.2 Concrete Syntax to Abstract Syntax Transformation Language . . . 79
4.2.1 Overview . . . 80
4.2.2 Expressions Syntax . . . 81
Shadow Expressions . . . 81
Trace Expressions . . . 82
Lookup Expressions . . . 83
4.2.3 Preliminaries Syntax . . . 83
4.2.4 Helpers Section Syntax . . . 84
4.2.5 Name Resolution Syntax . . . 84
Targets . . . 86
Inputs . . . 89
Providers . . . 90
Consumers . . . 101
4.2.6 Mappings Section Syntax . . . 102
Creation Mappings . . . 103

Reference Mappings 105
Multi-way mappings 107
4.2.7 Disambiguation Section Syntax 109
4.2.8 How Does the CS2AS-TL Address the Identified Concerns? 111
Concern 1: Mapping a CS Meta-class to an AS Meta-class 111
Concern 2: Mapping a CS Reference to an AS Reference 111
Concern 3: Mapping a CS Attribute to an AS Attribute 112
Concern 4: Mapping a CS Reference to an AS Reference and Additional AS Meta-classes 112
Concern 5: Mapping a CS meta-class to Many AS Meta-classes 112
Concern 6: Multi-way Mappings from CS Meta-classes to AS Meta-classes 113
Concern 7: Mapping AS Properties from non CS Meta-model Terms . 114
Concern 8: Name Resolution 114
4.2.9 Other CS2AS scenarios of mOCL 114
Constraints 116
Let Expressions 116
Literal Expressions 117
Loop expressions 117
Implicit Collects and Collection Conversions 120
4.2.10 Sufficiency of mOCL 120
OCL Types 122
mOCL Expressions 123
4.2.11 Related Work Discussion 125
General Purpose M2M Transformations 125
DSTL Design 125
Related DSTLs 126
4.3 Complete OCL-Based Internal DSTL 126
4.3.1 Overview 126
4.3.2 Helpers 127
4.3.3 Disambiguation Rules 127
4.3.4 Name resolution 127
4.3.5 Mappings 130
4.3.6 Related Work Discussion 133
4.4 Transformation Execution: QVTd 133
4.4.1 Overview 133
4.4.2 QVTm compilation: Micro-Mappings 134
4.4.3 Related Work Discussion 136
4.5 Language Workbench Integration: Xtext 136
4.5.1 AS Outline 137
4.5.2 Name resolution based code completion 138
4.6 Limitations 138
4.6.1 CS2AS-TL 139
4.6.2 Prototype Implementation 139
4.7 Summary 141

5 Evaluation **143**
5.1 General Purpose M2M Transformation Languages 143
5.1.1 Helpers Section 143
5.1.2 Mappings Section 143
5.1.3 Disambiguation Section 144

5.1.4 Name Resolution Section 146
5.1.5 Conclusion 146
5.2 Qualitative Study: Gra2Mol 148
5.2.1 Gra2Mol DSTL Introduction 148
5.2.2 How Does Gra2Mol Address the Identified Concerns? 149
Concern 1: Mapping an LHS non-terminal to a meta-class 149
Concern 2: Mapping an RHS non-terminal to a reference 149
Concern 3: Mapping a terminal to an attribute 150
Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes 150
Concern 5: Mapping an LHS non-terminal to many meta-classes . . . 151
Concern 6: Multi-way mappings from LHS non-terminals to meta-classes 151
Concern 7: Mapping properties from non-grammar terms 151
Concern 8: Name resolution 152
5.2.3 Discussion 152
Parsing Technology Dependency 153
Modeling Technology Dependency 153
Language Nature 153
Query Language 153
Name resolution 155
Disambiguation 155
Rule Inheritance 155
Language Extension Mechanism 156
Specification Generation 156
5.3 Qualitative Study: Spoofax 157
5.3.1 Spoofax Introduction 157
5.3.2 How Does Spoofax Address the Identified Concerns? 159
Concern 1: Mapping an LHS non-terminal to a meta-class 160
Concern 2: Mapping an RHS non-terminal to a reference 160
Concern 3: Mapping a terminal to an attribute 161
Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes 161
Concern 5: Mapping an LHS non-terminal to many meta-classes . . . 162
Concern 6: Multi-way mappings from LHS non-terminals to meta-classes 162
Concern 7: Mapping properties from non-grammar terms 162
Concern 8: Name resolution 162
5.3.3 Discussion 163
Parsing Technology Dependency 163
Modeling Technology Dependency 164
Language Nature 164
Query Language 164
Name resolution 164
Disambiguation 165
Type resolution 165
Specification Generation 165
5.4 Quantitative Study: Introduction 166
5.4.1 Goals 166
5.4.2 Metrics 166
Performance of the Implementation 166

Size of DSTL instances 166
Why these metrics? 167
5.4.3 Subjects 167
5.4.4 Hypotheses 168
Hypothesis A. 168
Hypothesis B. 169
5.4.5 Methodology 169
CS2AS transformations execution with different models 169
CS2AS bridges for different textual modeling languages 169
Experiment Reproducibility 170
Hypotheses Testing 170
5.4.6 Threats to Experiment Validity 171
Representative Sample 171
Equivalent CS2AS bridges 171
Same Experiment Environments 172
5.5 Quantitative Study: Hypothesis A 172
5.5.1 Detailed Experiment: 101 Companies Example 172
Example Introduction 172
CS2AS Bridge 173
Experiment Design 176
Experiment Results 177
Scalability Analysis 178
5.5.2 Another Experiment: Delphi2ASTM Example 181
Example Introduction 181
CS2AS Bridge: Missing Rules/Mappings 182
CS2AS Bridge: No information in CS 185
CS2AS Bridge: Refactoring CS grammar 185
Experiment Design 186
Experiment Results 186
5.5.3 Testing Hypothesis *A* 187
101 Companies Example 187
Delphi2ASTM Companies Example 188
5.6 Quantitative Study: Hypothesis B 188
5.6.1 Examples Introduction 189
Example: DDL2ASTM 189
Example: iMacros 189
Example: ABNF2Ecore 189
5.6.2 Obtained Results 190
5.6.3 Testing Hypothesis *B* 190
5.7 Summary 191

6 Future Work & Conclusions **193**
6.1 Future work 193
6.1.1 CS2AS-TL Improvements 193
CS2AS-TL Extension Mechanism 193
Type Resolution 193
Disambiguation Rules Inheritance 194
Bi-directional CS2AS Bridges 194
6.1.2 Prototype Improvements 194
Compiling to QVT Relations 194
Improved Error Reporting 195

Debugger 195
Incremental Execution 195
Xtext Integration 196
Bootstrapping CS2AS-TL 196
6.1.3 Further Research 197
Additional Experiments 197
CS2AS Bridges for Graphical Languages 197
Can Some CS2AS-TL Features be Adopted by General Purpose M2M Transformation Languages? 197
6.2 Conclusions 198
6.2.1 Industrial Perspective 200
6.2.2 Academic Perspective 201

A Mini-OCL CS grammar 203

B Mini-OCL AS meta-model 207

C CS2AS bridge for Mini-OCL 211

D Complete OCL documents of the Mini-OCL CS2AS bridge 219

E Using Spoofax To Support The 101 Companies Example 237
E.1 Grammar Definition 237
E.2 Name Resolution Definition 238
E.3 CS2AS Definition 239
E.4 *Treeware-to-Modelware* Mapping Definition 240
E.5 Experiment Results 240
E.5.1 Size of artefacts 240
E.5.2 Implementation performance 242

F Experiments Data 243

Glossary 245

Bibliography 249

List of Figures

1.1 AS alternatives for the *x.y* OCL expression 4
1.2 Current CS2AS design in Eclipse OCL 5
1.3 Proposed solution: a DSTL to describe CS2AS bridges 9

2.1 Some examples of models 15
2.2 The traditional software development life cycle, according to Kepple et al. (Figure 1-1 from [48]) 15
2.3 What does this visually notated model mean? 16
2.4 Modeling stack according to the MDA-initiative (Figure 5 from [5]) 18
2.5 Boundaries of different MD-* approaches (Figure 2.1 from [10]) 18
2.6 OMG vision of MDD, also known as MDA (Figure 1–3 from [48]) 19
2.7 High-level model transformations overview 20
2.8 Detailed M2M transformation process (Figure 10.1 from [34]) 20
2.9 Technological spaces and bridges between them (Figure 1 from [54]) 23
2.10 A *grammarware to modelware* bridging approach (Figure 1 from [81]) 25
2.11 Lexical, syntax and semantic analysis of a compiler 26
2.12 CS vs AS vs Semantics 27
2.13 Feature model for LWs (Figure 1 from [20]) 30
2.14 Features coverage by the ten LWs (Table 1 from [20]) 31
2.15 *2013 Language Workbench Challenge* results (Table 4 from [20]) 32
2.16 Brief Overview of Xtext. 33

3.1 AS of a *Package* definition 45
3.2 Example: *Package* notation and corresponding AS model 46
3.3 AS of a *Class* extension definition 46
3.4 Example: *Class* extension notation and corresponding AS model 47
3.5 AS of an *Operation* body definition 48
3.6 Example: *Operation* body notation and corresponding AS model 48
3.7 AS of a *PropertyCallExp* 49
3.8 Example: *PropertyCallExp* notation and corresponding AS model 50
3.9 Example: Two different outcomes of a multi-way mapping 54
3.10 AS of *ExpressionInOCL* 55
3.11 Example: *LetExp* notation and its corresponding AS model 56
3.12 Overview of a scope 58
3.13 Scope involved in the *LetExp* example from Figure 3.11 59
3.14 Scope propagation, according to the example from Figure 3.11 60
3.15 Example: Nested *LetExp* notation and corresponding AS model 61
3.16 Scopes involved in the nested *LetExp* example from Figure 3.15 61
3.17 Qualified name-based lookups 64
3.18 Example: *PropertyCallExp* accessing an out of the scope *Property* 65
3.19 Meta-model excerpt of the OCL AS (Figure 8.3 from [39]) 67

4.1 Overall approach 74
4.2 Compilation process from CS2AS-TL to Java 74
4.3 Grammarware-to-Modelware vs CS-to-AS 75
4.4 General CS meta-model vs language-specific CS meta-model 77
4.5 CS meta-model based approaches 78
4.6 General CST meta-model vs language-specific CS meta-model 78
4.7 How Xtext ❶, Gra2Mol ❷ and Spoofax ❸ tackle the *grammarware*-to-*modelware* techonological space shift. 79
4.8 Main CS2AS-TL sections and dependencies 81
4.9 Let expression defining multiple variables 96
4.10 Equality expression example (left) and corresponding AS model (right) 104
4.11 Call expression example (left) and corresponding AS model (right) 106
4.12 *CollectionLiteralExp* example (left) and AS meta-model (right) 107
4.13 *CollectionLiteralExp* grammar (left) and CS meta-model (right) 108
4.14 Left: Feature model to characterise the rule types of a DSTL (Figure 3 from [68]). Right: Configuration model that depicts the features selected by CS2AS-TL 125
4.15 Left: Some helper definitions using the CS2AS-TL. Right: Corresponding OCL operation definitions 128
4.16 Left: Some disambiguation rules using CS2AS-TL. Right: Corresponding OCL operation definitions 128
4.17 Compiling the set of Complete OCL files that declare a CS2AS bridge into an executable M2M transformation in Java 134
4.18 Left: a partial *ast()* operation definition that declares a mapping between *ClassCS* and *Class*. Right: the corresponding QVTm micro-mappings 137
4.19 Left: Textual mOCL file. Right: corresponding AS model representation of the outline 138
4.20 Textual mOCL file on the left, and corresponding AS model representation of the outline on the right 138

5.1 Two equivalent helper operations in CS2AS-TL (left) and ETL (right) 144
5.2 Two equivalent mapping definitions in CS2AS-TL (left) and ATL (right) . . . 144
5.3 Two equivalent mapping definitions in CS2AS-TL (left) and RubyTL (right) . 145
5.4 Two equivalent queries in Gra2Mol (left) and OCL (right) (Figure 6 from [42]) 154
5.5 Extension mechanism in Gra2Mol (Fig. 16 from [42]) 156
5.6 Left: CS grammar definition of a simple Families language. Right: the corresponding AS meta-model 158
5.7 Textual instance of the *Families* language and the corresponding internal AST representation 158
5.8 Left: CS grammar definition of a simple Families language. Right: the corresponding AS meta-model 159
5.9 Left: instance of the *Family* textual modeling language. Right: the corresponding AS model, including the properties of the *SBB* family 160
5.10 Mapping the position/index of a subterm (corresponding to an RHS nonterminal) to the name of a reference 161
5.11 101 Companies Example [22]. On the left, the CS (ANTLR-based) grammar of the language. On the right, the AS (Ecore-based) meta-model 173
5.12 Equivalent Xtext grammar and corresponding CS meta-model 174
5.13 CS2AS specification in Gra2Mol (left) and CS2AS-TL (right) 175
5.14 Experiment results: execution time 177
5.15 Scalability: execution times for 25 models 179

5.16 Scalability: performance when *Employee::mentor* is not computed 180
5.17 Scalability: performance when caching operation results 180
5.18 Left: Gra2Mol rule definition that specifies a binding that requires a missing rule. Right: the corresponding Complete OCL code with compilation errors . 183
5.19 *Delphi2ASTM* experiment results: execution time 187

6.1 High-level Comparison of alternatives to support CTMLs 199
6.2 Comparison between the DSTLs of Gra2Mol and the proposed solution . . . 200

B.1 MiniOCL AS Meta-Model: Main Structure 208
B.2 MiniOCL AS Meta-Model: OCL Expressions 209

E.1 Mapping sub-term indexes to property names of the *Company* meta-model, through the *EAnnotations*-based mechanism 241

Listings

1.1 Simple OCL property navigation 4
2.1 Production rules of a grammar 22
2.2 Xtext-based grammar excerpt for OCL 29
3.1 Meta-model import notation 40
3.2 Meta-model definition notation 40
3.3 Constraint definition notation 41
3.4 Literal expressions notation 41
3.5 Let expressions notation 42
3.6 Variable expressions notation 42
3.7 Call expressions notation 43
3.8 Call expressions with syntactic sugar 43
3.9 Equality expressions notation 43
3.10 CS of a *Package* definition 44
3.11 CS of a *Class* extension definition 46
3.12 CS of an *Operation* definition 47
3.13 CS of a *PropertyCallExp* 49
3.14 Navigation from *PropertyCallExp* ❶ to *Class 'c1'* ❺ 66
3.15 Xtext grammar of a *Package* definition 68
3.16 Xtext grammar of a *Class* extension definition 70
4.1 Shadow expression syntax 82
4.2 Shadow expression usage example 82
4.3 Trace expression syntax 83
4.4 Trace expression usage example 83
4.5 Source and target domain syntax 84
4.6 Source and target domain declaration 84
4.7 Helpers section syntax 85
4.8 Helpers section example 85
4.9 Name resolution section syntax 86
4.10 Targets definition syntax 87
4.11 Targets definition example 87
4.12 Filters definition syntax 88
4.13 Filters definition example 88
4.14 Qualification definition syntax 89
4.15 Qualified/qualifier targets example 89
4.16 Inputs definition syntax 90
4.17 Inputs definition example 90

4.18 Providers definition syntax 91
4.19 Current scope contributions syntax 91
4.20 Simple current scope provision example 92
4.21 Children selection syntax 92
4.22 Propagating a nested scope 93
4.23 Occlusion definition syntax 93
4.24 Occlusion definition example 94
4.25 Selective contribution definition syntax 95
4.26 Children selection example 95
4.27 Child definition syntax 96
4.28 Contribution using a child definition 97
4.29 Children selection syntax 97
4.30 Example of a contribution using preceding children 97
4.31 Exported scope declaration syntax 98
4.32 Exported scope declaration example 98
4.33 Syntax to use contributions from exported scopes 99
4.34 Example of using contributions from an exported scope 100
4.35 Alias declaration syntax 100
4.36 Alias declaration example 101
4.37 Lookup expression syntax 102
4.38 Lookup expression usage examples 102
4.39 Mappings section syntax 103
4.40 CS definition of equality expression 103
4.41 Creation mappings syntax 104
4.42 Example of a creation mapping 105
4.43 CS definition of call expression 106
4.44 Reference mappings syntax 107
4.45 Example of a reference mapping 107
4.46 When clause syntax 108
4.47 Multi-way mappings definition example 109
4.48 Disambiguation section syntax 110
4.49 Disambiguation section usage example 110
4.50 CS2AS definition of *Package* 111
4.51 CS2AS definition of *Operation* 112
4.52 CS2AS definition of *PropartyCallExp* (with implicit source) 113
4.53 Multi-way mappings definition for *NameExpCS* 115
4.54 CS2AS description of *Constraint* 117
4.55 CS2AS description of *LetExp* 118
4.56 CS2AS description of mOCL literal expressions 118
4.57 CS2AS description of mOCL loop expressions 119
4.58 CS2AS description of mOCL loop expressions 121
4.59 CS2AS conformance rules for some OCL types 122

4.60 Defining additional lookup criteria for *Tuple* types 123
4.61 CS2AS bridge to create *OperationCallExp* from a unary expression 123
4.62 CS2AS bridge to create *IfExp* model elements 124
4.63 Name resolution declaration corresponding to *Variable* lookups 129
4.64 OCL operation definitions corresponding to the name resolution declaration from Listing 4.63 131
4.65 OCL operation definition corresponding to the multi-way mappings whose source is *CollectionLiteralPartCS* 132
4.66 Simple QVTm micro-mapping 135
5.1 Example of name resolution described with CS2AS-TL 146
5.2 QVTo query operations equivalent to those from Listing 5.1 147
5.3 CS2AS definition for *PackageCS* using GraMol. 149
5.4 CS2AS definition for *CallExpCS* using Gra2Mol 150
5.5 CS2AS definition for *OperationCS* using Gra2Mol 150
5.6 CS2AS definition for *NameExpCS* using Gra2Mol 151
5.7 Multi-way mapping definition for *NameExpCS* using Gra2Mol 152
5.8 Name resolution declaration: *Block*s contribute *VariableDeclaration*s to the current scope 154
5.9 *Mixin* rule example (Fig. 11 from [42]) 156
5.10 CS2AS definition for *PackageCS* using Spoofax 160
5.11 CS2AS definition for *OperationBody* using Spoofax 161
5.12 CS2AS definition for *NameExpCS* using Spoofax 162
5.13 CS2AS definition for *NameExpCS* using Spoofax 163
5.14 Using NaBL to declare a site definition for mOCL *Variables* 163
5.15 Declaring Stratego rules to create navigation/query operations 164
5.16 A Gra2Mol rule that *skips* to its (sub)term when there is no *addOp* 184
5.17 A Gra2Mol rule that creates a *BinaryExpression* 185
5.18 A Gra2Mol rule which uses the path of the parsed file 185
5.19 An ANTLR syntactic rule to declare statements 186
5.20 The refactored syntactic rule to declare statements using Xtext 186
A.1 Mini-OCL CS grammar 203
C.1 CS2AS bridge for Mini-OCL 211
D.1 MiniOCLHelpers.ocl 219
D.2 MiniOCLDisambiguation.ocl 220
D.3 MiniOCLNameResolution.ocl 220
D.4 MiniOCLCS2AS.ocl 231
E.1 101 Companies example grammar definition using SDF 238
E.2 Name resolution definition for the 101 Companies Example 238
E.3 CS2AS definition for the 101 Companies Example 239

List of Tables

3.1 Correspondence between targets (T), providers (P), consumers (C) and mOCL meta-classes 58
3.2 Computed scopes for every AS model element of Figure 3.14 60

4.1 Valid property assignment scenarios based on LHS property multiplicity and RHS expression type 105
4.2 Name resolution topic and demonstrating listings 116
4.3 CS2AS-TL features required to address OCL types 123
4.4 CS2AS-TL features required to address OCL expressions 124

5.1 *101 Companies* example: models characterisation 176
5.2 New experiment models characterisation 178
5.3 *Delphi2ASTM* example: specification artefacts 181
5.4 *Delphi2ASTM* example: models characterisation 187
5.5 Wilcoxon statistical tests for H_A0 (Delphi Example). Paired data corresponds to the execution time obtained from the experiment 188
5.6 *DDL2ASTM* example: specification artefacts 189
5.7 *iMacros* example: specification artefacts 190
5.8 *ABNF2Ecore* example: specification artefacts 190
5.9 Measurements of size of specification artefacts 191
5.10 Wilcoxon statistical tests for H_B0. Paired data corresponds to the size of artefacts obtained from the experiment 191

E.1 Size (see Section 5.4.2) of the required artefacts specified within Spoofax . . . 240
E.2 Execution time (ms) required by Spoofax implementation to produce the AS models. 242

F.1 Hypothesis A: *101 Companies* example (Experiment 1) 243
F.2 Hypothesis A: *101 Companies* example (Experiment 2) 244
F.3 Hypothesis A: *Delphi2ASTM* example 244
F.4 Hypothesis B: Size of specification artefacts 244

Acknowledgements

There is a saying that states that "Gratitude is the least of the virtues, but ingratitude is the worst of vices (Thomas Fuller)". From my humble point of view, it is much more than a minor virtue. Since I was a child, my parents have taught me that gratefulness is the most open, direct and simplest way of building a fruitful relationship between yourself and others. In consequence, I do not miss a chance to sincerely thank everybody that has made me build something or progress in some way, regardless the size of the contribution. Obviously, submitting a thesis is a very important milestone in my life and I would probably require yet another lengthy appendix to properly acknowledge many of known people that have helped to achieve this milestone. It is impossible to mention everybody and I would probably and unfairly forget many of you. However, I will not miss this opportunity to say thanks.

I commence with my academic supervisor Richard who always have had a smiling face and positive attitude to guide me towards the right direction. His willing-to-help disposition has always been there to address from banal topics, such as 'Should I write modeling or modelling?' or 'Which restaurant do you suggest for Christmas dinner?' to tougher situations that I do not even dare to mention here. Second, my industrial supervisor Ed deserves his space within these lines. He not only firstly stepped in to make this EngD opportunity happen, but also this doctoral thesis would not have been possible without his technical support and his prompt responses to my calls for help. I also want to acknowledge the invaluable help of Dr. Paul Cairns, without his help this thesis would have been much worse, if not invalid. Likewise, I acknowledge the unconditional technical support of externals such as Javier Cánovas, Guido Wachsmuth and Pieter Van Gorp.

Mentioning all the colleagues that have helped during these four years would be impossible. Special mention to Horacio, who from the first day I landed in UK always did his best to facilitate my integration and helped me in this journey in many different ways. Discussions have appeared in many shapes and colours. William, Thanos, Kostas, Simos, Colin, Antonio, Faisal, Gabriel, James, Jimy, Dimitris, Louis, Nikos and every single member of the Enterprise Systems group have contributed a piece of enlightening discussion that have enriched me in some way or another. Needless to say that many of them have contributed to much more than a discussion: unforgettable moments, *trolling* and *epic* moments included ;). Of course, I can't forget many others in the department/University who helped me in my integration in this new environment: Antonio, Rosh, Sam, Barath, Chris, Alan, Russel, Pete, Luca, Javier, Sergio, Ana, Rick and a very long list of fellows that it is impossible to mention.

I'm surely missing many of you. For example, If I had to recall all the fruitful discussions I have had in conferences, workshop, etc. these acknowledgments could last forever. However, these acknowledgment lines can't be closed without some special mentions. Starting with Noe. Living abroad is not easy, and he has always made his best to bring a piece of Tenerife a little bit closer and it is something I'll not forget. Next my family. Despite being thousands of miles away, they have been always present in York. Without their unconditional support and love, this journey would not have been possible. The last lines of these acknowledgment focus on Inmaculada, my wife and the *leitmotiv* of my life. She has been my lungs during these four years, suffering my bad moments and enjoying the good ones as they were hers. She knows how important she has been to make this thesis happen, and I can only say: Thank you (L).

This research project has been funded by the Engineering and Physical Sciences Research Council (UK), with the collaboration of Willink Transformations Ltd. Additional acknowledgments are given to Open Canarias S.L. and Thales R&D (UK) for their support in the past.

Declaration of Authorship

I, Adolfo Sánchez-Barbudo Herrera, declare that this thesis titled, "Auto-tooling to Bridge the Concrete and Abstract Syntax of Complex Textual Modeling Languages" and the work presented in it are my own. I confirm that:

- This work was done wholly while in candidature for a research degree at this University.
- This work has not previously been presented for an award at this or any other institution.
- This work has been supervised by Professor Richard F. Paige and Dr. Edward D. Willink.
- I have acknowledged all sources of help as References.

The following list shows the authored and published work on which parts of this thesis is based:

- Adolfo Sánchez-Barbudo Herrera. *"Enhancing Xtext for General Purpose Languages"*. In: Proceedings of the Doctoral Symposium at MODELS'14. Vol. 1321. CEUR Workshop Proceedings. 2014.
- Adolfo Sánchez-Barbudo Herrera, Edward D. Willink, and Richard F. Paige. *"An OCL-based Bridge from Concrete to Abstract Syntax"*. In: Proceedings of the 15th OCL Workshop. Vol. 1512. CEUR Workshop Proceedings. 2015, pp. 19–34.
- Adolfo Sánchez-Barbudo Herrera, Edward D. Willink, and Richard F. Paige. *"A Domain Specific Transformation Language to Bridge Concrete and Abstract Syntax"*. In: Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016, pp. 3–18.

“I only know that I know nothing.”

Socrates

Chapter 1

Introduction

Throughout this thesis, the reader may note different coloured bubbles used to add side notes. In some cases, they comprise either a brief recap or an important note to emphasize and/or clarify parts of the thesis content. The semantics of the different colours are explained as follows:

- A blue bubble (bulb icon) represents just a hint that adds additional information. They are not required to understand the surrounding content.
- A yellow bubble (exclamation mark in a triangle icon) represents a warning note. Although they are not required to understand the surrounding content, they may help to clarify and/or remove a reader concern.
- A red bubble (exclamation mark in a circle icon) represents an important note. They are used to bring the attention to a particular part of the thesis because the discussed concern is relevant for further content.

1.1 Context

This document constitutes the doctoral thesis of a research project in an Engineering Doctorate (EngD) programme in *Large-Scale and Complex IT Systems (LSCITS)*. An EngD programme differs from traditional PhD studies in that the research project is directed by both an academic supervisor and an industrial one. In this research project, the industrial sponsor is Willink Transformations Ltd[1].

Understanding the context of the research is important. As noted in Section 1.3, the research project scope is partly established by the sponsoring industrial partner.

On the one hand, the industrial partner's interests focus on *Object Management Group (OMG)* standardised specifications, particularly Object Constraint Language (OCL) [39] and *Query/View/Transformation (QVT)* [38]. On the other hand, the industry supervisor, Dr Edward D. Willink is a committer in the official Eclipse [25] projects created to provide an implementation of these specifications.

[1] http://www.willinktransformations.co.uk/

OCL and QVT are open[2] specifications, which describe standardised languages that operate on models[3]. In the case of the former, the specification provides an expression language for different purposes [12], for instance, to traverse models (also known as models navigation) and to create queries with the aim of retrieving information from them. Likewise, OCL lets modelers create constraints on modeling languages. These constraints help the definition of more precise rules to specify when models are well formed. In the case of the latter, the specification describes three different languages used for defining Model-to-Model (M2M) transformations. Briefly, the QVT specification defines one imperative language, QVT Operational Mappings (QVTo), and two declarative [9] languages, QVT Relations (QVTr) and QVT Core (QVTc). The languages defined in the QVT specification are an extension of OCL; that is, they use the OCL expressions language to navigate or query the models that are transformed.

The modeling community has built a large ecosystem of technologies around the Eclipse [25] platform to provide a wide set of modeling tools to end users. OMG specifications, like OCL and QVT, have their own official Eclipse projects. Eclipse OCL [24] provides the corresponding language implementation, as well as facilities like textual editors and evaluation consoles. In addition, there are two projects to support the languages defined in the QVT specification: Eclipse QVTd [26] focuses on the two declarative languages, and Eclipse QVTo [27] focuses on the imperative one.

Eclipse is an open-source platform. In its umbrella modeling project, there are official Eclipse projects aimed at providing an implementation for the OCL and QVT specifications. The main development activities are tied to these open-source projects.

1.2 Motivation

OMG specifications, such OCL and QVT, are the focus of interest in this research project, whereas the Eclipse OCL, QVT Declarative (QVTd) and QVTo projects host the corresponding implementations. In this context, this section introduces the reasons that have motivated this research project, discussed from two different points of view: industry and academia.

1.2.1 Industry

Eclipse OCL has been an active project since 2006[4]. In 2010, it started to evolve by providing a new implementation to align with new changes in the OMG specification. Additionally, the project was enhanced with high-quality textual editors, with the aim of facilitating better tooling when adding model constraints. These high-quality editors have the particular

[2] In the sense of publicly available for download, as well as referring to the processes which take place to evolve them.
[3] The concept of model in this thesis is further discussed.
[4] https://www.eclipse.org/modeling/mdt/downloads/?project=ocl

feature of being largely automatically generated using Xtext [21]. Briefly, Xtext is a Language Workbench (LW) [31], which lets language engineers build tooling to support textual languages (e.g. Java, OCL, a textual Domain Specific Language (DSL) etc.), including parsers, editors, complementary views and, generally speaking, a high-quality Integrated Development Environment (IDE) for the Eclipse platform[5]. These IDEs aim to improve user experience when dealing with instances of the language.

By contrast, Eclipse QVTo is a mature project, which relies on Eclipse OCL (the old implementation). It has not evolved as Eclipse OCL has; thus, it is not aligned with the new Eclipse OCL implementation. This is where Willink Transformations Ltd become important for this research project, since they are pursuing an alignment between the new Eclipse OCL implementation and a new Eclipse QVTo one. As an additional note, despite the fact that Eclipse QVTo already has its own high-quality textual editor, this was completely handcoded. Therefore, moving the Eclipse QVTo project in the same direction as Eclipse OCL – i.e. towards the automatic generation of tools – has been considered as beneficial by the industrial partner.

The sponsoring industrial partner is also in charge of the Eclipse QVTd project. This project gives supports to the QVTd languages of the specification: QVTr and QVTc. The Eclipse QVTd project can also benefit from any results that can leverage and facilitate the development and maintainability of the tooling that supports the languages defined by OMG.

To conclude, the industrial partner is also interested in improving the open specifications defined by OMG: they not only present several typos and inconsistencies [66], but also – according to the industrial partner – there are parts of the specification that are not implementable. Therefore, one of the goals of the industrial partner is to have well-defined models from which both useful tools and parts of OMG specifications can be generated. In this way, with the right models and tools (validators, generators) specifications can be consistent, and implementations consistently compliant.

1.2.2 Academia

Although it is important to know the reasons that motivate the industrial partner to collaborate on this project, the research requires scientific motivations to qualify for an EngD. In particular, this EngD project offers the opportunity to investigate the improvement of existing approaches to support Complex Textual Modeling Language (CTML).

Xtext is a language workbench used by official Eclipse projects to create the tools (parsers, editors) that support the OCL and QVT specifications. Whilst Xtext is a good candidate [20] when dealing with DSLs, which do not usually place any imposition on its Abstract Syntax (AS) and Concrete Syntax (CS) (topic discussed in section 2.2.4), in specification-driven languages such as OCL and QVT there is a significant gap between its AS and CS definitions. For these kinds of complex languages [6], Xtext grammars are not powerful enough to produce a parser capable of processing an input conforming to the defined CS (a grammar) and producing the expected output conforming to the defined AS (a meta-model). Forthcoming

[5] From version 2.9 you can produce Xtext-based editors targeting other technologies (IntelliJ, web browsers).
[6] Section 1.3 describes the scope in which we investigate CTMLs.

```
x.y
```

LISTING 1.1: Simple OCL property navigation

subsections briefly introduce the problem, how it is currently overcome by the Eclipse OCL implementation, and the shortcomings of the current solution. These limitations, and the benefits of overcoming them, constitute the academic motivation of this research project.

Specification-based languages, such as OCL and QVT, are defined with a specific CS and AS. Pre-established CS and AS may present a wide gap between them, as exemplified by these OMG specifications.

Problem

As explained before, OCL and QVT languages have a specific CS grammar and AS metamodel defined by the corresponding OMG specifications. Although this situation is legitimate, there are difficulties when the specifications are implemented by means of a language workbench such as Xtext. The main reason is the gap that exists between the CS and AS imposed by the specification. A first example to understand the problem is shown by Listing 1.1, depicting a trivial OCL expression.

According to the CS of OCL, this navigation expression can be considered as two name expressions (e.g. *x* and *y*) separated by a navigation symbol (e.g. dot). According to the AS, this simple expression may entail different concepts. Whilst *y* unambiguously is a *PropertyCallExp*, *x* can be either a *VariableExp* – for instance, referring to a variable previously defined by an outer let expression – or another *PropertyCallExp* – referring to an *x*-named property of *self*, the context element in which that expression is evaluated. Figure 1.1 depicts the two different AS alternatives that can be obtained from the *x.y* OCL expression.

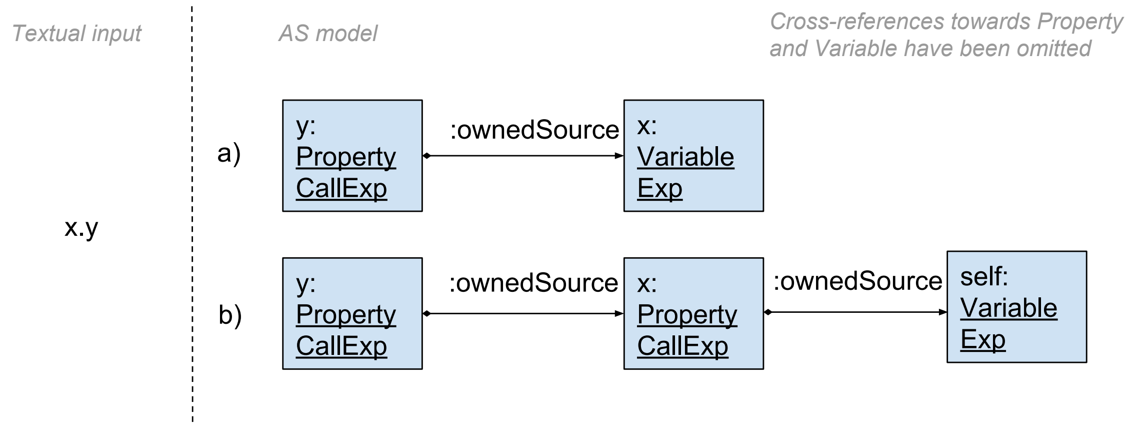


FIGURE 1.1: AS alternatives for the *x.y* OCL expression

This example shows the kind of gap between the CS and AS that requires attention. Whilst this gap is completely legitimate, problems arise when the tools do not provide adequate means to bridge it. In this case, by means of Xtext specification artefacts (i.e. a grammar), there is no way to generate automatically a parser that produces AS models (conforming to the OCL meta-model specified by OMG) from textual inputs (conforming to the OCL grammar defined by OMG).

Current Solution

The problem above does not prevent a language engineer from using Xtext when working on languages like OCL and QVT. As previously mentioned in Section 1.2.1, Eclipse OCL already has an Xtext-based implementation to give support (parsers, editors) to the OCL specification. However, due to the gap between the CS and AS of the language, a parser capable of directly producing models conforming to the AS meta-model cannot be fully generated from an Xtext grammar.

Therefore, the problem has been addressed within Eclipse OCL by splitting the process into two steps. The approach is depicted in Figure 1.2. From an Xtext-based grammar specification ❶, the generated parser ❷ is capable of producing models ❸ that conform to the CS meta-model ❹ of the language. In a second step, these CS models are processed by an analyser ❺ that finally produces the expected models ❻ that conform to the AS meta-model ❼.

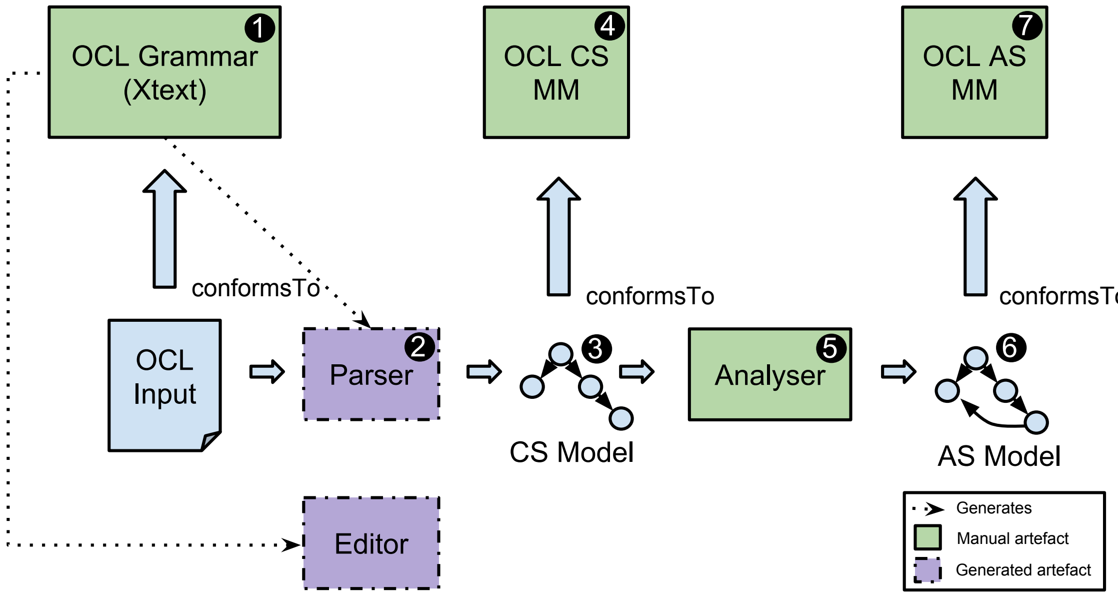


FIGURE 1.2: Current CS2AS design in Eclipse OCL

The main shortcoming of the current solution is that the analyser ❺ in charge of producing the AS models is implemented in Java, and it currently comprises a substantial amount of source code. Given that these kinds of analysers are required by other languages, such as QVTo, QVTr, QVTc etc., acquiring new mechanisms to reduce the amount of hand-written code required to support the mentioned activities, motivates this research from an academic point of view.

The main objective of this doctoral thesis is to provide adequate means to help language engineers that work on complex textual modeling languages. In particular, given that a language is defined by well-known specification artefacts, such as grammars and meta-models to define the CS and AS of a language respectively, additional specification artefacts to bridge the gap between the CS and AS are pursued.

1.3 Scope

This section establishes the scope of the research and development activities. Since this thesis is focused on Model-Driven Engineering and Language Engineering, it is important to set the boundaries of the kind of language that is actually targeted.

The title of this thesis refers to Complex Textual Modeling Language (CTML)s. When referring to them throughout this thesis, we mean the following:

Chapter 2 defines all the related terminology.

- **Modeling.** A language is considered to be modeling or model-based when working on instances of the language implies editing an underlying model that conforms to a meta-model. In particular, the meta-model defines the abstract syntax of the language. Although there are other forms of modeling (e.g. spreadsheets [33]), this thesis only focuses on languages whose abstract syntax is formalised by meta-models. When introducing the field survey, Section 2.1 defines models and meta-models.

- **Textual.** A language is considered to be textual when the instances of the language that the user edits are text that conforms to a grammar. In particular, the grammar defines the CS of the language. Although there are other ways to provide textual concrete syntax to modeling languages (e.g. via meta-model annotations [43]), this thesis only focuses on those languages whose concrete syntax is formalised by a grammar. When introducing the field survey, Section 2.2 defines grammars.

- **Complex.** A language is considered to be complex when there is a significant gap between the CS and AS of the language. Note that some tools, such as Xtext, allow the automatic generation of an AS meta-model from a grammar definition. Thus, a different way of seeing this complexity is the following: when a particular AS meta-model definition cannot be derived from a CS grammar definition, the textual modeling language is considered to be complex. When explaining the problems related to bridging the gap between the CS and AS of a language, Section 3.4.4 gives a more accurate explanation of what a complex language is.

Note that languages from the OCL and QVT specifications fall into this definition of CTML.

In the context of this EngD doctoral thesis, and according to the sponsoring company, there are additional requirements to highlight regarding the kind of technology and tools on which the solution development should be based:

- The language and tools created for this thesis must work on the Eclipse Modeling Framework (EMF), which is the official modeling framework within the Eclipse platform.
- The language and tools created for this thesis must work with Xtext, which is the official language workbench to work on textual modeling languages within the Eclipse platform.

1.4 Research Methodology and Questions

1.4.1 Research Method

The research method that has driven this project is the engineering method. According to Basili [3], this research method consists of "observing existing solutions, propose better solutions, measure and analyse". In this particular project, there has been a study of existing solutions:

- Xtext is a language workbench to support textual modeling languages. However, limitations have been identified that prevent the adequate support to CTMLs.
- Eclipse OCL and QVTd projects provide implementations for CTMLs and use Xtext. They have implemented ad-hoc solutions to the Xtext limitations. However, their solutions are based on writing Java source code.

Given the problems of existing solutions, this research project investigates better alternatives that should address Xtext specification artefacts limitations. At the same time, the new solution should remove – as far as possible – any need for additional hand-written Java code.

Additionally, other technologies that may be used to support CTMLs have been identified. These existing solutions (e.g. Gra2Mol [42] and Spoofax [46]) have been studied with the aim of helping the creation of the new alternative that can be used within the boundaries of this project (Section 1.3). Moreover, empirical research [83] has been conducted to show the benefits of the new solution compared to the existing ones.

1.4.2 Research Questions

This subsection presents the questions that drive the research and development activities of this doctoral thesis. The research questions are presented from two different points of view: a more specific technology-centred industrial perspective, and a wider academic perspective.

Industrial Perspective

Xtext is a language workbench that helps language engineers to generate tools (i.e. parsers and editors) that support textual modeling languages. However, CTMLs present a significant gap between the CS and AS that prevents these tools from being created by means of Xtext specification artefacts (i.e. grammars). Alternatives exist to complement the source code generated by Xtext, so that a parser can consume textual inputs and produce the expected AS models. However, these solutions are hand-coded in Java and the amount of manually written code is substantial compared to, for instance, the amount of manually written code for the corresponding grammar.

The following research questions can now be formulated:

- Can a new alternative solution decrease the amount of hand-written code required to support CTMLs within Xtext?
- If so, can this alternative solution also be used for automatically producing parts of OMG specifications?

Academic Perspective

⚠ Although the Xtext language workbench and OMG-based CTMLs are the main interests of this thesis from an industrial point of view, additional research questions have emerged from a wider academic perspective.

There exist different approaches to helping language engineers to create support for CTMLs. This support is realised in the form of tools (e.g. parsers) that end users can use to work on instances of their target CTML. As Chapter 3 further analyses, there are different identified concerns that need to be addressed in order to bridge the CS and AS of CTMLs.

The following research questions can now be formulated:

- Do existing approaches address all the identified concerns that are required to support CTMLs?
- Can a new approach improve the performance (in terms of execution time) of existing approaches to produce AS models from textual files of CTMLs?
- Can a new approach reduce the size of specification artefacts required by existing approaches to bridge the gap between the CS and AS of CTMLs?

1.5 Thesis Contributions

As part of this introductory chapter, this section discusses the contributions of this doctoral thesis. First, a brief description of the proposed solution to the problem introduced in Section 1.2.2 is shown. Finally, the contributions of this thesis are discussed.

1.5.1 Proposed Solution

When introducing the problem to explain the motivation of this research project in Section 1.2.2, Figure 1.2 showed how Eclipse OCL overcomes the limitations of Xtext. Now, Figure 1.3 depicts the overall solution that this thesis proposes as an alternative means of overcoming these limitations. The solution consists of having a Domain Specific Transformation Language (DSTL) that lets language engineers describe the bridges ❶ between the CS and AS of CTMLs. Then, additional tooling is responsible for generating the source code of an M2M transformation ❷ capable of producing AS models from CS ones. Finally, this M2M transformation can be integrated in the editors ❸ generated by Xtext, so that AS models can be produced every time the end user edits input files.

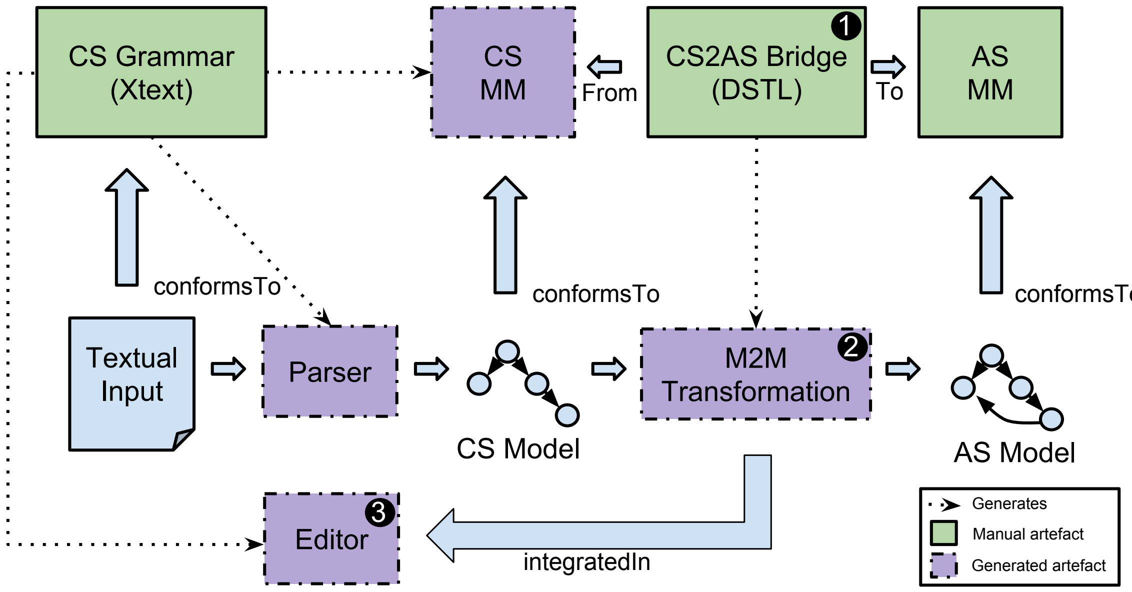


FIGURE 1.3: Proposed solution: a DSTL to describe CS2AS bridges

As discussed in Chapter 4, the following characteristics are highlighted:

- Producing tools (e.g. parsers and editors) that support CTMLs is a matter of providing specification artefacts rather than writing source code. These specification artefacts consist of meta-models, grammars and Concrete Syntax to Abstract Syntax (CS2AS) bridges.
- Although the developed prototype has been used within Xtext, the proposed solution is independent of the front-end (i.e. parsing technology). As long as textual information is comprised by a CS model, the CS2AS transition can be achieved by means of the proposed solution.

1.5.2 Contributions

This subsection lists the main contributions of this research project.

A DSTL to Specify CS2AS Bridges

The proposed DSTL (Section 4.2) comprises the main contribution of this thesis, which has been recently published [65] in the 9th International Conference in Model Transformations[7]. This transformation language is coined in this thesis as Concrete Syntax to Abstract Syntax Transformation Language (CS2AS-TL) and provides the required means to describe complex CS2AS scenarios existing in CTMLs. The CS2AS bridges can be seen as domain-specific M2M transformations[8]. As shown in subsequent chapters, some features of the proposed language are either not supported by existing approaches or are provided via out-of-the-box solutions (e.g. external black-boxed behaviour) to support them.

A Prototype to Execute CS2AS Model Transformations

Additionally, a prototype has been developed to make instances of the CS2AS-TL executable. After several compilation steps (Section 4.1.1), a Java class comprising an M2M transformation is generated. This Java class transforms CS models (generated by a parser) into AS models.

Note that this contribution is part of the development accomplished to produce the results of this thesis, but it is not coupled to CS2AS-TL. This means that, from instances of the same proposed CS2AS-TL, other M2M technologies could be targeted to produce executable M2M transformations.

1.6 Intended Audience

This section briefly discusses the target audience that may be interested in the performed activities and obtained results of this research project.

- **Xtext Users**. Language engineers that work with Xtext may be interested in CS2AS-TLs to give support to CTMLs. Note that if the language is simple enough that the AS meta-model can be directly obtained from (or mapped to) a grammar specification, the outcomes of this research are largely irrelevant.
- **Language Workbench Engineers**. Language Workbench engineers may be interested in CS2AS-TL or the fundamentals behind it. If their users (language engineers) need to create support to CTMLs, all the concerns identified in this thesis need to be supported. As is further discussed, the proposed solution is parsing-technology and language-workbench independent. Therefore, the proposed solution can be used not only within Xtext, but within other technologies such as IMP [14].
- **Specification writers, maintainers and readers**. The OMG is a consortium that produces specifications for several CTMLs. Instances of CS2AS-TL can be used as additional specification artefacts and/or to drive the generation of the part of the specification designed to describe CS2AS bridges (e.g. Clause 9.3 from [39]). The introduction

[7] `http://is.ieis.tue.nl/research/ICMT16/`

[8] Therefore, these bridges are set in the *modelware* technological space (Section 2.2).

of models to generate parts of the OMG specifications not only eases the task of specification maintainers, but also produces more robust specifications.

1.7 Thesis Overview & Structure

To conclude the chapter, this section gives an overview of the remaining thesis content and how it is structured.

Chapter 2 describes the **field survey** in depth, explaining concepts and ideas in the field of Model-Driven Engineering (MDE) (*modelware*), Language Engineering (particularly, *grammarware*), and introduces modern technologies designed to work on language creation called Language Workbenches. In this chapter, the relevant related work is also identified.

Chapter 3 shows a detailed **analysis of the problem** of working on CTMLs. In particular, it identifies the concerns that need to be addressed when bridging the CS and AS of CTMLs. The analysis is done from two different perspectives: a coarse-grained and a fine-grained view. Among others, several topics such as CS disambiguation or name resolution are explained in detail. The chapter concludes by demonstrating how a relevant technology (Xtext) that deals with textual modeling languages cannot adequately address some of the identified concerns.

Chapter 4 explains the **proposed solution**, including discussions of the approach and related work. After an overview of the solution, the chapter focuses on a detailed description of CS2AS-TL designed to define CS2AS bridges for CTMLs. Additionally, it is shown how the different identified concerns are addressed by the solution, including all the CS2AS scenarios of a running example. Then, a technical but brief explanation of the compilation process to make instances of CS2AS-TL executable is given. This compilation process includes the generation of instances of an OCL-based internal DSTL [66], QVTm transformations [77] and, eventually, the Java classes in charge of transforming CS models into AS models. Finally, the additional work developed to integrate the CS2AS functionality within the Xtext language workbench is shown.

Chapter 5 shows the **evaluation** of this thesis' contributions. After a brief discussion of the benefits of CS2AS-TL compared to general-purpose M2M transformation languages, two qualitative studies are presented. These studies aim at comparing the proposed solution to relevant related work: Gra2Mol and Spoofax. Finally, with the objective of answering some of the research questions, a quantitative study is conducted to show the benefits of CS2AS-TL, particularly with respect to the size of specification artefacts required to declare CS2AS bridges for several examples, and the performance (in terms of execution time) of the developed prototype.

Finally, Chapter 6 describes the future work that could build on this research project, and the conclusions of this thesis.

Chapter 2

Field Survey

In this chapter, core concepts related to the topics of this thesis are explained, along with some comments and citations to related work. The chapter is split into four sections. Section 2.1 introduces Model-Driven Development (MDD) and several related concepts that are used throughout this document. Since this thesis is focused on textual modeling languages, Section 2.2 explains important concepts, such as *grammarware* and *modelware* technological spaces, along with a summary of some tools that aim to bridge them. Modern tools called Language Workbench (LW)s are introduced in Section 2.3, along with a justification of their usage. Section 2.4 mentions the relevant work that this thesis focuses on. Finally, Section 2.5 concludes with a summary of the chapter.

2.1 Model-Driven Development

2.1.1 Introduction

Software Engineering (SE) [60] is a branch of the computer science field which aims to set standards, principles, techniques, methods and tools aimed at easing engineers' tasks when building software.

Since the middle of the last century, different types of programming languages have arisen to cover new needs of software developers. In the early *XXIth* century, a new trend named Model-Driven Development (MDD), also known as Model-Driven Software Development (MDSD), emerged to move beyond the traditional programming languages paradigm. This change of paradigm is a natural evolution in terms of raising the abstraction levels that engineers use to specify what needs to be done, and how it should be done, when software executes.

In traditional software development, programs are the main artefact used by software developers. In Model-Driven Development (MDD), the model is the key artefact to use for the same purposes. As in other science fields, models help scientists to work with abstractions, which are usually created to hide the complexity of the real world. These abstractions ease the understanding of the problem, the consequent reasoning and, in general, facilitate the scientists' labour. In MDD, a model is an abstraction of a piece of software[1], which is designed to be run by a computer.

[1] *A piece of software* ranges from a whole complex application to a small and simple part of it.

In the remainder of this section, some concepts are clarified, such as what models, meta-models and model transformations are, as well as why they are needed. In introducing common terminology, a broader term coined Model-Driven* (MD*) – also referred to as *Model-Driven Whatever* [10] – is introduced. Finally, model transformations are briefly explained, as well as their role within MDD.

2.1.2 What is a Model?

One obstacle encountered by engineers when they are introduced to any model-driven approach, is understanding the concept of **model**. It cannot be claimed that there is a clear and unambiguous definition about what a model, in the SE context, is. For instance, the Object Management Group (OMG) through their Model-Driven Architecture (MDA) initiative, defines a model in their MDA-guide [56] as follows:

> "A model of a system is a description or specification of that system and its environment for some certain purpose. A model is often presented as a combination of drawings and text. The text may be in a modeling language or in a natural language."

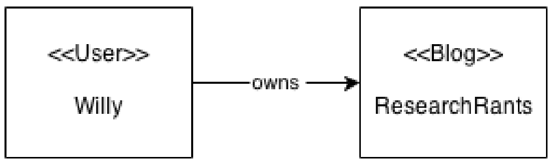
Nevertheless, in MDD a *'model'* can be rather interpreted in terms of some inherent characteristics:

- Models are abstractions of concrete specimens of the real world (e.g. a system).
- Models describe concepts (entities), their properties and the relationships between them.
- In the context of software development, models provide a higher level of abstraction than traditional programs do.
- Models are usually associated with graphical representations, but not necessarily.

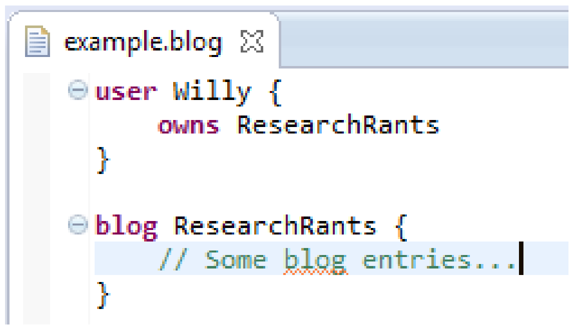
For instance, Figure 2.1a depicts an example of a model in a graphical syntax, whereas Figure 2.1b denotes the same model in a textual one. Both models are abstractions of the real world. In both models, there are two different entities: *User* and *Blog*, which have a property to identify them (a name), *Willy* and *ResearchRants* respectively, and there is a relationship between them indicating that the user *Willy* owns the *ResearchRants* blog.

2.1.3 Why are Models Used?

As Martin Fowler explains in his book *"UML Distilled: A Brief Guide to the Standard Object Modeling Language"* [32], visual modeling languages emerged from the need for providing a higher level of abstraction of than existing programming languages. In particular, after the success of Object Oriented Programming (OOP) languages to develop software, it happened that those languages did not provide enough level of abstraction to discuss properly,



(A) A graphically notated model



```
user Willy {
    owns ResearchRants
}

blog ResearchRants {
    // Some blog entries...
}
```

(B) A textually notated model

FIGURE 2.1: Some examples of models

for instance, software design. Therefore, the Unified Modeling Language (UML)[2] [40] was conceived to fill this gap.

By that time, models were mainly for communication and documentation purposes to help in the early stages of the software development process, during the analysis and design phases. These models were even passed to software developers as input documentation to understand the system they had to implement. Whilst there have been defenders of these model-based engineering processes, there have also been detractors who consider this kind of documentation as a mere overhead in the development process. Additionally, these models usually go out of date as soon as the initial software requirements, design and implementation evolve. Figure 2.2 depicts the traditional software development life cycle, as exposed by Kleppe et al. [48].

⚠ There is a subtle difference between model-based and model-driven development, explained in Section 2.1.5.

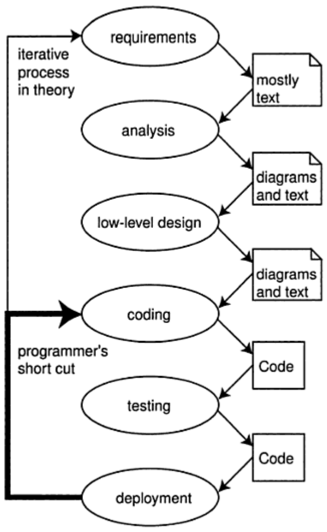


FIGURE 2.2: The traditional software development life cycle, according to Kepple et al. (Figure 1-1 from [48])

[2] The most widely known modeling language [69].

Nevertheless, by switching from a model-based development to a model-driven one in which (semi-)automation can be achieved to produce the source code to run an application, the usage of the models can be rather justified. Section 2.1.5 will further explain the key difference between the model-based and model-driven approaches.

2.1.4 What is a Meta-Model?

The concept of **meta-model** is one of the most important foundational concepts in MDD. Prior to its adoption, models were used as mere notational artefacts which helped the user to create abstractions upon which to reason and discuss. However, these artefacts required some conventions in the meaning of every visual element in the model. There was no specification about, for instance, what a square represented and what was its underlying meaning when it was drawn in a model. In other words, there was no language specification that defined the use of every visual notation element. Figure 2.3 shows an example of a model, which has a couple of squares, with some names on it, and an arrow between them. Although we could figure out that the model represents a object-oriented model, it is not certain what this model really means.

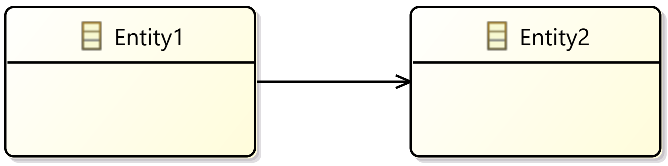


FIGURE 2.3: What does this visually notated model mean?

A meta-model is the required modeling language specification, which defines the concepts, their properties and relationships, that a modeler can make use of. Bézivin [5] explains a meta-model and its relation to *conforming* models as follows:

> "We say that the map conforms to its legend, i.e. that the map is written in the (graphical) language defined by its legend. This immediately generalizes to a model conforming to its metamodel, the second relation (*conformsTo*) associated with principle [P2].
>
> The relation between a model and its metamodel is also related to the relation between a program and the programming language in which it is written, defined by its grammar, or between an XML document and the defining XML schema."

Therefore, in order to create models, the meta-model to which they conform should be firstly defined.

More recent works about flexible modeling [52, 87] propose a different methodology in which the models and their notation are firstly created. Subsequently, the meta-model is (semi-)automatically inferred.

In our example, the meta-model could define that the modeler can make use of, for instance, *Class* elements that have a *String*-valued *name* property, and that they may refer to other *Class* elements. With that simple modeling language definition in mind, the model shown in Figure 2.3 would then make sense.

In the literature, a meta-model is also known as the abstract syntax of a modeling language.

The reader may note that the meta-model doesn't actually specify what a *square* is. That will be defined by the notation specification (also referred to as concrete syntax), including how these visual elements map to meta-model concepts, e.g. a *square* corresponds with a *Class* (see further discussion about concrete and abstract syntax in Section 2.2.4).

To conclude this subsection, we introduce the popular [5, 7] modeling stack, as described by the MDA initiative [56]. As we can see in Figure 2.4, real running systems (M0 level) are represented by models (M1 level). These models conform to their respective meta-models (M2 level). Finally, there is a third modeling layer, in which the so-called meta-meta-model (M3 level) is placed. This language is required to provide the concepts needed to create meta-models. No further modeling levels are required, because M3 can be defined in terms of itself.

2.1.5 What is *Model-Driven**?

When introduced to the *"Model-Driven"* world, different terminology and related acronyms can be found. Bambrilla et al. [10] include them in the broader concept they coin as *Model-Driven* (MD*)* (or *Model-Driven Whatever*). This subsection explains some of them.

When searching through the literature about Model-Driven* (MD*), it is very usual to find terms like Model-Driven Development (MDD), also referred to as Model-Driven Software Development (MDSD), Model-Driven Engineering (MDE) and Model-Driven Architecture (MDA). Figure 2.5 illustrates the boundaries between the different terms.

As the reader may note, all these terms have in common what was explained before: that models play a role in their underlying processes. These terms are explained as follows:

- **Model-Driven Development**. Also known as Model-Driven Software Development, it refers to the paradigm of using models as a driver (main artefact) to develop software. Once we have defined the required models, the implementation can be (semi-)automatically generated from them.

- **Model-Driven Architecture**. It is the name coined by the OMG for its *Model-Driven* initiative. It provides a particular vision about how the model-driven approach should

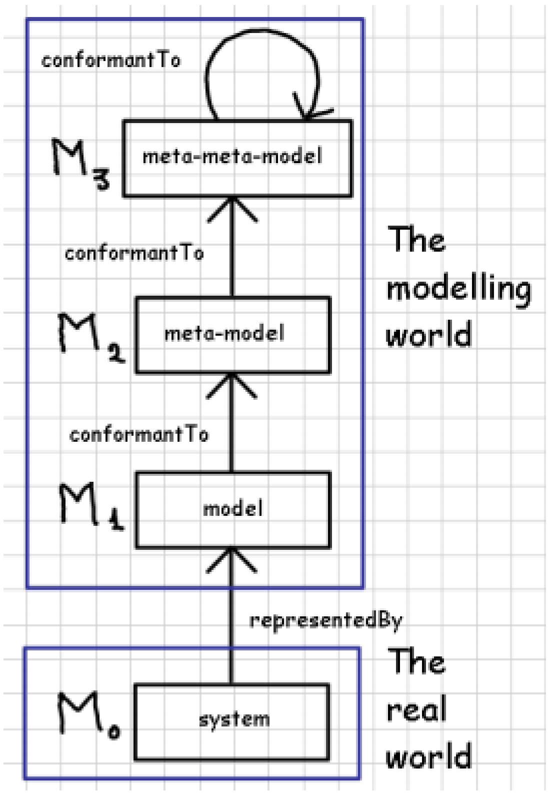


FIGURE 2.4: Modeling stack according to the MDA-initiative (Figure 5 from [5])

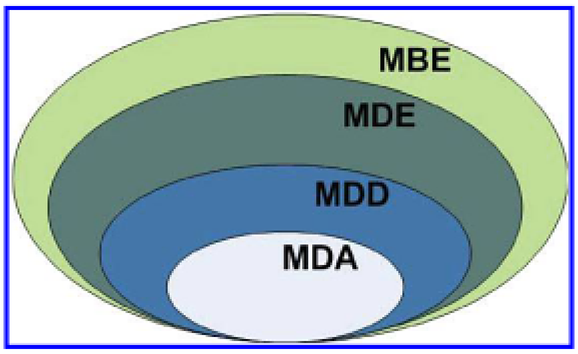


FIGURE 2.5: Boundaries of different MD-* approaches (Figure 2.1 from [10])

be undertaken (hence, a subset of MDD). The key points in the development process are the need of Computation-Independent Model (CIM)s and Platform Independent Model (PIM)s, which by means of model transformations should be transformed into Platform Specific Model (PSM)s (platform-dependent). From the latter, source code for a target platform can then be generated. Figure 2.6 depicts this particular MDD approach.

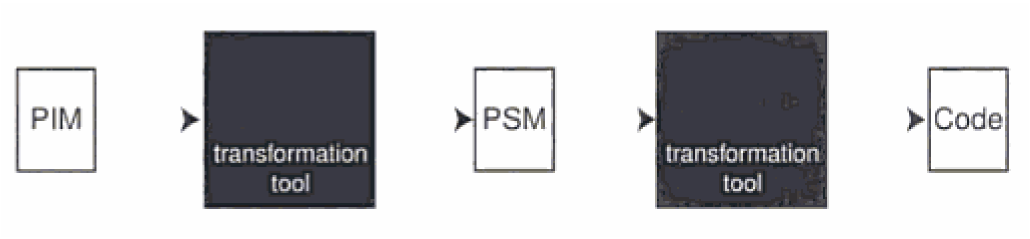


FIGURE 2.6: OMG vision of MDD, also known as MDA (Figure 1–3 from [48])

- **Model-Driven Engineering**. MDE stands for a wider concept, which again focuses on models as the main drivers for the underlying processes. In this case, other activities which are not related to the mere generation of software, such as reverse engineering or process engineering, can be scoped in this category.

- **Model-Based Engineering**. Finally, Marco Brambilla et al. [10] explain Model-Based Engineering (MBE) as "the process in which software models play an important role although they are not necessarily the key artefacts of the development". In essence, models are used as mere documentation; they are not meant to be involved in further automated processes (e.g. source code generation). Therefore, we have MDE as a particular case of MBE.

In this thesis, there is extensive use of models (and the modeling languages they conform to) and model transformations (introduced next). Although one of the ultimate goals is producing source code, hence an MDD approach, in the remainder of this thesis, MDD and MDE are used interchangeably.

2.1.6 What is a Model Transformation?

A model transformation can be defined, from a high-level point of view, as the process of producing one or more output artefacts from one or more input models, as depicted in Figure 2.7. In MDE, model transformations are critical artefacts because they provide the required automation that justifies the whole paradigm. Without model transformations, models could only play the role of documentation artefacts only.

The output artefacts could either be other models or textual artefacts [18], such as source code or some pretty documentation. In the former case, the transformations are called Model-to-Model (M2M). In the latter case, the transformations are called Model-to-Text (M2T).

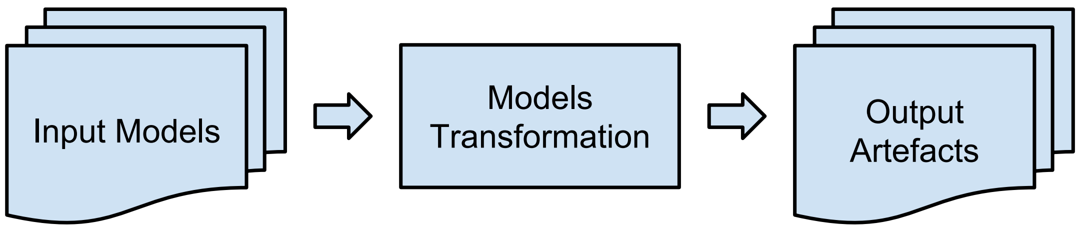


FIGURE 2.7: High-level model transformations overview

⚠ The input artefacts are usually models, although we can find particular scenarios in which they are not. For instance, in model-driven reverse engineering, models are created from source code. In this kind of scenarios, the output artefacts are models and the involved transformations are called Text-to-Model (T2M) [42].

The remainder of this subsection explains in more detail the process of an M2M transformation, which is depicted in Figure 2.8. The transformation process is driven by a transformation specification ❶, written in a model transformation language ❷. This transformation specification plays the role of an execution plan that is normally run by a transformation engine ❸. The transformation specification essentially relates concepts from the source meta-model/s ❹ to concepts from the target meta-model/s ❺. When the transformation is executed, the engine transforms the source model/s ❻ into the target model/s ❼, according to the rules defined in the transformation specification.

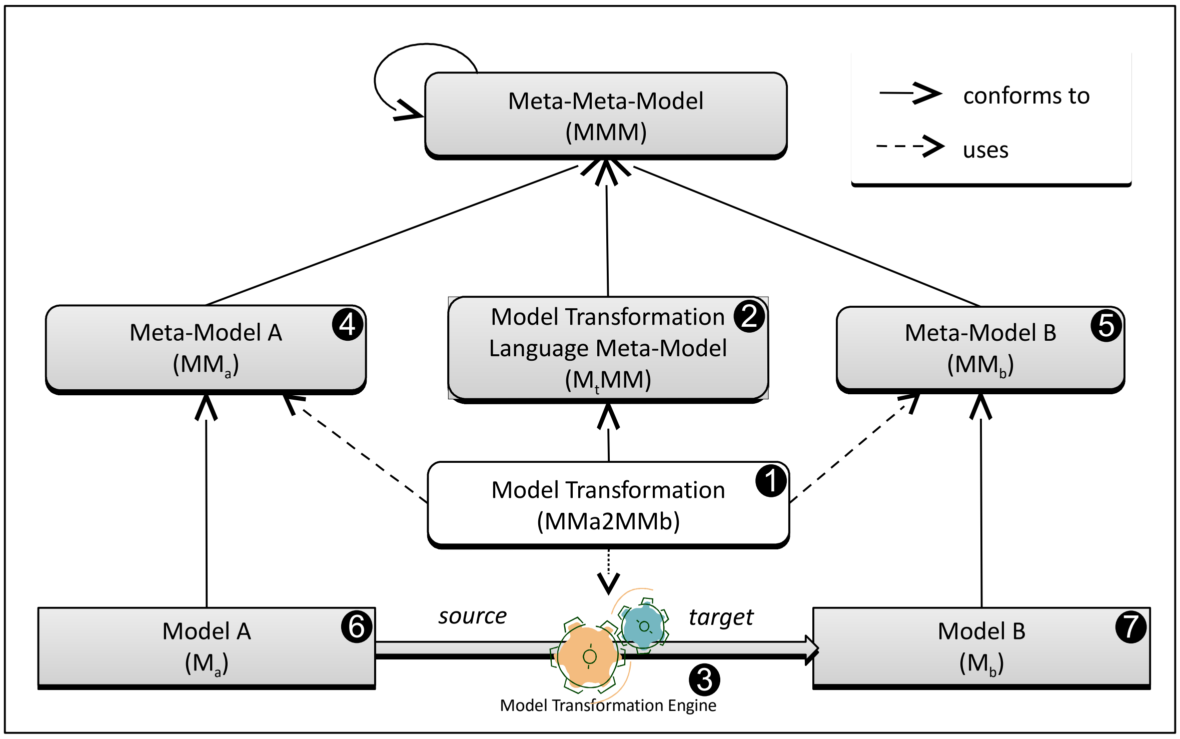


FIGURE 2.8: Detailed M2M transformation process (Figure 10.1 from [34])

Note that the source and target models and meta-models do not need to be unique. There may be many of them defined as sources or targets. Likewise, source and target meta-models do not need to be different. Additionally, for *in-place* transformations (also referred to as endogenous transformations), the source model/s are also the target ones. This kind of transformations is designed to modify existing models.

2.2 From Grammarware to Modelware

As the title of this thesis suggests, the research is focused on complex textual modeling languages. Therefore, this section introduces some concepts related to textual languages, such as grammars and *grammarware*. Additional concepts, such as *modelware*, are commented upon, along with how previous works have attempted to create bridges between *grammarware* and *modelware* (e.g. Marcus Alanen et al. [2], Wimmer et al. [81] and Eysholdt et al.[21]).

2.2.1 Introduction

As commented in Section 1.2, this thesis focuses on textual languages. In order to create instances of this kind of languages, the user will write and combine textual constructs to build a valid realisation of these languages. For instance, Java, Prolog, Matlab or OCL are examples of textual languages. Regarding this kind of language, there has been a long-standing computer science field named *Programming Language Theory*, which among other topics focuses on compilers theory for programming languages, code generation, code optimisation etc. One of the main concepts to highlight in this field is the term **Grammar**[3]. Aho et al. [1] introduce this concept as follows:

> "The analysis phase of a compiler breaks up a source program into constituent pieces and produces an internal representation for it, called intermediate code. The synthesis phase translates the intermediate code into the target program.
>
> Analysis is organised around the 'syntax' of the language to be compiled. The syntax of a programming language describes the proper form of its programs, while the semantics of the language defines what its programs mean; that is, what a program does when it executes. For specifying syntax, we present a widely used notation, called context-free grammars or BNF (for Backus-Naur Form)."

A grammar allows language designers to define the syntax of textual languages; in other words, which textual constructs are allowed to use and how to combine them in order to write correct realisations of that language (e.g. programs). Listing 2.1 shows a simple grammar excerpt in Backus Naur Form (BNF).

In traditional compilers theory these grammar definitions are used for creating parsers that, for a given programming language instance, produce intermediate code or directly the

[3] Also commonly referred to as context-free grammars.

```
exp −> exp op exp
exp −> digit
op −> + | − | ∗ | /
 digit  −> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

LISTING 2.1: Production rules of a grammar

target code to be executed by a computer. However, as a result of a Model-Driven approach, the output of these parsers will naturally be models. Therefore, from the point of view of a language engineer, there is a switch from the so-called *grammarware* technological space to the pursued *modelware* one.

In the following subsections, the origins of the *technological space* concept will be briefly explained, focusing on two specific ones: *grammarware* and *modelware*. Likewise, some important clarifications will be made regarding two main concepts related to the topic: the Concrete Syntax and the Abstract Syntax of a language. Finally, the section concludes with a survey of previous work related to bridging *grammarware* and *modelware*.

2.2.2 What are Technological Spaces?

The concept of *technological space* was introduced by Kurtev et al. [54] and was defined as follows:

> "A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings. It is at the same time a zone of established expertise and ongoing research and a repository for abstract and concrete resources."

⚠ The concept *technological space* may be also found in the literature as *technical space* [6].

In this work, authors envision the need for talking about different technological spaces and, more importantly, the need for creating bridges between them. Figure 2.9 depicts their initial proposal, including some terms of interest, such as grammars and meta-models. Nowadays, the technological spaces this research focuses on are commonly referred to as *grammarware* and *modelware*.

The concept of *grammarware*, associated to the *Abstract Concrete Syntaxes* technological space defined by Kurtev (see Figure 2.9), emerged years later in the work of Paul Klint et al. [49]. This work proposes a research agenda and highlights engineering activities around grammars. They define *grammarware* as follows:

> "We coin the term grammarware to comprise grammars and grammar dependent software:

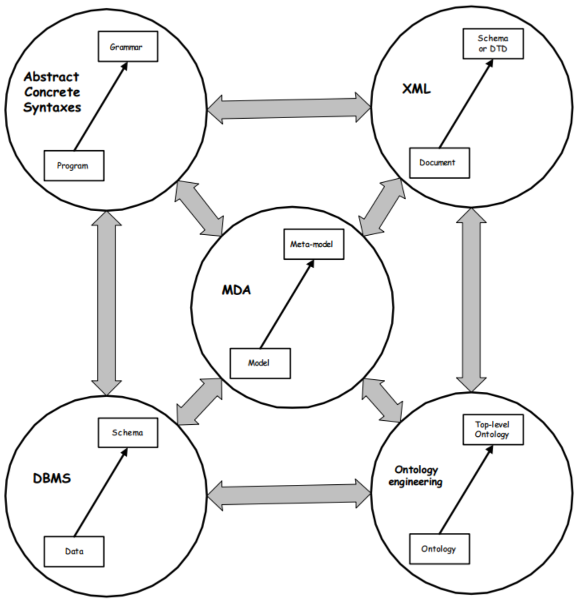


FIGURE 2.9: Technological spaces and bridges between them (Figure 1 from [54])

- The term grammar is used in the sense of all established grammar formalisms and grammar notations including context-free grammars, class dictionaries, and XML schemas as well as some forms of tree and graph grammars. Grammars are used for numerous purposes, for example, for the definition of concrete or abstract programming language syntax, and for the definition of exchange formats in component-based software applications.

- The term grammar-dependent software is meant to refer to all software that involves grammar knowledge in an essential manner. Archetypal examples of grammar-dependent software are parsers, program converters, and XML document processors. All such software either literally involves or encodes grammatical structure: compare generated versus hand-crafted parsers."

The concept of *modelware*, which is associated to the *MDA* technological space by Kurtev (see Figure 2.9), comprises all the terminology explained in Section 2.1. As a reference, Bézivin[6] introduces MDE as a kind of technological space:

"The main role of the M3-level in a technological space is to define the representation structure and a global typing system for underlying levels. The MOF for example is based on some kind of non-directed graphs where nodes are classes and links are associations. The notion of 'association end' plays an important role in this representation system."

Following up on the identification of the technological spaces of interest, the next subsection will explain the rationale behind the creation of bridges between them, and how previous work has attempted to do it.

2.2.3 Why Bridge *Grammarware* and *Modelware*?

Whilst the *grammarware* technological space has been researched since the 1960s, the *modelware* one is more recent. Therefore, the need to create bridges between both of them usually comes from MDE supporters. In essence, the need comes from textual language engineers who want to obtain models from the corresponding textual inputs.

The motivation to bridge these technological spaces is the following. On the one hand, the important role of textual languages is widely accepted [62]. On the other hand, as soon as models are obtained, engineers can benefit from all the advantages that MDE tooling provides. In these circumstances, there is a need to obtain models (i.e. *modelware*) corresponding to their textual representations (i.e. *grammarware*), which justifies the creation of bridges between these technical spaces.

Although Kurtev et al. [54] identified the need to bridge these technological spaces, the authors only provided a basis for further research. However, some works which aimed to bridge this gap started to appear afterwards, such as Marcus Alanen et al. [2]. In their technical report, they propose a relation between context-free grammars and Meta-Object Facility (MOF), so that, for instance, we can obtain Java models from Java programs and vice versa, by defining a relation between the corresponding Java BNF grammar and Java meta-model:

> "The relation between a metamodel and a BNF grammar can in practice be defined using two mappings, one transforming a BNF grammar to a MOF meta-model, and one transforming a MOF metamodel to a BNF grammar."

Other works have followed, including those of Wimmer et al. [81] and Eysholdt et al. [21]. The aim of these works is not only to establish the relations between these two technological spaces, but to enhance or complement the initial grammars with descriptive information about these relations. That information can then be used for generating vital artefacts (i.e. the meta-models) of the *modelware* technological space. For instance, Figure 2.10 shows the various bridges which apply to the different (meta-)levels between *grammarware* and *modelware*. The key point is what Wimmer et al. [81] call *Grammar Parser* (GP in the figure). By means of a correspondence between the Extended Backus Naur Form (EBNF) and MOF concepts, that grammar parser can take as input a given EBNF grammar and generate the corresponding (raw) meta-model. Additionally, from the same EBNF grammar definition, it could generate the corresponding *Program Parser* (PP in the figure). This *Program Parser* is responsible for generating, from a given program, the corresponding (raw) model conforming to the aforementioned meta-model.

Although the mentioned works are steps in the right direction, in this thesis we aim to delve into some overlooked issues that arise when dealing with complex textual languages. These issues are not solvable by their proposed approaches and they will be analysed and discussed in Chapter 3.

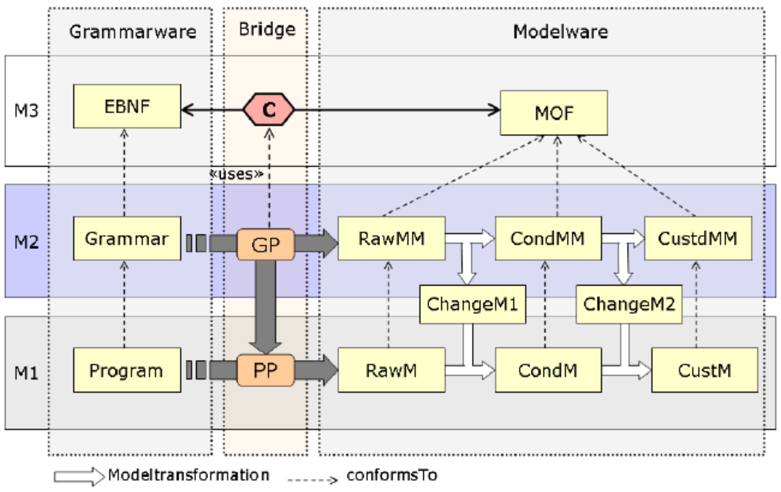


FIGURE 2.10: A *grammarware to modelware* bridging approach (Figure 1 from [81])

2.2.4 Concrete Syntax vs Abstract Syntax

The concept of syntax has been used in *grammarware* for decades and its meaning is clear. Conversely, when we are introduced to *modelware*, it is normal to find across the literature concepts such as Abstract Syntax (AS) and Concrete Syntax (CS). Similarly, whilst in *grammarware* it is common to talk about semantic analysis as part of a compiler process, in *modelware*, it is also common to find concepts such as language semantics.

Although all these concepts have similarities and correspondences between them, there are also some differences, and this normally leads to a confusing terminology and misunderstandings. In this subsection, the different terminology normally used in both technological spaces is explained, and the remainder of this thesis will adhere to these clarifications.

Terminology in *Grammarware*

When working on *grammarware*, this thesis follows the terminology defined in Aho et al. [1], a reference book in compilers theory. A language compiler will typically undertake a lexical analysis, a syntactic analysis and a semantic analysis (aside from further activities, such as code generation and code optimisation). Figure 2.11 shows these stages, and they are explained as follows:

- **Lexical analysis:** It is the process responsible for splitting the whole textual input into so-called tokens. These tokens, produced by a lexical analyser (also known as *lexer*), will constitute the further input of the parser to compute the syntax trees.
- **Syntax analysis:** After the lexical analysis has taken place, the syntax analysis is the process responsible for combining the tokens produced by that language *lexer*. The syntactic analyser (also known as *parser*) firstly produces the so-called parse trees based on the syntactic rules (defined in a grammar) of a language. The final goal is producing the so-called *Abstract Syntax Tree (AST)s* or simply *syntax trees*, which are

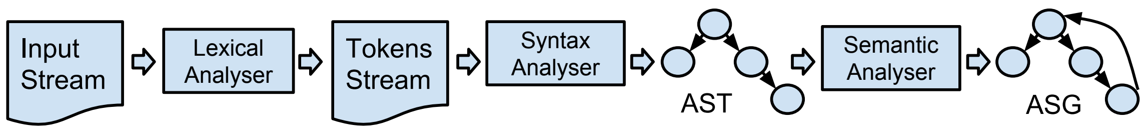


FIGURE 2.11: Lexical, syntax and semantic analysis of a compiler

created from the parse trees. These ASTs differ from the parse trees in that they suppress irrelevant tree nodes which are not needed in order to understand the syntactical structure of a textual language input, for instance, the semicolon which ends the statements in some programming languages (e.g. Java).

- **Semantic analysis:** After the syntactic analysis has taken place, a language-dependent semantic analysis might process the produced syntax trees to accomplish further activities like type checking or type conversions. These activities could lead to compiler error reports or even abstract syntax tree refinements. At this point, these trees are often referred to as *Abstract Syntax Graph (ASG)s* because they do not have a tree structure. Instead, the nodes of the abstract syntax tree may refer to other nodes; a graph rather than a tree. In the context of this semantic analysis, there is also a categorisation between static and dynamic semantic analysis. A simple *"2 + 2"* expression is used for showing the differences between them.

  Static semantics. The *"2 + 2"* expression denotes a simple sum of two operands. This expression could have a type associated, which is computed from the type of the operands. In this case, they are two integer literals, although they might be other constructs like variables or function call expressions. The semantic analysis accomplished to compute the type of an expression is considered as part of the static semantics of a language. This analysis takes place at compile time.

  Dynamic semantics. Conversely, the mentioned *"2 + 2"* expression has an intended meaning which consists of summing the two values and returning the result of the sum. This semantics, which defines how the expression is evaluated, is considered a part of the dynamic (or execution) semantics of the language. Dynamic semantics are relevant at runtime[4].

Terminology in *Modelware*

When working on *modelware*, the terms commonly used across the literature are the abstract syntax, the concrete syntax and the semantics of a language.

- **Abstract Syntax:** The AS of a language defines the concepts or entities, their properties and the relationships between them. This definition matches with the one previously used for a meta-model. Indeed, the AS of a language and a meta-model are treated as analogous in *modelware* [41]. The concept of AS (i.e. a meta-model), in *modelware*,

[4] Or by a compiler intermediate code generator.

would correspond with the result of performing the syntactic and the static semantic analysis, in *grammarware*.

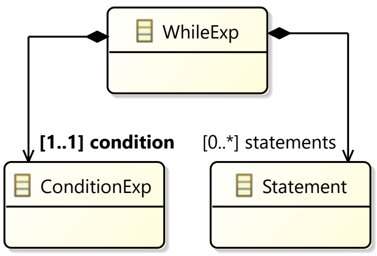
- **Concrete Syntax:** The CS of a language refers to the way that a user specifies valid instances of the abstract syntax, in other words, its notation. This could vary from textual syntax (e.g. as in traditional programming languages) to diagrammatic syntax (e.g. UML notation), or a tabular syntax (e.g. as in spreadsheets).

- **Semantics:** The semantics of a language refers to the meaning of its concepts. For example, whilst the abstract syntax of a typical expression language describes that there could be an *"add"* operator which expects two operands, the semantics of the language usually specifies that the operator computes a result by summing the first operand and the second one. In *modelware*, the semantics of a language might be specified in natural language (as in many language specifications proposed by the OMG), although it may lead to ambiguous interpretations of that semantics. Some works have emerged attempting to clarify the meaning of semantics in *modelware* [41], whereas others have worked towards a more formal way of specifying them [57]. The concept of semantics in *modelware* correspond with dynamic semantics in the *grammarware* technological space.

Figure 2.12 shows the differences between the abstract and concrete syntaxes of a language, as well as the semantics for a typical *while* expression.

```
WhileExp:
  'while'
  '('ConditionExp')'
  '{'Statements'}'
;
```

(A) CS example



(B) AS example

```
Evaluate the condition:
  – If true  =>
      evaluate statements
  – If false =>
      do nothing
```

(C) Semantics example

FIGURE 2.12: CS vs AS vs Semantics

Whilst a given language is usually defined by a unique AS, the language designer can choose among several CSs, e.g. textual or graphical notation. Moreover, instances of the same language can be manipulated by means of different CSs in a synchronised way [63].

2.2.5 Tools to Bridge Grammarware and Modelware

To conclude this section, a summary of tools designed to produce models from textual inputs is introduced. A valid classification of plausible approaches is presented by Javier Cánovas et al. [42]:

- **Dedicated parsers**. Many traditional parser generator technologies, such as YACC, ANTLR, LPG [61, 55], could be used for creating parsers which consume textual files and produce models conforming to a given meta-model. The classic/mature (see introductory Section 1.2.1) versions of Eclipse OCL/QVTo are based on this solution.

The main issue with this approach is that the language engineer needs to work on both sides, the grammar and the meta-model, and there is a considerable amount of additional hand-written source code involved in producing the conforming models. This issue turns the process of giving support to reasonable complex languages into a tedious and error-prone task.

Javier Cánovas et al. [42] also include in this category a more recent technology for reverse engineering, called MoDisco[29], which is an open source platform designed to, among other tasks, extract models from code written in different programming languages. For instance, it provides Java source file extractors (Java discoverer) via integrating the Java Development Tools (JDT) compiler and producing Java models.

- **DSL definition tools**. These more modern tools are an evolution of the traditional parser generator technologies and emerged in the context of *modelware* to give quick support to textual DSLs.

  The main improvement with respect to the previous category is that the language engineering only focuses either on the grammar or the meta-model, whereas the tool generates the other one for them. In addition, some technologies aim to provide language-specific IDEs which improve the experience of those users creating instances of their DSLs.

  With respect to these tools, Javier Cánovas et al. [42] identify two approaches which are inherent in working on bridging the two technological spaces in hand:

  Grammar-focused approach: Focusing on the grammar so that an enriched EBNF is provided, which allows one to map concepts from *grammarware* to *modelware*. See listing 2.2 as an example. With this approach, language engineers can firstly and automatically produce the output language meta-model, and secondly they are able to produce the parser[5] which will create the models conforming to that meta-model. Some examples of these tools are Xtext [30] and Spoofax [72].

  Meta-model focused approach: Focusing on the meta-model so that the additional concrete syntax information is provided in the form of annotations attached to the meta-model or as separate files with their own syntax specification languages. Similarly, the grammars to generate the parser that will create those models conforming to that meta-model can be produced. Some examples of these tools are EMFText [19] and TCS [43].

- **Program transformation languages**. Although they were not originally designed to create models, program transformation languages, such as Stratego/XT [75] and TXL [16], could be used for that purpose. However, as Javier Cánovas et al. [42] identify, "the result of a program transformation execution, is a program conforming to a grammar,

[5] Using a third party parser generator such as ANTLR [61] or LPG [55]

and a tool for bridging *grammarware* and *modelware* would still, therefore, be needed to obtain the model conforming to the target metamodel".

With the consolidation of Spoofax [46], there is an accurate identification of the role of Stratego/XT in the required *grammarware* and *modelware* bridge: a grammar-like specification language called SDF [47] is firstly used for producing a parser capable of consuming the textual inputs. According to the strategy rules defined in a specification based on Stratego/XT, models conforming to a meta-model are obtained from the outputs of that parser. A more recent publication [63] demonstrates how this *grammarware* and *modelware* bridge can be achieved within Spoofax.

- **Model Transformation Languages**. Similarly, but more specific to MDE, M2M transformation languages can also be used. As Javier Cánovas et al. [42] explain, firstly, an intermediate model (e.g. a syntax tree model) is obtained from a dedicated parser. Then, an M2M transformation can be in charge of producing the target AS models.

```
LetExpCS returns LetExpCS:
'let' variable+=LetVariableCS (',' variable+=LetVariableCS)*
'in' in=ExpCS;
;
```

LISTING 2.2: Xtext-based grammar excerpt for OCL

2.3 Language Workbenches

Although grammars and parsers are a key part of this research, modern technologies have made substantial progress in providing better tools to support a given language. These modern technologies are currently under the umbrella of a concept called Language Workbench (LW). This section focuses on explaining some notions about this concept and how it is related to this thesis.

2.3.1 Introduction

Supporting a textual language not only consists of producing the corresponding parser that produces AS models from a given textual input, but also of producing some high-quality tooling which eases the writing of textual inputs. High-quality tools usually consist of editing facilities, such as syntax highlighting, content assistant, text-folding, error markers etc. (more features exist [20]). In this section, the concept of language workbench will be introduced, focusing on a particular tool called Xtext [21]. Although the reasons for choosing this particular tool were briefly introduced in Section 1.2, they will be expanded in Section 2.3.5.

2.3.2 What are Language Workbenches?

The concept of LW was introduced by Martin Fowler in his book *Domain Specific Languages* [31]:

"Language workbenches are, in essence, tools that help you build your own DSLs and provide tool support for them in the style of modern IDEs. The idea is that these tools don't just provide an IDE to help create DSLs; they support building IDEs for editing these DSLs."

An LW is a meta-tool which helps engineers build the tools required to support a language, including the IDE to edit instances of that language. An LW can give support to languages with different kinds of notations (e.g. textual, graphical etc.). Provided that some LWs can also target textual modeling languages, they should be categorised in the classification of tools that bridge *grammarware* and *modelware* (see Section 2.2.5). In this case, they would belong to the *DSL definition tools* category, although to be more accurate, an LW can also target General Purpose Language (GPL)s.

Nowadays, we can find several types of language workbenches based on different technologies: some of them are the result of academic research and some of industrial Research and Development.

An overview of the state-of-the-art [20] bore some results on comparing different language workbenches all at once. The work provides the following contributions:

- Firstly, it contributes a feature model highlighting all the features that a language workbench should support. This feature model is shown in Figure 2.13. Detailing all these features is beyond the scope of this thesis. However, some of these features are related to the already mentioned topic of interest for this thesis: producing a high-quality editor for a particular language.

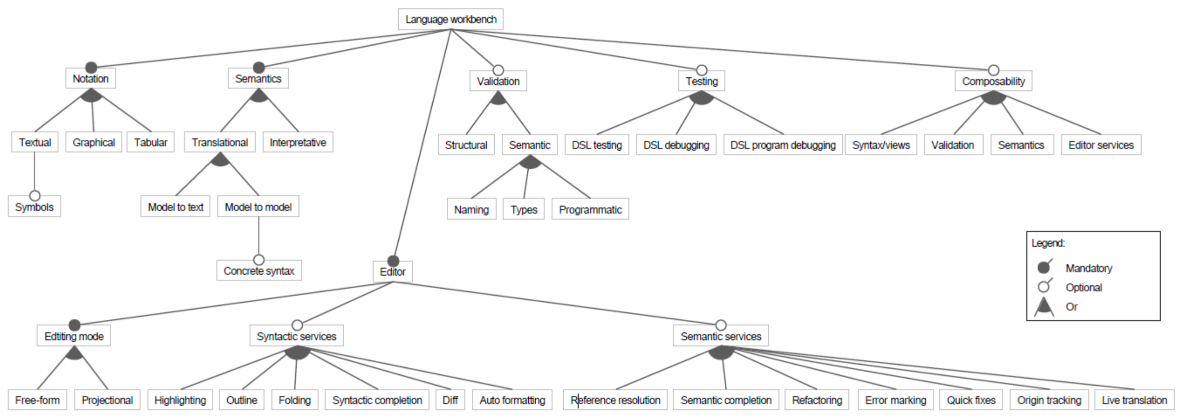
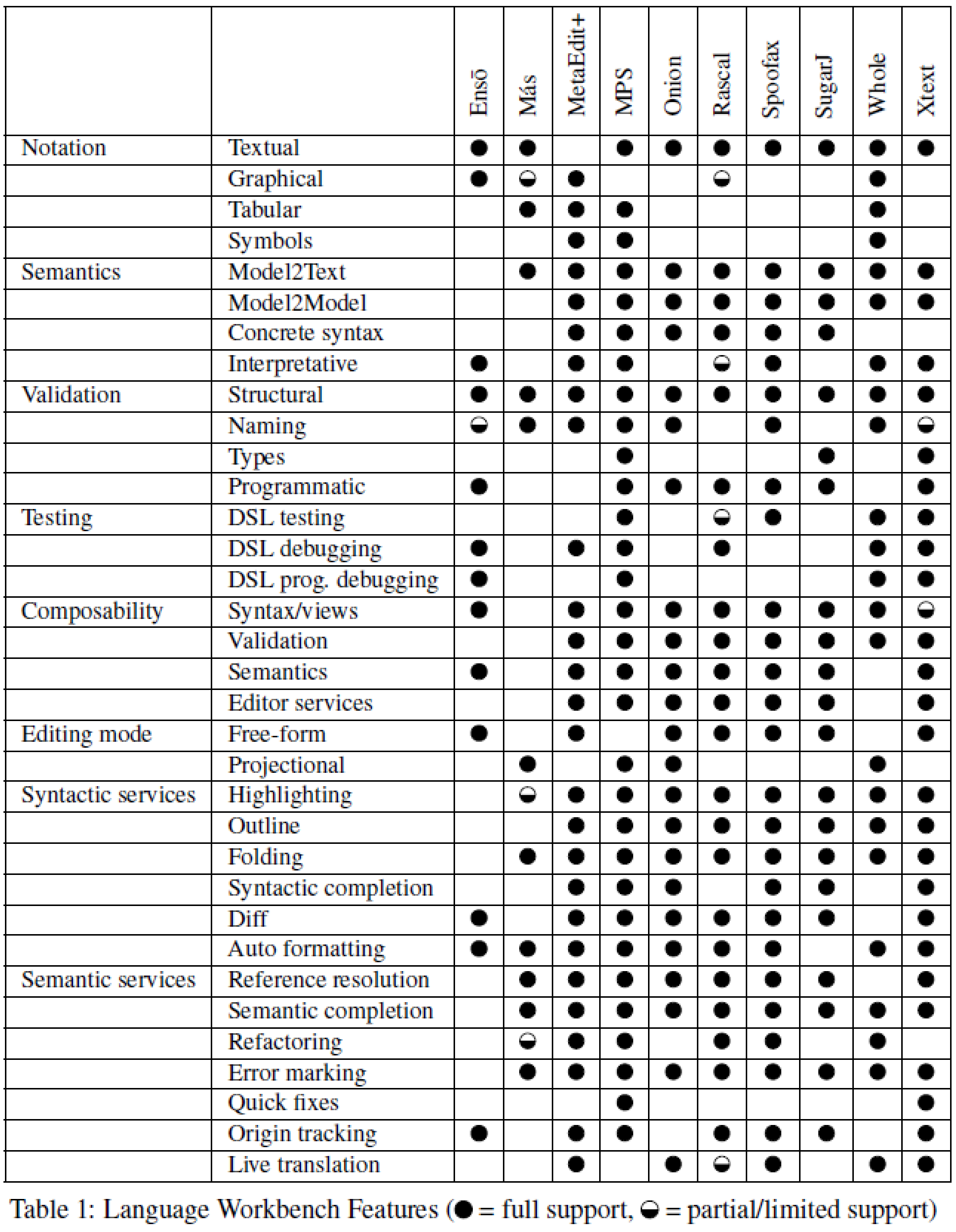


FIGURE 2.13: Feature model for LWs (Figure 1 from [20])

- Secondly, it briefly introduces ten LWs which participated in the 2013 Language Workbench Challenge [71]. Additionally, it provides a table which shows how these LWs cover the different features shown in Figure 2.13. Figure 2.14 depicts the feature model coverage by the different implementations.

  For this research, Xtext will be the target LW. According to Figure 2.14, it provides a fair coverage of the features expected for an LW. In fact, certain features that are not covered, such as providing support for graphical, tabular or symbol notations, are not of interest for this research project.

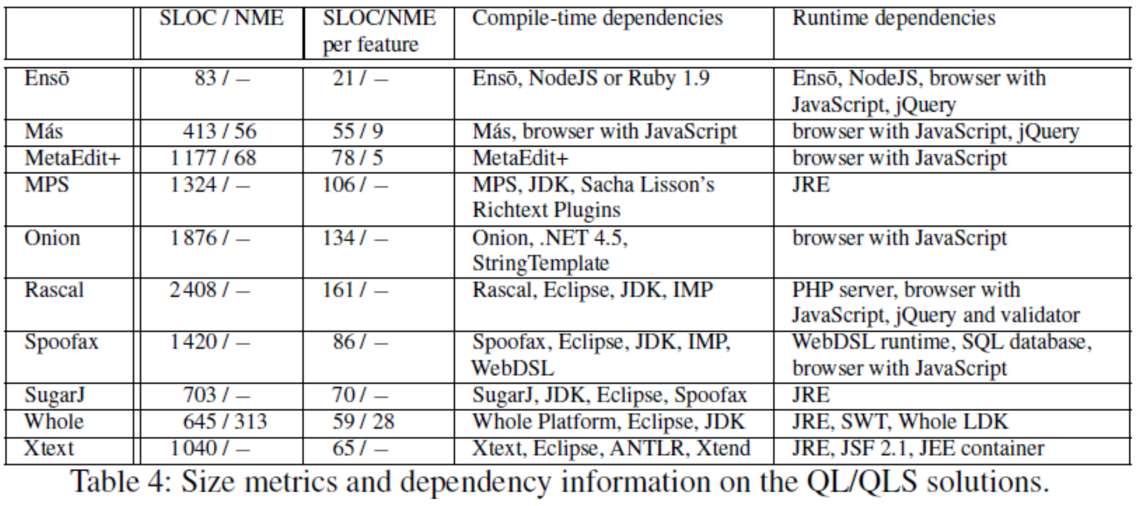


		Ensō	Más	MetaEdit+	MPS	Onion	Rascal	Spoofax	SugarJ	Whole	Xtext
Notation	Textual	●	●		●	●	●	●	●	●	●
	Graphical	●	◒	●			◒			●	
	Tabular		●	●	●					●	
	Symbols			●	●					●	
Semantics	Model2Text		●	●	●	●	●	●	●	●	●
	Model2Model			●	●	●	●	●	●	●	●
	Concrete syntax			●	●	●	●	●	●		
	Interpretative	●		●	●		◒	●		●	●
Validation	Structural	●	●	●	●	●	●	●	●	●	●
	Naming	◒	●	●	●	●		●		●	◒
	Types				●				●		●
	Programmatic	●			●	●	●	●	●		●
Testing	DSL testing				●		◒	●		●	●
	DSL debugging	●		●	●		●			●	●
	DSL prog. debugging	●			●					●	●
Composability	Syntax/views	●		●	●	●	●	●	●	●	◒
	Validation			●	●	●	●	●	●	●	●
	Semantics	●		●	●	●	●	●	●		●
	Editor services			●	●	●	●	●	●		●
Editing mode	Free-form	●		●		●	●	●	●		●
	Projectional		●		●	●				●	
Syntactic services	Highlighting		◒	●	●	●	●	●	●	●	●
	Outline			●	●	●	●	●	●	●	●
	Folding		●	●	●	●	●	●	●	●	●
	Syntactic completion			●	●	●		●	●		●
	Diff	●		●	●	●	●	●	●		●
	Auto formatting	●	●	●	●	●	●	●		●	●
Semantic services	Reference resolution		●	●	●	●	●	●	●		●
	Semantic completion		●	●	●	●	●	●	●	●	●
	Refactoring		◒	●	●		●	●		●	
	Error marking		●	●	●	●	●	●	●	●	●
	Quick fixes				●						●
	Origin tracking	●		●	●		●	●	●		●
	Live translation			●		●	◒	●		●	●

Table 1: Language Workbench Features (● = full support, ◒ = partial/limited support)

FIGURE 2.14: Features coverage by the ten LWs (Table 1 from [20])

- Finally, this comparative study provides some results on how different LWs can provide support for a specific example (see [20] for details). In justifying the rationale behind the hypotheses of this thesis, some metrics are introduced here in order to measure the solutions based on the different technologies. Figure 2.15 shows these results. As can be observed, Xtext is not the ideal solution in terms of the amount of artefacts that need to be manually produced to give support to the challenge example.



	SLOC / NME	SLOC/NME per feature	Compile-time dependencies	Runtime dependencies
Ensō	83 / –	21 / –	Ensō, NodeJS or Ruby 1.9	Ensō, NodeJS, browser with JavaScript, jQuery
Más	413 / 56	55 / 9	Más, browser with JavaScript	browser with JavaScript, jQuery
MetaEdit+	1 177 / 68	78 / 5	MetaEdit+	browser with JavaScript
MPS	1 324 / –	106 / –	MPS, JDK, Sacha Lisson's Richtext Plugins	JRE
Onion	1 876 / –	134 / –	Onion, .NET 4.5, StringTemplate	browser with JavaScript
Rascal	2 408 / –	161 / –	Rascal, Eclipse, JDK, IMP	PHP server, browser with JavaScript, jQuery and validator
Spoofax	1 420 / –	86 / –	Spoofax, Eclipse, JDK, IMP, WebDSL	WebDSL runtime, SQL database, browser with JavaScript
SugarJ	703 / –	70 / –	SugarJ, JDK, Eclipse, Spoofax	JRE
Whole	645 / 313	59 / 28	Whole Platform, Eclipse, JDK	JRE, SWT, Whole LDK
Xtext	1 040 / –	65 / –	Xtext, Eclipse, ANTLR, Xtend	JRE, JSF 2.1, JEE container

Table 4: Size metrics and dependency information on the QL/QLS solutions.

FIGURE 2.15: *2013 Language Workbench Challenge* results (Table 4 from [20])

2.3.3 Why Use Language Workbenches?

In Section 2.2, the motivation for using existing technologies to bridge *grammarware* and *modelware* was explained. In essence, in order to support textual modeling languages, there is a need for creating parsers capable of consuming textual inputs, and of producing output models. However, when working on textual languages, modern tools (e.g. editors, outlines etc.) that help to work with instances of a language are also desired. Provided that LWs not only generate these parsers from specification artefacts (i.e. a grammar), but also some default edition facilities designed to help the final user of a language, the usage of modern LWs is justified.

Projects like Eclipse OCL [24], Eclipse QVTo [27] pursue the provision of textual editing facilities. Indeed, the former currently uses an LW called Xtext to generate textual editors. The following subsections provide some more details on that.

2.3.4 What is Xtext?

Xtext is an LW, which provides facilities to build high-quality IDEs for textual languages. Figure 2.16 shows a simplified overview. By means of writing annotated EBNF-style grammars ❶ and source code generation techniques, Xtext produces source code comprising a parser ❷. This parser is capable of consuming instances of the corresponding textual language ❸. The output of the parser is a model ❹ conforming to an AS meta-model ❺ which will be auto-generated from its own grammar definition.

Xtext allows one to either auto-generate a meta-model from a grammar definition, or make the grammar work with a pre-existing meta-model.

Additionally, it will generate a high-quality editor ❻ with advanced features, such as syntax highlighting, content assistance and auto-completion, hyper-linking etc.

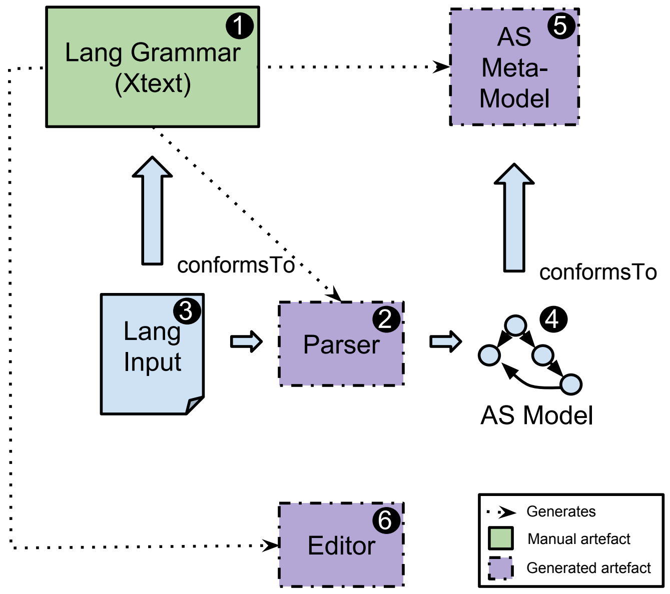


FIGURE 2.16: Brief Overview of Xtext.

2.3.5 Why Use Xtext?

Following the introduction of Xtext, this subsection discusses the rationale behind its choice as an LW.

Figure 2.14 (in Section 2.3.2) indicates that, among the compared LWs, Xtext is a good candidate for building languages based on textual CS: it does not support some features which are out of scope for that kind of languages, such as *graphical notation*, *tabular notation*, *symbol notation* and *projectional editing form*. To the contrary, it supports most of the remaining features excluding but a few: *naming validation*, *concrete syntax semantics*, *composeable syntax/views* and *refactoring editor service*.

Details about these specific unsupported features can be found in [20], although they are not relevant for the purposes of this thesis.

Further, some of the arguments introduced in Chapter 1 need to be recalled.

- **Eclipse OCL currently uses Xtext**. As stated in Section 1.2.1, QVT languages reuse Essential OCL as expressions language. A dependency between the Eclipse QVTo/QVTd projects and Eclipse OCL is natural. Indeed, many artefacts and functionalities, such as grammars and editor's facilities, can be freely reused by the Eclipse QVTo/QVTd implementation. Since Eclipse OCL grammars and editors are based on Xtext, it is also natural that Eclipse QVTo and QVTd adhere to the same technology.

- **Xtext is an official Eclipse project**. Due to the fact that Xtext is an official and mature project participating in the Eclipse simultaneous release train [28], the processes and efforts to develop and deliver the required implementation for the Eclipse QVTo and QVTd projects are simplified and decreased respectively. In fact, this has been one of the reasons why Eclipse OCL moved onto an Xtext-based implementation [80].

Although the rationale has been identified as a constraint for the research project, using Xtext as the target LW is not really an inconvenience. Instead, it actually brings some advantages:

- **Official Eclipse project**. Xtext is an official Eclipse open source project belonging to the simultaneous release train [28], which means that it has an open project plan, and an advanced and stable release schedule. This will ensure that the Eclipse OCL and QVTo components will be available and integrated with the upstream dependent projects (including Xtext) in every planned release.
- **Good documentation**. In order to use third party projects, good documentation is usually a valuable asset to consider and, indeed, Xtext provides an extensive documentation in various formats [85].
- **Active community**. As can be observed in its forum [86], Xtext has a wide and active community which is indicative that not only the project is not dead, but that one can also find support when encountering particular issues.

2.4 Related Work of Interest

To conclude the chapter, this section focuses on related work that is considered of interest to this thesis and is, therefore, referred to in later chapters. In particular, this subsection enumerates three technologies considered as relevant, and how they are linked to the corresponding thesis chapters. The rationale of why this three particular technologies are considered as relevant follows. From the wide spectrum of technologies mentioned in this thesis (e.g. see Figure 2.14), Xtext, Spoofax and Gra2Mol have documented or showed, at the time of writing this thesis, some support to CTMLs (characterised in Section 1.3).

2.4.1 Xtext

As commented in the previous section, considering Xtext as a relevant language workbench mainly stems from the motivation and scope of this EngD project, as presented in Chapter 1. Xtext can provide straightforward support to textual modeling languages, but it presents some limitations when the languages are complex. Chapter 3 shows the Xtext limitations in solving some identified concerns presented in that chapter. Additionally, Xtext is involved in the approach presented in Chapter 4, as part of the complete solution for supporting CTMLs (editors, parsers etc.).

2.4.2 Spoofax

Spoofax is another language workbench designed to give support (e.g. parsers, editors) to complex textual languages. Although it was not originally designed to work with modeling languages, there is work [63] demonstrating that Spoofax can be used for this purpose. Spoofax provides the means to overcome some of the identified concerns described in Chapter 3. Therefore, Chapter 5 presents a qualitative study to compare in depth Spoofax and the proposed solution.

More specifically, Chapter 5 introduces Spoofax languages and features, explaining which of the identified concerns described in Chapter 3 are actually addressed. Finally, it discusses how the proposed solution differs to Spoofax taking into account features such as parsing technology dependency, language nature, name resolution support, etc.

2.4.3 Gra2Mol

Gra2Mol is a text-to-model technology used for software modernization. It is designed to produce models from textual inputs, including instances of CTMLs. However, it is not a language workbench. Gra2Mol provides a DSTL to overcome some of the identified concerns described in Chapter 3. Therefore, qualitative and quantitative studies to evaluate in depth Gra2Mol and the proposed solution are presented in Chapter 5.

More specifically, Chapter 5 introduces the Gra2Mol language and features, explaining which of the identified concerns described in Chapter 3 are actually addressed. As done with Spoofax, it discusses how the proposed solution differs to Gra2Mol taking into account features such as parsing technology dependency, language nature, name resolution support, etc. Additionally, Section 5.4, Section 5.5 and Section 5.6 provide a detailed quantitative study which compares Gra2Mol and the proposed solution in terms of performance (particularly, execution time of CS2AS transformations) and the size of artefacts (particularly, transformation language instances).

2.5 Summary

Chapter 2 presented a survey of the field related to this thesis. It focused on three main topics: Model-Driven Development, bridging *grammarware* and *modelware*, and Language Workbenches.

MDD is an area of software engineering that proposes models as the first class artefacts to use during the software development process. Several concepts such as models, meta-models and model transformations were explained.

Grammarware and *Modelware* were introduced as two types of technological spaces. The former focuses on textual languages and grammars, whilst the latter focuses on models and meta-models. Some important clarifications about concepts such as concrete syntax, abstract syntax and semantics were explained. Since this thesis focuses on creating textual modeling languages, which require bridging *grammarware* and *modelware*, different approaches (and related technologies) to achieve this goal were introduced.

The concept of LW was introduced. This kind of modern tool allows language engineers to generate fit-for-purpose editors. Since these editors help final users to manipulate instances of a language, the adoption of LWs becomes justified. Due to the context and scope of this thesis, the choice in favour of Xtext as the target LW was also clarified.

Finally, the relevant work that is discussed throughout this thesis was described. Whilst Xtext is part of the solution (Chapter 4), Spoofax and Gra2Mol are used for evaluating the proposed approach (Chapter 5).

Chapter 3

Problem Analysis

In Chapter 1, the problem that has motivated this research was introduced. This chapter provides a deeper analysis of the different concerns that need to be addressed to bridge the gap between the CS and AS of CTMLs. Section 3.1 summarises key concepts from *grammarware* and *modelware*. Section 3.2 introduces a running example that will be used throughout the rest of the thesis. Section 3.3, Section 3.4 and Section 3.5 analyse the different concerns identified during this research. Before introducing the solution (Chapter 4) that addresses these concerns, Section 3.6 shows how Xtext can cope with some but not all them. Finally, Section 3.7 concludes the chapter.

3.1 Summary of Required Concepts

To understand the problem analysis adequately, this section introduces a brief summary of key concepts related to grammars and meta-models.

When referring to grammar terminology (the source involved in a CS2AS bridge), the following concepts [1] are relevant:

- **Terminals**, sometimes referred to as **tokens**, are the textual units into which a textual input is split. Their definitions are language dependent and they comprise the leafs of the so-called syntax trees (produced by a parser). Some examples of tokens might be *12* (an integer number), *var1* (an identifier) or *"foo"* (a string).

- **Keywords** are special terminals (tokens), whose main characteristic is that they cannot be used as normal identifiers (which are used for defining, and referring to, named elements, e.g. a variable called *var1*). For instance, *class* and *package* are keywords in Java.

- **Non-terminals** are the syntactic variables, used within a context-free grammar, that define and participate in the syntactic rules of the grammar. They represent a set of strings of the language. For instance, the syntactic rule of a variable declaration in Java could be *VarDecl –> TypeRef ID ('=' VarInit)?*. This syntactic rule comprises two terminals – the token *ID* (an identifier) and the symbol *'='*–, and three non-terminals –*VarDecl*, *TypeRef* and *VarInit*–. The non-terminal definition *VarDecl* represents a set of language strings, such as *'int count'* or *'String foo = "bar"'*.

- **Left Hand Side (LHS) non-terminals** are the non-terminals that appear at the left of the arrow of a syntactic rule, i.e. the non-terminal that defines the syntactic rule. For instance, in the previous example, *VarDecl* is the LHS non-terminal.
- **Right Hand Side (RHS) non-terminals** are the non-terminals that appear at the right of the arrow of a syntactic rule, i.e. the non-terminal(s) that participate in the syntactic rule definition. For instance, in the previous example, *TypeRef* and *VarInit* are RHS non-terminals.

When referring to meta-model terminology (the target involved in a CS2AS bridge), the following concepts [70] are relevant:

- **(Meta-)Classes** are the concepts involved in a model, i.e. the kind (or type) of model elements. For instance, a *Department* may be a (meta-)class denoting the type of *Computer Science* and *Biology* model elements.

  Model elements are instances of a (meta-)class. For example, in this thesis, an instance of *Department* can be referred to as a *Department* model element. In other words, *Computer Science* and *Biology* are instances of the *Department* metaclass.

- **Data types** are the kind of (primitive) values involved in a model, i.e. the kind of values that a model element can hold. For instance, *Integer* might be a data type denoting the type of integer values, such as *46*.
- **Properties** are the additional features of (meta-)classes. They may be either **Attributes** or **References**.
- **Attributes** are a kind of property with data type values for model elements. For instance, *age* may be a *Department* attribute to denote that the *Computer Science* department is *46* years old.
- **References** are a kind of property with (meta-)class reference values. They may be either **Containment References** or **Cross-References**.
- **Containment References** are a kind of reference that implies an 'owning' (or containing) relationship between the source and target model element, so that the former owns (or contains) the latter. This relationship has strong implications. For instance, if a model element owns (or contains) another one, whenever the former is deleted, the latter should also be deleted. Another important implication is that a model element can be referred to by at most one containment reference. In other words, a model element can be owned (or contained) by only one model element[1]. For instance, a *Department* (meta-)class may have containment reference *modules*, so that the *Computer Science* department may contain a *Theory and Practice of Programming* module.

[1]Model elements cannot contain themselves.

- **Cross-References** are a kind of reference that does not imply owning; therefore, model elements refer to existing model elements located anywhere in the model[2]. An important characteristic is that the same model element may be referred to by many model elements, via the same (or even different) cross-reference. For instance, a *Deparment* (meta-)class may have a cross-reference *collaboratesWith*, so that the *Computer Science* department may collaborate with the *Biology* department.

3.2 Running Example: Mini OCL

This section introduces the target language that is used throughout the thesis as the running example. This target language is called Mini OCL (mOCL) and consists of a restricted subset of OCL. The following subsections explain the rationale behind using Mini OCL (mOCL), as well as a description of the language.

3.2.1 Why Mini OCL?

The rationale behind creating and using a new language (rather than using OCL directly) is the following:

- mOCL comprises the essence of concerns that need to be addressed for OCL. Whilst OCL is a substantial language with a wide variety of expressions, many of them are irrelevant (or duplicated) with respect to the concerns that are explained throughout this chapter. mOCL will be used in this thesis to show a complete solution for a CTML, without the need to inflate the thesis with the additional exposition that supports the irrelevant[3] or duplicated parts of OCL. Likewise, a smaller language is better for debugging and testing.

- Whilst being a smaller language than OCL, mOCL captures the essence of OCL as a language to define queries and constraints on models. Conversely, it provides the opportunity to add useful syntax to explain the analysed concerns. In this case, mOCL additionally provides some basic support to define meta-models textually.

- Although providing a solution for a smaller language may bring external validity threats (i.e. can the proposed solution deal with the entire OCL language?), these threats are addressed in the next chapter (Section 4.2.10). After explaining the solution, Section 4.2.10 justifies why the missing OCL concepts can be excluded from mOCL (see Section 4.2.10).

3.2.2 Mini OCL Description

This subsection describes mOCL, which reuses a subset of OCL constructs and introduces new ones. Rather than inflating this thesis with a detailed CS and AS of the language, the

[2]Model elements could (cross-)refer to themselves.
[3]From the point of view of the presented concerns that arise in implementing CS2AS bridges.

constructs are shown by means of examples. As a reference, Appendix A shows the whole CS grammar definition and Appendix B shows the AS meta-model.

Some knowledge about OCL [39] specification is assumed.

Meta-model Import

mOCL supports meta-model import as a new construct. Imports can be declared within mOCL files with the goal of explicitly making meta-models available. In this way, the different constraints and expressions can refer to the imported packages, classes, operations and properties. The imported meta-model may declare an alias. Listing 3.1 shows an example of the notation.

```
import mm : './aMetamodel.xmi';
```

LISTING 3.1: Meta-model import notation

Meta-model Definition

mOCL supports meta-model definition as a new construct. Alternative to importing meta-models, meta-models can be explicitly defined with a simple notation. The notation allows defining packages, classes, properties and operations. Properties can declare its multiplicity (by default, they are optional and single-valued). Operation bodies consist of a single expression whose context is the class in which they are defined. Listing 3.2 shows an example of the notation.

```
package APackage {
  class AClass {
     prop p1 : AnotherClass[1];
     op getAString() : String = p1.getProp();
  }
  class AnotherClass {
     prop aString : String;
     op getProp() : String = self.aString;
  }
  class ClassExtension extends AClass {}
}
```

LISTING 3.2: Meta-model definition notation

Constraint Definition

OCL provides notation to declare constraints. mOCL provides a similar notation for the same purpose. In this case, only invariant definitions on meta-classes can be declared. Listing 3.3 shows an example of the notation.

```
context APackage::AClass {
  inv : getAString() <> null;
}
```

LISTING 3.3: Constraint definition notation

Types

OCL provides a non-trivial type system; in particular, when dealing with collections and the uncommon[4] invalid value. From the point of view of bridging the CS and AS, the complexity of the type system only impacts on the following:

- Computing expression types. The more complex the type system is, the more complex the required logic to compute the expression types (e.g. more complex type conformance rules).
- Special CS2AS scenarios. There are some special CS2AS scenarios that are highly coupled to the type system (particularly, for expression types).

The mOCL type system has been reduced to support the type system-related CS2AS scenarios that are present in OCL. The decision of simplifying the type system for mOCL does not compromise the ability of the proposed solution to be used for a more complex language like OCL. This assertion is properly discussed in Section 4.2.10.

The mOCL built-in type system consists of:

- *String Boolean* and *Integer* primitive types to represent some primitive values.
- *OclAny* type to represent all possible model elements and primitive values.
- *OclVoid* type to represent the absence of value/model element.
- *Collection* type to represent a collection of model elements or primitive values.
- Finally, all user meta-classes (e.g. classes defined in a meta-model definition) are also types within the language.

Literal Expressions

As with OCL, mOCL provides literal expressions to specify the values corresponding to several types of the built-in type system. With respect to collection literals, collection ranges can be used. Listing 3.4 shows an example of the notation of different literal expressions.

```
true                          -- a boolean literal expression
10                            -- an integer literal expression
Collection {1 ,2, 3..10}    -- a collection literal expression
null                          -- a null literal expression (of type OclVoid)
```

LISTING 3.4: Literal expressions notation

[4]When compared to traditional object-oriented languages.

Let Expressions

As with OCL, mOCL supports let expressions to provide reusable variable declarations that can be referred to by inner expressions. Listing 3.5 shows an example of the notation.

```
let foo : String = 'foo',
    bar = foo
in bar
```

LISTING 3.5: Let expressions notation

Variable Expressions

As with OCL, mOCL supports variable expressions to refer variables that have been previously defined. Candidate variables are the implicit contextual *'self'* variable, operation parameters or variables defined in outer let expressions. Listing 3.6 shows several examples.

```
class AClass {
  prop a : Integer;
  op same(b : Integer) : Integer =
    let letVar = self.a -- Variable expression referring to 'self' context variable
    in letVar =         -- Variable expression referring to 'letVar' let variable
       b;               -- Variable expression referring to 'b' parameter
}
```

LISTING 3.6: Variable expressions notation

Call Expressions

As with OCL, mOCL supports model navigation and queries by means of call expressions. These call expressions are used on a source expression which evaluates to a model element, a (primitive) value or a collection. There are three kinds of call expressions:

- Property call expressions to navigate through model elements. The source has to be a model element.
- Operation call expressions to invoke an operation. The source can be model elements, values or collections.
- Loop expressions to iterate over a collection. The source has to be a collection.

⚠ The loop expressions included in mOCL are the *collect* iterator expression and the *iterate* expression.

mOCL also provides two kinds of operators for these call expressions: the '.' symbol is used for either navigating through a model element, or invoking an operation on a model element or a (primitive) value. The '->' symbol is used for either invoking an operation on a collection or iterating over a collection. Listing 3.7 shows several examples.

```
self.aProperty                    -- a property call expression on a model element
1.max(2)                          -- an operation call expression on a primitive value
Collection{1,2}->size()                  -- an operation call expression on a collection
Collection{1,2}->collect(x | x = 1)   -- collect iterator expression on a collection
Collection{1,2}->iterate(x : Integer; acc : Integer = 0
                            | acc.sum(x)) -- iterate expression on a collection
```

LISTING 3.7: Call expressions notation

Although these rules concerning the call expressions operators are easy to understand, mOCL is complicated because the syntax permits using the '.' operator with collections, as well as the '->' operator with single model elements or values. In the former case, there is an implicit *collect* iteration expression, so that the call expression is performed on every single element of the collection. In the latter case, there is an implicit collection conversion, so that the call expression is performed on a new collection with the single model element (or primitive value). Listing 3.8 shows examples of the notation, where the expression in line 1 is equivalent to the expression in line 2, and the expression in line 3 is equivalent to the expression in line 4.

```
1 self.multipleValueProperty.size() -- equivalent to the next expression
2 self.multipleValueProperty->collect(x | x.size())
3 self.singleValueProperty->size()  -- equivalent to the next expression
4 self.singleValueProperty.asCollection()->size()
```

LISTING 3.8: Call expressions with syntactic sugar

Equality Expressions

mOCL has reduced the set of expressions to deal with one kind of binary expression, in this case an equality to test if two operands are equal (or distinct) between them. Listing 3.9 shows an example.

```
1 <> 2           -- simple equality expression that evaluates to true
1 <> 2 = false   -- more complex equality expression that evaluates to false
```

LISTING 3.9: Equality expressions notation

3.3 Bridging Concrete and Abstract Syntax: Coarse Grained Analysis

Now that the running example has been introduced, this section analyses the concerns to be addressed when bridging the gap between the CS and AS of CTMLs. In particular, this section makes a coarse grained classification of the kind of mappings that are required to bridge that gap. This classification will be explained by showing particular examples of mapping scenarios that need to be addressed for mOCL.

> ⚠ In this and future chapters, the concept of **mapping** will be used for relating CS terms to AS terms. In other words, bridging the gap between the CS and AS of a language will be described by defining mappings between CS and AS terms.

On the one hand, this thesis targets languages whose CS is textual. Therefore, this section shows the different mapping scenarios by referring to grammar terms (e.g. terminals and non-terminals), as the source CS concepts to be mapped. On the other hand, this thesis targets languages whose AS is defined by meta-models. Therefore, this section shows the different mapping scenarios by referring to meta-model terms (e.g. meta-classes and their properties), as the target AS concepts to be mapped.

The following subsections will go into detail about the presented coarse grained classification:

- **1-to-1 mappings**, in which each CS grammar term can be directly mapped to a single AS meta-model term.
- **N-to-1 mappings**, in which many CS grammar terms can be mapped to a single AS meta-model term.
- **1-to-N mappings**, in which a single CS grammar term can be mapped to many AS meta-model terms.
- **N-to-M mappings**, as a generalisation of the previous ones, in which many CS grammar terms can be mapped to many AS meta-model terms.

3.3.1 1-to-1 Mappings

This subsection explains 1-to-1 mappings, in which each CS grammar term is directly mapped to an AS meta-model term. The following example is used for exposing the mapping scenario.

As shown in Section 3.2.2, mOCL supports *Package* definitions. Listing 3.10 shows the relevant excerpt of the CS grammar in EBNF notation.

```
PackageCS:
  'package' ID '{'
     (ClassCS | PackageCS)*
  '}'
;
```

LISTING 3.10: CS of a *Package* definition

According to the CS, the syntactic rule for the non-terminal *PackageCS* consists of the keyword *'package'* followed by the token *ID*, followed by the token *'{'*, optionally followed by a number of non-terminals (either a *ClassCS* or another *PackageCS*), and finally followed by the token *'}'*.

The relevant abstract syntax of a *Package* definition is shown in Figure 3.1.

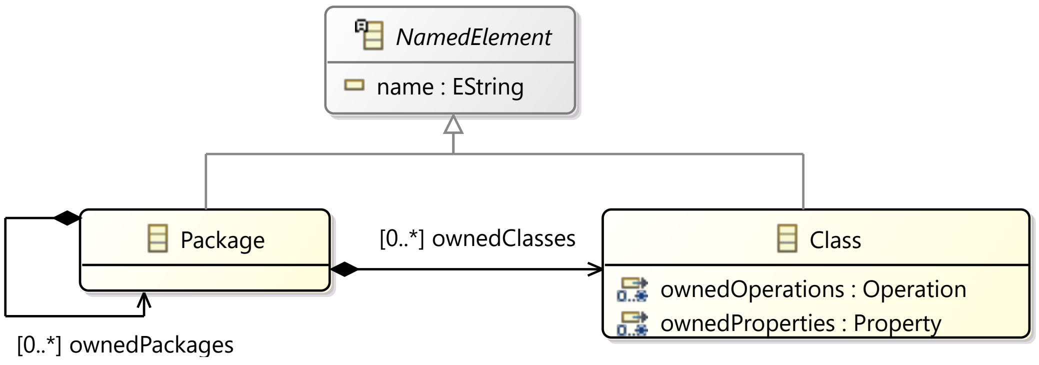


FIGURE 3.1: AS of a *Package* definition

According to the AS, a *Package* definition has a *name*, and may contain a number of *Classes* and/or *Packages*.

This example requires trivial 1-to-1 mappings between grammar and meta-model terms, because all AS terms are mapped from a single CS term. For instance:

- The LHS non-terminal *PackageCS* maps to the *Package* meta-class.
- The terminal *ID* maps to the *name* attribute of the *NamedElement* meta-class.
- The RHS *ClassCS* maps to the *ownedClasses* reference of the *Package* meta-class. In this way, any *Class* model element obtained by a further *ClassCS-to-Class* mapping can be referred to via this *Package::ownedClasses* reference.
- The RHS *PackageCS* maps to the *ownedPackages* reference of the *Package* meta-class. In this way, any (nested) *Package* model element obtained by the mentioned *PackageCS-to-Package* mapping can be referred to via that *Package::ownedPackages* reference.

According to these mapping definitions, Figure 3.2 shows an example of a textual input of a *Package* definition, and the expected AS model.

3.3.2 N-to-1 Mappings

This subsection explains N-to-1 mappings, in which many CS grammar terms are mapped to a single AS meta-model term. The following example is used for exposing the mapping scenario.

As introduced in Section 3.2.2, mOCL supports *Class* definitions. In particular, a *Class* may extend to additional *Classes*. Listing 3.11 shows the relevant excerpt of the CS grammar in EBNF notation.

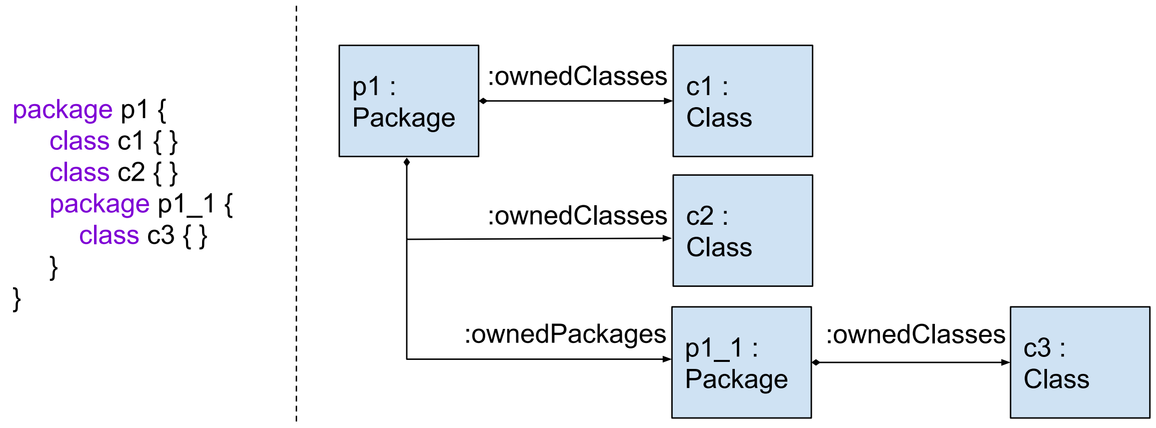


FIGURE 3.2: Example: *Package* notation and corresponding AS model

```
ClassCS:
  'class' ID ClassExtensionCS? '{'
    // the remainder of Class definition syntax has been omitted
  '}'
ClassExtensionCS:
  'extends' PathNameCS (',' PathNameCS)*
PathNameCS:
  PathElementCS ('::' PathElementCS)*
PathElementCS:
  ID
```

LISTING 3.11: CS of a *Class* extension definition

According to the CS, a *Class* definition may optionally include extension definitions. The syntactic rule for the non-terminal *ClassExtensionCS* consists of the keyword *'extends'* followed by a non-terminal *PathNameCS*, and optionally followed by a comma-separated, undetermined number of additional *PathNameCS*. The syntactic rule for the non-terminal *PathNameCS* consists of the non-terminal *PathElementCS*, optionally followed by a double colon-separated, undetermined number of additional *PathElementCS*. Finally, the syntactic rule for the non-terminal *PathElementCS* consists of the token *ID*.

The relevant abstract syntax of a *Class* definition is shown in Figure 3.3.

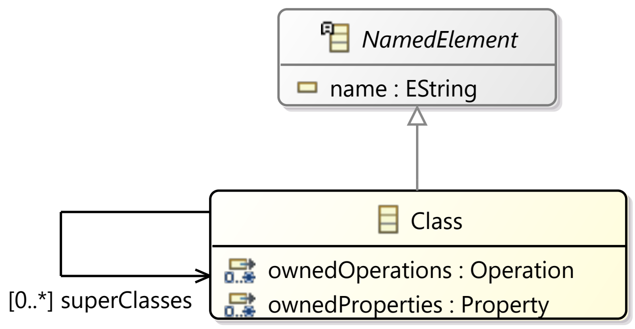


FIGURE 3.3: AS of a *Class* extension definition

According to the AS, a *Class* may optionally refer to a number of additional *Class*es, via the *Class::superClasses* reference.

This example conforms to an N-to-1 mapping between grammar and meta-model terms:

- A set of LHS non-terminals, *PathNameCS* and *PathElementCS*, maps to the *Class* meta-class. In this way, the *Class* model element that corresponds to a particular textual input – which conforms to the *PathNameCS* syntactic rule – can be referred to by the *Class* on which the class extension is defined, via that *Class::superClasses* reference.

According to this mapping definition, Figure 3.4 shows an example of a textual input of a *Class* extension definition, and the corresponding AS model.

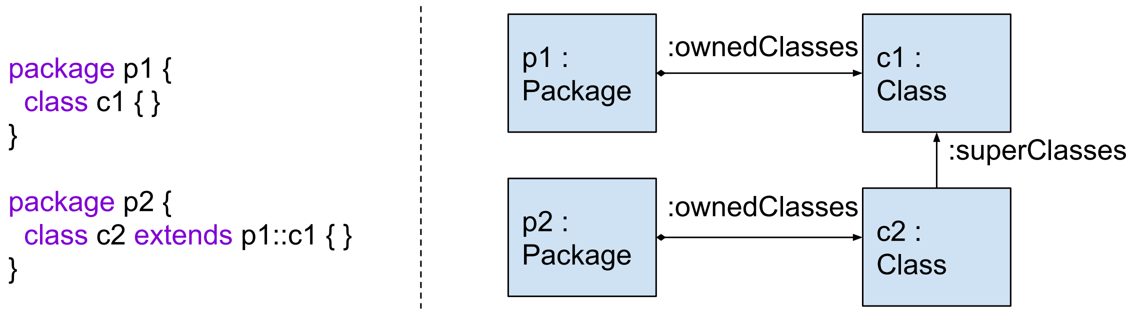


FIGURE 3.4: Example: *Class* extension notation and corresponding AS model

3.3.3 1-to-N Mappings

This subsection explains 1-to-N mappings, in which a single CS grammar term is mapped to many AS meta-model terms. The following example is used for exposing the mapping scenario.

As introduced in Section 3.2.2, mOCL supports *Operation* definitions. In particular, we are interested in the definition of the operation body. Listing 3.12 shows the relevant excerpt of the CS grammar in EBNF notation.

```
OperationCS:
  'op' ID '(' ParametersDeclarationCS ')'
  ':' PathNameCS
  '=' ExpCS ';'
```

LISTING 3.12: CS of an *Operation* definition

According to the CS, the syntactic rule for the non-terminal *OperationCS* starts with the keyword *'op'*. The non-terminal related to the operation body is *ExpCS*, which follows the token '=' and precedes the token ';'. This non-terminal comprises an arbitrary expression, which is the body of the operation.

The relevant abstract syntax of an *Operation* definition is shown in Figure 3.5.

According to the AS, an *Operation* owns an *ExpressionInOCL*, which owns a self *Variable* and the target *OCLExpression* – i.e. the operation body expression.

This example requires a 1-to-N mapping between grammar and meta-model terms:

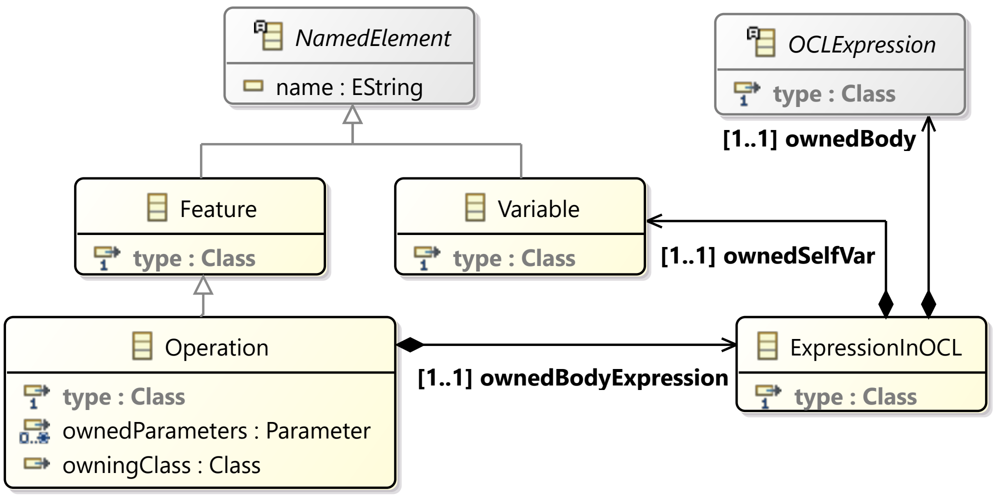


FIGURE 3.5: AS of an *Operation* body definition

- The RHS non-terminal *ExpCS* maps to the *ownedBodyExpression* reference of the *Operation* meta-class. However, the scenario also requires mapping to additional meta-classes, such as *ExpressionInOCL* and *Variable*. In this way, any *OCLExpression* model element obtained by a further *ExpCS-to-OCLExpression* mapping can be adequately contained in the corresponding *ExpressionInOCL* model element, via the *ExpressionInOCL::ownedBody* reference. Likewise, this *ExpressionInOCL* model element can be contained via the aforementioned *Operation::ownedBodyExpression* reference.

.

Figure 3.6 shows an example of a textual input of an *Operation* body definition, and the corresponding AS model.

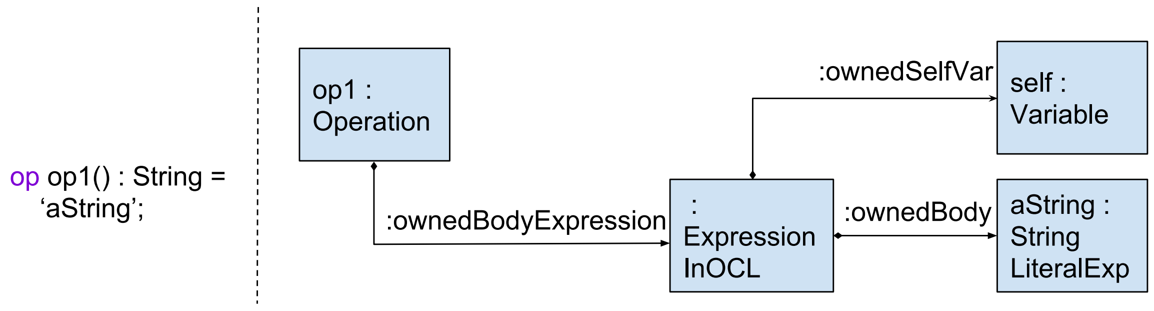


FIGURE 3.6: Example: *Operation* body notation and corresponding AS model

3.3.4 N-to-M Mappings

Finally, this subsection explains N-to-M mappings, in which, as a generalisation of the previous kind of mappings, many CS grammar terms are mapped to many AS meta-model terms. The following example is used for demonstrating the mapping scenario.

As introduced in Section 3.2.2, mOCL supports property call expressions (i.e. model property navigation). Listing 3.13 shows the relevant excerpt of the CS grammar in EBNF notation.

```
CallExpCS:
  PrimaryExpCS (('.' | '->') NameExpCS)∗
NameExpCS:
  PathNameCS (RoundedBracketClauseCS)?
PathNameCS:
  PathElementCS ('::' PathElementCS)∗
PathElementCS:
  ID
```

LISTING 3.13: CS of a *PropertyCallExp*

According to the CS, the syntactic rule for the non-terminal *CallExpCS* starts with the non-terminal *PrimaryExpCS*, optionally followed by a number of call-operator separated *NameExpCS*. The latter consists of a non-terminal *PathNameCS*, optionally followed by a non-terminal *RoundedBracketClauseCS* (used for operation call expressions).

The relevant abstract syntax of a *PropertyCallExp* is shown in Figure 3.7.

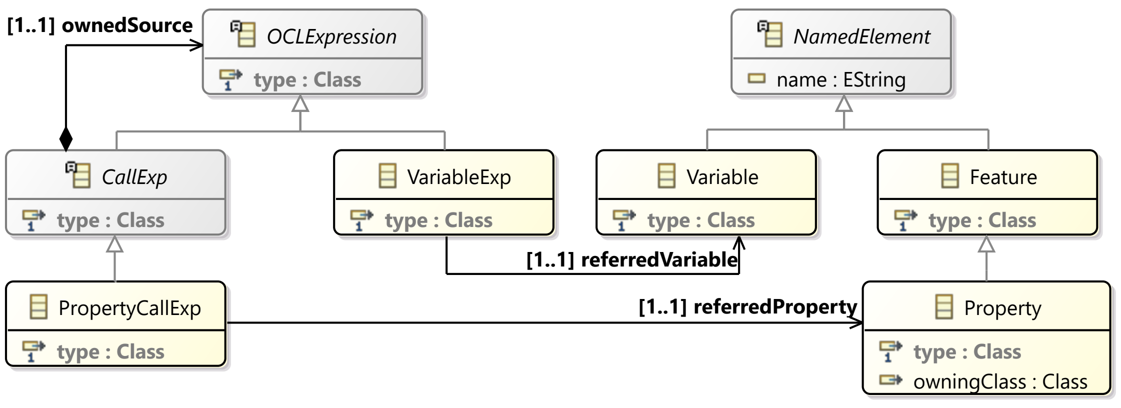


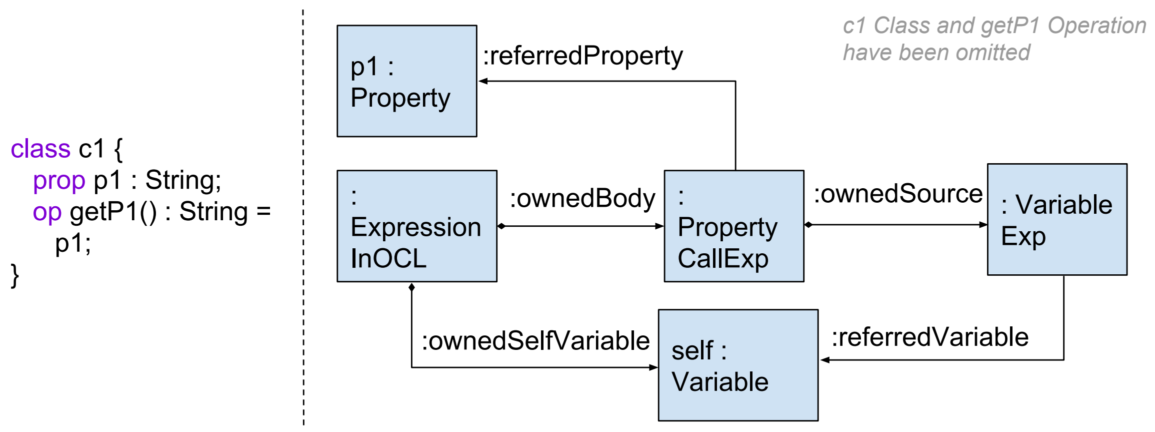
FIGURE 3.7: AS of a *PropertyCallExp*

According to the AS, a *PropertyCallExp* owns another *OCLExpression* via the *CallExp::ownedSource* reference, and it refers to a *Property* via the *PropertyCallExp::referredProperty* reference.

This example requires an N-to-M mapping between grammar and meta-model terms:

- As the introduction explained (Section 1.2.2), a simple expression *'x'* may be a shorthand for the expression *'self.x'*. It turns out that a set of LHS non-terminals, *NameExpCS*, *PathNameCS* and *PathElementCS* may map to a set of meta-classes, *VariableExp* and a *PropertyCallExp*.

Figure 3.8 shows an example of a textual input of a *PropertyCallExp* usage, and the corresponding AS model.



```
class c1 {
  prop p1 : String;
  op getP1() : String =
    p1;
}
```

FIGURE 3.8: Example: *PropertyCallExp* notation and corresponding AS model

3.4 Bridging Concrete and Abstract Syntax: Fine Grained Analysis

Following the introduction of a coarse grained view of the kind of mappings necessary to bridge the CS (text) and AS (models) of a textual modeling language, this section goes into detail regarding the fine grained concerns that need to be addressed.

3.4.1 Concern 1: Mapping an LHS non-terminal to a meta-class

The first concern relates to the need for mapping an LHS non-terminal to a meta-model concept. In this case, LHS non-terminals are mapped to meta-classes. This kind of mapping is split into two different categories: mappings with the goal of creating new AS model elements and mappings with the goal of referring to AS model elements already created by other mappings. These categories are explained below.

LHS non-terminal mappings that create new AS elements

When introducing 1-to-1 mappings in Section 3.3.1, it was mentioned that "the LHS non-terminal *PackageCS* maps to the *Package* meta-class". In this particular mapping, for every textually-defined package that has been parsed, according to the syntactic rule for the non-terminal *PackageCS*, a new *Package* model element must be created.

For instance, according to the example shown in Figure 3.2, for the textually-defined *p1* and *p1_1* packages, the AS model must contain two *Package* model elements called *p1* and *p1_1* respectively.

LHS non-terminal mappings that refer to existing AS elements

When introducing N-to-1 mappings in Section 3.3.2, it was mentioned that "a set of LHS non-terminals, *PathNameCS* and *PathElementCS*, maps to the *Class* meta-class". In this case, the corresponding *Class* model element is not meant to be created, but to be referred to by the extending *Class* that defines the class extension. The referred *Class* model element should have been created by another *ClassCS-to-Class* mapping.

For instance, according to the example shown in Figure 3.4, the class extension definition *'p1::c1'* should be parsed conforming to the syntactic rules corresponding to *PathNameCS* and *PathElementCS*. The first *'p1' PathElementCS* corresponds to the *'p1' Package*, which should have been created according to a mapping defined on the non-terminal *PackageCS*. Likewise, the second *'c1' PathElementCS* corresponds to the *'c1' Class*, which should have been created according to a mapping defined on the non-terminal *ClassCS*. Finally, the *'p1::c1' PathNameCS* corresponds to the 'c1' *Class*, which is the AS model element corresponding to the last *PathElementCS* of that *PathNameCS*.

3.4.2 Concern 2: Mapping an RHS non-terminal to a reference

The second concern relates to the need for mapping one RHS non-terminal to a meta-model concept. As mentioned in Section 3.1, an RHS non-terminal appears in a syntactic rule and is related to an LHS non-terminal defined somewhere else in the grammar. Whilst LHS non-terminals are mapped to meta-classes, RHS non-terminals are mapped to references.

When introducing 1-to-1 mappings in Section 3.3.1, it was mentioned that "the RHS *ClassCS* maps to the *ownedClasses* reference of the *Package* meta-class". As a reminder, the RHS non-terminal *ClassCS* appears in the syntactic rule of the LHS non-terminal *PackageCS* (see Listing 3.10). In this case, the RHS non-terminal *ClassCS* maps to the containment reference *Package::ownedClasses*.

According to the example shown in Figure 3.2, for the textually-defined *'c1'* and *'c2'* classes, the AS model must contain the *'c1'* and *'c2' Class* model elements. In particular, these model elements end up contained by the *'p1' Package* model element, via the *Package::ownedClasses* containment reference.

Note that there is no distinction between containment and cross-references, because LHS non-terminal mappings may map to either new or existing model elements. However, a correct mapping specification should take into account whether the mapped reference is a containment or not. On the one hand, if a mapping for an LHS non-terminal is designed to create new AS model elements, any RHS non-terminal should map to a containment reference, otherwise the created element may end up as an orphan model element[5]. On the other hand, if a mapping for an LHS non-terminal is designed to refer to already existing AS model elements, any RHS non-terminal should map to a cross-reference, otherwise already existing AS model elements would be removed from their current container (child-stealing in [79]), which may not be the original intention.

3.4.3 Concern 3: Mapping a terminal to an attribute

The third concern relates to the need for mapping one terminal to a meta-model concept. Terminals correspond to information from the textual input, and comprise the (primitive) values to be held inside the model. Therefore, they are naturally mapped to attributes of

[5]Orphan model elements are those model elements that are not contained by any other. Therefore, they are located at the root of the model.

meta-classes. In this way, whatever token is parsed from the text may end up as an attribute value of an AS model element.

Note that not every terminal has to be mapped to an attribute. For example, some of them are keywords and punctuation symbols, which are not normally relevant to the AS of a language.

When introducing 1-to-1 mappings in Section 3.3.1, it was mentioned that "the terminal *ID* maps to the *name* attribute of the *NamedElement* meta-class". For instance, according to the example shown in Figure 3.2, the identifier *p1* corresponds to the *name* of the *p1 Package*.

3.4.4 Recap: Simple vs. Complex CS2AS Bridges

So far, the fine grained concerns that have been explained cover the 1-to-1 and N-to-1 mappings that were introduced in Section 3.3.

On the one hand, by means of these fine grained mappings, every single grammar term (or as many as required) could be mapped to a meta-model term. In this thesis, if a language AS meta-model can be fully mapped from the CS grammar (by means of the fine grained mappings discussed), the language is considered sufficiently trivial that it only requires a **simple** CS2AS bridge.

On the other hand, by means of these fine grained mappings, it cannot be assured that every single AS meta-model term has been likewise mapped. If this is the case, this thesis considers the language sufficiently complex that it requires a **complex** CS2AS bridge.

Giving support to the 1-to-N mappings presented in the previous Section 3.3, and more generally N-to-M mappings, turns out to be necessary for supporting complex CS2AS bridges. The following subsections detail the additional fine grained concerns to be addressed to cover 1-to-N and N-to-M mappings.

3.4.5 Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes

The fourth concern relates to the need for mapping one RHS non-terminal to additional meta-classes, apart from the target reference explained in Section 3.4.2. More precisely, this concern comprises the need for creating an arbitrary number of connected model elements so that:

- One (or many) of these additional model elements are referred to via the target reference.
- The AS model element obtained from a further mapping (for the corresponding LHS non-terminal) is referred to by one (or many) of these additional model elements.
- In essence, this kind of mapping allows us to produce more complex structures of model elements that are referred to via the target reference.

When introducing 1-to-N mappings in Section 3.3.3, it was mentioned that "the RHS non-terminal *ExpCS* maps to the *ownedBodyExpression* reference of the *Operation* meta-class.

However, it also requires mapping to additional meta-classes, such as *ExpressionInOCL* and *Variable*". For instance, according to the example shown in Figure 3.6, the textually-defined *'aString'* corresponds to a *StringLiteralExp* model element, to be added to the *ownedBody-Expression* reference of the *Operation*. However, this *StringLiteralExp* model element is not directly considered as the model element to refer to. Instead, an *ExpressionInOCL* model element is created for that purpose. This *ExpressionInOCL* also includes an implicit *Variable* model element named *'self'*. Finally, *ExpressionInOCL* is responsible for containing the former *StringLiteralExp* model element, via the *ExpressionInOCL::ownedBody* reference.

3.4.6 Concern 5: Mapping an LHS non-terminal to many meta-classes

The fifth concern relates to the need for mapping one LHS non-terminal to additional meta-classes, apart from the target meta-class explained in Section 3.4.1. More precisely, this concern comprises the need for creating an arbitrary number of additional connected model elements so that:

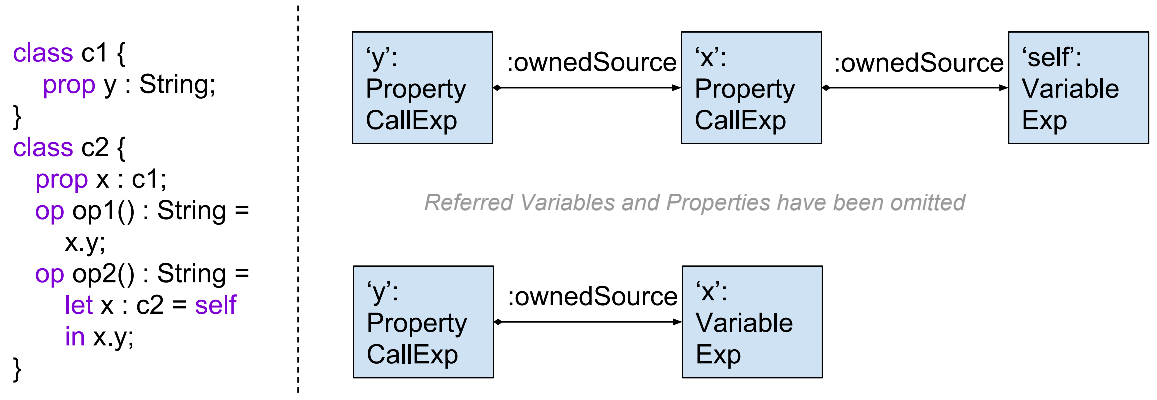
- One (or many) of these additional model elements is (are) referred to by the model element that conforms to the mapped target meta-class.
- In essence, this kind of mapping allows us to produce more complex structures of model elements, rather than a single one.

When introducing N-to-M mappings in Section 3.3.4, it was mentioned that "a simple expression *'x'* may be a shorthand for the expression *'self.x'*. It turns out that a set of LHS non-terminals, *NameExpCS*, *PathNameCS* and *PathElementCS* may map to a set of meta-classes, *VariableExp* and *PropertyCallExp*". For instance, according to the example shown in Figure 3.8, the textually-defined *'p1'* corresponds to a *PropertyCallExp* model element, which contains (via the *CallExp::ownedSource* reference) a *VariableExp* model element (that refers to the *'self' Variable*).

3.4.7 Concern 6: Multi-way mappings from LHS non-terminals to meta-classes

The sixth concern relates to the need for expressing multi-way mappings from one LHS non-terminal to its corresponding meta-class(es). In this thesis, a multi-way mapping refers to the situation where a non-terminal can be mapped to different meta-classes, in an exclusive manner, driven by some conditions.

The *x.y* expression from the introduction (Section 1.2.2) exposes the problem. According to the CS, *x* and *y* are simple *NameExpCS* (definition presented in Listing 3.13). However, this expression may correspond to different AS model elements, depending on some conditions. In this case, it depends on the context in which they are used. Whilst *y* is unambiguously a *PropertyCallExp*, *x* may be either a *VariableExp* (referring, for instance, to a variable previously defined by an outer *let* expression) or another *PropertyCallExp* (referring to an *x* named property of *self*, the context model element in which that expression is evaluated). Figure 3.9 shows a complete example.



```
class c1 {
   prop y : String;
}
class c2 {
  prop x : c1;
  op op1() : String =
     x.y;
  op op2() : String =
     let x : c2 = self
     in x.y;
}
```

FIGURE 3.9: Example: Two different outcomes of a multi-way mapping

The way in which these multi-way mappings are executed depends on several conditions. In this thesis, these conditions are called **disambiguation rules**. It may arise that, depending on the design of the source CS grammar and the target AS meta-model, some CS non-terminals turn out to be ambiguous from the point of view of the AS, and these rules serve to disambiguate.

In the case of mOCL, *NameExpCS* is an example of ambiguous non-terminal (from the point of view of the AS) because there is a need to determine if it corresponds to a *VariableExp* or a *PropertyCallExp*. The disambiguation rule to decide if *NameExpCS* corresponds to a *VariableExp* (instead of a *PropertyCallExp*), consists of checking that the contained *PathNameExpCS* corresponds to a *Variable* (instead of a *Property* of *self*).

For instance, according to the example shown in Figure 3.9, in the *'x.y'* expression within the *'op1'* operation, the *'x'* parsed by the syntactic rule of the non-terminal *PathNameCS* corresponds to a *PropertyCallExp* model element. On the contrary, in the *'x.y'* expression within the *'op2'* operation, the *'x'* parsed by the syntactic rule of the non-terminal *PathNameCS* corresponds to a *VariableExp* model element.

3.4.8 Concern 7: Mapping properties from non grammar terms

The seventh concern relates to the need for computing additional AS meta-class properties that do not necessarily map from CS terms. When the differences between simple and complex CS2AS bridges were explained (Section 3.4.4), it was identified that all meta-model concepts need to be mapped. Whilst the two concerns from Section 3.4.5 and Section 3.4.6, ensure that all AS model elements can be created, this concern ensures that all its corresponding properties can be likewise computed (ensuring the required completeness).

The following exposition explains this need. mOCL is a statically typed language (i.e. it defines a type system), and every *OCLExpression* must refer to a *Type* model element, via the *TypedElement::type* cross-reference. The information about types for *OCLExpressions* is not provided in the CS, but there exist additional rules (i.e. a type system definition) to compute the type for every expression. Consequently, any CS2AS bridge requires the means that allow the mapping of any meta-class property, even though the required information does not come directly from the textual input.

It could be argued that:

- A type system should not be embedded within the AS meta-model. For instance, XSemantics [4] allows the definition of a separate type system for Xtext-based languages. Although this is a valid approach, it is not the case for the OCL language, whose specification [39] states that the type system is embedded within the AS meta-model.

- The definition of a type system embedded within the AS meta-model should be done as a set of derived properties[a]. Although this is a valid approach, it is not the case for the OCL language, whose specification [39] states that *type* property is not derived. For the sake of giving support to a wider range of target meta-models this concern should be considered as part of the CS2AS activity.

[a]Derived properties are special in the sense that their values are automatically computed by an expression (which, for instance, can use the values from other properties) defined at the meta-model level. In consequence, they are transient properties and the modeler cannot establish their value.

Note that these mappings do not specify the kind of properties (e.g. either references or attributes) for which this concern applies. The example above applies to (cross-)references. Regarding attributes, they are normally initialised with information from the textual input (see Section 3.4.3). However, this is not always the case. For instance, Figure 3.10 shows the relevant AS definition of an *ExpressionInOCL*.

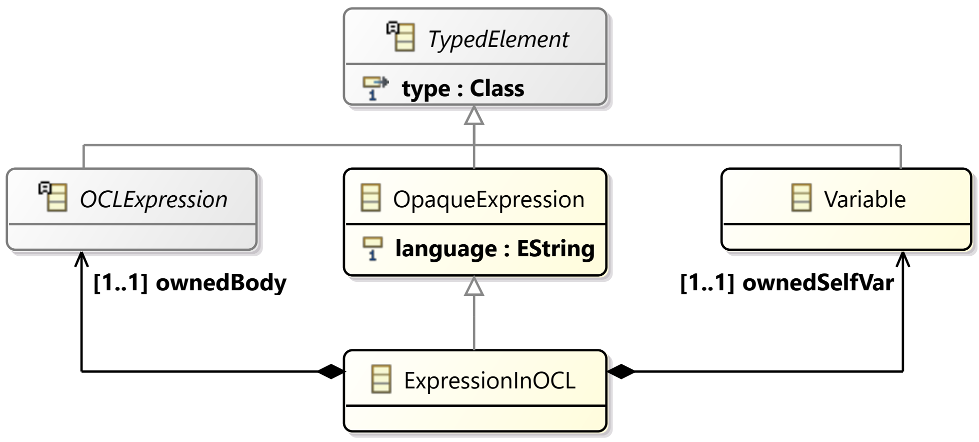


FIGURE 3.10: AS of *ExpressionInOCL*

According to the OCL specification, an *ExpressionInOCL* is a kind of *OpaqueExpression*. The latter has a *String*-valued attribute called *language*. The specification states that every *ExpressionInOCL* instance must have this attribute initialised to the *'OCL' String* value.

3.5 Concern 8: Name Resolution

Name resolution is an important activity of *Programming Language Theory and Practice* (e.g. in compilers [1]) and it plays a key role in this thesis. As further explained, name resolution is related to several fine grained concerns that were introduced in Section 3.4. Due to its length and the amount of different topics to study, name resolution is separately analysed in its own section.

3.5.1 Overview

Name resolution, also known as named-based lookup, consists of performing a lookup activity with the aim of finding (named) model elements located in the AS model. The input of this lookup activity is normally CS related information (e.g. a name), whilst the output is the desired AS model element.

Name resolution is relevant to this thesis, because it can be used for relating two AS model elements via a cross-reference. Therefore, this lookup activity is closely related to Concern 1 (when a mapping pursues the retrieval of already existing AS model elements) from Section 3.4.1, and Concern 7 (where arbitrary property computations can be done to relate AS model elements via a cross-reference) from Section 3.4.8. Additionally, these name-based lookups can also be used in disambiguation rules to decide which AS model element should be created. Therefore, name resolution is also related to Concern 6 from Section 3.4.7.

The following example illustrates the role of name resolution. The left hand side of Figure 3.11 denotes a *LetExp* usage in mOCL. It declares and initialises a variable named *var* for use within the *LetExp 'in'* expression. In this example, the *'in'* expression comprises a *VariableExp* usage that refers to the former variable *'var'*. The right hand side of Figure 3.11 exposes the corresponding AS model, in which it can be observed how the *VariableExp* model element refers to the *Variable* one, via the *referredVariable* cross-reference.

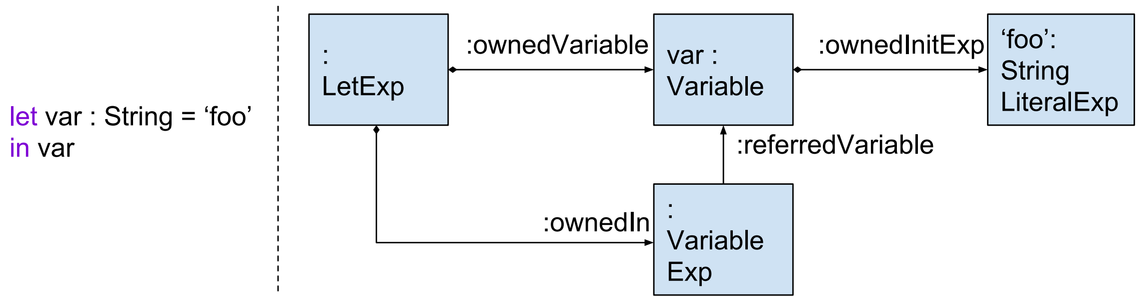


FIGURE 3.11: Example: *LetExp* notation and its corresponding AS model

In order to make the *VariableExp* model element refer to the *Variable* one, name resolution is required, so that the declared variable (i.e. the *Variable* model element) is looked up (and found) in the scope of the variable usage (i.e. the *VariableExp* model element).

The class extension example introduced in Section 3.3.2 can also be referred to as an example that requires name resolution. In order to make a derived class extend to a super class, via the *Class::superClasses* cross-reference, a name resolution activity is required to find the actual *Class* that is meant to be extended.

3.5.2 Targets, Inputs, Providers and Consumers

In a name resolution activity, the following roles are identified:

- **Targets**. These are the AS model elements that need to be found via name resolution. In the example shown in Figure 3.11, the *Variable* model element is the target of the name resolution activity.

  In a name resolution activity, targets are identified by a name, i.e. the corresponding meta-classes have a *String*-valued attribute to identify these instances. This attribute is normally called *name*, although it could be any other. In this thesis, targets can also be referred to as target (model) elements or named (model) elements.

- **Inputs**. These are the values that are used as the input to find targets via name resolution. In the example shown in Figure 3.11, the *String "var"* is the input of the name resolution activity.

- **Providers**. These are the AS model elements that add targets to a scope or environment (see further Section 3.5.3) so that they can be found via name resolution. In the example shown in Figure 3.11, the *LetExp* model element is the provider during the name resolution activity.

  In this thesis, providers can also be referred to as provider (model) elements.

- **Consumers**. These are the AS model elements that need to find a target via name resolution. In the example shown in Figure 3.11, the *VariableExp* model element is the consumer during the name resolution activity.

  In this thesis, consumers can also be referred to as consumer (model) elements.

From the point of view of these roles, a name resolution activity can be explained as follows. A consumer requires performing name resolution in order to find a target that is located somewhere else in the AS model. The goal of the name resolution is to make the consumer refer to the found target via a cross-reference. To perform the name resolution, an input is required so that the target can be found by matching its name with the provided input. The target can be matched as long as a provider contributes that target to the consumer's scope (see next Section 3.5.3).

For instance, according to the example shown in Figure 3.11, a *VariableExp* model element requires performing name resolution in order to find a *Variable* model element that is located somewhere else in the AS model. The goal of the name resolution is to make the *VariableExp* model element refer to the found *Variable* model element via the *referredVariable* cross-reference. To perform the name resolution, a *String* value, *'var'* in this case, is required

mOCL meta-class	T	P	C	Explanation
Class	X	X	X	It may be referred to by other classes It contributes properties and operations It may refer to other classes
ExpressionInOCL		X		It contributes the 'self' contextual variable
Import		X		It contributes the imported elements
IterateExp		X		It contributes the iterator and accumulator variables
IteratorExp		X		It contributes the iterator variable
LetExp		X		It contributes the let variable
Package	X	X		It contributes classes and nested packages It may be target of qualified name-based lookups
Property	X			It may be referred to by property call expressions
PropertyCallExp			X	It refers to properties
Operation	X			It may be referred to by operation call expressions
OperationCallExp			X	It refers to operations
NamedElement	X			It may be target of name-based lookups
Root		X		It contributes packages and imports
Variable	X			It may be referred to by variable expressions
VariableExp			X	It refers to variables
TypedElement			X	It refers to a class (as type of the element)

TABLE 3.1: Correspondence between targets (T), providers (P), consumers (C) and mOCL meta-classes

so that the *Variable* model element can be found by matching its name with the provided *String* value. The *Variable* model element can be matched because a *LetExp* model element contributes that *Variable* model element to the *VariableExp* model element scope.

To conclude this subsection, Table 3.1 summarises a correspondence between targets, providers, consumers and the related AS meta-class of mOCL. The following subsection goes into detail regarding the concept of scopes.

3.5.3 Lookup Scopes

The concept of scope, or lookup scope, was previously mentioned. It is another important concept relevant to name resolution. Scopes, also known as environments in [39], are data structures that keep a *name-to-target* key-value map. Figure 3.12 shows how this data structure is used by consumers and providers, with target and input being the data that flows to make a consumer find a target contributed by a provider.

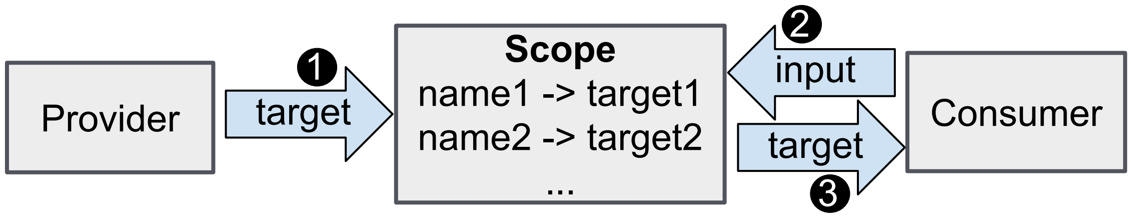


FIGURE 3.12: Overview of a scope

This *name-to-target* map comprises all the candidate (named) targets of a name-based lookup for a particular consumer. The candidate targets inside a scope are those contributed by providers ❶. A consumer can look up a target by means of a given input ❷. If the lookup scope contains the target whose name matches the input, the target is found ❸. Figure 3.13 shows how the scope is used during the name resolution required for the example from Figure 3.11.

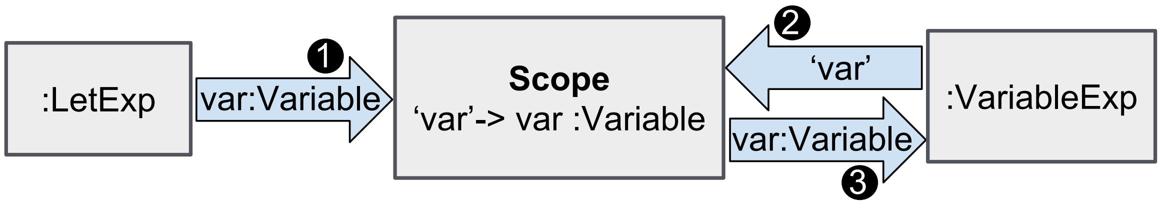


FIGURE 3.13: Scope involved in the *LetExp* example from Figure 3.11

In this case, a *LetExp* model element contributes a *Variable* named *'var'* to the scope ❶. Conversely, a *VariableExp* model element performs a name resolution activity using the *String 'var'* as an input ❷. Since the lookup scope contains a *Variable* model element whose name matches the provided input, this *Variable* model element is retrieved as the result of the name resolution activity ❸.

Scope propagation

Scopes are data structures that let consumers find a target contributed by a provider. The key characteristic of this kind of information passing is that the consumer is unaware of which are the target providers and where they are located in the AS model, and the provider is unaware of which are the consumers and where they are located in the AS model.

In order to explain how this mechanism of information passing works, the concept of scope propagation is introduced. Since name resolution is defined to perform name-based lookups across the AS model, scope propagation consists of transferring the scope in a top-down fashion. In other words, the scope flows between model elements, starting from the root, from parent to children across the containment hierarchy. By default, lookup scopes are propagated as they are (initially, they are empty). However, when providers propagate their current scope to their children, the former have the ability to add new targets (selectively) to the propagated scope. In this way, whenever a consumer needs to perform a name-based lookup, its current scope already contains all candidate targets that can be found by that particular consumer.

A provider selectively configures the lookup scope for their children model elements. This means that it has the ability to decide if the scope is propagated as it was received (from its parent element), or it is modified to include more targets. This configuration is selective in the sense that, if a provider contains different model elements (e.g. via several containment references), the propagated scope can likewise be different.

For instance, according to the example shown in Figure 3.11, the scope propagation mechanism is depicted in Figure 3.14 (the final content of each scope is shown in Table 3.2). The AS model has been enhanced so that the target *LetExp* model element is contained by a proper *ExpressionInOCL* model element. The actual context in which the *LetExp* usage may appear is not relevant for the following exposition.

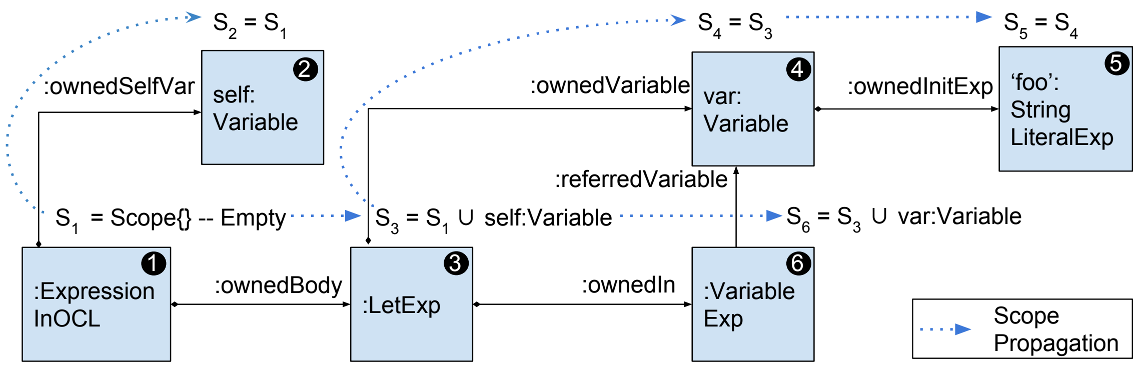


FIGURE 3.14: Scope propagation, according to the example from Figure 3.11

In this example, the *ExpressionInOCL* model element ❶ corresponds to the root model element and, therefore, its lookup scope S_1 is empty. This scope must be propagated to its children. *ExpressionsInOCL* is a kind of provider in mOCL. On the one hand, the scope is propagated as it is for its child *'self' Variable* model element ❷, contained via the *ownedSelfVar* reference. Therefore, the corresponding scope S_2 must also be empty. On the other hand, the scope S_3 that is propagated for its child *LetExp* model element ❸, contained via the *owned-Body* reference, additionally includes the mentioned target *'self' Variable* model element.

LetExp is another kind of provider in mOCL. On the one hand, the scope is propagated as it is for its child *'var' Variable* model element ❹, contained via the *ownedVariable* reference. Therefore, the corresponding scope S_4 is the same as S_3. Note that *Variable* is not a provider in mOCL, so the current scope is propagated as it is to the *'var' StringLiteralExp* model element ❺. As a result, S_5 is the same scope as S_4 and S_3. On the other hand, the scope S_6 that is propagated for the child *VariableExp* model element ❻, contained via the *ownedBody* reference, additionally includes the mentioned target *'var' Variable* model element.

For this particular example, Table 3.2 shows the values of all the propagated scopes of their corresponding AS model element.

Model Element	Corresponding Scope
❶ :ExpressionInOCL	S_1 = {}
❷ self:Variable	S_2 = {}
❸ :LetExp	S_3 = {self:Variable}
❹ var:Variable	S_4 = {self:Variable}
❺ 'var':StringLiteralExp	S_5 = {self:Variable}
❻ :VariableExp	S_6 = {self:Variable, var:Variable}

TABLE 3.2: Computed scopes for every AS model element of Figure 3.14

Nested scopes

Nested scope is a concept that emerges to give support to name occlusion; in other words, to let some target model elements be occluded by others that have exactly the same name. Figure 3.15 shows an example for this particular language requirement.

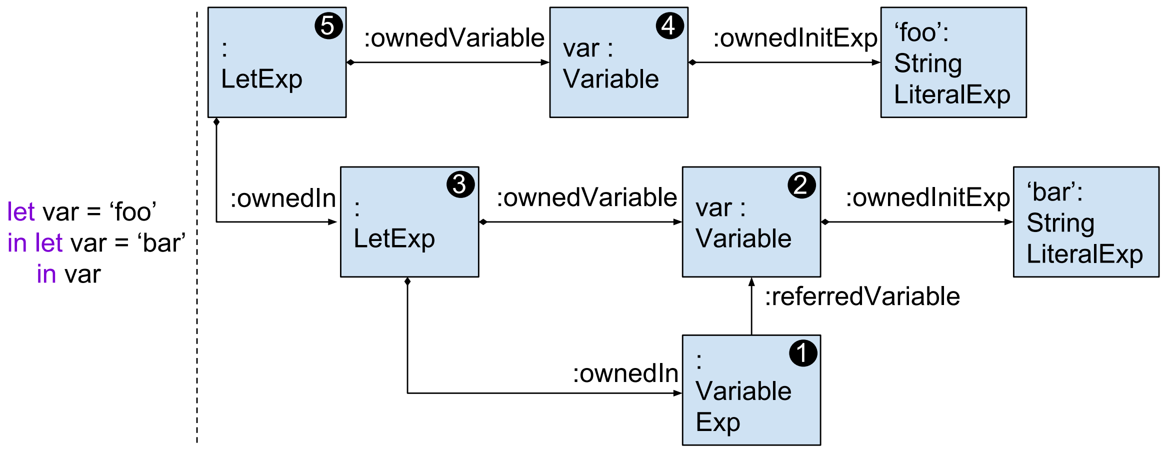


FIGURE 3.15: Example: Nested *LetExp* notation and corresponding AS model

The nested *LetExp* example notation, in the left hand side of the figure, represents a valid expression of mOCL. According to the AS, in the right hand side of the figure, the *VariableExp* model element ❶ refers to the *Variable* (initialised with the *String 'bar'*) model element ❷ contained by the inner *LetExp* model element ❸. This occurs because that *Variable* model element ❷ occludes the equally named *Variable* (initialised with the *String 'bar'*) model element ❹ contained by the outer *LetExp* model element ❺.

This scenario is supported by the concept of scopes nesting, which allows the creation of a hierarchy of composed scopes. In this way, any target model element that belongs to an inner (nested) scope may occlude any other target model element with the same name that belongs to the outer (parent) scope. Figure 3.16 shows how the scopes are configured and used during the name resolution required for this example from Figure 3.15.

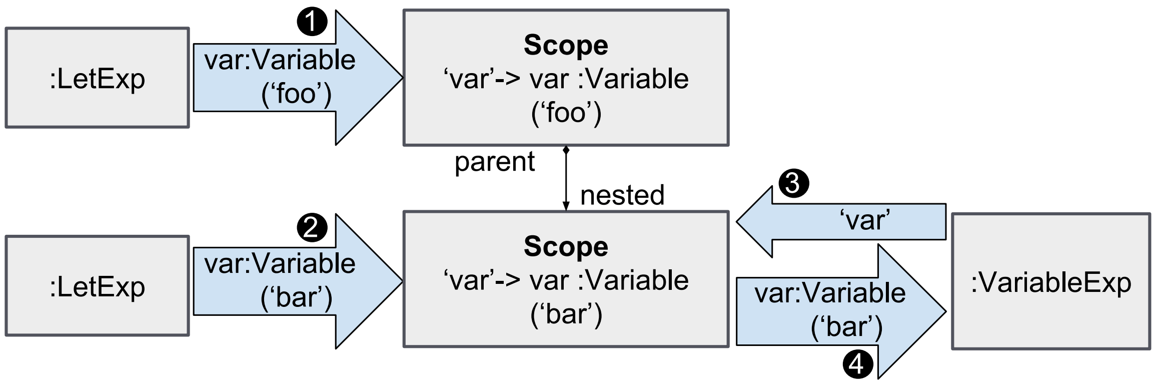


FIGURE 3.16: Scopes involved in the nested *LetExp* example from Figure 3.15

In this case, the outer *LetExp* model element contributes a target *Variable* named *'var'* (initialized with the *String 'foo'*) to the lookup scope ❶. The inner *LetExp* model element contributes another target *Variable* named *'var'* (initialized with the *String 'bar'*) to the corresponding lookup scope ❷. However, this second scope is not the same as the first one, but a nested one. In this way, the outer variable gets occluded by the inner one. Finally, a *VariableExp* model element performs a name resolution activity using the *String 'var'* as an input ❸. Since the nested lookup scope contains a *Variable* model element whose name matches the provided input, this *Variable* model element is retrieved as the result of the name resolution activity ❹.

3.5.4 Name-based Lookups

The previous subsection explained how the scopes are configured by providers and how they are propagated from parents to children across the entire AS model. This subsection explains how these scopes are used by consumers of the name resolution activity.

Any AS model element has access to its corresponding scope, which comprises all the visible targets that the former can find. As commented in subsection 3.5.2, a name-based lookup is performed by consumers in order to find a target element located elsewhere in the AS model. In this case, these targets are the model elements that have been contributed to its corresponding scope by providers.

A name-based lookup simply consists of querying the corresponding scope of the consumer. This query requires the provision of an input (e.g. a *String* value) that is used for matching a target, and the type of that target. This information required to perform the lookup is also referred to throughout this thesis as **lookup criteria**. When the scopes contain a map entry, whose entry-key matches the provided input, and the entry-value (i.e the target) conforms to the provided type, the corresponding target is returned as the result of the name-based lookup. If such a match is not found, the result would be empty, meaning that there is no visible model element in that scope that matches the lookup criteria. If several entries match the lookup criteria, there is a case of ambiguous name usage (see the next sub-subsection).

The additional information on the type is needed to support selective name-based lookups. The type information is used as a filtering criterion. For instance, in mOCL, a *Class* model element can contribute, within the propagated scope, a *Property* model element and an *Operation* model element with exactly the same name. In this case, any children model element names should not obtain ambiguous results as the outcome of named-based lookup. If consumers (e.g. *PropertyCallExp*) have the ability to provide additionally the type of target they are looking for (e.g. *Property*), ambiguous lookups can be avoided.

According to the previous example shown in Figure 3.14, the *VariableExp* model element ❻ requires performing a name-based lookup to locate the *Variable* that it needs to refer to, via the *referredVariable* cross-reference. In this case, this cross-reference can be set because the corresponding scope (S_6) of that *VariableExp* model element comprises a *Variable* ❹ model element whose name is *'var'* (see Table 3.2). In terms of this name-based lookup, the input is the *String 'var'*, the target type is *Variable* and the outcome of the lookup is the *Variable* model element with name *'var'*.

Ambiguous Names

Name-based lookups are not guaranteed to be successful. A trivial example of this is the case where no target element is found in the corresponding scope. The other possible erroneous situation is where the outcome comprises more than one candidate target. In this case, it turns out that the input, i.e. the name, provided to perform the name-based lookup is ambiguous. In this thesis, this situation is referred to as an **ambiguous lookup**. Ambiguous names can be cured by means of qualified name-based lookups (see further Section 3.5.5).

For instance, in UML and OCL (hence mOCL), multiple inheritance is allowed among *Class* model elements. *OperationCallExp* model elements refer to the corresponding *Operation* model element, via the *referredOperation* cross-reference. Then, it may occur that when an *OperationCallExp* performs a name-based lookup to locate the corresponding *Operation* model element, the input value used for performing the lookup may correspond to two different *Operation* model elements in that particular scope. In this case, the name-based lookup has been performed with an ambiguous name. As it is explained in the next Section 3.5.5, using the fully qualified name of the operation can solve the ambiguity.

Nested Scopes

The previous subsection explained why nested scopes are needed and how they are populated. In this subsection, it is described how the name-based lookup is performed providing that it operates on nested scopes.

A name-based lookup works with nested scopes in nearly the same way as if they were not nested. The difference is that, if there is no target in the nested scope that fulfils the lookup criteria, the lookup is retried in its parent scope.

Nesting scopes prevents ambiguous lookups from occurring, provided that in a particular language an ambiguity is not expected for that particular name-based lookup. For instance, as was previously explained in the example from Figure 3.15, the *'var'* name is unambiguously resolved to the *Variable* model element that belongs to the inner (nested) scope. This occurs because, within the nested scope, the *Variable 'var'* is unique. Besides, nesting scopes does not prevent targets belonging to outer (parent) scopes from being found, providing that no target fulfils the lookup criteria in the inner (nested) scope.

3.5.5 Qualified Name-based Lookups

So far, basic name-based lookups, which are performed by AS model elements to find another one located somewhere else in the AS model, have been introduced. Whilst this mechanism allows name resolution to occur without letting a consumer know where the provider is located in the AS model (and vice versa), the lookups are restricted to accessing the targets visible in the corresponding current scope of a consumer. This is the normal case when resolving the name resolution, but there are special scenarios where a consumer has to refer legally to a target which is not within its corresponding scope. This and the following subsection explain these scenarios and the corresponding name-based lookup mechanisms.

This subsection introduces a new kind of name-based lookup required to overcome new scenarios that cannot be tackled with the lookup mechanisms shown in the previous subsection. For instance, in mOCL, it may occur that an *OperationCallExp* model element has to refer to a static *Operation* model element. The problem arises when the latter is not visible because the former's scope is not propagated from the *Class* model element that contributes that *Operation* model element.

The required particular lookup is called qualified name-based lookup and is described by means of the following characteristics:

- The lookup relies on giving the targets the ability to qualify other target elements. In essence, a target element can be qualified by another preceding target. A sequence of preceding qualifying targets comprises the qualification of the desired target to look up.
- The lookup criteria are enhanced with a more complex input. The lookup is performed with a list of inputs, rather than a single one. Each input is used for sequentially matching each of the preceding targets of the desired one.
- The actual wanted target does not necessarily need to be within the corresponding scope of a consumer. A target that is not visible in the corresponding scope of a consumer could be found, as long as: the target corresponding to the first input of the list is in the consumer's scope, and the subsequent names of the input's list match all the subsequent targets of the qualification, including the final wanted target.

An example of this qualified name-based lookup can be found in mOCL, where *Package* model elements qualify *Class* and other nested *Package* model elements. Moreover, *Class* model elements also qualify *Operation* and *Property* model elements. In this way, a fully qualified *Operation* model element can be found by any consumer, such as an *OperationCallExp* model element, regardless of whether the former is in the scope of the latter.

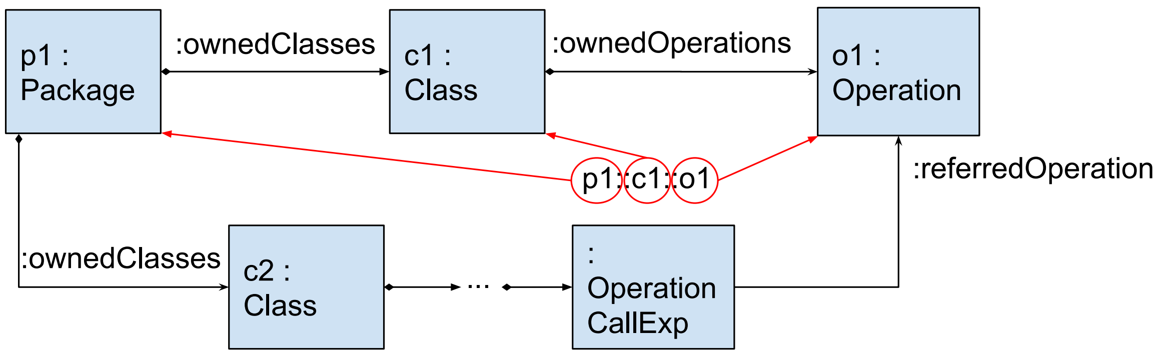


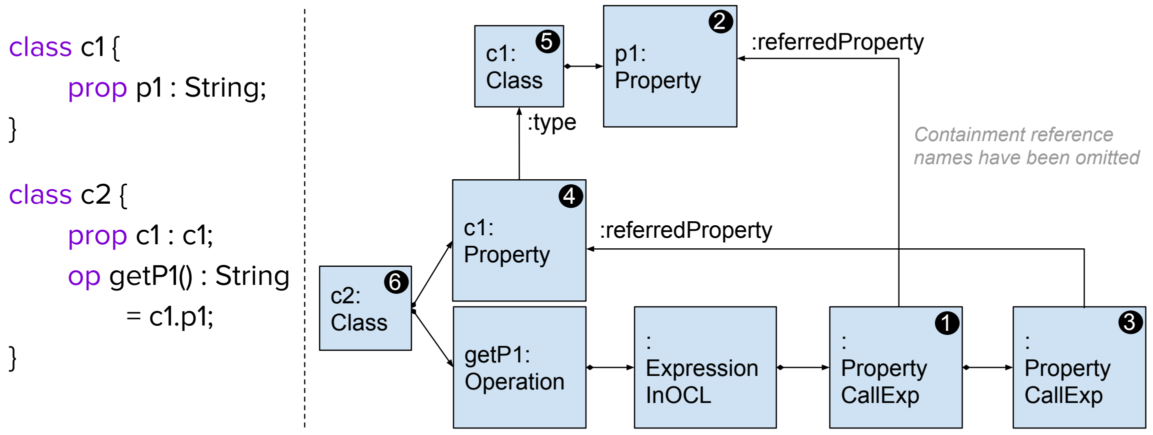
FIGURE 3.17: Qualified name-based lookups

Figure 3.17 shows a concrete example. The *Operation 'o1'* is qualified by the *Class 'c1'*, which is qualified by the *Package 'p1'*. An *OperationCallExp* model element that is indirectly contained by the *Class 'c2'* may need to refer to that *Operation 'o1'*. However, the target model element is contained by a different *Class 'c1'*, which does not contribute the wanted *Operation 'o1'* to the scope of the *OperationCallExp* model element. Therefore, a qualified name-based lookup can be performed to locate that operation. In mOCL, fully qualified names are specified by separating simple names with a double colon (i.e. *'::'*), comprising the list of simple names that the lookup needs as part of the lookup criteria. Providing that there is a *Package* model element with name 'p1' (the first name of the list) within the scope of the *OperantionCallExp* model element, the lookup can proceed to find the subsequent *Class 'c1'* and, finally, the wanted *Operation 'o1'*.

Qualified name-based lookups can be used for disambiguating lookups (explained in the previous subsection). Provided that a name lookup (within the scope of a consumer) returned more than one target (ambiguous lookup), the ambiguity could be solved by performing a qualified name-based lookup that uses the corresponding fully qualified name of the wanted target.

3.5.6 Name-based External Lookups

This subsection introduces the analysis of another problem related to name resolution, namely the need for performing lookups out of the corresponding scope of a consumer. Specifically, this external lookup is done in an exported scope of a particular provider that the consumer has access to. Figure 3.18 shows a scenario in mOCL that motivates the need for external lookups.



```
class c1 {
      prop p1 : String;
}

class c2 {
      prop c1 : c1;
      op getP1() : String
            = c1.p1;
}
```

FIGURE 3.18: Example: *PropertyCallExp* accessing an out of the scope *Property*

According to the example's notation, in the left hand side of the figure, there is an expression *'c1.p1'* which is the body of the *Operation 'getP1'*. This valid expression corresponds to a *PropertyCallExpression* model element ❶ that refers to the *Property 'p1'* ❷. The source of that *PropertyCallExpresion* ❶ is another *PropertyCallExpression* model element ❸ that refers to the *Property 'c1'* ❹.

Whilst the *Property* 'c1' ❹ is within the corresponding scope of the *PropertyCallExp* ❸, the *Property* 'p1' ❷ is not. In other words, the target property belongs to the *Class 'c1'* ❺ rather than the *Class 'c2'* ❻; hence, it is not visible from *PropertyCallExp* ❶. However, the contributor *Class 'c1'* of that *Property 'p1'* can be accessed from that *PropertyCallExp* model element ❶ (the navigation to access the exported scope of the contributing *Class* model element is shown in Listing 3.14). Therefore, a name-based external lookup can be performed to find that *Property 'p1'*, within the exported scope of the *Class 'c1'*.

```
ownedSource.referredProperty.type
```

LISTING 3.14: Navigation from *PropertyCallExp* ❶ to *Class 'c1'* ❺

As will be explained in the proposed solution (Section 4.2.5), the current AS model element's scope (that can only be accessed by the provider's children) and the exported one (that can be accessed by anyone) do not necessarily comprise the same targets. In essence, a provider may contribute different targets in each scope.

3.5.7 Additional Lookup Criteria

This subsection illustrates another problem related to name resolution, namely the need for enhancing the lookup criteria to perform additional filtering of the candidate targets. This requirement can be explained when going into more detail regarding *Operation* model element lookups that are performed from *OperationCallExp* model elements. In this case, the lookup criteria, which comprise the type of model element to look up (*Operation*) and the input name to match an instance of that type, are not enough.

The reason is that different *Operation* model elements, within the same scope, can legally have the same name. Therefore, they have to be further filtered according to an additional criterion, for instance, the *Operation* signature. In this case, the lookup criteria have to be enhanced with additional inputs related to the arguments involved in the operation call expression.

3.5.8 Looking Up into External Models

Finally, this subsection introduces the last relevant concern regarding name resolution. In this case, it is related to the need for performing name-based lookups into external models. For instance, OCL is a language designed to write constraints defined on (meta-)models. On the one hand, the AS defines some language constructs, such as *PropertyCallExp* and *OperationCallExp*, that are required to link to external (meta-)models with the aim of referring to the corresponding *Property* or *Operation*. Figure 3.19 shows the excerpt for this kind of AS concepts.

On the other hand, unlike other object-oriented languages such as Java, OCL does not provide any concrete syntax to define these external (meta-)models. Instead, the specification [39] delegates to implementors to provide the required mechanism to make the external (meta-)models available. Practical implementations, such as Eclipse OCL, provide an *Import* construct so that the involved meta-models can be explicitly loaded by the tools (e.g. parser, editors).

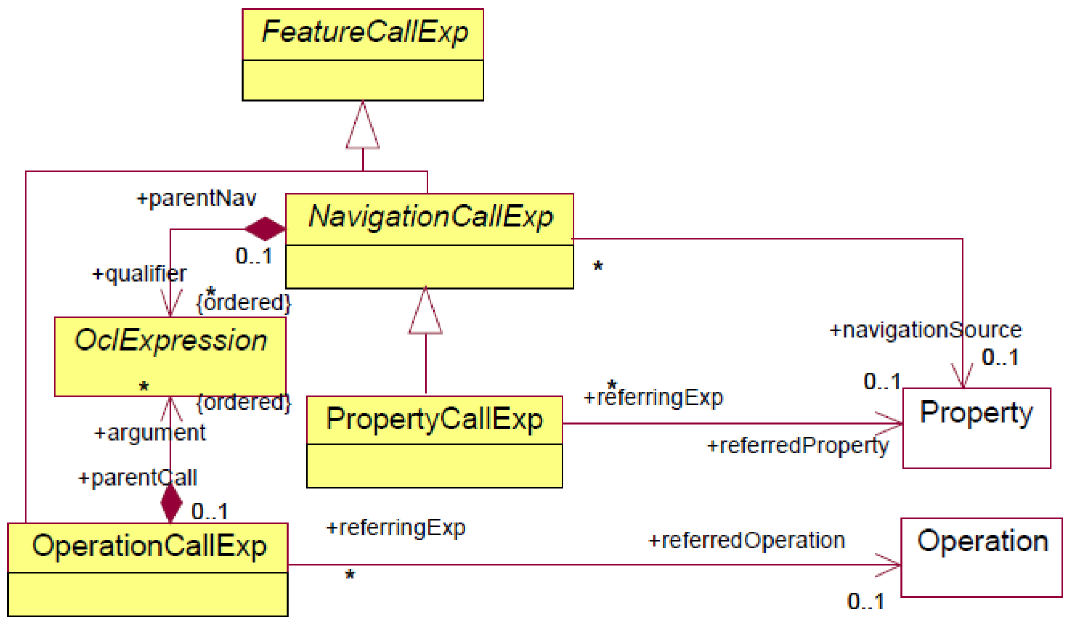


Figure 8.3 - Abstract syntax metamodel for FeatureCallExp in the Expressions package

FIGURE 3.19: Meta-model excerpt of the OCL AS (Figure 8.3 from [39])

In general, the concern here consists of letting AS models parsed from some textual inputs refer to external models. These models do not even mandate to have a specific textual notation. They can be just external models loaded by the involved modeling tools.

3.6 How Does Xtext Address These Concerns?

In the previous sections, the different concerns that have to be addressed to bridge the *CS* and *AS* of CTMLs were analysed. Prior to explaining the proposed solution for all these problems, we expose how Xtext is able to cope with these concerns, exposing the limitations that have motivated this research project. Note that Xtext is not only one of the related of work of interest (see Section 2.4), but also an industrial sponsor requirement (see Section 1.3).

3.6.1 Xtext Grammar Introduction

When introducing Xtext in Section 2.3.4, a brief overview of the technology was given in Figure 2.16. This subsection introduces the language specification artefacts required by Xtext, i.e. the grammars that let language engineers define the CS of a language and, at the same time, map the CS with the AS.

In Section 3.3.1, Listing 3.10 showed an EBNF excerpt of the mOCL grammar, particularly a *Package* definition. The following Listing 3.15 shows the corresponding excerpt conforming to the Xtext grammar language.

Compared to EBNF, Xtext grammars provide additional syntax to declare how the different CS terms map to the AS concepts. For instance:

```
1 PackageCS returns Package:
2    'package' name=ID '{'
3       (ownedClasses+=ClassCS | ownedPackages+=PackageCS)*
4    '}'
5 ;
```

LISTING 3.15: Xtext grammar of a *Package* definition

- In line 1, the LHS non-terminal *PackageCS* maps to the meta-class *Package*.
- In line 2, the terminal *ID* maps to the attribute *name*.
- In line 3, the RHS non-terminal *ClassCS* maps to the multi-valued containment reference *ownedClasses*.
- In line 3, the RHS non-terminal *PackageCS* maps to the multi-valued containment reference *ownedPackages*.

Xtext grammars support 1-1 mappings directly when declaring the instances of their specification artefacts (i.e. Xtext grammars); hence, they provide direct support to simple textual modeling languages. The following subsections explain how Xtext can or cannot address the different fine grained concerns introduced in Section 3.4, and name resolution from Section 3.5.

3.6.2 Concern 1: Mapping an LHS non-terminal to a meta-class

According to Listing 3.15, in line 1, this concern is supported by Xtext by declaring the name of the meta-class with the preceding keyword *returns*. In this example, the LHS non-terminal *PackageCS* maps to the meta-class *Package*.

3.6.3 Concern 2: Mapping an RHS non-terminal to a reference

According to Listing 3.15, in line 3, this concern is supported by Xtext by assigning that RHS non-terminal to the name of a reference. This reference must exist within the meta-class (or any of their super meta-classes) that is mapped to the LHS non-terminal of the corresponding syntactic rule. According to Listing 3.15, the RHS non-terminal *ClassCS* maps to the reference *Package::ownedClasses*.

With respect to the kind of reference, the following comments apply:

- The '=' or '+=' assignment operator is used, depending on whether the reference is mono-valued or multi-valued.
- Only containment references are expected. Cross-references cannot be used with RHS non-terminals. However, Section 3.6.9 further shows how some cross-references can be assigned to what Xtext denominates as cross-referenceable terminals.

3.6.4 Concern 3: Mapping a terminal to an attribute

According to Listing 3.15, in line 2, this concern is supported by Xtext by assigning the terminal to the name of an attribute. The attribute must exist within the meta-class (or any of their super meta-classes) that is mapped to the LHS non-terminal of the corresponding syntactic rule. According to Listing 3.15, the terminal *ID* maps to the attribute *NamedElement::name*.

3.6.5 Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes

By means of their language specification artefacts, this concern cannot be addressed with Xtext.

3.6.6 Concern 5: Mapping an LHS non-terminal to many meta-classes

By means of their language specification artefacts, this concern cannot be addressed with Xtext. That said, from a technical point of view, a language engineer could customise the generated Java code so that a post-processor can actually create the additional model elements and add them to the model element that conforms to the mapped meta-class.

3.6.7 Concern 6: Multi-way mappings from LHS non-terminals to meta-classes

By means of their language specification artefacts, this concern cannot be addressed with Xtext.

3.6.8 Concern 7: Mapping properties from non grammar terms

By means of their language specification artefacts, this concern cannot be addressed with Xtext. That said, from a technical point of view, a language engineer could customise the generated Java code so that a post-processor can actually compute all the missing properties.

3.6.9 Concern 8: Name Resolution

By means of their language specification artefacts, Xtext can partially cope with the name resolution concerns. That said, from a technical point of view, a language engineer could customise the generated Java code so that the required lookup algorithms are consumed by the Xtext name resolution infrastructure.

With respect to how name resolution concerns are partially addressed, Xtext provides a default name resolution support so that Xtext grammars can be enough for certain textual modeling languages. To explain the underlying behaviour, Listing 3.16 shows the Xtext grammar corresponding to the Listing 3.11 from Section 3.3.2. The EBNF grammar excerpt has been reworked so that it is a valid Xtext example that exposes the name resolution default behaviour.

The default name resolution support is characterised as follows:

```
1  ClassCS returns Class:
2    'class' name=ID
3      'extends' (superClasses+=[Class|QualifiedID] (',' superClasses+=[Class|
           QualifiedID] )* )? '{'
4    // the remainder of Class definition syntax has been omitted
5    '}'
6  ;
7
8  QualifiedID returns Ecore::EString:
9    ID ('::' ID)*
10 ;
```

LISTING 3.16: Xtext grammar of a *Class* extension definition

- Firstly, name resolution is performed to compute cross-references. In line 3, the *superClasses* reference name that appears before the '+=' operator is a cross-reference. In this case, it is a cross-reference of the *Class* meta-class (the return type of that syntactic rule).
- Name resolution is expressed by means of the *'['* and *']'* operators that appear after the assignment operator ('+=') (see line 3). The type of the element to look up is provided in square brackets, followed by a separator '|' to specify which is the token that provides the input (i.e. String value) to perform the name-based lookup.
- All AS meta-classes with an attribute *name* are possible targets of Xtext name resolution. In other words, any named target forcibly requires an attribute called *name* (see line 2).
- There is a default scoping policy, as long as inner named targets are directly contained by another named target. Otherwise, all non-scoped targets are visible across the entire AS model. In our example, the default scoping policy would apply, because *Class* model elements are directly contained by *Package* model elements.
- More complex qualified named-based lookups are also supported by default, based on the default scoping policy. In other words, a non-visible *Class* model element is accessible, as long as the fully qualified name of that model element (i.e. writing *<myPackageName>::<myClassName>*) is provided.

Making name resolution work in Xtext requires designing the grammar with some restrictions in mind (i.e. forcing the AS to adopt the Xtext conventions). Although many of the Xtext conventions are reasonable, they may not adjust to a particular AS meta-model definition. Moreover, some name resolution scenarios introduced previously do not adjust to the default behaviour, or simply the default name resolution scenario may not apply to a particular textual modeling language.

3.7 Summary

Chapter 3 presented an analysis of the different concerns that need to be taken into account when bridging the CS and AS of CTMLs.

Firstly, some additional terminology and the running example were introduced in Section 3.1 and Section 3.2 respectively.

Then, the difficulties related to bridging CS and AS were detailed in Section 3.3 and Section 3.4, from a coarse grained and fine grained perspective respectively.

Name resolution is a substantial concern that was introduced in its own Section 3.5, showing the different scenarios that a name resolution activity needs to address.

Finally, it was pointed out how Xtext can address some of the presented concerns. At the same time, the limitations that prevent it from addressing some of the other concerns were likewise introduced.

Chapter 4

Solution Design & Implementation

In the previous chapter, all the different concerns that arise when bridging the CS and AS of a Complex Textual Modeling Language (CTML) were explained. This chapter focuses on describing a solution proposed to address all these problems, including the technical description of the prototype developed to realise that solution and the limitations that the solution provides.

4.1 Solution Overview

This section introduces the overall solution, identifying the existing technologies that are reused within the whole approach, and the different layers or stages in which that approach is divided, including discussions about related work. Further sections focus individually on the technical details of each stage, including how the solution integrates within a modern language workbench such as Xtext.

4.1.1 Overall Approach

The overall approach is depicted in Fig. 4.1. As occurs with traditional approaches, the CS for a particular textual language is given in the form of a grammar ❶. Likewise, the approach relies on existing related work to move from *grammarware* to *modelware*, so that a meta-model ❷ can be derived automatically from the grammar definition. However, this meta-model (referred to as the CS meta-model) represents the syntactic structure of the textual input. Therefore, it can differ substantially from the actual meta-model ❸ that represents the AS of the modeling language. Expressing potentially complex bridges ❹ between the CS and AS of a CTML consists of establishing mappings between the concepts of the CS and AS meta-models. In the same way that current technologies are able to generate the parser ❺ responsible for consuming textual inputs to produce the corresponding CS models, the proposed solution generates the M2M transformation ❻ responsible for transforming CS models into AS models.

With respect to how the M2M transformation is generated from CS2AS bridge descriptions, Figure 4.2 shows the overall approach. Instances of a new external DSTL ❶ [65], called CS2AS-TL, are in charge of bridging the gap between CS and AS meta-models. These instances are then transformed into a set of Complete OCL files, according to an OCL-based internal DSTL ❷ [66]. This set of Complete OCL files are transformed into a declarative

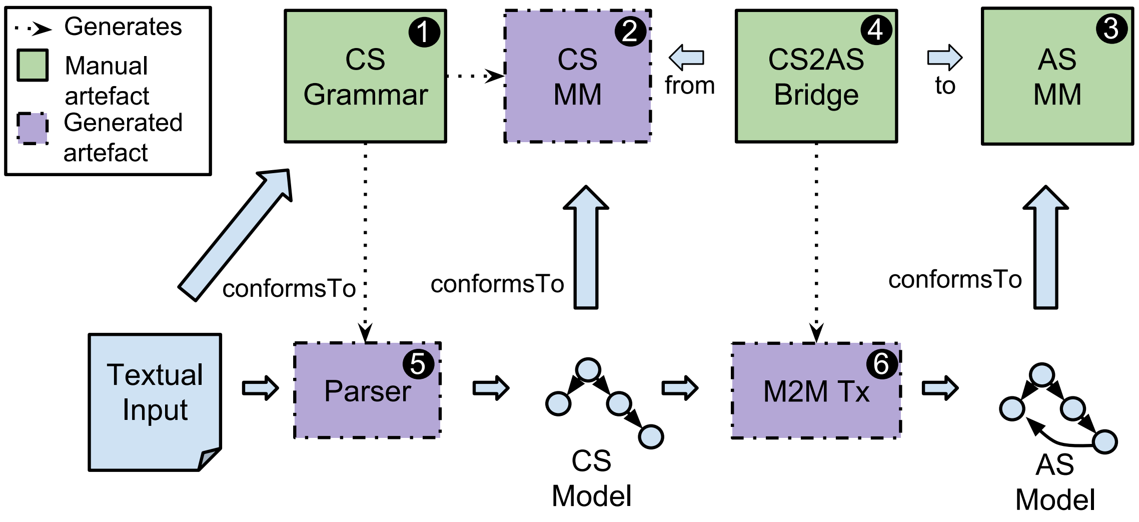


FIGURE 4.1: Overall approach

M2M transformation, in particular one of the low-level declarative QVT languages ❸ [76]. Finally, a Java-based transformation ❹ is generated that is capable of consuming CS models to produce AS ones.

Design and implementation details about this overall solution are shown in further sections of this chapter. The following subsections discuss the overall approach, including comments about related work.

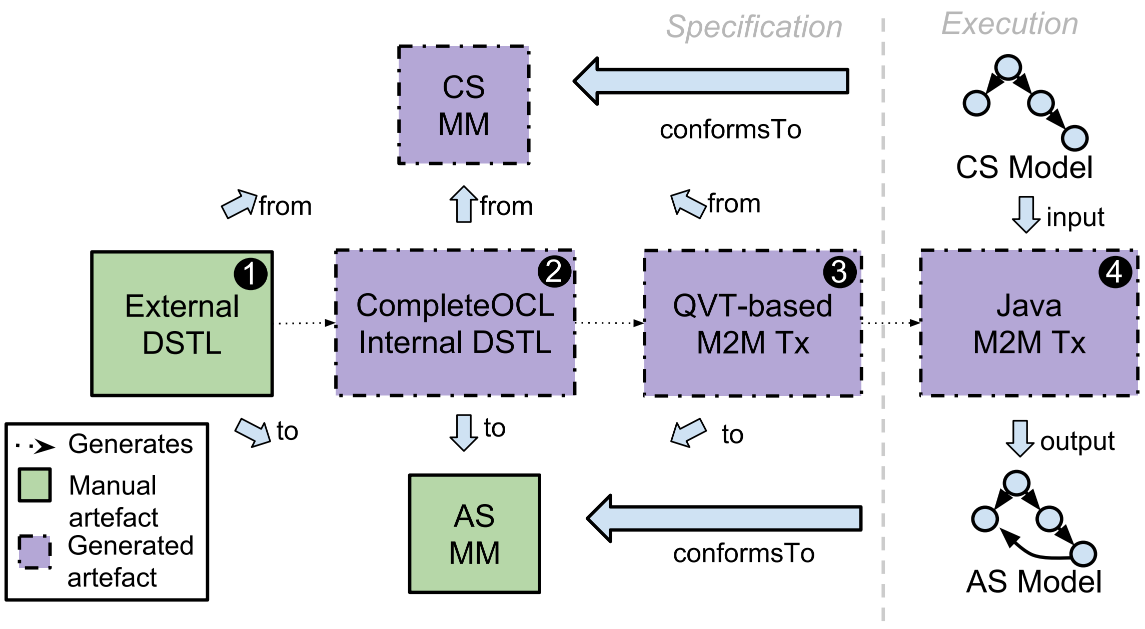


FIGURE 4.2: Compilation process from CS2AS-TL to Java

All the developed source code is publicly available in a GitHub repository[a] so that it can be consulted and used as required. The third party code used by the prototype is also publicly available in official Eclipse repositories[bc].

[a]`https://github.com/adolfosbh/cs2as`
[b]`https://git.eclipse.org/r/ocl/org.eclipse.ocl`
[c]`https://git.eclipse.org/r/mmt/org.eclipse.qvtd`

4.1.2 Approach Discussion

Grammarware-to-*Modelware* vs CS2AS

The gap between CS and AS requires to be bridged, and there should be a technological space shift from *grammarware* to *modelware*. At runtime, it turns out that a textual input (conforming to the CS) needs to be transformed into a model (conforming to the AS).

From a technological space perspective, the solution proposed in this thesis works as shown in Figure 4.3. Firstly, a *grammarware* to *modelware* technological space shift is performed, so that a CS model is obtained from the corresponding textual input file. This model conforms to the so-called CS meta-model, which can be derived automatically (as Xtext does) with no requirement to comply with a particular structure or design. Then, bridging the gap between the CS and AS can be performed by means of M2M transformations so that the final AS model can be obtained. Therefore, this CS2AS activity entirely operates in the *modelware* technological space.

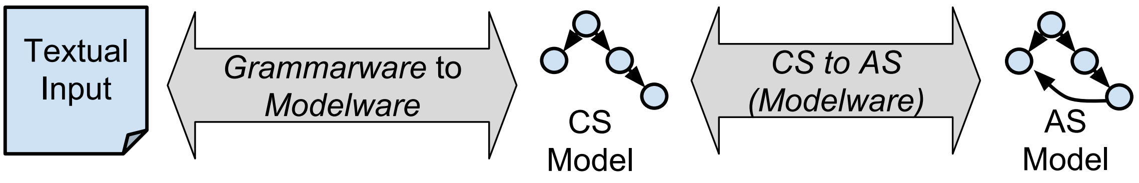


FIGURE 4.3: Grammarware-to-Modelware vs CS-to-AS

Reformulating the CS2AS Fine Grained Concerns

In order to understand how the solution can address the fine grained concerns explained in Chapter 3, a reformulation is required. This requirement comes from the fact that the CS2AS bridges are established in the *modelware* technological space. Therefore, concepts from the *grammarware* technological space (e.g. terminals, non-teminals etc.) no longer apply.

The reformulation relies on how current related work deals with *grammarware-to-modelware* bridges, i.e:

- An LHS non-terminal corresponds to a CS meta-class.
- An RHS non-terminal corresponds to a CS reference.
- A terminal corresponds to a CS attribute.

In consequence, the seven concerns presented in Section 3.4 are reformulated as follows:

- Concern 1: Mapping a CS meta-class to an AS meta-class.
- Concern 2: Mapping a CS reference to an AS reference.
- Concern 3: Mapping a CS attribute to an AS attribute.
- Concern 4: Mapping a CS reference to an AS reference and additional AS meta-classes.
- Concern 5: Mapping a CS meta-class to many AS meta-classes.
- Concern 6: Multi-way mappings from CS meta-classes to AS meta-classes.
- Concern 7: Mapping AS properties from non CS meta-model terms.

CS2AS As a *Modelware* Activity

The activity of bridging the CS2AS gap has been reformulated as an activity of transforming models. In consequence, this part of the solution operates in the *modelware* technological space. Likewise, all the facilities that M2M transformations offer can be used for solving the complexities of dealing with CTMLs. Section 4.2 shows how they are particularly addressed within CS2AS-TL.

Moreover, moving away from the *grammarware* technological space brings additional advantages that are summarised below:

- Parsing-related concerns do not have to be considered by CS2AS-TL. Dealing with ambiguous context-free grammars, left/right recursion, operators precedence etc. [1] are *grammarware* concerns that still need to be addressed by the particular parser technology or language workbench.
- Although this thesis presents a complete solution based on Xtext, the overall approach is independent of a parsing technology (ANTLR [61], LPG [55]) or a particular language workbench (Xtext [21] or IMP [14]). The requirement for the proposed approach is the existence of a CS meta-model.

Testing the Approach

One of the concerns that the reader may find when analysing the proposed approach is how it can be tested. Taking into account that the approach proposes the generation of M2M transformations, the way of testing the original CS2AS bridges consists of applying testing solutions from the M2M field. In particular, test cases are designed to check that output AS models produced from M2M transformations are the same as the AS models expected by the particular CS2AS scenario that is tested.

During this research project, different CS2AS scenarios have been tested. They are hosted in the examples folder of the Eclipse QVTd project[1]. The test cases load generated M2M

[1] `https://git.eclipse.org/r/mmt/org.eclipse.qvtd`

transformations, which take a set of textual inputs conforming to the concrete syntax of test languages. Then, the loaded transformations produce the corresponding AS models, which are compared to reference AS models.

4.1.3 Related Work Discussion

To conclude the solution overview, this subsection presents a brief discussion on how the proposed overall solution relates to previous work.

CS Meta-Model

The first body of work to mention is Muller et al. [58, 59], in which they introduce the concept of the CS meta-model. However, this concept has some differences with respect to the one introduced in the previous subsection. Figure 4.4 shows these differences. Muller et al. [58, 59] propose just one CS meta-model ❶ that plays the role of the EBNF notation ❷. In this way, specifying the CS of a language consists of creating a CS model ❸ conforming to that CS meta-model ❶. Additional features of this meta-model let a language engineer relate CS concepts to AS meta-classes and properties of the target AS meta-model ❹. In the case of the proposed solution, a language-specific CS meta-model ❻ is derived automatically from a grammar definition ❺. Then, an external artefact ❼ is in charge of relating concepts from the specific CS meta-model and the AS one ❽.

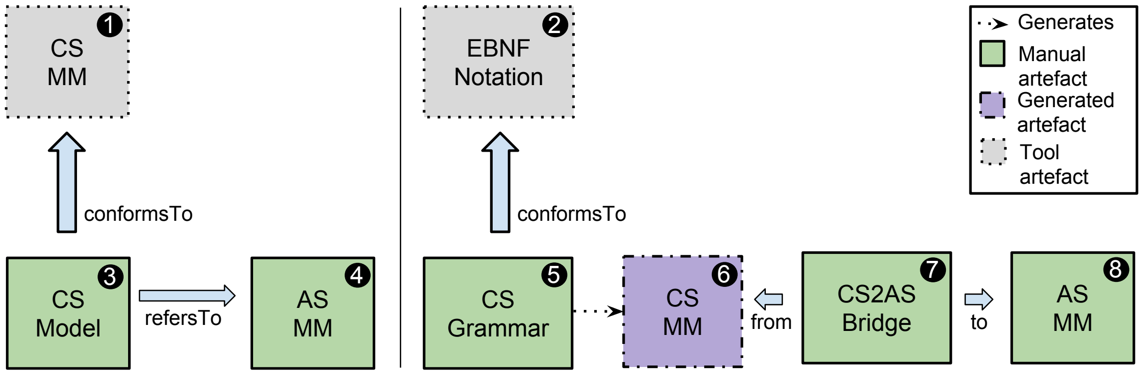
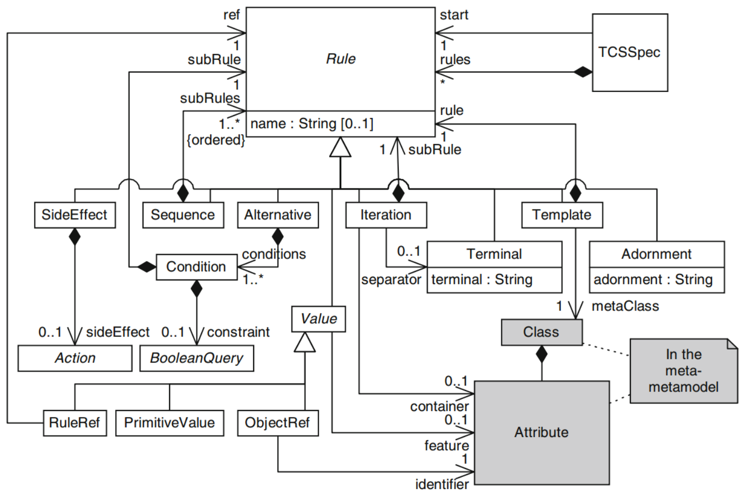


FIGURE 4.4: General CS meta-model vs language-specific CS meta-model

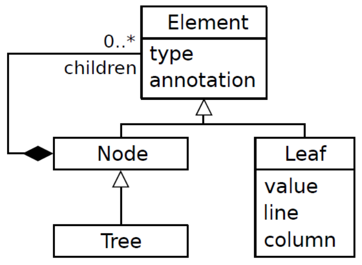
Looking at the principles of their CS meta-model [59], their approach has the same limitations as Xtext (see Section 3.6). Although Muller et al. do not go into detail about the different scenarios they do not support, they mention for instance that they cannot define a "full type system". As will be mentioned in Section 4.2, CS2AS-TL allows the computation of the properties in charge of defining the type system of a language.

Cánovas et al. [13, 42] also introduce a similar concept, called Concrete Syntax Tree (CST) meta-model. However, contrary to Muller et al.'s approach, this meta-model only creates a structured model with the syntactic information of the input file. Figure 4.5 shows Muller et al.'s CS meta-model (4.5a) and Cánovas et al.'s CST meta-model side by side (4.5b).

When compared to the proposed solution, Cánovas et al.'s CST meta-model [13] is discussed along with Figure 4.6. As with the proposed solution, a language CS definition is



(A) A general CS meta-model from [59]



(B) A CST meta-model from [13]

FIGURE 4.5: CS meta-model based approaches

specified by means of a CS grammar ❶ that conforms to the EBNF notation ❷. However, there is no language-specific CS meta-model generation. Instead, an internal and homogeneous CST meta-model ❸ is used, so that the specific parser ❹ can produce a first model ❺ with the syntactic tree structure of the corresponding input file. Besides, although the definition of the CS2AS bridge is defined by means of another external DSTL ❻, the instances of that language refer (by name) to the grammar terms ❶, rather than the CS meta-model terms.

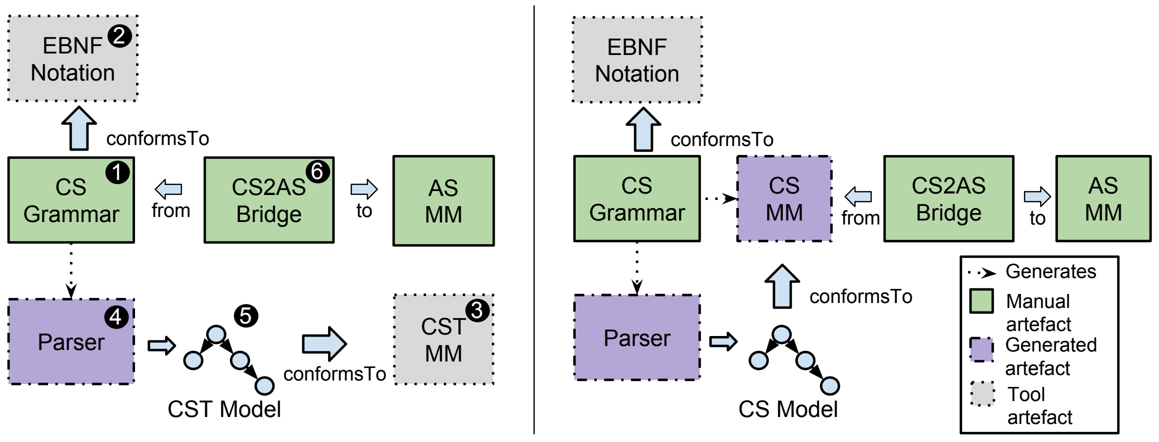


FIGURE 4.6: General CST meta-model vs language-specific CS meta-model

Grammarware-to-*Modelware* vs CS2AS

The previous subsection discussed how bridging *Grammarware-to-Modelware* and the CS2AS was distinguished within the proposed approach. This subsection shows a simplified vision of how related work tackles these activities, depicted in Figure 4.7.

In contrast to how the proposed approach addresses these activities (see Figure 4.3), Xtext ❶ tackles them in one shot. This is done by enhancing the grammar specification language with additional syntax to map the different grammar terms to meta-model terms. Gra2Mol ❷ presents a similar approach, although the mappings between grammar and meta-model terms are done by means of an external DSTL. Spoofax ❸ works with its own representation of AST and, therefore, it requires an additional step to transform ASTs into AS models.

⚠ Although it was previously mentioned that Gra2Mol uses an intermediate CST meta-model, the latter is an implementation internal artefact. Conceptually, they present their CS2AS bridges as a bridge between grammars and meta-models.

With respect to Xtext, Section 3.6 presented some limitations of the corresponding approach. Regarding Gra2Mol and Spoofax, Chapter 5 presents a comparative study.

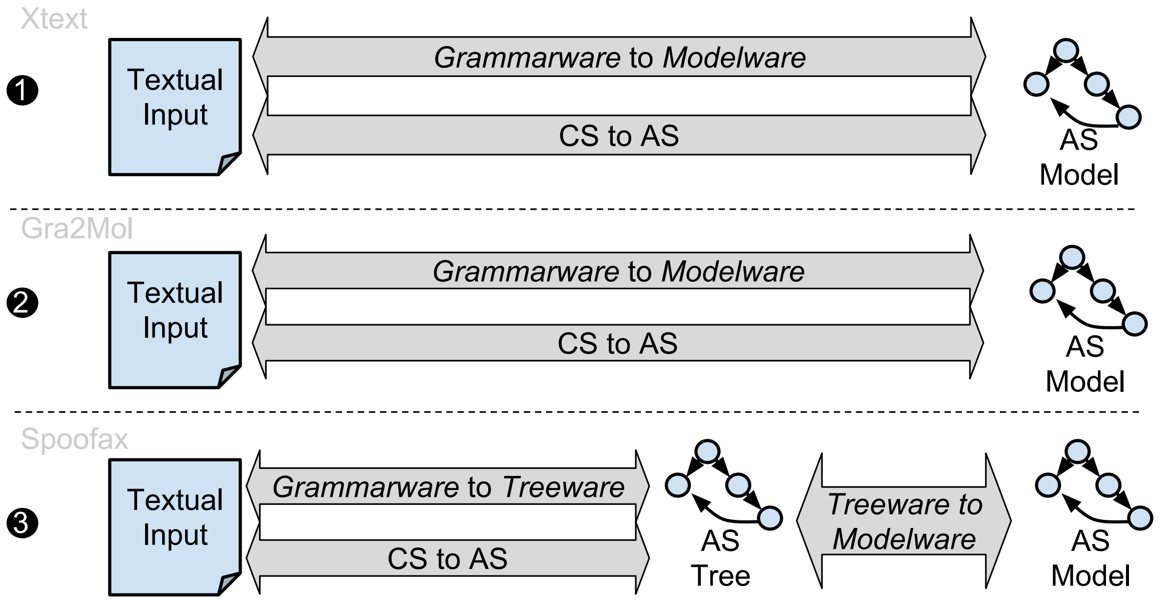


FIGURE 4.7: How Xtext ❶, Gra2Mol ❷ and Spoofax ❸ tackle the *grammarware-to-modelware* techonological space shift.

4.2 Concrete Syntax to Abstract Syntax Transformation Language

Having outlined the overall solution, this section presents the main contribution of this thesis: a Domain Specific Transformation Language (DSTL) designed to define bridges between the CS and the AS of Complex Textual Modeling Language (CTML)s. This new language is called Concrete Syntax to Abstract Syntax Transformation Language (CS2AS-TL).

4.2.1 Overview

As part of the overall solution that was shown in the previous Section 4.1.1, a new external DSTL is proposed to define CS2AS bridges (see Figure 4.1). In fact, given that a *grammarware-to-modelware* shift previously takes place, CS2AS-TL is a M2M transformation language that maps CS meta-models to AS meta-models.

Rather than using a general purpose M2M transformation language, a domain-specific one has been designed and developed to provide a more concise language that addresses the concerns presented in Chapter 3. In particular, it is considered as domain-specific because:

- **One input and output domain**. CS2AS model transformations involve just one source input domain and one target output domain. There is no need to support in-place transformations.
- **Specific name resolution related constructs**. The language comprises additional constructs to define name resolution in a declarative and concise manner.
- **Specific disambiguation rules**. With the aim to support multi-way mappings, there is a dedicated section to specify the disambiguation rules that drive this kind of mapping execution.

Some high-level features that characterise this CS2AS-TL are the following:

- It is a declarative language, meaning that the comprising mappings specify which AS concepts (meta-classes, properties etc.) are mapped from which CS ones, but there is no control specification about how the transformation actually executes. Because of this, bridges expressed in the language tend to be more concise than bridges expressed in imperative languages.
- The base expression language corresponds to Essential OCL [39]. This facilitates the development of the subsequent compilation steps. Additionally, this feature also simplifies the development of tools designed to generate parts of OMG specifications, such as OCL and QVT.

The language has been developed with Xtext [85]. It is divided into five parts: preliminaries, helpers section, mappings section, disambiguation section and name resolution section. The upcoming subsections delve into more detail on the CS2AS-TL syntax, but a brief description about these five parts is introduced here. Additionally, Figure 4.8 shows the four main language parts (called sections) and the dependencies between them.

- The **preliminaries** part declares the source and target domains involved during the transformation.
- The **helpers section ❶** declares reusable functionality in the form of helper operations.
- The **name resolution section ❷** declares which AS concepts are involved in name resolution. In essence, targets, inputs and providers (including how they contribute targets to lookup scopes) are declared in this section. Helpers may be reused within this section.

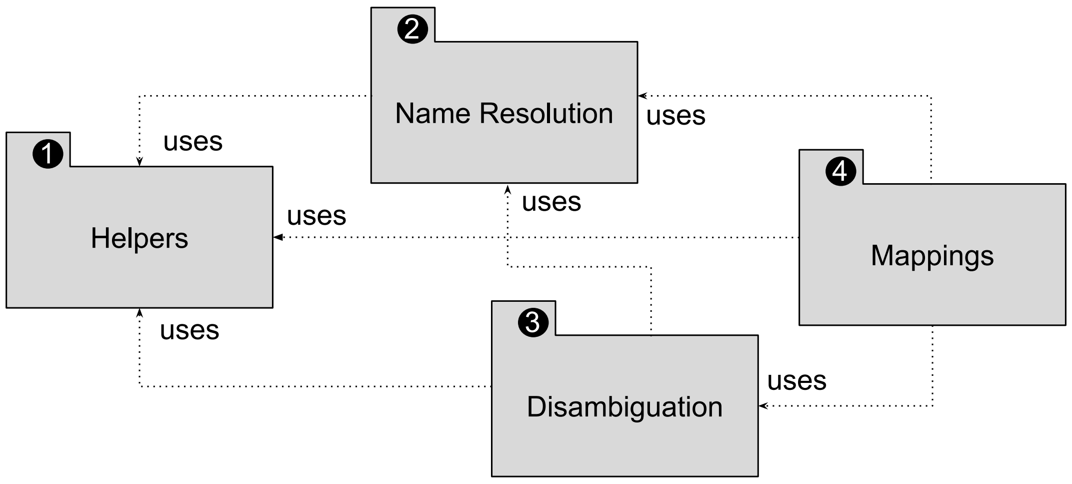


FIGURE 4.8: Main CS2AS-TL sections and dependencies

- The **disambiguation section** ❸ declares the CS disambiguation rules required by multi-way mappings. Helpers may be reused within this section. Name resolution may be performed from these rules.
- The **mappings section** ❹ declares how AS concepts are mapped from the CS ones. Helpers may be reused and name resolution may be performed to compute AS properties. Multi-way mappings refer to the rules declared in the disambiguation section.

4.2.2 Expressions Syntax

The previous subsection introduced an overview of CS2AS-TL. In the following subsections the whole syntax of the language is explained. Since CS2AS-TL has been designed by means of Xtext, the language syntax is presented in the form of Xtext grammar excerpts, along with some example scenarios to show syntax usage.

CS2AS-TL reuses Essential OCL [39] as the expression language to support navigation of the models involved. Additionally, Essential OCL is expressive enough for creating elaborated expressions with potentially complex computations. In addition to the set of expressions from Essential OCL, new expressions are required which are introduced in the following sub-subsections.

Shadow Expressions

Shadow expressions have been discussed as a possible addition (Section 3.1 from [11]) to the OCL specification. Shadow expressions permit the construction of objects in a record-based representation. Aligned with OCL tuples approach, these constructed *shadow* objects do not have an *object ID* and do not impact on the system state. Therefore, they can be considered as side-effect free shared objects.

Listing 4.1 shows the syntax to declare shadow expressions. The syntax is similar to OCL *Tuple* literal expression declarations. Firstly, instead of a *Tuple* keyword, a path name is used to identify the type of the object to create (line 2). Then, a list of comma-separated parts can be specified (line 3), enclosed by a pair of curly brackets (lines 1 and 4). Each part consists of a property (of the constructed object type) name, followed by the '=' symbol and the corresponding expression that computes the property value (line 7).

```
1 ShadowExpCS:
2   pathName=PathNameCS '{'
3   (parts+=ShadowPartCS (',' parts+=ShadowPartCS)*)?
4   '}'
5 ;
6 ShadowPartCS:
7   propertyName=ID '=' initExp=ExpCS
8 ;
```

LISTING 4.1: Shadow expression syntax

Eclipse OCL supports the usage of shadow expressions. Since CS2AS-TL Xtext grammar extends the Essential OCL one from the Eclipse project, CS2AS-TL does not actually need to provide new syntax for them.

Listing 4.2 shows a simple example in which a *Variable* model element is constructed using a shadow expression. Line 1 declares the name of the shadow object type. Line 2 declares a part of the shadow expression, in which the *Variable::name* attribute is initialised to *'self' String* value.

The usage of these shadow expressions is common in the mappings section, and they are mainly used to declare which additional model elements are created as a result of a mapping execution. They are used for addressing Concerns 4 and 5 from the previous chapter.

```
1 as::Variable {
2   name = 'self'
3 }
```

LISTING 4.2: Shadow expression usage example

Trace Expressions

Trace expressions are an addition to the set of Essential OCL expressions that can be used across the transformation language. The concept is borrowed from M2M languages like Epsilon Transformation Language (ETL), ATL Transformation Language (ATL) and QVTo, which provide special constructs to return target (output) model elements from source (input) ones. During execution, this traceability is internally managed by means of traces that are updated when the different mappings are executed.

Listing 4.3 shows the syntax to declare trace expressions. The syntax simply consists of the *trace* keyword (line 3).

```
TraceExp:
  {TraceExp}
  'trace';
```

LISTING 4.3: Trace expression syntax

Listing 4.4 shows a simple example in which trace expressions are used. The example corresponds to a simplified version of a mapping that produces *LetExp* model elements from *LetExpCS* model elements (line 1). Line 4 shows the usage of a trace expression. Whatever CS model element is contained by the contextual *LetExpCS* model element, via the *LetExpCS::inExp* containment reference, the corresponding AS model element (another *OCLExpression*) is returned. The type of the created *LetExp* model element is the type of its inner *OCLExpression*.

The usage of these trace expressions is common in the mappings section, particularly when specifying the OCL expression designed to provide the initial value of an AS property. They are mainly used for addressing Concern 7, explained in Section 3.4.8.

```
1 create LetExp from LetExpCS {
2   ownedVariable := letVar; -- equivalent to  letVar.trace
3   ownedIn := inExp; -- equivalent to  inExp.trace
4   type := inExp.trace.type;
5 }
```

LISTING 4.4: Trace expression usage example

Trace expressions are a kind of *CallExp*, which requires that the object on which they are evaluated belong to the source (input) domain. This means that only the target domain can be accessed from the source domain, and not the other way around.

Lookup Expressions

Finally, lookup expressions are added to the set of Essential OCL expressions that are supported by CS2AS-TL. This kind of expressions is used for performing name-based lookups, so that the result obtained by the lookup can be used to set cross-references of target AS model elements (name resolution consumers). They are mainly used for addressing Concern 8, explained in Section 3.5.

Lookup expressions are explained in Section 4.2.5, where the syntax to declare name resolution is shown.

4.2.3 Preliminaries Syntax

The main goal of the preliminaries is to declare the source and target domains, i.e. which files correspond to the CS and AS meta-models. A domain is defined by one or more meta-models, comprising all the meta-classes and data types that are available at design time.

Listing 4.5 shows the corresponding syntax. Source (lines 1–2) and target (lines 3–4) domain declarations comprise a list of comma-separated meta-model declarations (at least one is required). The latter (lines 5–6) consists of an optional identifier to denote an alias and a Uniform Resource Identifier (URI) path name, which permits specifying the URI of the corresponding meta-model file. The previous Listing 4.6 shows an example of the notation.

```
1 SourceDomain:
2   'source' metamodels+=MetaModelDecl (',' metamodels+=MetaModelDecl)* ;
3 TargetDomain:
4   'target' metamodels+=MetaModelDecl (',' metamodels+=MetaModelDecl)* ;
5 MetaModelDecl:
6   (alias=ID ':')? uri=STRING;
```

LISTING 4.5: Source and target domain syntax

Listing 4.6 shows an example of this declaration, where the mOCL CS meta-model generated by Xtext is declared (line 1). The manually created AS meta-model is likewise declared (line 2).

```
1 source cs : 'generated/MiniOCLCS.ecore#/'
2 target as : '/org.eclipse.qvtd.doc.miniocl/model/MiniOCL.ecore#/'
```

LISTING 4.6: Source and target domain declaration

4.2.4 Helpers Section Syntax

The helpers section is designed to declare reusable functionality. In particular, contextual operations can be defined so that they can be invoked by any OCL expression.

Listing 4.7 shows the syntax of the helpers section. Lines 1–4 show the syntax of the top level declaration of the section. It consists of the *helper* keyword (line 2) and an arbitrary number of helper class declarations (line 3) enclosed by a pair of curly brackets (lines 2 and 4). Lines 5–8 show the syntax of the helper class declaration. It consists of a context declaration (line 6) and an arbitrary number of class helper operation declarations (line 7) between a pair of curly brackets (lines 6 and 8). Lines 9–13 show the syntax of the helper operation declaration. It consists of a name (line 10), an arbitrary number of parameter definitions between rounded brackets (line 11), the operation result type (line 12) and the helper body (line 13), which is an expression. Finally, lines 15–16 show a parameter definition. It consists of a name and the corresponding type declaration (line 16).

Listing 4.8 shows some general helpers that are required by mOCL. For instance, the *commonSupertype* helper operation (lines 3–6) can be invoked on a *Class* model element (line 2) to determine which is the common super class between that *Class* model element and an *another Class* model element (line 3). An additional *conformsTo* helper operation (lines 7–9) can be invoked to determine whether the contextual *Class* model element (line 2) is the same type or subtype of *another Class* model element (line 7).

4.2.5 Name Resolution Syntax

The name resolution section is designed to declare how name resolution (Concern 8 from Section 3.5) is addressed for a particular language.

```
HelpersSect:
    {HelpersSect} 'helpers' '{'
        classHelpers+=ClassHelper∗
    '}' ;
HelperClass:
    context=PathNameCS '{'
        helpers+=HelperDef∗
    '}' ;
HelperOperation:
    name=ID
    '(' (params+=ParameterDef (',' params+=ParameterDef)∗)? ')'
    (':' type=PathNameCS)?
    ':=' helperBody=ExpCS
    ';' ;
ParameterDef:
    name=ID ':' type=PathNameCS ;
```

LISTING 4.7: Helpers section syntax

```
helpers {
  as :: Class {
    commonSupertype(another : Class) : Class :=
      let allSupertypes = self−>asOrderedSet()−>closure(superClasses),
          allOtherSupertypes = another−>asOrderedSet()−>closure(superClasses)
      in allSupertypes−>intersection(allOtherSupertypes)−>any(true) ;
    conformsTo(another : Class) : Boolean :=
        self = another or
        superClasses−>exists(conformsTo(another));
  }
}
```

LISTING 4.8: Helpers section example

When the concern was analysed in Section 3.5, four different roles were identified: targets, inputs, providers and consumers. This name resolution section deals with the first three. Regarding the fourth, consumers make use of *lookup* expressions to perform a name-based lookup. Lookup expressions are explained at the end of this subsection.

Listing 4.9 shows the syntax of the top level declaration of the section. It consists of the *name_resolution* keyword (line 3). Then, a pair of curly brackets (lines 3 and 7) enclose the declaration of *targets* (line 4), *inputs* (line 5) and *providers* (line 6) clauses.

```
1 NameResolutionSect:
2     {NameResolutionSect}
3     'name_resolution' '{'
4         targetsDef=Targets?
5         inputsDef=Inputs?
6         providersDef=Providers?
7     '}';
```

LISTING 4.9: Name resolution section syntax

The following sub-subsections delve into the details of these three clauses. The fourth sub-subsection details how the consumers perform name resolution by means of lookup expressions.

Targets

The first clause, called *targets*, is used for declaring which are the AS meta-classes of targets of name-based lookups (including an identifying identifier expression). Optionally, an additional filter definition can be used for filtering the possible candidates that match a particular name. Targets can also qualify other targets.

Listing 4.10 shows the syntax to declare the targets involved in a name resolution activity. It consists of the *targets* keyword (line 3), followed by an arbitrary number of target declarations (line 4) between a pair of curly brackets (lines 3 and 5). A target declaration consists of the name of an AS meta-class (line 7), optionally followed by the *using* keyword and an expression that evaluates to a (primitive) value to identify the target (line 8); this expression is referred to as **identifying expression**. An optional *ignore_case* keyword (line 8) can be specified, so that the target name-based lookups are name case-insensitive. An optional definition of escape sequences (line 9) can be defined, so that a prefix, and optionally a suffix (line 16), can be used to match target names. Then, an optional filter definition (line 10) and a qualification definition (line 11) complete the target definition (further explained).

Escape sequences for name lookups are used in order to find targets that clash with language keywords (e.g. a property of a meta-class named *and*). In previous versions of OCL, this was supported by preceding the target name with an underscore symbol (e.g. *_and*). Although this approach is not recommended (e.g. an *and* property cannot be accessed if there exists also an *_and* property), CS2AS-TL supports it by means of these escape sequence definitions.

```
Targets:
    {Targets}
    'targets' '{'
        targets+=Target∗
    '}' ;
Target:
    classRef=PathNameCS
    ('using' identifyingExp=ExpCS ignoreCase?='ignore_case'?
        escapingSeqDef=EscapeSequenceDef? )?
     filter =FilterDef?
     qualification =QualificationDef?
    ';' ;
EscapeSequenceDef:
    'escaped_with' escapes+=EscapeSequence (',' escapes+=EscapeSequence)∗ ;
EscapeSequence:
    prefix=StringLiteral  ('and' suffix=StringLiteral)? ;
```

LISTING 4.10: Targets definition syntax

The specification of the identifying expression can be omitted as long as the target meta-class extends (directly or indirectly) another target meta-class for which an identifying expression has been defined (See the example below).

Listing 4.11 shows a simple example where a meta-class *NamedElement* is declared as a target type and the expression *name* is used for identifying *NamedElement* targets. Additionally, the meta-class *Class* is also declared as a possible lookup target. Since *Class* extends *NamedElement*, the declaration of the identifying expression is omitted.

```
targets {
    NamedElement using name escaped_with '_';
    Class;
}
```

LISTING 4.11: Targets definition example

As explained in Section 3.5.7, when looking up a particular target, additional inputs may enhance the lookup criteria to choose among several candidates that may match a particular name. An example of this scenario in mOCL occurs when looking up *Operation* model elements. Not only is the name of the operation required, but also additional arguments (from the operation call) are used to discriminate among different operations with the same name.

Listing 4.12 shows the syntax of a filter definition. It consists of the *filtered_by* keyword, followed by an arbitrary number of parameters to declare the additional input that is required, and a *when* condition expression that refers to these parameters. This condition is used for evaluating whether the particular target fully complies with the lookup criteria.

Listing 4.13 shows an example of how the *Operation* meta-class is declared as a target of name resolution. It requires an additional *arguments* filtering parameter (line 3) to enhance

```
1 FilteringDef:
2    'filtered_by'
3    params+=ParameterDef (',' params+=ParameterDef)*
4    'when' expression=ExpCS;
```

LISTING 4.12: Filters definition syntax

the lookup criteria. This parameter is used by a filtering condition (lines 4–7) to filter the adequate *Operation* model elements. In this case, the condition consists of checking that the number of operation parameters is the same as the number of arguments (the additional input that forms the lookup criteria). Additionally, the type of every argument must conform to the type of the corresponding operation parameter.

```
1 targets {
2    NamedElement using name;
3    Operation filtered_by arguments : OrderedSet(OCLExpression)
4       when ownedParameters->size() = arguments->size() and
5            arguments->forAll(x |
6                  let argIdx = arguments->indexOf(x)
7                  in x.type.conformsTo(ownedParameters->at(argIdx).type));
8 }
```

LISTING 4.13: Filters definition example

In mOCL, *OperationCallExp* model elements are the consumers of *Operation* model elements. The former contain and provide the required *arguments* to perform lookups of the latter.

As explained in Section 3.5.5, a target can qualify other targets, with the aim of facilitating qualified name-based lookups. For instance, *Property* model elements can be looked up by using their qualified name, which consists of the name of their own *Property* model element, preceded by the '::' symbol-separated list of its transitive qualifiers. In this case, *Property* model elements are qualified by *Class* model elements, which are qualified by *Package* model elements. In CS2AS-TL, the qualification definitions rely on declarations specified by the qualifier target. In this way, the qualifier target declares, by means of OCL expressions, which are the actual model elements that it qualifies.

Listing 4.14 shows the syntax of a target qualification definition. It consists of the *qualifies* keyword (line 2), followed by an arbitrary number (at least one) of qualifications (line 2). A qualification consists of one or more path names to refer to the qualified class/es (line 4), followed by the *using* keyword and a comma-separated list of OCL expressions that comprise the model elements that are actually qualified (line 5).

CS2AS-TL requires that the referred meta-class/es of a qualification definition is/are declared as a target.

```
QualificationDef:
   'qualifies' qualifications+=Qualification (',' qualifications+=Qualification)∗ ;
Qualification:
   qualifiedClasses=MultiplePathNames
   'using' contributions+=ExpCS (',' contributions+=ExpCS)∗ ;
MultiplePathNames:
   pathNames+=PathNameCS (',' pathNames+=PathNameCS)∗ ;
```

LISTING 4.14: Qualification definition syntax

Listing 4.15 shows an example of the declarations required to allow qualified name lookups of *Property* targets. Lines 3–5 declare that *Package* model elements qualify other (nested) *Package* model elements and *Class* model elements. Regarding the latter, the actual qualified model elements are those owned via the *Package::ownedClasses* containment reference. Additionally, lines 6–7 declare that *Class* model elements qualify *Property* model elements, particularly those owned via the *Class::ownedProperties* containment reference.

```
1  targets {
2     NamedElement using name;
3     Package qualifies
4                 Package using ownedPackages
5                 Class using ownedClasses
6     Class qualifies
7                 Property using ownedProperties;
8     Property;
9  }
```

LISTING 4.15: Qualified/qualifier targets example

Inputs

The second clause, called *inputs*, is used for declaring which kind of CS information takes part in the lookup criteria. By default, any (primitive) value that conforms to the type of the identifying expression of a target can be used as a lookup input. However, additional model elements that conform to CS meta-classes can also be used as alternative lookup inputs. These additional input types have to be declared in this clause. An input type declaration may refer to CS meta-classes designed to comprise qualified names.

Listing 4.16 shows the syntax to declare the type of inputs involved in a name resolution activity. It consists of the *inputs* keyword (line 3), followed by an arbitrary number of input declarations (line 4) enclosed by a pair of curly brackets (lines 3 and 5). An input declaration consists of a name of a CS meta-class (line 8), followed by the *using* keyword and a expression that is used for matching targets. Therefore, the type of this expression must be another input type (explicitly defined or any default input type). Finally, the input type declaration may be preceded by the *qualifier* keyword (line 7), meaning that the input comprises a qualified name (or a list of inputs).

```
Inputs:
    {Inputs}
    'inputs' '{'
        inputs+=Input*
    '}';
Input:
    ( qualifier ?='qualifier')?
    typeRef=PathNameCS ('using' matchExp=ExpCS)? ';' ;
```

LISTING 4.16: Inputs definition syntax

Listing 4.17 shows the type of inputs defined for mOCL. Apart from the default *String* data type to match *NamedElement* targets, two additional input types are declared. A first input type *PathElementCS* is declared (line 3), being the *elementName* expression (whose type is *String*) used for matching targets. A qualifier input type *PathNameCS* is also declared (line 4), being the *pathElements* expression (whose type is a collection of *PathElementCS*), used for comprising the list of inputs for a qualified name.

```
1 inputs {
2   -- NB String is a default input type
3   PathElementCS using elementName;
4   qualifier PathNameCS using pathElements;
5 }
```

LISTING 4.17: Inputs definition example

Providers

This clause is presented in an incremental way, so that the syntax for simple scenarios is shown first, along with an illustrative example, and then the syntax is incrementally enhanced to show more elaborated scenarios.

The third clause, called *providers*, is used for declaring how AS meta-classes contribute targets to lookup scopes. The contributions can be made into the current lookup scope (see Section 3.5.4) or into the exported lookup scope (see Section 3.5.6).

Recall that there is a default behaviour with respect to the scopes. In the case of the current lookup scope, this is propagated from parent to children as it stands, without modification. In the case of the exported lookup scope, it is empty. Therefore, this clause includes contribution declarations on the particular providers that modify the default behaviour, in this case by adding targets to the scope.

Listing 4.18 shows the syntax to declare the providers involved in a name resolution activity. It consists of the *providers* keyword (line 3), followed by an arbitrary number of provider declarations (line 4) enclosed by a pair of curly brackets (lines 3 and 5). A provider declaration consists of a name (line 7) of an AS meta-class, followed by an optional variables declaration (line 8) and optional declarations of the current and exported scopes (lines 9–10), enclosed by a pair of curly brackets (lines 7 and 11). The provider variables declaration is a sub-clause to declare variables that can be reused across the provider declaration. It consists of the *vars* keyword (line 13), followed by an arbitrary number (at least one) of variable declarations (line 13). The current scope definition (line 9) and the exported scope definition (line 10) are explained below.

```
1  Providers:
2     {Providers}
3     'providers' '{'
4        providers+=Provider∗
5     '}' ;
6  Provider:
7     classRef=PathNameCS '{'
8        varsDecl=ProviderVarsDecl?
9        currentScope=CurrentScopeDecl?
10       exportedScope=ExportedScopeDecl?
11    '}' ;
12 ProviderVarsDecl:
13    'vars' varDecl+= LetVariableCS (',' varDecl+=LetVariableCS)∗ ';' ;
```

LISTING 4.18: Providers definition syntax

A provider can declare how targets are contributed to the current scope. Listing 4.19 shows the syntax to deal with this requirement for simple scenarios. The syntax consists of the keywords *in* and *current_scope* (line 2), followed by at least one provision definition (line 3). The syntax of a provision definition starts with the keyword *provides* (line 5), followed by an arbitrary number (at least one) of provisions (line 5), and ends with a semicolon (line 6). These provisions declare which are the actual targets contributed to the current scope. The syntax consists of an optional declaration of the names (line 8) of the kind of targets that are provided, and the keyword *using* (line 8). Then, a contribution definition (line 9) is required, whose syntax consists of at least one comma-separated contribution (line 13). For the simple case, the contribution syntax comprises the OCL expression (line 15) that collects one or many contributed targets.

```
1  CurrentScopeDecl:
2    'in' 'current_scope'
3    provisionDefs+=CurrentScopeProvisionDef+ ;
4  CurrentScopeProvisionDef:
5    'provides' provision+=Provision+
6    ';' ;
7  Provision:
8    (providedClasses=MultiplePathNames 'using')?
9    contributionDef=ContributionDef ;
10 MultiplePathNames:
11   pathNames+=PathNameCS (',' pathNames+=PathNameCS)∗ ;
12 ContributionDef:
13   contributions+=Contribution (',' contributions+=Contribution)∗ ;
14 Contribution:
15   expression=ExpCS ;
```

LISTING 4.19: Current scope contributions syntax

⚠ The names of the provided meta-classes can be omitted because they can be inferred from the expression/s involved in the contribution definition.

Listing 4.20 shows an example. It declares that the AS meta-class *Class* (line 2) is a target provider. It contributes to the current scope (line 3) its owned properties (line 6) and its owned operations (line 7). In this way, any of their children could find these targets by means of a name-based lookup.

```
1 providers {
2   Class {
3     in current_scope provides
4       -- The target type specification could be omitted because
5       -- it could be inferred from the contribution expression
6       Property using ownedProperties
7       Operation using ownedOperations;
8   }
9 }
```

LISTING 4.20: Simple current scope provision example

By default, current scopes are propagated to children as they are passed from parents, and specific providers can include new targets in them. However, Section 3.5.3 explained that nested scopes can be rather propagated, with the aim of occluding equally named targets contributed by the provider's ancestors.

Listing 4.21 shows the syntax to comply with this requirement. In this case, the current scope provision definition syntax is enhanced by an optional declaration of keywords (line 3) that precedes the provisions (line 4). These keywords specify in which kind of scope the provided targets are included:

- *adding* means that targets are contributed to the same scope propagated from the parent. This is the default, in the case of keyword absence.
- *occluding* means that targets are contributed to a new nested scope (see Section 3.5.3).
- *resetting* means that targets are contributed to a new empty scope.

```
1 CurrentScopeProvisionDef:
2   'provides'
3   ('occluding' | sameScope?='adding' | emptyScope?='resetting')?
4   provisions+=Provision+
5 ';' ;
```

LISTING 4.21: Children selection syntax

There is no case in mOCL in which *resetting* a propagated scope is actually needed. However, for completeness, CS2AS-TL supports the case where targets available in a model element current scope are no longer so for its children.

Listing 4.22 shows an example. It is similar to the previous example, except that this one includes the *occluding* keyword (line 4). On the one hand, in mOCL the propagation of a nested scope from *Class* model elements is not required. According to the containment hierarchy, the possible parents of *Class* model elements (e.g. a *Package* model element) do not contribute properties or operations. On the other hand, there are object-oriented languages like Java, where inner class definitions are allowed. As a language requirement, the properties from inner classes must occlude identically named fields defined in outer classes. In that case, the propagation of a nested scope would be mandatory.

```
1 providers {
2   Class {
3     in current_scope provides
4       occluding
5         ownedProperties, ownedOperations; -- Target type spec can be omitted
6   }
7 }
```

LISTING 4.22: Propagating a nested scope

In object-oriented languages like OCL and Java, property and operation call expressions can refer not only to the properties and operations of the contextual class in which they are used, but also to those properties and operations that belong to super classes that are (directly and indirectly) extended by the contextual class. Additionally, any property of the contextual class should occlude any property that belongs to any of their super classes. This means that there is a need for creating an arbitrary number of nested scopes, rather than just one with all the contributed elements.

Listing 4.23 shows the syntax to comply with this requirement. In this case, the provision syntax is enhanced by an optional declaration of an arbitrary number of occluding definitions (line 4). The syntax of the latter is a normal contribution definition, preceded by the keyword *occluding* (line 6). By introducing many occluding definitions, more nested environments are created.

```
1 Provision:
2   (providedClasses=MultiplePathNames 'using')?
3   contribution=ContributionDef
4   occludingDefs+=OccludingDef* ;
5 OccludingDef:
6   'occluding' contribution=ContributionDef ;
```

LISTING 4.23: Occlusion definition syntax

Listing 4.24 shows an example. The example comprises a more elaborate declaration for the provider *Class* (line 2). Firstly, a variable declaration appears (line 3). Then, the set of target model elements contributed to the current scope is declared. In this case, the contribution consists of the properties and operations belonging to the contextual class (line 6). Finally, this first set of targets occludes a new set of properties and operations owned by the super classes (line 8).

```
1  providers {
2    Class {
3      vars allSuperClasses = self->closure(superClasses);
4      in current_scope provides
5        occluding
6          ownedProperties, ownedOperations
7        occluding
8          allSuperClasses.ownedProperties, allSuperClasses.ownedOperations;
9    }
10 }
```

LISTING 4.24: Occlusion definition example

When contributing targets to the current scope, there are situations where the contribution needs to be different depending on whether the propagated scope is for a child or any other. In other words, there is a selective contribution depending on which is the containment reference the child is contained in.

Listing 4.25 shows the syntax to comply with this requirement. In this case, the current scope provision definition syntax is enhanced by the optional declaration of a children selection definition (line 2) preceding the *provides* keyword. This definition is used for declaring to which provider's children the provision applies. By default, in the absence of any selection definition, the scope with the contributed targets is propagated to all their children. The syntax of a selection definition consists of the keyword *for* (line 7) followed by either an *all selection* (lines 8–9) or a *specific selection* (lines 10–11) syntax:

- The former is used for declaring that the targets provision is propagated to all its children. The syntax consists of the keyword *all* (line 9). Optionally, a list of exceptions can be declared by means of a comma-separated list of the containment reference names (line 9). This declares that the targets provision applies to all the provider's children, excepting those contained by the specified containment reference/s.

- The latter is used for declaring that the targets provision is propagated to specific children. The declaration consists of an arbitrary number (at least one) of containment reference names (line 11). This declares that the targets provision applies to the provider's children contained by the specified containment reference(s).

Listing 4.26 shows two analogous examples that conform to the explained syntax. On the left hand side, it is declared how *LetExp* model elements contribute target *Variable* model elements. The key point of this scenario is the following. The init expression of a *Variable* contained by a *LetExp* is not allowed to refer to the own *Variable* that is initialising. Therefore, it is declared that *LetExp* model elements (line 2), in the current scope (line 3), for all

```
CurrentScopeProvisionDef:
  selectionDef=SelectionDef? 'provides'
  ('occluding' | sameScope?='adding' | emptyScope?='resetting')?
  provisions+=Provision+
  ';' ;
SelectionDef:
  'for'
  ( {SelectionAll}
   'all'('excepting' exceptions+=PathNameCS(',' exceptions+=PathNameCS)
       *)?
  | {SelectionSpecific}
   selections+=PathNameCS (',' selections+=PathNameCS)*
  ) ;
```

LISTING 4.25: Selective contribution definition syntax

children excepting those contained via the *LetExp::ownedVariable* containment reference (line 4), provide the target *Variable* model elements obtained from the *'ownedVariable'* expression (line 5).

On the right hand side of the figure, there is another way of declaring the same provision. Given that there is just one additional containment property belonging to *LetExp*, a selective provision can be declared. Therefore, it is declared that *LetExp* model elements (line 11), in the current scope (line 12), for the children contained via the *LetExp::ownedIn* containment reference (line 13), provide the target *Variable* model elements obtained from the *'ownedVariable'* expression (line 14).

```
1  providers {                               10  providers {
2    LetExp {                                11    LetExp {
3      in current_scope                      12      in current_scope
4        for all excepting ownedVariable     13        for ownedIn
5          provides ownedVariable;           14          provides ownedVariable;
6    }                                       15    }
7  }                                         16  }
8
9
```

Listing 4.26: Children selection example

Some languages prevent the creation of forward cross-references between a consumer and a target that is defined further on. For instance, in Java, a variable expression can only refer to a variable that has been previously defined. To provide a concrete example related to the running example, let's assume that multiple variable declarations are supported in mOCL by keeping them in a multiple valued *LetExp::ownedVars* containment reference. The language requirement to support is the following: every expression that initialises a let variable can only refer to previously defined variables. Figure 4.9 shows an example with valid and invalid situations. In this case, there is a problem with the *'a'* variable initialisation because the corresponding variable expression refers to a further defined variable (see the red dashed arrow from the figure). From the point of view of name resolution, the provider requires contributing different targets depending on the particular child that is performing the name-based lookup.

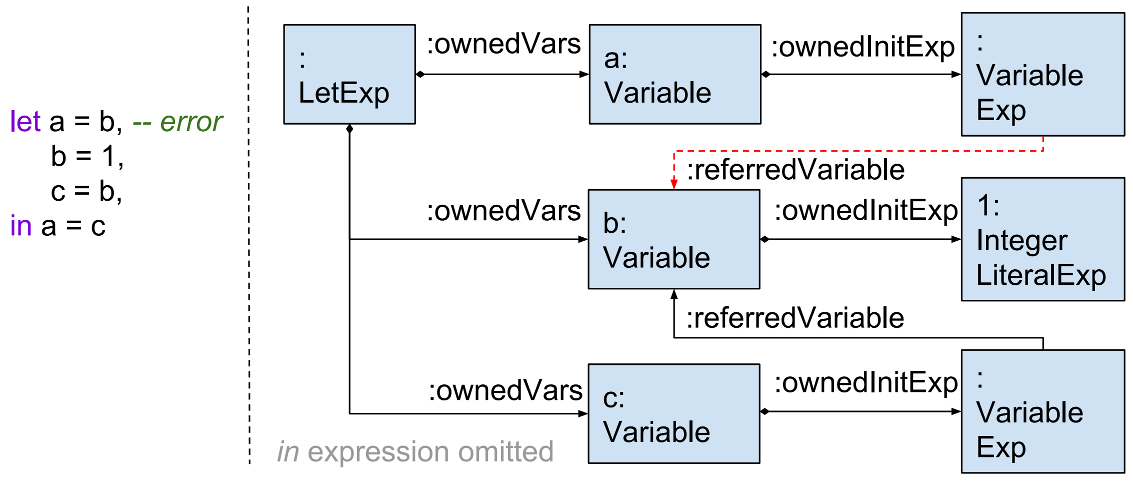


FIGURE 4.9: Let expression defining multiple variables

Listing 4.27 shows the syntax. In this case, the contribution syntax is enhanced with an optional declaration of a child definition between the keywords *child* and *in* (line 2). The syntax of a child definition consists of a name that represents the child, and is optionally followed by the ':' symbol and a type name (line 5). This child definition represents the actual child for which the scope is computed, so that the child can be used as a variable in the contribution expression.

```
Contribution:
  ('child' child=ChildDef 'in')?
  expression=ExpCS ;
ChildDef:
  name=ID (':' typeRef=PathNameCS)? ;
```

LISTING 4.27: Child definition syntax

Listing 4.28 shows an example that conforms to the explained syntax and addresses the scenario shown in Figure 4.9. The contribution of targets for the children contained via the *LetExp::ownedIn* containment reference (line 4) is the same as before. However, now there is a specific contribution defined for any child contained via the *ownedVars* containment reference (line 6). In this case, a child variable is defined (line 7) so that the contributed targets for a particular child variable are all the variables that are previously defined to that child.

⚠ A child definition is tied to a particular child type, so that a child definition is subject to being used along with a children selection definition. Therefore, the type of the child can be omitted. It can be inferred from the type of the reference used for the children selection definition.

```
1  providers {
2    LetExp {
3      in current_scope
4        for ownedIn provides
5          ownedVars;
6        for ownedVars provides
7          child childVar -- type is inferred from the children selection definition
8          in ownedVars->select(x | self.ownedVars->indexOf(x) <
9                                self.ownedVars->indexOf(childVar));
10
11   }
12 }
```

LISTING 4.28: Contribution using a child definition

Due to the fact that this expression (line 8) is quite verbose and it encodes a pattern that is common to several languages (e.g. imperative languages only allow referring to variables that are previously defined), CS2AS-TL defines a more concise syntax to deal with this kind of scenario. Listing 4.29 shows the syntax that enhances a contribution with an optional *preceding* keyword prior to the expression.

```
Contribution:
(('child' child=ChildDef 'in')
 | isPreceding?='preceding')?
expression=ExpCS ;
```

LISTING 4.29: Children selection syntax

Listing 4.30 shows the previous example reformulated with the more concise syntax.

```
providers {
  LetExp {
    in current_scope
      for ownedIn provides ownedVars;
      for ownedVars provides preceding ownedVars;
  }
}
```

LISTING 4.30: Example of a contribution using preceding children

A provider can declare how targets are contributed to the exported scope. The syntax is similar to the one required for current scope declarations but, in general, it is simpler since there is no concept of scope propagation. Listing 4.31 shows the syntax to deal with this requirement. It consists of the keywords *in* and *exported_scope* (line 2), followed by an arbitrary number (at least one) of provision definitions (line 3). The syntax of a provision definition starts with the keyword *provides* (line 5), followed by an arbitrary number (at least one) of provisions (line 5), ending with the ';' symbol (line 6). These provisions declare which are the actual targets contributed to the exported scope. The syntax of a provision was previously shown in Listing 4.19.

```
ExportedScopeDecl:
  'in' 'exported_scope'
  provisionDefs+=ExportedScopeProvisionDef+;
ExportedScopeProvisionDef:
  'provides' provisions+=Provision+
  ';' ;
```

LISTING 4.31: Exported scope declaration syntax

> The syntax of a provision is reused for exported scope declarations. However, the additions explained for the simple contribution syntax are forbidden. In essence, using the *preceding* keyword or a *child* definition does not make sense for exported scope declarations; hence, it is forbidden.

Listing 4.32 shows an example. It is a continuation of the example shown in Listing 4.24. In this case, several reusable variables are declared (lines 3–5). The contribution consists of a set of the properties and operations the class contains (line 13), which occlude (line 14) the properties and operations of all the super classes (line 15).

```
1  providers {
2    Class {
3      vars allSuperClasses = self->closure(superClasses),
4           allSuperProperties = allSuperClasses.ownedProperties,
5           allSuperOperations = allSuperClasses.ownedOperations;
6      in current_scope provides
7        occluding
8          ownedProperties, ownedOperations
9        occluding
10         allSuperProperties, allSuperOperations;
11
12     in exported_scope provides
13       ownedProperties, ownedOperations
14       occluding
15         allSuperProperties, allSuperOperations;
16   }
17 }
```

LISTING 4.32: Exported scope declaration example

According to the standard [39], "The OCL specification puts no restriction on the visibility declared for a property defined in the model (such as 'private', 'protected' or 'public')". Therefore, the same group of properties and operations are contributed to the current and exported scopes. However, different expressions could be used to add additional contributions, for instance, to filter the contributed targets based on their visibility.

Exported scopes are normally used by consumers (e.g. by means of lookup expressions, see next sub-subsection). However, these scopes can also be reused within the name resolution section. In this way, targets exported by a provider can be added to the scope of another provider. In particular, this is useful for mOCL imports, which can refer to the root model element of an imported external model. In this way, the high-level requirement consists of making the exported targets from the imported model available throughout the entire importing model.

Listing 4.33 shows the syntax to comply with this requirement. In this case, the contribution syntax is enhanced with additional alternatives that precede the contribution expression.

On the one hand, an optional declaration of the keyword *exported* (line 4) may precede the contribution expression (line 8). The keyword may be followed by an arbitrary number of comma-separated type names and the keyword *from* (line 5). In this way, rather than contributing the model elements evaluated from the contribution expression, the actual targets to contribute are those that exist in the exported scope of these model elements. Given that in an exported scope different kinds of targets may coexist, the optional type names are used for specifying the specific targets that are wanted.

On the other hand, an optional declaration of the keyword *loaded* (line 6), followed by a type name and the keyword *'from'* (line 7), may precede the contribution expression (line 8). This expression must be *String* valued. The declaration expresses that the model element to contribute has to be loaded using the URI comprised by the contribution expression's value. In case that the computed URI corresponds to a model file, the root model elements are considered as the ones to load. Since the contribution expression has to be *String* valued, and there is no static information about the kind of object to be contributed, the type name declaration is mandatory.

```
1 Contribution:
2   ( ('child' child=ChildDef 'in')
3   | isPreceding?='preceding'
4   | (isExported?='exported'
5     (typeRefs+=PathNameCS (',' typeRefs+=PathNameCS)* 'from')?)
6   | (isLoaded?='loaded'
7     typeRefs+=PathNameCS 'from'))?
8   expression=ExpCS ;
```

LISTING 4.33: Syntax to use contributions from exported scopes

Taking the above into account, Listing 4.34 shows an example that conforms to the explained syntax and exposes how external model imports are supported. Firstly, *Import* model elements (line 2) provide targets to the exported scope (line 3). In this case, target *Package* model elements will be loaded from an external model (line 4). The URI of the external model is held by the *Import::uri* attribute (line 4). Finally, in order to make this imported target available throughout the entire AS model, *Root* model elements (line 6) add to their current scope (line 7) all the targets exported by their owned imports (line 9).

```
1  providers {
2    Import {
3      in exported_scope
4        provides loaded Package from uri;
5    }
6    Root {
7      in current_scope
8        provides ownedPackages,
9                 exported ownedImports;
10   }
11 }
```

LISTING 4.34: Example of using contributions from an exported scope

To conclude this sub-subsection, the final additional syntax to support aliases is introduced. A scope consists of a map between a name-key and a target-value (see Section 3.5.3), where that name-key is normally the name that identifies the target-value. However, this is not always the case, and another name can be used. This is useful to create aliases for the model elements imported by means of imports. These aliases turn out to be crucial when the two different imported model elements have a clashing name.

Listing 4.35 shows the syntax to comply with this requirement. In this case, the contribution syntax is enhanced by introducing an optional keyword *with_alias*, followed by an expression that evaluates the new name that is used for the contribution of the target to the scope. The alias expression type must conform to the type of target identifying expression.

```
Contribution:
  ( ('child' child=ChildDef 'in')
  | isPreceding?='preceding'
  | (isExported?='exported'
    (typeRefs+=PathNameCS (',' typeRefs+=PathNameCS)* 'from')?)
  | (isLoaded?='loaded'
     typeRef+s=PathNameCS 'from'))?
  expression=ExpCS
  ('with_alias' alias=ExpCS)?
  ;
```

LISTING 4.35: Alias declaration syntax

Listing 4.36 shows an example. The previous example is enhanced with new syntax. In this case, the loaded *Package* model element is contributed to the exported scope using the *Import::alias* (line 4) name rather than the *NamedElement::name* that identifies target *Packages*.

```
providers {
  Import {
    in exported_scope
      provides loaded Package from uri with_alias alias;
  }
  Root {
    in current_scope
      provides ownedPackages, ownedContraints,
                 exported ownedImports;
  }
}
```

LISTING 4.36: Alias declaration example

Consumers

Finally, this sub-subsection introduces how consumers perform name-based lookups. As mentioned in Section 4.2.2, additional lookup expressions are introduced in CS2AS-TL with the aim of looking up targets. These expressions can be used throughout CS2AS-TL, for instance, to compute the initial value of an AS reference, or as part of a disambiguation rule.

Listing 4.37 shows the syntax to declare lookup expressions, which are similar to OCL operation call expressions, but there is a dedicated keyword to classify them. The syntax consists of either the *lookup* or *lookupExported* keyword (line 2). The former is used for performing a lookup within the current scope, whereas the latter is used for performing lookups within the exported scope. Then, a rounded-brackets clause follows (line 3), which consists of an arbitrary number of comma-separated OCL expressions enclosed by a pair of parentheses (line 5).

Note that both lookups are actually performed from the AS model element corresponding to the source expression of the lookup one. The arguments expected by both expressions are different:

- A name-based lookup in the current scope expects: firstly, the kind of target that is required, followed by an expression that provides the lookup input (see Inputs sub-subsection above), optionally followed by an expression providing additional filtering arguments (see Targets sub-subsection above).

- A name-based lookup in an exported scope expects: firstly, the kind of target that is required, followed by an expression that evaluates the AS model element that provides the exported scope (see Providers sub-subsection above), followed by an expression that provides the lookup input (see Inputs sub-subsection above), optionally followed by an expression providing additional filtering arguments (see Targets sub-subsection above).

Listing 4.38 shows some examples of lookup expressions. They are related to the disambiguation and mappings sections (further explained). Here, only the usage of the lookup expressions is considered.

```
LookupExp:
   ('lookup' | isExported?='lookupExported')
   RoundedBracketCS ;
RoundedBracketsCS:
   '(' (args+=ExpCS (',' args+=ExpCS)*)? ')' ;
```

LISTING 4.37: Lookup expression syntax

A lookup in the current scope is used in the context of a disambiguation rule (lines 1–5), in particular, the one required to determine whether *NameExpCS* model elements correspond to *VariableExp* model elements. In essence, the rule involves discovering whether a **Variable** with the name held by the *NameExpCS* is found in the current scope. In this case, the *lookup* expression (line 4) receives the name *Variable* as the kind of target to look up, and an expression that provides the input *PathNameCS* model element (which is held via the *NameExpCS::expName* containment reference).

A lookup in an exported scope is used in the context of a mapping declaration (lines 7–14), in particular, the one that creates *OperationCallExp* model elements from *NameExpCS* model elements. In essence, the mapping requires looking up an *Operation* model element in the exported scope of a *Class* referred to by the *type* cross-reference of the created *OperationCallExp* model element. In this case, the *lookupExported* expression (line 11) receives the name *Operation* as the kind of target to look up, an expression to provide the mentioned *Class* model element, an expression that provides the input *PathNameCS* model element, and an expression that provides the additional filtering arguments to match the right *Operation* model element.

```
1  disambiguation {
2    NameExpCS {
3      isVariableExp :=  roundedBrackets = null and not isNavExpOfACallExpCS()
4                        and lookup(Variable, expName) <> null;
5    }
6  }
7  mappings {
8    create  OperationCallExp from NameExpCS
9      when isOpCallExpWithExplicitSource {
10     -- other property  initializations  are  omitted
11       referredOperation := lookupExported(Operation, trace.ownedSource.type,
12                                           expName, trace.ownedArguments);
13   }
14 }
```

LISTING 4.38: Lookup expression usage examples

4.2.6 Mappings Section Syntax

The mappings section is the main section where the CS meta-model terms are related to the AS meta-model terms. It is a completely declarative section, so the order in which the mappings are declared is not relevant. There can only be one mapping per CS meta-class, unless the term is ambiguous. In that case, all the mappings for the same meta-class must be accompanied by a disambiguation rule name.

Listing 4.39 shows the syntax of the top-level declaration of the section. It starts with the *mappings* keyword (line 3). Then, a pair of curly brackets (lines 3 and 5) enclose the declaration of an arbitrary number of mapping definitions (line 4). A mapping definition can either be a creation mapping or a reference mapping (line 7).

```
1 MappingSect:
2   {MappingSect}
3   'mappings' '{'
4     mappings+=MappingDef*
5   '}';
6 MappingDef:
7   MappingCreation | MappingReference ;
```
LISTING 4.39: Mappings section syntax

The following sub-subsections present the mapping definitions in detail.

Creation Mappings

A creation mapping is designed to relate source CS meta-classes to target AS meta-classes, and to declare how the properties of the latter are computed. The particular semantics of this kind of mapping consists of creating instances of the target meta-classes and initialise the values of their properties.

A new CS2AS scenario of mOCL is introduced to explain creation mappings. Listing 4.40 shows the CS definition of one kind of mOCL expression: equality expression. In this case, *EqualityExpCS* model elements are produced (line 3), when the corresponding (in)equality operators (line 4) are processed. The model element (left sub-expression) obtained when processing the first RHS non-terminal *CallExpCS* (line 2) is held by the *EqualityExpCS::left* containment reference. The model element (right sub-expression) obtained when processing the second RHS non-terminal *CallExpCS* (line 5) is held by the *EqualityExpCS::right* containment property.

```
1 EqualityExpCS:
2   CallExpCS
3   ({EqualityExpCS.left=current}
4   opName=('=' | '<>')
5   right=CallExpCS)* ;
```
LISTING 4.40: CS definition of equality expression

A concrete example of this kind of expression is shown in Figure 4.10. On the left, there is the CS model corresponding to the *'1 <> 2 = true'* expression. On the right, the expected AS model is shown. In this case, *EqualityExpCS* model elements (e.g. ❶) are transformed into *OperationCallExp* model elements (e.g. ❷). The left sub-expression (e.g. ❸) corresponds to the source (e.g. ❹) of the *OperationCallExp* model element, whereas the right sub-expression (e.g. ❺) corresponds to the argument (e.g. ❻).

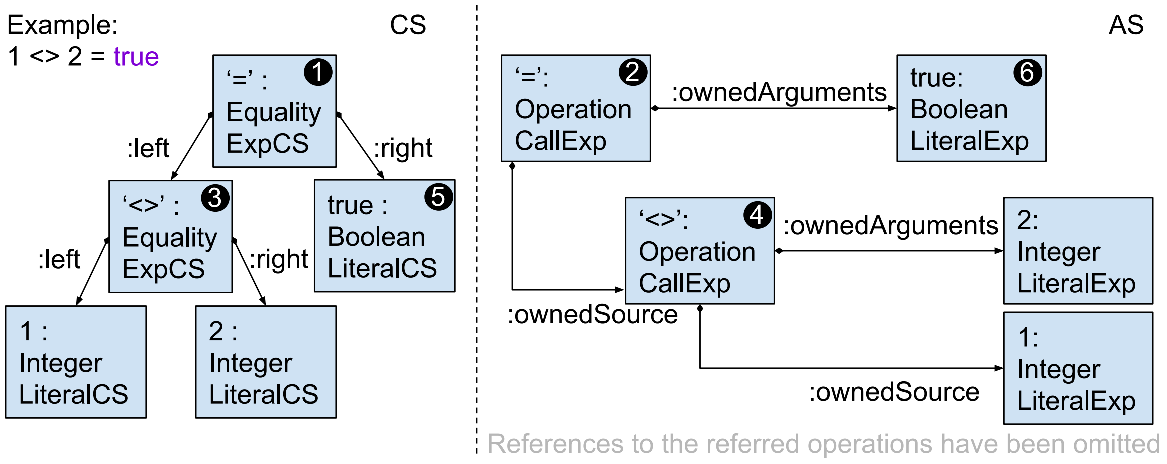


FIGURE 4.10: Equality expression example (left) and corresponding AS model (right)

Listing 4.41 shows the syntax to declare creation mappings. It consists of the keyword *create*, followed by the class name of an AS meta-class, followed by the keyword *from*, followed by the class name of a CS meta-class. Then, an arbitrary number of property assignments (line 3) can be enclosed by a pair of curly brackets (lines 2 and 4). The syntax of a property assignment consists of an LHS property name (of the created AS meta-class), followed by the ':=' symbol and an RHS OCL expression that declares how the property is mapped or computed. The context of that OCL expression is the source CS meta-class.

```
1 MappingCreation:
2   'create' to=PathNameCS 'from' from=PathNameCS '{'
3   properties += PropertyAssignment*
4   '}';
5 PropertyAssignment:
6   propRef=PathNameCS ':='
7   propInit=ExpCS
8   ';';
```

LISTING 4.41: Creation mappings syntax

The RHS of the property assignment is either ❶ an OCL property call expression that refers to a compatible CS property, or ❷ a compatible OCL expression that returns AS model elements. This compatibility is explained by introducing the semantics of a property assignment. In terms of the semantics, the following applies:

- In the former case ❶, the RHS expression is evaluated so that it returns one or many CS model elements. All these CS model elements have the corresponding AS model elements that are assigned or added to the LHS AS property. Statically, it can be determined if there exists a mapping defined for the CS meta-class that is a type of the RHS property call expression, as well as whether the corresponding AS meta-class target of that mapping conforms to the type of the LHS AS property.

- In the latter case ❶, the RHS expression is evaluated so that it returns one or many AS model elements, or (primitive) values that are assigned or added to the LHS AS property. Statically, it can be determined if the AS meta-class that is a type of the RHS expression conforms to the type of the LHS AS property.

Apart from type conformance, there are additional restrictions with respect to multiplicities. Table 4.1 explains the different scenarios. In essence, the only invalid situation is having a single-valued AS property name in the LHS of the property assignment, and an RHS OCL expression that returns many model elements or values.

	❶ *PropertyCallExp* in the CS domain		❷ *OCLExpression* in the AS domain	
AS Property	Single-Value	Multi-Value	Single-Value	Multi-Value
Single-Value	OK	Error	OK	Error
Multi-Value	OK	OK	OK	OK

TABLE 4.1: Valid property assignment scenarios based on LHS property multiplicity and RHS expression type

Listing 4.42 shows the creation mapping that addresses the explained CS2AS scenario. In this case, the mapping declares that *OperationCallExp* model elements are created from *EqualityExpCS* model elements (line 2). The *OperationCallExp::ownedSource* property is mapped from the *EqualityExpCS::left* property (line 3), whereas the *OperationCallExp::ownedArguments* property is mapped from the *EqualityExpCS::right* property (line 4). Finally, there is a property computation of the *OperationCallExp::referredOperation* property (line 5). In this case, a name-based lookup is performed in order to locate the corresponding (in)equality *Operation* model element.

```
1 mappings {
2   create OperationCallExp from EqualityExpCS {
3       ownedSource := left;
4       ownedArguments := right;
5       referredOperation := lookupExported(Operation, trace.ownedSource.type,
6                                                  opName, trace.ownedArguments);
7   }
8 }
```

LISTING 4.42: Example of a creation mapping

Reference Mappings

A reference mapping is designed to relate source CS meta-classes to target AS meta-classes. However, in contrast to creation mappings, they are not designed to create new AS model elements, but rather to refer to a particular one by means of an OCL expression. These mappings permit the creation of trace links between CS model elements and AS model elements, even though the former are not meant to create the latter. In this way, trace expressions can be used on CS model elements that do not create AS model elements.

A new scenario from mOCL syntax is introduced to explain reference mappings. Listing 4.43 shows the CS definition of another kind of mOCL expression: call expression. In this case, *CallExpCS* model elements are produced (line 3) when the corresponding call operators (line 4) are processed. The model element (left sub-expression) obtained when processing the RHS non-terminal *PrimaryExpCS* (line 2) is held by the *CallExpCS::source* containment reference (line 3). The model element (right sub-expression) obtained when processing the RHS non-terminal *NavigationExpCS* (line 5) is held by the *CallExpCS::navExp* containment property (line 5).

```
1  CallExpCS:
2    PrimaryExpCS
3    ({CallExpCS.source=current}
4    opName=('.' | '->')
5    navExp=NavigationExpCS)* ;
6  PrimaryExpCS:
7    LiteralExpCS | NameExpCS | SelfExpCS | LetExpCS;
8  NavigationExpCS:
9    LoopExpCS | NameExpCS ;
```

LISTING 4.43: CS definition of call expression

A concrete example is shown in Figure 4.11. On the left, there is the CS model corresponding to the *'self.x.y'* expression. On the right, the expected AS model is shown. With respect to the previous example, the main differences can be observed. In this case, *CallExpCS* model elements (e.g. ❶) are not transformed into AS model elements. As expected, different *PropertyCallExpression* model elements (e.g. ❷) are produced, but they are created from the *NameExpCS* model elements (e.g. ❸) contained by these *CallExpCS* model elements. However, there is a need for relating these *CallExpCS* model elements to AS model elements, so that any *trace* expressions defined on the former can return sensible AS model elements.

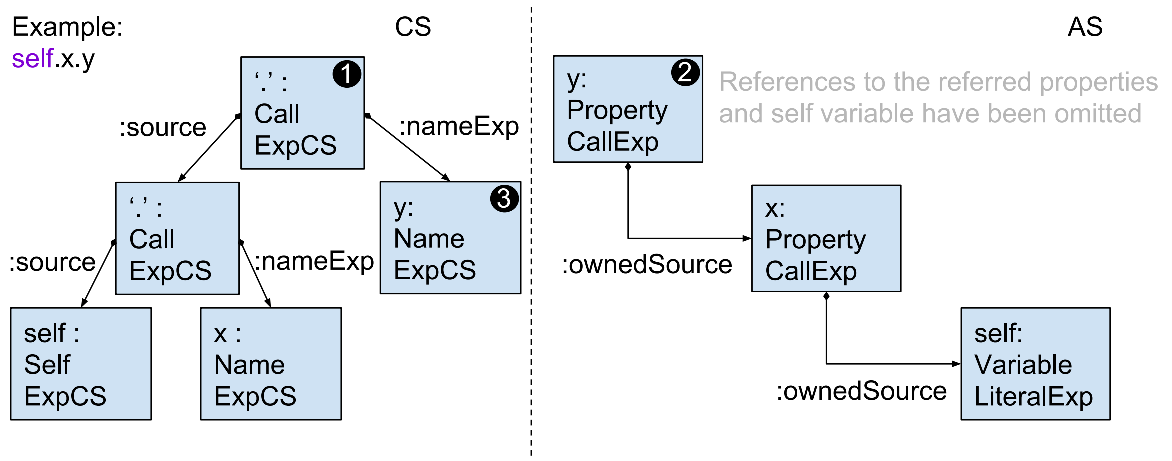


FIGURE 4.11: Call expression example (left) and corresponding AS model (right)

Listing 4.44 shows the syntax to declare reference mappings. It consists of the keyword *refer*, followed by the class name of an AS meta-class, followed by the keyword *from*, followed by the class name of a CS meta-class, followed by the *':='* symbol. Then, an OCL expression declares how the corresponding AS model element can be accessed from the contextual CS model element.

```
MappingReference:
  'refer' to=PathNameCS 'from' from=PathNameCS
  ':=' expression=ExpCS
  ';' ;
```

LISTING 4.44: Reference mappings syntax

Listing 4.45 shows the reference mapping that addresses the previous CS2AS scenario. In this case, the mapping declares that *CallExp* model elements are referred to from *CallExpCS* model elements (line 2). The OCL expression (line 3) that follows the ':=' declares which is the corresponding AS model element of the contextual *CallExpCS* model element. In this case, it corresponds to the AS model element corresponding to the *NavigationExpCS* model element contained via the *CallExpCS::navExp* containment reference.

```
1  mappings {
2    refer CallExp from CallExpCS :=
3      self.navExp.trace;
4  }
```

LISTING 4.45: Example of a reference mapping

Multi-way mappings

Multi-way mappings are a set of different mappings that are defined on the same source CS meta-class. The execution of this kind of mapping is subject to the successful evaluation of a disambiguation rule. Therefore, multi-way mappings require the specification of the corresponding disambiguation rules. These rules are declared within a separate disambiguation section (see Section 4.2.7), so the mappings only need to refer to them by name.

In order to explain multi-way mappings, a new mOCL scenario is introduced. Figure 4.12 shows an example of an mOCL collection literal expression and the AS definition designed to deal with them. According to the AS definition, a *CollectionLiteralExp* optionally comprises an arbitrary number of *CollectionLiteralParts*, which can be either a *CollectionItem* (consisting of a single *OCLExpression*) or a *CollectionRange* (consisting of two *OCLExpressions* to form the range).

```
Collection{1, 2, 3...5}

-- equivalent to:

-- Collection{1, 2, 3, 4,
   5}
```

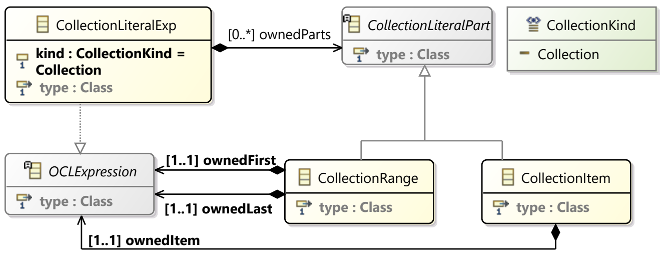


FIGURE 4.12: *CollectionLiteralExp* example (left) and AS meta-model (right)

Figure 4.13 shows the Xtext grammar definition (on the left) to deal with *CollectionLiteralExp* textual syntax, and the corresponding automatically derived CS meta-model (on the right). According to the CS definition, a *CollectionLiteralExpCS* (line 4) consists of the kind of collection (line 5), followed by an arbitrary number of the expression parts (line 6), enclosed by a pair of curly brackets (lines 5 and 7). A *CollectionLiteralPartCS* (line 9) consists of a first expression (line 10), and is optionally followed by the '..' symbol and a second expression (line 11), to represent integer ranges.

```
1   enum CollectionKindCS:
2     'Collection'
3   ;
4   CollectionLiteralExpCS:
5     kind=CollectionKindCS '{'
6     (parts+=CollectionLiteralParCS)*
7     '}'
8   ;
9   CollectionLiteralPartCS:
10     first =ExpCS
11    ('..' last=ExpCS)?
12  ;
```

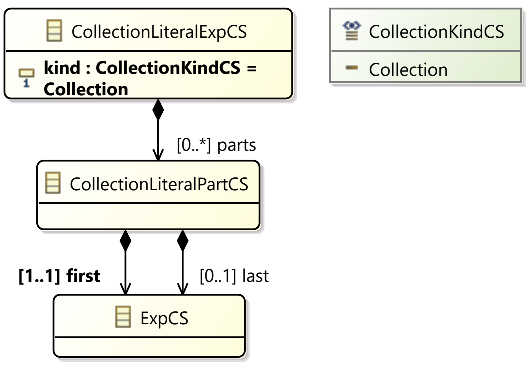


FIGURE 4.13: *CollectionLiteralExp* grammar (left) and CS meta-model (right)

The key point of the scenario is the following: from a *CollectionLiteralPartCS* model element, either a *CollectionItem* or a*CollectionRange* model element has to be created. In this case, it depends on whether the *CollectionLiteralPartCS* declares a last expression. This kind of scenario requires multi-way mappings.

Listing 4.46 shows the syntax introduced to deal with this requirement. In this case, the syntax of a creation mapping is enhanced by the addition of an optional *when* clause (line 2). It consists of the keyword *when*, followed by either an identifier of a rule defined within the disambiguation section, or the keyword *fall_back*. The latter is used for designing a mapping to be executed, when no disambiguation rule succeeds for a particular CS model element.

```
1   MappingCreation:
2     'create' to=PathNameCS 'from' from=PathNameCS ('when' (ruleName=ID |
          isFallback?='fall_back'))? '{'
3       properties += PropertyAssignment*
4     '}';
```

LISTING 4.46: When clause syntax

Listing 4.47 shows the mappings section that addresses the scenario. Lines 2–6 comprise a simple mapping to create AS *CollectionLiteralExp* model elements from *CollectionLiteralExpCS* model elements. Between the curly brackets, there are three assignments: on the left hand side, property references of mapped AS meta-classes; on the right hand side, different OCL expressions are used for declaring how the assignments take place.

```
1  mappings {
2    create CollectionLiteralExp from CollectionLiteralExpCS {
3      kind := kind;
4      ownedParts := parts;
5      type := lookup(Class,'Collection');
6    }
7    create CollectionItem from CollectionLiteralPartCS
8    when withoutLastExpression {
9      ownedItem := first;
10     type := first .trace.type;
11   }
12   create CollectionRange from CollectionLiteralPartCS
13   when withLastExpression {
14       ownedFirst := first ;
15       ownedLast := last;
16       type := first .trace.type;
17   }
18 }
```

LISTING 4.47: Multi-way mappings definition example

After the first mapping declaration, there is an example of two-way mappings (lines 7–11 and 12–17). In this case, the *CollectionLiteralPartCS* meta-class is mapped to two different AS meta-classes, particularly *CollectionItem* and *CollectionRange*. This kind of mapping incorporates an additional *when* clause (lines 8 and 13) that refers to a disambiguation rule defined in the corresponding section (see next Section 4.2.7).

4.2.7 Disambiguation Section Syntax

The disambiguation section comprises the disambiguation rules required by multi-way mappings. These rules are boolean expressions that determine which one of the multi-way mappings is actually executed. They are similar to the guards defined in rule-based M2M transformations. However, these rules are grouped together within the same section rather than belonging to individual mappings. This characteristic allows the section to provide additional semantics to the declared rules. Specifically, the rules are prioritised based on the order in which they are defined within the disambiguation section. In this way, during execution time, the first rule to succeed is the one that determines how the CS element is disambiguated (i.e. which is the mapping that is actually executed). The explained semantics are useful in arbitrating the situation where the involved disambiguation rules are not exclusive (i.e. when more than one disambiguation rule evaluates to true for the same CS model element).

Listing 4.48 shows the syntax of the top-level declaration of the section. It starts with the *disambiguation* keyword (line 3). Then, a pair of curly brackets (lines 3 and 5) encloses the declaration of the disambiguation definition (line 4). The syntax of the latter consists of the class name of a CS meta-class (line 7), followed by an arbitrary number of disambiguation rules enclosed by a pair of curly brackets (lines 7 and 9). The syntax of a disambiguation rule consists of an identifier followed by the ′:=′ symbol (line 11) and an OCL expression that must be boolean-valued (line 12).

```
1  DisambiguationSect:
2    {DisambiguationSect}
3    'disambiguation' '{'
4    disambiguations+=DisambiguationDef*
5    '}' ;
6  DisambiguationDef:
7    classRef=PathNameCS '{'
8    rules+=DisambiguationRule*
9    '}' ;
10 DisambiguationRule:
11   name=ID ':='
12   exp=ExpCS ';' ;
```

LISTING 4.48: Disambiguation section syntax

Listing 4.49 shows the disambiguation rules required for the *CollectionLiteralExp* example introduced earlier. The name *CollectionLiteralPartCS* (line 2) represents the CS meta-class on which the disambiguation rules are declared. Between the curly brackets, there are two assignments corresponding to the two rules. On the left hand side, the rule name is declared. On the right hand side, a boolean-valued OCL expression evaluates if the rule applies for the particular expression context (i.e. *CollectionLiteralPartCS* model elements).

```
1 disambiguation {
2   CollectionLiteralPartCS    {
3     withoutLastExpression := last = null;
4     withLastExpression := last <> null;
5   }
6 }
```

LISTING 4.49: Disambiguation section usage example

The reader may note that both expressions are mutually exclusive, meaning that only one of the disambiguation rules can apply to a particular *CollectionLiteralPartCS* model element. Therefore, the order in which they are declared is not relevant. However, there may be CS2AS scenarios in which this is not always true. In these cases, the priority in which the rules are defined for a particular CS meta-class is important.

Section 4.2.8 shows how the different scenarios introduced in Chapter 3 are addressed. When resolving the scenario concerned with multi-way mappings, the subsection shows an example in which the order of the disambiguation rules is relevant.

4.2.8 How Does the CS2AS-TL Address the Identified Concerns?

The previous subsections introduced the syntax of CS2AS-TL, including some examples from mOCL to demonstrate its usage. The upcoming subsections show how the syntax is used to solve other CS2AS scenarios that are present in mOCL, including those presented during the problem analysis.

After presenting the problem analysis in Chapter 3, Section 3.6 discussed how Xtext specification artefacts partly addressed the different concerns introduced in the chapter. In consequence, following the proposed solution explanation, this subsection shows how these concerns are fully addressed with CS2AS-TL.

Appendix C shows the complete CS2AS bridge for mOCL. Additionally, the source code for the mOCL example can be found in the examples folder of the Eclipse QVTd project repository[a]

[a] `https://git.eclipse.org/r/mmt/org.eclipse.qvtd`

Concern 1: Mapping a CS Meta-class to an AS Meta-class

Concern 1 is supported within CS2AS-TL mappings section, particularly by declaring a mapping between a CS meta-class and an AS meta-class.

Listing 4.50 shows the CS2AS declaration for the *Package* example (see Section 3.4.1). Line 2 shows how both meta-classes are mapped.

```
1  mappings {
2    create  Package from PackageCS {
3      name := name;
4      ownedPackages := packages;
5      ownedClasses := classes ;
6    }
7  }
```

LISTING 4.50: CS2AS definition of *Package*

According to the concern explanation, two kinds of mappings are required. They can be designed either to create new AS elements at execution time, or to refer to existing AS elements. As was explained in Section 4.2.6, CS2AS-TL supports the distinction by means of the keywords *create* and *refer*.

Concern 2: Mapping a CS Reference to an AS Reference

Concern 2 is supported within CS2AS-TL mappings section, particularly by declaring a property assignment. On the left hand side, a reference of the AS meta-class is specified. On the right hand side, there is an OCL property call expression that refers to the mapped CS reference.

Lines 4–5 from the previous Listing 4.50 show two examples of how this concern is addressed.

Concern 3: Mapping a CS Attribute to an AS Attribute

Concern 3 is supported within CS2AS-TL mappings section, particularly by declaring a property assignment. On the left hand side, an attribute of the AS meta-class is specified. On the right hand side, there is an OCL property call expression that refers to the mapped CS attribute.

Line 3 from the previous Listing 4.50 shows an example of how this concern is addressed.

Concern 4: Mapping a CS Reference to an AS Reference and Additional AS Meta-classes

Concern 4 is supported within CS2AS-TL mappings section, particularly by declaring a property assignment. On the left hand side, a reference of the AS meta-class is specified. On the right hand side, a more complex OCL shadow expression is used to declare a pattern that involves the set of the required meta-classes (and their corresponding properties).

Listing 4.51 shows the CS2AS declaration for the *Operation* example (see Section 3.4.5). Lines 6–10 show how this concern is addressed. In this case, the *Operation::ownedBodyExpression* reference is assigned (line 6), but it is not directly mapped from the expected *OperationCS::body* reference (line 8). Instead, an *ExpressionInOCL* meta-class is used by means of an OCL shadow expression (lines 6–10), and the *OperationCS::body* reference is used for computing the *ExpressionInOCL::ownedBody* reference (line 8). Finally, an additional *Variable* meta-class is involved to produce the implicit *self Variable* model element owned by the *ExpressionIn-OCL* one.

```
1  mappings {
2    create  Operation from OperationCS {
3      name := name;
4      type :=  lookup(Class, resultType);
5      ownedParameters := params;
6      ownedBodyExpression := ExpressionInOCL {
7              language = 'OCL',
8              ownedBody = body.trace,
9              ownedSelfVar = Variable {name = 'self' , type = trace.owningClass }
10     };
11   }
12 }
```

LISTING 4.51: CS2AS definition of *Operation*

Concern 5: Mapping a CS meta-class to Many AS Meta-classes

Concern 5 is supported within CS2AS-TL mappings section, particularly by declaring a mapping between a CS meta-class and the primary AS meta-class, and by declaring property assignments to involve the additional meta-classes. On the assignment's left hand side, a reference of the primary AS meta-class is specified. On the right hand side, a more complex OCL shadow expression is used to declare a pattern that involves the set of the required meta-classes (and their corresponding properties).

```
1  mappings {
2    create PropertyCallExp from NameExpCS
3    when isPropCallExpWithImplicitSource {
4      ownedSource := let referredVar = lookup(Variable, 'self')
5                     in VariableExp {
6                          referredVariable = referredVar,
7                          type = referredVar.type
8                     };
9      referredProperty := lookupExported(Property,trace.ownedSource.type,expName);
10     type := trace.referredProperty?.type;
11   }
12 }
```

LISTING 4.52: CS2AS definition of *PropartyCallExp* (with implicit source)

Listing 4.52 shows the CS2AS declaration for the *PropartyCallExp* example (see Section 3.4.6). Line 2 shows how the CS meta-class *NameExpCS* is mapped to the primary AS meta-class *PropartyCallExp*. Lines 4–8 show the additional property assignment that involves the required *VariableExp* meta-class. In this case, on the assignment's left hand side, the reference *ownedSource* is used (line 4). On the right hand side, an OCL shadow expression (lines 5–8), wrapped by an OCL let expression (lines 4–8), is used.

Concern 6: Multi-way Mappings from CS Meta-classes to AS Meta-classes

Concern 6 is supported within CS2AS-TL mappings section, in particular, by declaring several multi-way mappings with the corresponding *when* clauses.

Listing 4.53 shows a partial solution for the multi-way mappings for *NameExpCS*. The declarations required to produce *OperationCallExp* model elements have been omitted because they are similar to the ones required to produce *PropertyCallExp* model elements.

Firstly, in the helpers section (lines 1–12), two helper operations are declared. The aim of the first helper consists of returning the container of the *NameExpCS* model element as a *CallExpCS* model element, provided the container conformed to that type. The aim of the second helper consists of checking whether a *NameExpCS* is contained by a *CallExpCS* model element, via the *CallExpCS::navName* rather than *CallExpCS::source*. This would mean that the *NameExpCS* model element is a navigation expression (i.e. it corresponds to a name appearing after a navigation operator).

Secondly, in the disambiguation section (lines 13–23), three disambiguation rules are declared. The three rules require that the *NameExpCS* model element does not contain a child *RoundedBracketCS* (lines 16, 18 and 21). This condition excludes those *NameExpCS* model elements used for denoting *OperationCallExp* model elements. Looking at the remaining conditions involved, the three following situations can occur:

- If the previously explained *NameExpCS::isNavExpOfACallExpCS()* helper evaluates to true (line 16), the *NameExpCS* model element corresponds to a *PropertyCallExp* model element with an explicit source expression.

- Otherwise, a name-based lookup is performed to seek a *Variable* model element using the corresponding CS information (line 19). If a result is found, the *NameExpCS* model element corresponds to a *VariableExp* model element.
- Otherwise, a name-based lookup is performed to seek a *Property* using the corresponding CS information (line 22). If a result is found, the *NameExpCS* model element corresponds to a *PropertyCallExp* model element with an implicit source expression.

As commented in previous subsection 4.2.7, the order in which these rules have been declared matters. The *isPropCallExpWithExplicitSource* rule is defined first because it just requires CS information. In contrast, the other two rules require to perform a name-based lookup. Additionally, the *isVariableExp* and *isPropCallExpWithImplicitSource* rules are non-exclusive. It may arise that the same name used in the name-based lookup corresponds to both a defined *Variable* model element and a *Property* model element of the implicit source object. In this particular case, disambiguating towards a *VariableExp* model element is the priority.

Finally, in the mappings section (lines 24–50), multi-way mappings are declared to address the scenario: firstly, a mapping to produce a *VariableExp* model element (lines 25–29) whenever the *isVariableExp* disambiguation rule applies to the contextual *NameExpCS* model element; secondly, a mapping to produce a *PropertyCallExp* model element (lines 30–35) whenever the *isPropCallExpWithExplicitSource* disambiguation rule applies to the contextual *NameExpCS* model element; thirdly, a mapping to produce a *PropertyCallExp* model element (lines 36–45) whenever the *isPropCallExpWithImplicitSource* disambiguation rule applies to the contextual *NameExpCS* model element. Finally, a fall-back mapping is defined in which a *VariableExp* model element is created with no referred variable, and an *OclVoid* type.

Concern 7: Mapping AS Properties from non CS Meta-model Terms

Concern 7 is addressed by including additional property assignments to the mapping that deals with the AS meta-class of interest.

Listing 4.53 contains several examples. For instance, lines 28, 34 and 44 show specific assignments to compute the *TypedElement::type* property for the different AS meta-classes involved in those mappings.

Concern 8: Name Resolution

The previous Section 4.2.5 introduced several mOCL examples to explain the whole syntax that addresses the name resolution concern. This subsection summarises in Table 4.2 how the different name resolution topics relate to the shown OCL examples.

4.2.9 Other CS2AS scenarios of mOCL

So far, most of the CS2AS scenarios that mOCL embodies have been exposed, regardless of whether they were used for explaining the problem analysis or the CS2AS-TL syntax. This

```
helpers {
  NameExpCS {
    parentAsCallExpCS() : CallExpCS :=
      let container = self.oclContainer()
      in if container.oclIsKindOf(CallExpCS)
         then container.oclAsType(CallExpCS)
         else null
         endif;
    isNavExpOfACallExpCS() : Boolean :=
      let parentCallExpCS = parentAsCallExpCS()
      in parentCallExpCS <> null and parentCallExpCS.navExp = self;
  } }
disambiguation {
  NameExpCS { -- Note: order of the disambiguation rules matters
    isPropCallExpWithExplicitSource :=
      roundedBrackets = null and isNavExpOfACallExpCS();
    isVariableExp :=
      roundedBrackets = null and not isNavExpOfACallExpCS()
      and lookup(Variable, expName) <> null;
    isPropCallExpWithImplicitSource :=
      roundedBrackets = null and not isNavExpOfACallExpCS()
      and lookup(Property, expName) <> null;
   } }
mappings {
  create VariableExp from NameExpCS
  when isVariableExp {
    referredVariable := lookup(Variable, expName);
    type := trace.referredVariable?.type;
  }
  create PropertyCallExp from NameExpCS
  when isPropCallExpWithExplicitSource {
    ownedSource := parentAsCallExpCS()._source;
    referredProperty := lookupExported(Property,trace.ownedSource.type,expName);
    type := trace.referredProperty?.type;
  }
  create PropertyCallExp from NameExpCS
  when isPropCallExpWithImplicitSource {
    ownedSource := let referredVar = lookup(Variable, 'self')
                   in VariableExp {
                        referredVariable = referredVar,
                        type = referredVar.type
                   };
    referredProperty := lookupExported(Property,trace.ownedSource.type,expName);
    type := trace.referredProperty?.type;
  }
  create VariableExp from NameExpCS
  when fall_back {
      referredVariable := null;
      type := lookup(Class, 'OclVoid');
  } }
```

LISTING 4.53: Multi-way mappings definition for *NameExpCS*

Name resolution topic	Listings from Section 4.2.5
Targets	Listing 4.11
Inputs	Listing 4.17
Providers	Listing 4.20
Consumers	Listing 4.38
(Nested) Lookup scopes	Listings 4.20 and 4.22
Name-based lookups	Listing 4.38
Qualified name-based lookups	Listings 4.15 and 4.17
Name-based external lookups	Listings 4.32 and 4.38
Additional lookup criteria	Listing 4.13
Looking up into external models	Listing 4.34

TABLE 4.2: Name resolution topic and demonstrating listings

section shows the remaining scenarios, and how the CS2AS scenario is addressed by means of CS2AS-TL. As a reminder, Appendix A and Appendix B respectively contain the mOCL CS grammar and the AS meta-model.

Constraints

When introducing the running example, Section 3.2.2 stated that mOCL provides syntax to declare invariants on *Class* model elements.

Listing 4.54 shows the mOCL CS2AS bridge that deals with the creation of *Constraint* model elements.

From this CS2AS scenario, it can be seen that there is a misalignment between the CS and AS. On the one hand, according to the mOCL CS (see Appendix A), there is a single *ConstraintsDefCS* model element with the information related to the constrained *Class*. However, that *ConstraintsDefCS* model element may contain many *InvariantCS* model elements comprising many invariant expressions. On the other hand, according to the mOCL AS, *Root* model elements may contain many *Constraint* model elements via the *Root::ownedConstraints* containment reference.

This misalignment is reflected in the CS2AS bridge. Firstly, an extra navigation of the CS model is required for the *Root::ownedConstraints* property assignment (line 5). Secondly, another CS model navigation is required for accessing the *ConstraintsDefCS::typeRef* from the contextual *InvariantCS* model element (line 13) .

Let Expressions

Although let expressions have been discussed during previous subsections to illustrate some name resolution scenarios, a complete exposition of how *LetExp* model elements are created from the corresponding CS has not been shown. This kind of expression presents a challenging scenario because a single *LetExpCS* model element may comprise several variable declarations and, according to the AS, many (nested) let expressions must be created as variables are declared.

Listing 4.55 shows the mOCL CS2AS bridge that deals with the creation of *LetExp* model elements. In this case, the scenario has been addressed by means of multi-way mappings.

```
mappings {
  create Root from RootCS {
     ownedImports := imports;
     ownedPackages := packages;
     ownedConstraints := constraints.invariants;
  }
  create Constraint from InvariantCS {
     ownedSpecification := ExpressionInOCL {
                              ownedBody = exp.trace,
                              ownedSelfVar = Variable { name = 'self',
                                                        type = trace.constrainedElement }
                            };
     constrainedElement := lookup(Class, self.ConstraintsDefCS.typeRef);
  }
}
```

LISTING 4.54: CS2AS description of *Constraint*

Depending on the number of declared variables (lines 3–4), a different *LetExp* model element will be created (lines 8–13 and 14–26). The simple scenario occurs when a single let variable is declared. In that case, the *as* property *LetExp::ownedIn* is mapped from the *cs* property *LetExpCS::inExp* (line 11). The complex scenario occurs when several let variables are declared, because many (nested) *LetExp* model elements have to be created. In this case, the property *LetExp::ownedIn* (line 17) is computed by assigning an accumulation of all the required (nested) *LetExp* model elements (lines 17–23). For that, an iterate expression is used, so that starting from the last declared variable, in reverse order (line 17), a *LetExp* model element (lines 19–23) is accumulated, so that it is added to the *LetExp::ownendIn* property of the previous *LetExp* model element (line 21).

Literal Expressions

mOCL provides different kinds of literal expressions. The creation of *CollectionLiteralExp* model elements was already explained in Section 4.2.6, when introducing multi-way mappings. However, mOCL contains additional literal expressions.

Listing 4.56 shows the mOCL CS2AS bridge that deals with the creation of *BooleanLiteralExp*, *IntegerLiteralExp* and *NullLiteralExp* model elements. Since they do not present new challenges, no additional comments are added.

Loop expressions

When introducing the running example, Section 3.2.2 stated that mOCL provides syntax to declare some loop expressions, in particular, iterate and collect expressions. Listing 4.57 shows how the CS2AS bridge for loop expressions is expressed with CS2AS-TL. With respect to the mappings section, the different way of creating distinct model elements conforming

```
disambiguation {
  LetExpCS {
    singleVarDecl := letVars->size() = 1;
    multipleVarDecls := letVars->size() > 1;
  }
}
mappings {
  create LetExp from LetExpCS
  when singleVarDecl {
    ownedVariable := letVars->at(1);
    ownedIn := inExp;
    type := inExp.trace.type;
  }
  create LetExp from LetExpCS
  when multipleVarDecls {
    ownedVariable := letVars->first();
    ownedIn := letVars->excluding(letVars->first())->reverse()
                   ->iterate(x : LetVarCS; acc : OCLExpression = inExp.trace |
                          LetExp {
                            ownedVariable = x.trace,
                            ownedIn = acc,
                            type = acc.type
                          });
    type := inExp.trace.type;
  }
}
```

LISTING 4.55: CS2AS description of *LetExp*

```
mappings {
  create BooleanLiteralExp from BooleanLiteralExpCS {
    booleanSymbol := boolSymbol;
    type := lookup(Class, 'Boolean');
  }
  create IntegerLiteralExp from IntegerLiteralExpCS {
    integerSymbol := intSymbol;
    type := lookup(Class, 'Integer');
  }
  create NullLiteralExp from NullLiteralExpCS {
    type := lookup(Class, 'OclVoid');
  }
}
```

LISTING 4.56: CS2AS description of mOCL literal expressions

to the same AS meta-class is highlighted. The distinction resides in how a particular AS reference is computed (line 11). In this case, the two different outcomes are handled by an OCL if expression. If no iterator variable is declared, an implicit one is created. With respect to name resolution, both expressions create a new nested scope for their children (lines 22 and 26). In this way, the implicit iterator (variable) occludes any other implicit variable declared in outer scopes (e.g. in another outer loop expression). For *IterateExp*, the contribution is explicitly specified to apply to the child contained via the *IterateExp::ownedBody* reference property. The rationale is that the accumulator (contained via *IterateExp::ownedResult* reference) has an init expression, which is not allowed to refer to the contributed targets.

```
 1  mappings {
 2    create IterateExp from IterateExpCS {
 3      ownedIterator := itVar;
 4      ownedResult := accVar;
 5      ownedBody := exp;
 6      ownedSource := parentAsCallExpCS();
 7      type := trace.ownedResult.type;
 8    }
 9    create IteratorExp from CollectExpCS {
10      name := 'collect';
11      ownedIterator := if itVar = null
12                       then Variable { name='__implicit__', type=lookup(Class,'
                             OclAny') }
13                       else itVar.trace
14                       endif;
15      ownedBody := exp;
16      ownedSource := parentAsCallExpCS()._source;
17      type := lookup(Class,'Collection');
18    }
19  }
20  name_resolution {
21    providers {
22      IteratorExp {
23        in current_scope
24          provides occluding ownedIterator;
25      }
26      IterateExp {
27        in current_scope
28          for ownedBody
29            provides occluding ownedIterator, ownedResult;
30      }
31    }
32  }
```

LISTING 4.57: CS2AS description of mOCL loop expressions

Implicit Collects and Collection Conversions

When explaining how Concern 7 (multi-way mappings) is addressed by CS2AS-TL, an exposition that deals with the *NameExpCS* model elements was shown. However, this exposition was incomplete. One missing part was the creation mappings for *OperationCallExp*. The omission was justified by the fact that these mappings are rather similar to the ones specified for *PropertyCallExp*. However, the other missing part addresses the mOCL characteristic related to dealing with the navigation operators (see Section 3.2.2), particularly with the creation of the additional implicit collect expression, and the operation call expression that performs a collection conversion.

Listing 4.58 shows the specific helpers, disambiguation rules and creation mappings that address these scenarios. A new helper is introduced (lines 4–5), which aims to determine whether the type of the source of the navigation is of type collection or not. Two additional disambiguation rules are introduced. Apart from the conditions that check if it is a navigation with an explicit source, these disambiguation rules perform additional checks:

- For operation call expressions (line 12), it is checked whether the operator is the ′->′ symbol and the source is not a collection. In that case, the creation of an implicit collection conversion is required.
- For property call expressions (line 15), it is checked whether the operator is the ′.′ symbol and the source is a collection. In that case, the creation of an implicit collect is required.

Looking at the mappings section (lines 17–45), the solutions for the two scenarios differ.

With respect to the creation of an operation call expression with the preceding implicit collection conversion (line 18), the solution consists of creating an intermediate *OperationCallExp* model element (lines 22–27) that refers to an *asCollection Operation* model element (lines 25–26). This new intermediate *OperationCallExp* model element is the new *ownedSource* (line 20) of the *OperationCallExp* model element created by the mapping. The original source expression is now the *ownedSource* (line 23) of the new intermediate *OperationCallExp* model element.

With respect to the creation of a collect iterator expression comprising the property call expression (line 33), the approach is different. The mapping creates an *IteratorExp* model element rather than a *PropertyCallExp* one (line 33). The expected property call expression is now the *ownedBody* (line 38) of the new iterator expression. The new *ownedSource* (line 39) of the expected property call expression is a *VariableExp* model element (lines 39–40) that refers to the *ownedIterator* (line 39) of the *IteratorExp* model element created by the mapping.

4.2.10 Sufficiency of mOCL

Once the CS2AS bridge[2] of mOCL has been presented, this subsection aims to address external validity threats that may arise by using mOCL as a simplified version of OCL .

[2] The whole CS2AS bridge definition for mOCL can be found in Appendix C.

```
helpers {
   NameExpCS {
     -- other helpers omitted
     isTheSourceACollection() : Boolean :=
        parentAsCallExpCS()._source.trace.type = lookup(Class,'Collection');
   } }
disambiguation {
  NameExpCS {
    -- other disambiguation rules omitted
    isOperationCallExpWithExplicitSourceAndImplicitCollection :=
      roundedBrackets <> and isNavExpOfACallExpCS()
      and opName = '->' and not isTheSourceACollection();
    isPropCallExpWithExplicitSourceAndImplicitCollect :=
      roundedBrackets = null and isNavExpOfACallExpCS()
      and opName = '.' and isTheSourceACollection();
  } }
mappings {
  create OperationCallExp from NameExpCS
  when isOperationCallExpWithExplicitSourceAndImplicitCollection {
    ownedSource := let emptyArgs = OrderedSet{},
                       collType = lookup(Class, 'Collection')
                    in OperationCallExp {
                        ownedSource = parentAsCallExpCS()._source.trace,
                        ownedArguments = emptyArgs,
                        referredOperation = lookupExported(Operation, collType,
                                                 'asCollection', emptyArgs),
                        type = collType };
    ownedArguments := args;
    referredOperation := lookupExported(Operation,trace.ownedSource.type,
                                        expName,trace.ownedArguments);
    type := trace.referredOperation.type;
  }
  create IteratorExp from NameExpCS
  when isPropCallExpWithExplicitSourceAndImplicitCollect {
    name := 'collect';
    ownedSource := parentAsCallExpCS()._source.trace;
    ownedIterator := Variable { name='self', type=lookup(Class,'OclAny') };
    ownedBody := PropertyCallExp {
                   ownedSource = VariableExp{referredVariable=trace.ownedIterator,
                                             type=trace.ownedIterator.type },
                   referredProperty = lookupExported(Property,
                                            trace.ownedIterator.type,expName),
                   type = trace.referredProperty.type };
    type := trace.ownedSource.type;
  } }
```

LISTING 4.58: CS2AS description of mOCL loop expressions

OCL specification splits its AS definition into two different packages: the types package and the expressions package. In this subsection, a similar distinction is also made and both are commented on below.

OCL Types

mOCL presents a reduced type system compared with OCL. In the former case, types are represented with a simple meta-class *Class* with a name to identify the type. The more complex OCL type system contrasts in two different areas: firstly, the logic of type conformance (required to compute expression types) needs to address more complex scenarios. Secondly, name resolution description is also more elaborate because type lookups involve more than a simple name.

With respect to computing expression types in a more complex type system, creating the logic to deal with additional cases is not a relevant challenge. OCL expressions provide enough expressiveness, for instance, to compute complex type conformance rules as part of CS2AS-TL *helpers* section. According to the OCL specification [39], Listing 4.59 shows how some of these conformance rules could be defined with CS2AS-TL.

```
helpers {
  SequenceType { -- conforms to OclAny and SequenceTypes with conformant element type
    conformsTo(anotherType : Classifier) : Boolean :=
      not anotherType.isOclAny() implies
      anotherType.oclKindOf(SequenceType) and
        self.elementType.conformsTo(anotherType.oclAsType(SequenceType).
            elementType);
  }
  InvalidType { -- InvalidType conforms to all types
    conformsTo(anotherType : Classifier) : Boolean :=
       true;
  }
  VoidType { -- VoidType conforms to all types, except InvalidType
    conformsTo(anotherType : Classifier) : Boolean :=
      not anotherType.isOclInvalid()
  }
}
```

LISTING 4.59: CS2AS conformance rules for some OCL types

With respect to name-based lookups, the OCL built-in type system contains more complex types in which simple name-based lookups are not enough. For instance, *Tuples* are not identified by their name (which is simply *Tuple*), but by the parts that identify them. However, Section 4.2.5 explained how the lookup criteria can be augmented with new lookup information, which could be used to address the name resolution concern for more complex types. Listing 4.60 shows an example of name resolution definition to locate tuples.

Table 4.3 summarises the different CS2AS-TL features that are required to deal with each OCL type.

```
name_resolution {
  inputs {
    TupleType filtered_by tParts : OrderedSet(Property)
              when ownedProperties->size() = tParts->size() and
                   ownedProperties->
                    forAll(x | let tPart= tParts->at(ownedProperties->indexOf(x))
                               in x.name = tPart.name and x.type = tPart.type);
  }
}
```

LISTING 4.60: Defining additional lookup criteria for *Tuple* types

OCL Type	CS2AS-TL Features
AnyType	Contextual Helper Operations
InvalidType	Contextual Helper Operations
VoidType	Contextual Helper Operations
ClassType	Contextual Helper Operations
TupleType	Contextual Helper Operations and Additional Lookup Criteria
CollectionType	Contextual Helper Operations and Additional Lookup Criteria
OrderedType	Contextual Helper Operations and Additional Lookup Criteria
SequenceType	Contextual Helper Operations and Additional Lookup Criteria
OrderedSetType	Contextual Helper Operations and Additional Lookup Criteria
BagType	Contextual Helper Operations and Additional Lookup Criteria
SetType	Contextual Helper Operations and Additional Lookup Criteria
MessageType	Contextual Helper Operations and Additional Lookup Criteria

TABLE 4.3: CS2AS-TL features required to address OCL types

mOCL Expressions

mOCL presents a reduced number of expressions compared with OCL. The rationale is that, from the point of view of bridging CS2AS, many of the OCL expressions do not present a different challenge to the mOCL ones that have been addressed during this chapter.

Looking at the CS definition of OCL [39], varied syntax to express different kinds of expressions can be found. For instance, there are different operators to depict logic expressions (e.g. *and*, *or*, *xor* and *implies*), arithmetic expressions (e.g. +, -, *, /) etc.

The main reason why this kind of expression can be omitted is that, from the point of view of bridging CS2AS, they all present the same scenario: in essence, creating an operation call expression with a source and arguments. Therefore, regardless of whether the operators are binary (one argument) or unary (no arguments), all CS2AS-TL features presented to address *equality expressions* can be used for other similar expressions that represent operation call expressions. Listing 4.61 shows the excerpt for unary expressions (e.g. *not* expression).

```
mappings {
  create OperationCallExp from UnaryExpCS {
    ownedSource := left;
    referredOperation := lookupExported(Operation, trace.ownedSource.type,
                                        opName, OrderedSet{});
}
```

LISTING 4.61: CS2AS bridge to create *OperationCallExp* from a unary expression

With respect to the remaining expressions, such as *IfExp* or the different sorts of *LiteralExp*, there are no additional challenges to explain. Here, the already mentioned requisite of computing the *TypedElement::type* cross-references for all the expressions is discussed. Contextual helpers can be used for creating the required logic. As Listing 4.62 shows, in the case of an *IfExp*, it should be the common super type of both branch expressions.

```
helpers {
   IfExp {
     computeType() : Classifier  :=
       ownedThenExp.type.commonSupertype(ownedElseExp.type);
   }
}
mappings {
  create IfExp from IfExpCS {
    ownedCondition := condition;
    ownedThenExp := _then;
    ownedElseExp := _else;
    type := computeType();
  }
}
```

LISTING 4.62: CS2AS bridge to create *IfExp* model elements

Table 4.4 summarises the different CS2AS-TL features that are required to deal with each kind of OCL expression.

OCL Expression	CS2AS-TL Features
PropertyCallExp	Contextual helpers, multi-way mappings, exported name-based lookups
OperationCallExp	Contextual helpers, multi-way mappings, exported name-based lookups
PrimitiveLiteralExp	Contextual helpers, one-way mappings, name-based lookups
CollectionLiteralExp	Contextual helpers, one-way mappings, name-based lookups
TupleLiteralExp	Contextual helpers, one-way mappings, name-based lookups
VariableExp	Contextual helpers, multi-way mappings, name-based lookups
IfExp	Contextual helpers, one-way mappings
TypeExp	Contextual helpers, one-way mappings, name-based lookups
IteratorExp	Contextual helpers, multi-way mappings, selective targets provision
IterateExp	Contextual helpers, multi-way mappings, selective targets provision
MessageExp	Contextual helpers, multi-way mappings, name-based lookups
StateExp	Contextual helpers, one-way mappings, name-based lookups

TABLE 4.4: CS2AS-TL features required to address OCL expressions

4.2.11 Related Work Discussion

This subsection discusses how CS2AS-TL relates to previous work.

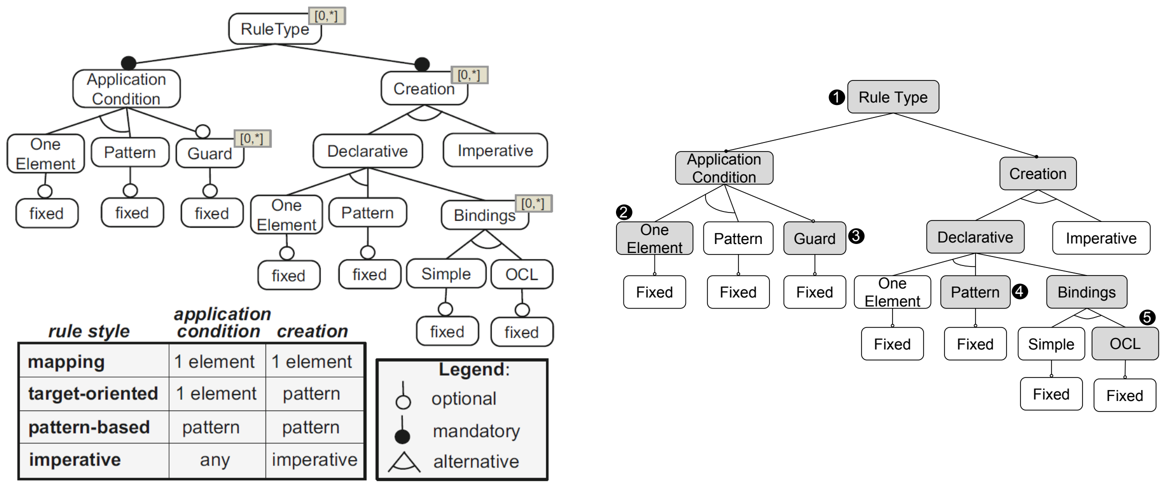
General Purpose M2M Transformations

The presented solution is based on domain-specific M2M transformation language. One question that may arise is the benefits that this new language provides with respect to the already existing general purpose M2M transformation languages. When evaluating the solution in the next chapter, this topic is particularly discussed in Section 5.1.

DSTL Design

With respect to designing the DSTL, Sánchez-Cuadrado et al. [68] propose a systematic approach to construct domain-specific transformation languages. In this case, by means of the DSL they propose, a language designer can declare new DSTLs for which a CS definition, an AS definition, an IDE, and even some execution infrastructure are automatically generated.

Figure 4.14 shows, on the left hand side, a feature model describing the possible features of DSTL, according to Sánchez-Cuadrado et al. [68]. On the right hand side of the figure, a configuration model shows the specific features (in grey) that characterise our solution: declarative rules ❶ are specified on one element ❷ type of the source CS meta-model, which may define guards (disambiguation rules) ❸ for multi-way mappings. The outcome of these mappings may be several model elements (a pattern) ❹, and the properties of the created model elements may be initialised by means of OCL-based bindings ❺.



rule style	application condition	creation
mapping	1 element	1 element
target-oriented	1 element	pattern
pattern-based	pattern	pattern
imperative	any	imperative

FIGURE 4.14: Left: Feature model to characterise the rule types of a DSTL (Figure 3 from [68]). Right: Configuration model that depicts the features selected by CS2AS-TL

Although it is possible to classify CS2AS-TL by means of Sánchez-Cuadrado et al.'s feature model, there are some features that are not appropriately classified. For instance, there is no concept of reference rule in the feature model, or room for the specific name resolution section.

Related DSTLs

Moving deeper into the specific domain of bridging the CS and AS of a CTML by means of CS2AS-TL, there are several relevant works:

- Gra2Mol [42] defines a DSTL to bridge CS grammars and AS meta-models.
- Spoofax [46] defines a set of languages to create tools (parsers, editors) for textual languages. In particular, they define a language called *NaBL* to declare name resolution.

A qualitative study to evaluate these two works is presented in Chapter 5. The reader is referred to Section 5.2 and Section 5.3 for more details.

4.3 Complete OCL-Based Internal DSTL

Having presented CS2AS-TL, the following sections explain how the CS2AS transformations are actually executed. In particular, the different compilation steps introduced during the solution overview (see Figure 4.2) are explained. This section introduces the first compilation step which is responsible for compiling instances of CS2AS-TL into instances of a Complete OCL-based internal DSTL.

4.3.1 Overview

The first compilation step consists of generating a set of Complete OCL files that comprise another declarative exposition of how outputs (AS models) are computed from inputs (CS models). In this way, many of the higher level of abstraction constructs introduced by CS2AS-TL are expanded into a pure OCL-based CS2AS bridge.

The rationale behind the existence of this compilation step is the following:

- In terms of prototype development, this Complete OCL-based internal DSTL was developed prior to CS2AS-TL. This internal DSTL was used to execute CS2AS transformations by means of existing tooling (from the Eclipse OCL and QVTd projects), whilst it helped to obtain the requirements for CS2AS-TL.
- In terms of industrial partner interest, there was interest for this pure OCL-based language, so that a CS2AS bridge can be completely modeled in an OMG specification language. In this way, these CS2AS bridges can be added to the standardised specifications.

The Complete OCL-based DSTL is internal [31] and uses only facilities proposed for OCL 2.5 [11]. In essence, the DSTL constrains the use of the general purpose OCL language to define a set of idioms that express CS2AS bridges.

The compiler has been developed with Xtend [84] and, from instances of CS2AS-TL, it generates the corresponding Complete OCL documents in a source folder of the user workspace. More precisely, the generation consists of four files:

- A file with the OCL helper operations corresponding to the helpers section of CS2AS-TL.
- A file with *Boolean*-valued OCL operations corresponding to the disambiguation rules declared in the disambiguation section of CS2AS-TL.
- A file with a set of OCL lookup operations that encodes the lookup algorithms corresponding to the name resolution section of CS2AS-TL.
- A file with a set of OCL operations that declare how AS model elements are computed from CS model elements, corresponding to the mappings section of CS2AS-TL.

The following subsections introduce some details about how the Complete OCL documents are produced.

⚠ Exposing all the implementation details of the Complete OCL-based internal DSTL would make this thesis unnecessarily long. The following subsections present some specific examples of mOCL, whilst the entire exposition can be consulted in Appendix D. The reader is also referred to this published work [66].

4.3.2 Helpers

The helpers section is straightforwardly generated as a collection of operation definitions declared in a Complete OCL file. Figure 4.15 shows an example of how two helpers are translated. Since CS2AS-TL reuses OCL as the expressions language, the body of the helper operations is translated as it is to the body of the OCL operation definitions.

4.3.3 Disambiguation Rules

The disambiguation section is translated to a new Complete OCL document as a collection of *Boolean*-valued operation definitions, one per disambiguation rule. Figure 4.16 shows an example where the disambiguation rules defined to disambiguate *CollectionLiteralPartCS* are translated to the corresponding Complete OCL operation definitions.

4.3.4 Name resolution

The name resolution section is translated to a new Complete OCL document as a set of operation definitions that resolve name-based lookups. These OCL operations encode the different lookup algorithms explained in the previous Section 4.2. When compared to other CS2AS-TL sections, this part of offers the higher level of abstraction constructs. Therefore, the amount of OCL code corresponding to the name resolution section is substantial. Listing 4.63 shows a simple example that includes the name resolution declaration related to variable lookups.

Listing 4.64 shows the corresponding Complete OCL operations that encode the name-based lookups. The first four operations consist of a set of lookup operations. The first

```
source cs : 'generated/MiniOCLCS
    .ecore#/'
target as : '/org.eclipse.qvtd.
    doc.miniocl/model/MiniOCL.
    ecore#/'
helpers {
  Class {
    commonSupertype(another : Class)
        : Class :=
    let allSupertypes = self->
        asOrderedSet()->closure(
        superClasses),
        allOtherSupertypes = another->
            asOrderedSet()->closure(
            superClasses)
    in allSupertypes->intersection(
        allOtherSupertypes)->any(true);
    conformsTo(another : Class) :
        Boolean :=
    self = another or
    superClasses->exists(conformsTo(
        another));
  }}
```

```
import cs : 'generated/MiniOCLCS
    .ecore#/'
import as : '/org.eclipse.qvtd.
    doc.miniocl/model/MiniOCL.
    ecore#/'
package as
context Class
def: commonSupertype(another:Class)
    : Class =
  let allSupertypes = self->
      asOrderedSet()->closure(
      superClasses),
      allOtherSupertypes = another->
          asOrderedSet()->closure(
          superClasses)
  in allSupertypes->intersection(
      allOtherSupertypes)->any(true);
def: conformsTo(another : Class) :
    Boolean =
  self = another or
  superClasses->exists(conformsTo(
      another))
endpackage
```

FIGURE 4.15: Left: Some helper definitions using the CS2AS-TL. Right: Corresponding OCL operation definitions

```
disambiguation {
  CollectionLiteralPartCS {
    withoutLastExpression :=
      last = null;
    withLastExpression :=
      last <> null;
  }
}
```

```
package cs
context CollectionLiteralPartCS
def : withoutLastExpression() : Boolean =
          last = null
def : withLastExpression() : Boolean =
          last <> null
endpackage
```

FIGURE 4.16: Left: Some disambiguation rules using CS2AS-TL. Right: Corresponding OCL operation definitions

```
1  name_resolution {
2    targets {
3      NamedElement using name escaped_with '_';
4      Variable;
5    }
6    inputs {
7      PathElementCS using elementName;
8    }
9    providers {
10     LetExp {
11       for all excepting ownedVariable
12         provides occluding ownedVariable;
13     }
14   }
15 }
```

LISTING 4.63: Name resolution declaration corresponding to *Variable* lookups

receives a *String* with the name of the variable to look up (lines 4–5). Since the name resolution section declares an additional input (lines 6–8 from Listing 4.63), another *lookupVariable* method is generated that receives that type of input (lines 6–7). The third and fourth operations (lines 8–21) encode the lookup algorithm: the third operation (lines 8–15) receives the *String* value of the variable to look up, and it invokes (line 10) the fourth one with an extra argument that provides the current lookup environment[3]. This environment comprises all the candidate *Variable* model elements contributed by their parent model elements. Note that if no variable is found, and the variable name is escaped (using the '_' prefix), a new lookup is performed using the variable name without the escape prefix (see line 12). The fourth operation (lines 16–21) queries the passed lookup environment and selects the *Variable* model elements that have the same name with the passed variable name (line 17). Note that if no variable is found, and the passed lookup environment is a nested one, a lookup in the parent environment is tried (line 19).

The three operations that follow comprise the default implementation of lookup environment computations (lines 23–30). The first operation (lines 23–24) is always invoked to compute the current environment of a particular AS model element (line 9). By default, the operational behaviour simply consists of invoking the third operation (line 26) with the aim of obtaining the current lookup environment propagated by its parent. This third operation (lines 27–30) consists of obtaining the lookup environment computed by its parent model element (line 30). In the case of a root model element (no parent), an empty environment is created (line 29).

[3] OCL specification [39] refers to *Environments* in its Clause 9. They are the actual implementation of the current *scopes* explained in the previous subsection.

According to the explained OCL operations, an empty lookup environment is propagated from the root model element to the particular children that require it. For every contribution to the current scope made by a particular AS model element, a new operation is generated (lines 31–37). This last operation overloads the second one explained above (line 25–26). In this way, the default operational behaviour is modified according to the target contribution. Note that the *child* operation parameter can be used to drive different contributions based on the child for which the lookup environment is propagated. In this example (lines 31–37), for every child that is not the *LetExp::ownedWariable*, the propagated environment is a new nested environment (from the one propagated by the *LetExp* parent), including the *Variable* contained by the *LetExp::ownedVariable* reference. In the opposite case, in which the child is contained via the *LetExp::ownedVariable* reference, the default behaviour applies.

The whole name resolution declaration and the corresponding Complete OCL document can be found in Appendix C and Appendix D respectively. It can be observed that **65** lines of code of CS2AS-TL name resolution section correspond to **480** lines of code of a Complete OCL file.

4.3.5 Mappings

Finally, a fourth Complete OCL document is generated, comprising the mappings definition declared within the CS2AS-TL mappings section. The concept of mapping does not exist in OCL. Therefore, the Complete OCL-based internal DSTL uses the following conventions to express mapping declarations:

- A mapping consists of a parameter-less *ast()* operation definition. The context of the operation definition corresponds to the input (source) type of the mapping, whereas the operation result type corresponds to the output (target) type of the mapping.
- For creation mappings, the body of the *ast()* operation definition is a shadow expression[4], which represents the type of element that is actually created by the mapping. Every property assignment of the creation mapping is translated as a shadow part of that shadow expression.
- For reference mappings, the body of the *ast()* operation definition is the expression declared within the reference mapping.
- For multi-way mappings, all the mapping output results (corresponding to the same input type) are *"merged"* within the same *ast()* operation definition, whereas the multi-way execution is controlled by nested if expressions, one per multi-way mapping. Every *IfExp::ownedCondition* comprises a call expression of the operation corresponding to a disambiguation rule. The *IfExp::ownedThen* comprises the *ShadowExp* of the corresponding output model element to create. Finally, the *IfExp::ownedElse* comprises the

[4] Concept proposed for inclusion in OCL 2.5 [11]. It has been implemented in Eclipse OCL and OCLT [45].

```
package as
context Element -- API for variable lookups
-- Operations for variable lookup
def : lookupVariable(vName : String) : Variable[?] =
  _lookupUnqualifiedVariable(vName)
def : lookupVariable(aPathElementCS : cs::PathElementCS) : Variable[?] =
  _lookupUnqualifiedVariable(aPathElementCS.elementName)
def : _lookupUnqualifiedVariable(vName : String) : Variable[?] =
  let env = unqualified_env_Variable() in
  let foundVariable = _lookupVariable(env, vName)
  in  if foundVariable->isEmpty()
      then if vName.isScaped() then _lookupVariable(env, vName.unscape())
           else null endif
      else foundVariable->first()
      endif
def : _lookupVariable(env : LookupEnvironment, vName : String):OrderedSet(
    Variable) =
  let foundVariable = env.namedElements->selectByKind(Variable)->select(name =
      vName)
  in  if foundVariable->isEmpty() and not (env.parentEnv = null)
      then _lookupVariable(env.parentEnv, vName)
      else foundVariable
      endif
-- Operations for environment computation
def : unqualified_env_Variable() : LookupEnvironment[1] =
  _unqualified_env_Variable(null)
def : _unqualified_env_Variable(child : OclElement) : LookupEnvironment[1] =
  parentEnv_Variable()
def : parentEnv_Variable() : LookupEnvironment[1] =
  let parent = oclContainer()
  in if parent = null then LookupEnvironment {} -- Empty Environment
     else parent._unqualified_env_Variable(self) endif -- The environment of my
         parent
context LetExp -- LetExp contributes variables to the current scope
def : _unqualified_env_Variable(child : ocl :: OclElement) : LookupEnvironment[1] =
  if not (ownedVariable->includes(child))
  then parentEnv_Variable().nestedEnv()
           .addElements(ownedVariable)
  else parentEnv_Variable()
  endif
```

LISTING 4.64: OCL operation definitions corresponding to the name resolution declaration from Listing 4.63

next nested *IfExp* of the next multi-way mapping. The last *IfExp::ownedElse* comprises the *fall-back* multi-mapping or *invalid* literal expression in case no one is defined. The order in which the *IfExp* are nested is driven by the order in which the corresponding disambiguation rules are declared. An explanatory example is shown below.

- Trace expressions are translated as *ast()* call expressions.

Listing 4.65 shows the *ast()* OCL operation that results from the two-way mapping declared on *CollectionLiteralPartCS* (see Section 4.2.6). A parameter-less *ast()* operation definition (line 3) is declared on *CollectionLiteralPartCS* (line2). The result type of that operation is a *CollectionLiteralPart* (line 3), which is the common super type of the AS model elements to be created by the shadow expressions involved, i.e. *CollectionItem* (lines 5–8) and *CollectionRange* (lines 10–14). Since the translation comes from multi-way mappings, the body of the *ast()* operation definition consists of a set of nested *if* expressions. The condition of the first *if* expression (line 4), invokes the boolean-value operation corresponding to the first disambiguation rule *'withoutLastExpression'* (see previous Figure 4.16). If the disambiguation rule applies, a *CollectionItem* model element is created (lines 5–8), otherwise the next nested if expression is considered. The condition of this second *if* expression (line 9), invokes the boolean-value operation corresponding to the second disambiguation rule *'withLastExpression'*. If the disambiguation rule applies, a *CollectionRange* model element is created (lines 10–14). If no disambiguation rule applies, there is an invalid situation. Note that this invalid situation cannot ever happen because a *CollectionLiteralPartCS* has either a last expression or none.

```
1  package cs
2  context CollectionLiteralPartCS
3  def : ast() : as::CollectionLiteralPart =
4    if withoutLastExpression()
5    then as::CollectionItem {
6      ownedItem = first.ast(),
7      type = ast().ownedItem.type
8    }
9    else if withLastExpression()
10     then as::CollectionRange {
11       ownedFirst = first,
12       ownedLast = last,
13       type = ast().ownedFirst.type
14     }
15     else
16       invalid
17   endif endif
18 endpackage
```

LISTING 4.65: OCL operation definition corresponding to the multi-way mappings whose source is *CollectionLiteralPartCS*

4.3.6 Related Work Discussion

Although OCL (and variants) can be found as the expression language of several M2M transformation languages (e.g. QVT [37], ETL [51] and ATL [44]), there is little related work that proposes OCL on its own as an M2M transformation language. In this thesis, and in published work [66], OCL is proposed as the host language of an internal DSTL.

Alternatively, recent work from Jouault et al. [45] proposes OCL as a functional model transformation language called *OCLT*. Apart from shadow expressions, it relies on other constructs such as pattern matching to enable the declaration of functional model transformations. In terms of execution, it relies on "a specific execution layer to enable traceability and side effects on models". Instead, the prototype developed for this thesis compiles to, and reuses, existing M2M technologies.

4.4 Transformation Execution: QVTd

Previous sections explained CS2AS-TL, and showed how the corresponding Complete OCL files (conforming to an internal DSTL) are produced. The last steps of the compilation process (see Figure 4.2 from Section 4.1) consist of producing the final artefacts in charge of executing CS2AS model transformations. The executable M2M transformation engine on which the proposed prototype relies, belongs to the Eclipse QVTd [26] project. The following subsections give an overview of the approach, and explain details of the compilation process.

4.4.1 Overview

The prototype relies on the Eclipse QVTd project. This Eclipse project contains the implementation of the two declarative M2M transformation languages of the QVT specification: QVTr and QVTc. The aim of reusing this third-party project is to execute M2M transformations in charge of producing AS models from CS ones. Whilst the proposed Complete OCL-based internal DSTL is expressive enough to define declarative CS2AS transformations, a pure OCL-based engine cannot execute them. The next step within the proposed approach is compiling to M2M transformations. In particular, as part of the solution implementation, the goal is producing executable QVT transformations in the context of the Eclipse QVTd project.

The rationale behind choosing this particular technology is the following:

- The QVT family of languages includes OCL as the expression language. Therefore, when translating expressions from the Complete OCL-based internal DSTL to any of the QVT languages, no special considerations are required to translate the expressions.
- The M2M transformation languages corresponding to QVTd are declarative. Since the OCL-based internal DSTL is also declarative, targeting a declarative M2M transformation language is very convenient, because the implementation does not have to switch from a declarative transformation exposition to the required imperative execution.

- The Eclipse QVTd project has a code generator capable of producing the Java class responsible for transforming models. Recent works [78, 77] have started to demonstrate the utility of this approach and, in particular, the significant gains in terms of execution time.
- From a pragmatic perspective, the sponsor of this EngD project is interested in focusing on this technology (to test it, improve it and benefit from it).

Figure 4.17 shows an overview of the compilation process. From the set of four Complete OCL documents ❶ corresponding to an instance of the CS2AS internal DSTL, only the one corresponding to the mappings is translated to a QVT Minimal (QVTm) file ❷. This file comprises a QVT M2M transformation containing a set of so called micro-mappings [77], and it imports the other three Complete OCL documents that contain all the callable OCL operation definitions (helpers, disambiguation rules and lookup operations). Finally, this QVT transformation conforms the input to a code generator (belonging to the Eclipse QVTd project) that produces that final Java class ❸ in charge of transforming CS models into AS models.

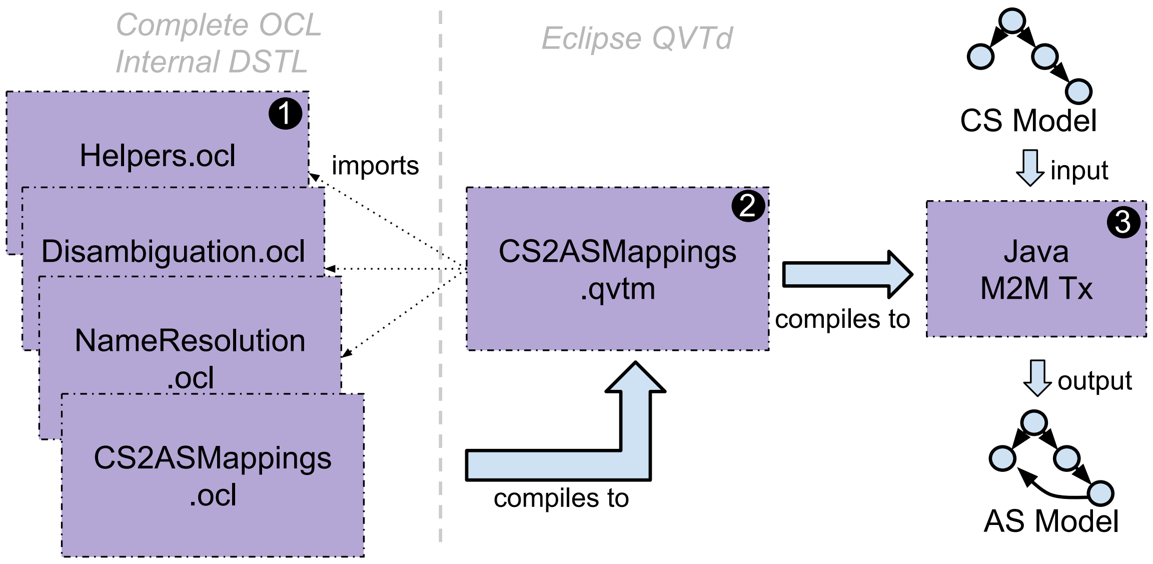


FIGURE 4.17: Compiling the set of Complete OCL files that declare a CS2AS bridge into an executable M2M transformation in Java

4.4.2 QVTm compilation: Micro-Mappings

Explaining and clarifying the details of the Eclipse QVTd functionality go beyond the objectives of this thesis. That said, this subsection briefly introduces the concept of micro-mapping, and gives an overview of how these micro-mappings are created from the Complete OCL document that declares a CS2AS bridge.

The concept of QVT Minimal (QVTm) first appears in [76], where a transformation is considered as minimal when features such as multi-directionality, mapping refinement and composition, have been removed from the original (QVTc) transformation. More recently, the concept of primitive micro-mapping has been explained [77]: "A declarative transformation may therefore execute one commit action at a time, where a commit action either creates a node or assigns an edge. The type of object created, or the value assigned by the commit-action, is computed from zero or more objects or values that must be ready for use. We therefore wrap up the commit-action inside a primitive Micro-Mapping to include the input parameters, predicates and computations that influence the commit-action. A primitive Micro-Mapping is therefore similar to the mapping or rule or relation of declarative transformation languages, but is constrained to a single commit-action".

In this case, as long as the input objects of a micro-mapping are ready, a single commit-action permits the micro-mapping to be successfully executed from start to end. Then, it is only a matter of a transformation scheduler adequately sequencing the invocation of the micro-mappings. When sequencing these micro-mapping invocations, the scheduler statically ensures that the inputs of every micro-mapping are ready upon invocation. The simpler and smaller the micro-mappings are, the easier to sequence the invocations without suffering deadlocks between mappings (e.g. a mapping requires an input that another one produces, and vice versa). Listing 4.66 shows an example of a micro-mapping where there is a *check* (input) domain that declares an input object of type *InputType* (line 3). There is an *enforce* (output) domain that realises an output object of type *OutputType* (line 6). The micro-mapping additionally contains a property assignment so that the property *aProp* of the output object is initialised with the value of the property *anotherProp* of the input object.

```
1  map input_2_output in myTransformation
2  {
3    check inputDomain(inputObj : InputType[1]
4    |) {}
5    enforce outputDomain() {
6      realize  outpuObj : OutputType[1]
7      |}
8    where() {
9      outputObj.aProp := inputObj.anotherProp;
10   }
11 }
```

LISTING 4.66: Simple QVTm micro-mapping

In the prototype implementation, when generating the micro-mappings of a QVTm transformation, two kinds of micro-mappings are produced:

- A creation micro-mapping. This receives an input model element, and creates an output model element, as well as a trace link between them by updating the traceability property of the input model element.
- An update micro-mapping. This receives an input model element but it does not create an output model element. Instead, it updates a property from an output model element. The corresponding output model element is accessed via the traceability property of the input model element. Note that according to the micro-mapping definition,

update micro-mappings cannot be executed until the corresponding creation micro-mapping has been firstly executed. Otherwise, the output model element would not have been created and the mapping execution would fail.

With these two different kinds of micro-mapping, the compilation process of producing QVTm transformations from the set of OCL operation definitions that declare CS2AS bridges is as follows. The shadow expressions that are in the body of *ast()* operations produce the following micro-mappings:

- One creation micro-mapping, where the input type is the context of the OCL operation definition and the output type is the type of the shadow expression. There is a single property assignment to update the trace property that links the input model element to the output one.
- One update micro-mapping per shadow part of the shadow expression, where the input type is the context of the OCL operation definition. There is a single property assignment to update a property of the output model element.

For instance, Figure 4.18 shows, on the left, a partial *ast()* operation definition corresponding to the mapping that creates *Class* model elements from *ClassCS* model elements. On the right, the corresponding two micro-mappings are shown. A creation mapping (lines 1–11) receives a *ClassCS* model element (line 3), and it realises a *Class* model element (line 6). Additionally, the traceability property (called *ast*) of the *ClassCS* model element is assigned to the created *Class* model element (line 9). An update mapping (lines 12–20) receives a *ClassCS* model element (line 14). In this case, no additional model element is created. Instead, the *Class* model element corresponding to the input *ClassCS* model element is updated (line 18). In particular, the value of the property *name* of the *ClassCS* model element is assigned to the property *name* of the *Class* model element.

4.4.3 Related Work Discussion

The idea of compiling to existing languages to reuse execution engines is not new. With the added value of working on transformation languages that are defined by meta-models [8], this task is reduced to create another (high-order) M2M transformation [73]. In their approach for a systematic construction of DSTLs, Sánchez-Cuadrado et al. [68] also rely on a similar approach. In this case, the target transformation language is a different one (Eclectic [67]), but the goals are the same: reusing the existing engine and the tools around the target transformation language.

4.5 Language Workbench Integration: Xtext

This section introduces additional work that identifies further benefits from having CS2AS-TL to declare CS2AS bridges. In this case, from instances of CS2AS-TL, additional functionality could be generated to complement the default textual editor produced by Xtext. In

```
context ClassCS
def : ast () : as :: Class =
  as :: Class {
    name = name,
    -- Other shadow parts
  }
```

```
map cClassCS_2_Class in MiniOCLCS2AS
{
  check leftCS(lClassCS : cs :: ClassCS[1]
  | ) {}
  enforce rightAS() {
    realize rClass : as :: Class[1]
  | }
  where() {
    lClassCS.ast := rClass;
  }
}
map uClassCS_2_Class_name in MiniOCLCS2AS
{
  check leftCS(lClassCS : cs :: ClassCS[1]
  | ) {}
  enforce rightAS() {}
  where() {
    lClassCS.ast.oclAsType(as::Class).name :=
        lClassCS.name;
  }
}
```

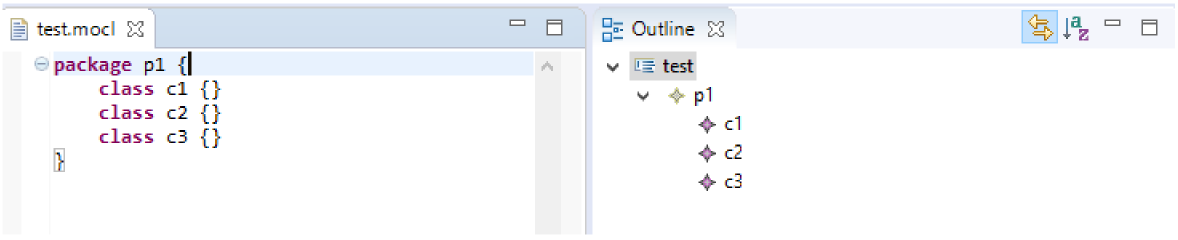
FIGURE 4.18: Left: a partial *ast()* operation definition that declares a mapping between *ClassCS* and *Class*. Right: the corresponding QVTm micro-mappings

particular, the following subsections show two integration features. The first one focuses on providing an outline that shows the final AS model obtained from CS2AS transformations. The second one shows how the editor code completion assistant can be enhanced by showing the available targets that are visible in a particular AS model element.

⚠ This section is not intended to formalise Xtext-based tooling creation, but to show some hints of the benefits of language workbench independent DSLs to describe CS2AS bridges. Showing how this kind of tooling can be created in the context of a different language workbench is beyond the goal of this research project. Moreover, technical details about the Xtext integration have been omitted.

4.5.1 AS Outline

One typical helper view of Eclipse-based IDEs is the outline view. This view is normally linked to the active editor, and it shows information relevant to the file that is currently edited. When editing textual modeling languages, the information that has been considered to show in the outline is the structure of the underlying AS model. Every time that the edited file comprises a valid syntactic instance of the language, the CS2AS transformation is executed and the corresponding AS is shown in the outline, where each node corresponds to an element of the AS model. Figure 4.19 shows a screen-shot of the enhanced editor feature.

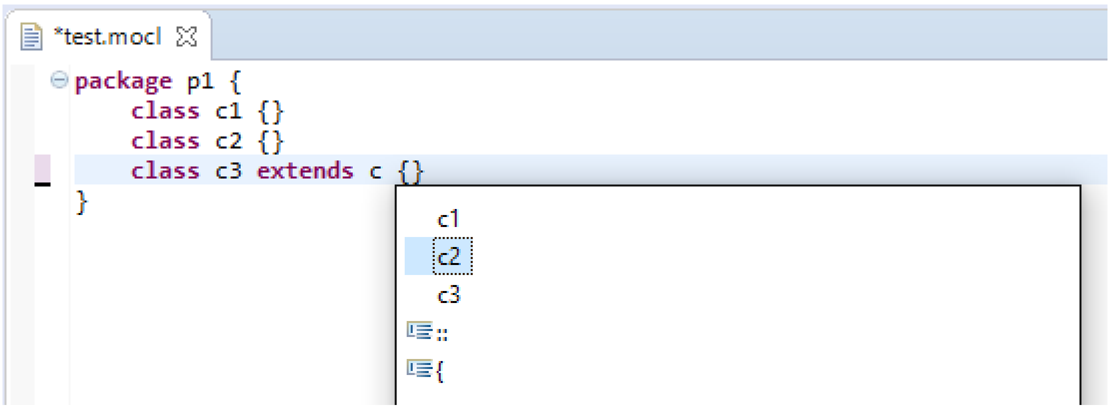


```
package p1 {
    class c1 {}
    class c2 {}
    class c3 {}
}
```

FIGURE 4.19: Left: Textual mOCL file. Right: corresponding AS model representation of the outline

4.5.2 Name resolution based code completion

Another useful feature of modern IDEs is the code auto-completion/assistant[5]. By default, Xtext provides some useful alternatives based on grammar definition (e.g. keywords and punctuation symbols). However, when having additional specification artefacts, such as CS2AS bridges, more options could be included. In particular, the name resolution describes the candidate targets that are contributed to a lookup environment, hence, all the model elements that are visible within a particular scope. All these candidate targets can be additionally included as part of the suggestions of a code completion assistant. Figure 4.20 shows a screen-shot where, after the *extends* keyword, the candidate target classes appear in the code assistant.



```
package p1 {
    class c1 {}
    class c2 {}
    class c3 extends c {}
}
```

FIGURE 4.20: Textual mOCL file on the left, and corresponding AS model representation of the outline on the right

4.6 Limitations

To conclude the chapter, this section identifies limitations of the proposed solution. They are categorised from two different points of view: the presented CS2AS-TL (main contribution of this thesis) and the prototype implementation.

[5] In Eclipse, it appears after pressing ctrl+space.

4.6.1 CS2AS-TL

Given the kind of required M2M transformations – which only involve one input domain (CS meta-model/s) and one output domain (AS meta-model/s) – and the expressive power that OCL provides, CS2AS-TL offers sufficient means to address varied and complex CS2AS scenarios. Even the constructs that provide a higher level of abstraction[6], such as the name resolution section, can be replaced by a set of reusable helper operations.

Beside the varied CS2AS scenarios presented over the course of this chapter, the evaluation shows additional examples in Chapter 5 to demonstrate the suitability of the language. However, there are a few limitations that have been identified and they are commented on below:

- The main limitation of the approach is the following: the information that has to be obtained in the AS model has to be computable from the information present in the CS model. For instance, if the AS model required some attribute value to be initialised with the date in which the model was created, the creation of such a model would not be possible unless that information were present in the CS model. All computation capability is limited to working on CS information with the help of the additional operational behaviour that OCL provides. In other words, there is no extension mechanism (i.e. black-box operations) to incorporate more computation facilities that operate beyond the CS2AS-TL (i.e. OCL) capabilities.

- Reference mappings do not currently accept disambiguation rules. Hypothetically, a CS2AS scenario may require that, depending on disambiguation rules, a CS model element could either create new AS model elements or refer to them.

- Although language extension and composition is not explicitly addressed throughout the thesis, working on a family of languages is supported and it has been tested with small examples within the Eclipse QVTo project [7]. Providing that a CS2AS bridge has been declared for a base (extended) language, another CS2AS bridge could be separately declared for the derived (extending) one. Declaring mappings on new or derived CS meta-classes is straightforward, and the final mappings follow the mapping inheritance rules [82] of the target QVTm transformation language. However, there is a shortcoming in the current CS2AS-TL design due to the semantics of the disambiguation rules section. The order in which these rules are declared for a base meta-class is rigid, and it cannot be changed by a derived CS2AS bridge.

4.6.2 Prototype Implementation

The solution presented in this chapter has been prototyped using official Eclipse technologies. Although all the ideas behind the CS2AS-TL are technology agnostic, the actual prototype is not so. The following list enumerates the relevant limitations of the current prototype implementation:

[6] Therefore, they impose more restrictions on what needs to be expressed.

[7] `git://git.eclipse.org/gitroot/mmt/org.eclipse.qvto.git`

- The main limitation is that it has been developed for and using a specific modeling framework, Eclipse Modeling Framework (EMF). Therefore, the CTMLs that the current prototype supports need to be developed using this technology. Other modeling tools such as Epsilon [50] can work with varied forms of models and meta-models, but this is not the case for the current prototype.

- The final M2M transformation is a Java class generated from a code generator hosted by the Eclipse QVTd project. Reflective EMF is not supported by the generated transformation, meaning that the Java classes corresponding to the source and target meta-models need to have been generated as well.

- The type inference stated during the CS2AS-TL explanation is not actually implemented. This means that for a neat compilation process from an instance of CS2AS-TL to the final Java class transformation, the (otherwise unnecessary) long-form of the presented syntax is required. For instance, every contribution requires specifying the type of the target that is contributed to a lookup scope. This negatively impacts only on CS2AS-TL when measuring the size of the artefacts required to declare a valid CS2AS bridge (see the evaluation in Chapter 5).

- CS2AS-TL does not impose restrictions on the CS and AS meta-models that participate in CS2AS bridges. However, there are limitations in the current implementation that need to be taken into account when the transformations are executed. For instance, a reference cannot be typed as *EObject*: the underlying scheduler – based on static analysis – of the M2M transformation engine complains when the domain of the model elements held via that reference cannot be determined.

- The current mappings section may support creating or referring to a collection of AS model elements. However, the implementation only traces one AS model element from one CS model element. Therefore, this kind of mapping would not actually be executed. That said, the limitation could be worked around by means of helpers: a helper wraps the creation of the required collection of AS model elements. In consequence, operation call expressions to these helpers are used, rather than trace expressions.

- The CS2AS-TL feature related to alias support is not currently implemented. Although the CS2AS-TL syntax admits the specification of aliases, the first compilation step responsible for producing the Complete OCL file with the name resolution functionality does not currently produce the required OCL code to support aliases.

- When integrating the generated transformation with the Xtext editor, the current prototype does not have a good error reporting in the end-user editor, whenever a problem has occurred during the transformation execution. Some errors related to name resolution are reported (e.g. if a target is not found). However, if the M2M transformation fails (e.g. a mistake was made in the CS2AS bridge), the end user does not have appropriate feedback on what is going on.

4.7 Summary

This chapter explained the proposed solution to address the different concerns introduced in Chapter 3. The solution includes the main contribution of this thesis: a CS2AS-TL with the aim of mapping concepts from source CS meta-models (that are automatically derived from grammar definitions) to target AS meta-models. In consequence, instances of CS2AS-TL constitute the CS2AS bridges required by CTMLs. Although the running example (i.e. mOCL) has been presented using Xtext excerpts to present the CS of the varied CS2AS scenarios, CS2AS-TL is independent from the front-end (language workbench or parser technology). On the contrary, the prototype implementation relies on EMF and the Eclipse QVTd project, and the AS needs to be defined by Ecore-based meta-models.

Section 4.2 explained CS2AS-TL, which is split in four main sections to address different concerns: helpers, disambiguation rules, name resolution and mappings. Sections 4.3 and 4.4 explained the compilation process to obtain executable M2M transformations. Section 4.5 focused on some integration features that showed how instances of CS2AS-TL can be used to complement the default Xtext-based editors. Finally, some limitations of the approach were presented in Section 4.6.

Chapter 5

Evaluation

Chapter 4 introduced a solution to address the different concerns explained in Chapter 3. This chapter focuses on evaluating the proposed solution from the following points of view. First, the chapter begins with a comparison between CS2AS-TL and general purpose M2M transformation languages. Second, a qualitative study is performed to compare the presented CS2AS-TL with two related works: Gra2Mol and Spoofax. Finally, a quantitative study is shown. This study pursues two goals: demonstrating the suitability of the proposed language to deal with different languages of varied nature (beyond OCL-like languages), and providing a quantitative comparison in terms of size of specification artefacts and performance (execution time) of the solution's implementation.

5.1 General Purpose M2M Transformation Languages

This section introduces a discussion about general purpose M2M transformation languages, such as ETL [51], ATL [44], RubyTL [17] or QVT [38]. The rationale behind including this discussion in the thesis is the following: the proposed solution is a domain specific M2M transformation language and, therefore, it should provide some benefits when compared to specifying equivalent M2M transformations written in a GPL.

The following subsections discuss the four technical sections of CS2AS-TL and, in particular, whether these technical sections provide any benefit when compared to using general purpose M2M transformation languages.

5.1.1 Helpers Section

The helpers section defines helper operations on meta-classes. Apart from the differences in the concrete syntax, general purpose M2M transformation languages provide similar mechanisms for defining contextual operations. In this case, CS2AS-TL does not provide any significant benefit. Figure 5.1 shows side by side excerpts of a helper operation definition, declared with CS2AS-TL (on the left) and ETL (on the right).

5.1.2 Mappings Section

The mappings section is designed to express how model elements of the output domain are created from the model elements of the input domain. Mapping (or rule) definitions are a key concept present in any M2M transformation language. Again, apart from the

```
Class {
  conformsTo(another : Class) : Boolean :=
    self = another or
    superClasses−>exists(conformsTo(
        another));
}
```

```
operation as!Class conformsTo(
    another : as!Class) : Boolean {
  return self = another or
  self.superClasses.exists(c | c.
      conformsTo(another));
}
```

FIGURE 5.1: Two equivalent helper operations in CS2AS-TL (left) and ETL (right)

differences in syntax, there are no apparent benefits when declaring mappings using CS2AS-TL. Figure 5.2 shows side by side excerpts of a mapping definition, declared with CS2AS-TL (on the left) and ATL (on the right).

```
create Package from PackageCS {
  name := name;
  ownedClasses := classes;
  ownedPackages := packages;
}
```

```
rule PackageCS2Package {
  from
    s : cs!PackageCS
  to
    t : as!Package (
        name <− s.name,
        ownedClasses <− s.classes,
        ownedPackages <− s.packages
    )
}
```

FIGURE 5.2: Two equivalent mapping definitions in CS2AS-TL (left) and ATL (right)

5.1.3 Disambiguation Section

The disambiguation section is designed to define the disambiguation rules (conditions) that drive multi-way mappings execution. According to the proposed solution, only one mapping can be executed on a single model element, and the mapping guards are not mandated to be exclusive. As such, the order in which the disambiguation rules are declared in the section is important.

On the one hand, this disambiguation section does not comprise more than the mapping guards that any general purpose M2M transformation language has. On the other hand, the guards are declared all together, away from the mapping definition. In this way, the order of mapping declarations is irrelevant, whereas the order in which disambiguation rules are declared is not so. That said, there is no evidence that this section provides any apparent benefits to the M2M transformation writer. Figure 5.3 shows side by side excerpts of some multi-way mapping definitions and the corresponding disambiguation rules, declared with CS2AS-TL (on the left) and RubyTL (on the right).

```
mappings {
  create CollectionItem from
      CollectionLiteralPartCS
  when withoutLastExpression {
    ownedItem := first;
    type := trace.ownedItem.type;
  }
  create CollectionRange from
      CollectionLiteralPartCS
  when withLastExpression {
    ownedFirst := first ;
    ownedLast := last;
    type := trace.ownedFirst.type;
  }
}
disambiguation {
  CollectionLiteralPartCS {
    withoutLastExpression := last =
        null;
    withLastExpression := last <>
        null;
  }
}
```

```
rule 'CollLitPartCS2CollItem' do
  from cs::CollectionLiteralPartCS
  to as::CollectionItem
   filter do |collPart|
     collPart.last == null
  end
  mapping do |collPart, collItem|
    collItem.ownedItem = collPart.first
    collItem.type = collItem.ownedItem
        .type
  end
end
rule 'CollLitPartCS2CollRange' do
  from cs::CollectionLiteralPartCS
  to as::CollectionRange
   filter do |collPart|
     collPart.last != null
  end
  mapping do |collPart, collRange|
    collRange.ownedFirst = collPart.
        first
    collRange.ownedLast = collPart.last
    collRange.type = collRange.
        ownedFirst.type
  end
end
```

FIGURE 5.3: Two equivalent mapping definitions in CS2AS-TL (left) and RubyTL (right)

5.1.4 Name Resolution Section

The name resolution section is designed to specify declaratively how name-resolution based lookups are performed. The specific constructs that support these declarations raise the level of abstraction compared to general purpose languages: in this case, by encoding some lookup algorithms to locate named elements throughout the AS model. The name resolution section provides a significant benefit compared to describing the equivalent name-based lookup algorithms in a general purpose M2M transformation language. Listing 5.1 shows an example of the name resolution description required to locate let expression variables, so that they can be referred by any variable expression that is inside the let expression. Listing 5.2 shows the corresponding description written in QVTo.

```
name_resolution {
  targets {
    NamedElement using name;
    Variable;
  }
  inputs {
    PathElementCS using elementName;
  }
  providers {
    LetExp {
      in current_scope
        for all excepting ownedVariable
          provides occluding ownedVariable;
    }
  }
}
```

LISTING 5.1: Example of name resolution described with CS2AS-TL

5.1.5 Conclusion

According to the proposed approach to declare complex CS2AS bridges, using a general purpose M2M transformation language is a plausible alternative: models conforming to an AS meta-model are obtained from models conforming to a CS meta-model. However, although the differences in syntax are cosmetic rather than a real benefit of CS2AS-TL, the domain-specific concern related to name resolution provides more adequate abstractions that let language engineers declare the concern in a more concise manner. In consequence, this feature provides a measurable benefit to CS2AS-TL when compared to general purpose M2M transformation languages.

```
query LookupEnvironment::nestedEnv() : LookEnvironment {
  return object LookupEnvironment {
      parentEnv := self;
  }
}
query Element::unqualified_env_Variable() : LookupEnvironment {
  return unqualified_env_Variable(null);
}
query Element::unqualified_env_Variable(in child : Element) :  LookupEnvironment {
  return parentEnv_Variable();
}
query Element::parentEnv_Variable() : LookupEnvironment {
  var parent :=  oclContainer();
  return if  parent = null
           then object LookupEnvironment { }
           else  parent.unqualified_env_Variable(self)
           endif;
}
query Element::lookupVariable(in env : LookupEnvironment, in vName : String) :
    OrderedSet(Variable) {
  var foundVariable := env.namedElements−>selectByKind(Variable)−>select(name =
       vName);
  return if  foundVariable−>isEmpty() and not (env.parentEnv = null)
           then lookupVariable(env.parentEnv, vName)
           else  foundVariable
           endif;
}
query Element::lookupUnqualifiedVariable(vName : String) : Variable {
  var foundVariable := lookupVariable(unqualified_env_Variable(), vName);
  return if  foundVariable−>isEmpty()
           then null
           else  foundVariable−>first()
           endif;
}
query Element::lookupVariable(in vName : String) : Variable {
  return lookupUnqualifiedVariable(vName);
}
query Element::lookupVariable(in aPathElementCS : PathElementCS) : Variable {
  return lookupUnqualifiedVariable(aPathElementCS.elementName);
}
query LetExp::unqualified_env_Variable(in child : Element) :  LookupEnvironment {
  return if  not (ownedVariable−>includes(child))
           then parentEnv_Variable().nestedEnv()
                   .addElements(ownedVariable)
           else  parentEnv_Variable()
           endif;
}
```

LISTING 5.2: QVTo query operations equivalent to those from Listing 5.1

5.2 Qualitative Study: Gra2Mol

This section comprises the first qualitative study. It compares Gra2Mol with the proposed solution. Gra2Mol was originally designed as a text-to-model tool for software modernisation. In this context, the goal is to create AS models conforming to a particular meta-model (e.g. Knowledge Discovery Metamodel (KDM) [36] meta-model[1]), from textual inputs that conform to the CS grammar of a language. Although the technology is tied to working with ANTLR grammars, the target is an arbitrary AS meta-model. Because of this, the tool provides a DSTL that deals with the concerns we have presented in this thesis.

When introducing the overall approach of the proposed solution (see Section 4.1.1), there was a discussion about how it relates to Gra2Mol (see Section 4.1.3). Now, this qualitative study focuses on DSTL features and capabilities.

5.2.1 Gra2Mol DSTL Introduction

A more detailed presentation of Gra2Mol can be found in Cánovas et al. [42]

This subsection is a brief introduction of the means that Gra2Mol offers to bridge the CS and AS of CTMLs. These consist of a DSTL that allows language engineers to bridge CS grammars to AS meta-models. In this case, the source CS grammars need to be specified in ANTLR, and the target AS meta-models need to be specified in *Ecore* (EMF). To specify these bridges, a set of declarative mapping rules are defined so that they map LHS non-terminals to AS meta-classes. The language provides a structure-shy (like Xpath [15]) query language to traverse the CST model. These queries are grouped within a sub-clause *queries* belonging to the rule, and they can be used in the sub-clause *mappings* that also belongs to the same rule. The sub-clause *mappings* contains a set of bindings for AS properties.

Listing 5.3 shows a Gra2Mol excerpt corresponding to the mOCL scenario that deals with *Package* definitions (see Section 3.3.1). A rule called *'mapPackage'* (line 1) is declared so that the LHS non-terminal *PackageCS* (line 2) is mapped to the meta-class *Package* (line 3). Then, two queries (lines 5–6) are declared using the mentioned query language. The '/' symbol is used for navigation, in this case, starting from the source of the rule (i.e. using the *pckg* variable). The results of these queries are used in the mappings sub-clause to compute the corresponding *ownedPackages* (line 8) and *ownedClasses* (line 9) properties of the target meta-class. Finally, the attribute name is initialised with the value of the terminal *ID*.

In terms of conciseness, the key of the Gra2Mol DSTL consists of navigation operators, such as '/' and '//', which allow for the traversing of all directly and indirectly contained model elements. This permits the navigation of CSTs, without specifying every navigation step from parent to children. In consequence, these operators remove the need for lengthy expressions to access model elements that are not located close to a particular CST node.

[1] KDM is an OMG standard for analysing existing software artefacts.

```
1  rule 'mapPackage'
2    from PackageCS pckg
3    to     miniocl::Package
4    queries
5      nestedPackages : /pckg/PackageCS;
6      classes         : /pckg/ClassCS;
7    mappings
8      ownedPackages = nestedPackages;
9      ownedClasses = classes;
10     name           = pckg.ID;
11 end_rule
```

LISTING 5.3: CS2AS definition for *PackageCS* using GraMol.

5.2.2 How Does Gra2Mol Address the Identified Concerns?

After the brief introduction to Gra2Mol DSTL, this subsection introduces how the different concerns presented in Chapter3 are addressed by the tool. In this way, more Gra2Mol features are introduced and its DSTL capabilities are shown.

Concern 1: Mapping an LHS non-terminal to a meta-class

Concern 1 is supported by Gra2Mol by means of rule definitions that declare an LHS non-terminal as the source, and a meta-class as the target.

With respect to mappings that create AS model elements, normal rule definitions are used. The previous Listing 5.3 showed an example where the *lhs* non-terminal *PackageCS* (line 2) is mapped to the meta-class *Package* (line 3).

With respect to mappings that do not create AS model elements, but refer to other model elements, Gra2Mol introduces special rules called *skip* rules. They receive this name because element creation should be skipped. Listing 5.4 shows how the mOCL call expression example (see Listing 4.43) is addressed by means of the Gra2Mol DSTL. In this case, the *CallExpCS* non-terminal (line 2) is mapped to the *CallExp* meta-class (line 3). The *skip* keyword (line 7) is used within the mappings sub-clause to declare which AS model element corresponds to the rule source. In this case, the corresponding AS model element is computed from the *next* query (line 7). This query retrieves the *as* model element corresponding to the directly contained *NavigationExpCS* CST node (line 5).

Concern 2: Mapping an RHS non-terminal to a reference

Concern 2 is supported by Gra2Mol by means of the bindings (declared in the mappings sub-clause) that assign the results of a query to a reference. Listing 5.3 showed an example where two containment references – *Package::ownedPackages* (line 8) and *Package::ownedClasses* (line 9) – are initialised from two query results. These results are the AS model elements corresponding to the RHS non-terminal *PackageCS* (line 5) and the RHS non-terminal *ClassCS* (line 6) respectively.

```
skip_rule 'mapCallExp'
  from CallExpCS ce
  to    miniocl::CallExp
  queries
    next : /ce/NavigationExpCS;
  mappings
    skip next;
end_rule
```

LISTING 5.4: CS2AS definition for *CallExpCS* using Gra2Mol

Concern 3: Mapping a terminal to an attribute

Concern 3 is supported by Gra2Mol by means of the bindings (declared in the mappings sub-clause) that assign a specific terminal value to an attribute. Listing 5.3 showed an example where a binding is defined for the *NamedElement::name* attribute (line 10). In this case, the attribute is initialised with the value of the terminal *ID* that belongs to the rule source (the *PackageCS* non-terminal).

Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes

Concern 4 is supported by Gra2Mol by means of *new* expressions that allow the instantiation of AS model elements – as the RHS of a binding – and by allowing the initialisation of the properties of these new AS model elements with an additional set of bindings. Listing 5.5 shows how the mOCL scenario that deals with operation definitions (see Section 3.4.5) is addressed using Gra2Mol.

```
rule 'mapOperation'
  from OperationCS op
  to    miniocl::Operation
  queries
    parameters : /op/ParmeterDeclarationCS;
    body       : /op/ExpCS;
  mappings
    name = op.ID;
    ownedParameters = parameters;
    ownedBodyExpression = new miniocl::ExpressionInOCL;
    ownedBodyExpression.ownedBody = body;
    ownedBodyExpression.ownedSelfVar = new miniocl::Variable;
    ownedBodyExpression.ownedSelfVar.name = 'self';
    ownedBodyExpression.ownedSelfVar.type = ext computeType();
end_rule
```

LISTING 5.5: CS2AS definition for *OperationCS* using Gra2Mol

The usage of the *ext* keyword (line 14) is clarified when explaining how Gra2Mol addresses Concern 7.

Concern 5: Mapping an LHS non-terminal to many meta-classes

Concern 5 is similarly supported by Gra2Mol by means of the already mentioned *new* expressions. Listing 5.6 shows how the mOCL scenario that deals with mOCL property call expressions (see Section 3.4.6) is addressed using Gra2Mol.

```
1  rule 'mapPropCallExpWithImplicitSource'
2    from NameExpCS { -- Check if NameExpCS maps to a propety call with implicit source
3                   } nExp
4    to PropertyCallExp
5    queries
6      refVar : ext lookup(miniocl::Variable, 'self');
7    mappings
8      ownedSource = new miniocl::VariableExp;
9      ownedSource.referredVar = refVar;
10     ownedSource.type = refVar.type;
11     referredProperty = ext lookupExported(miniocl::Property,refVar,nExp.expName);
12     type = ext computeType();
```

LISTING 5.6: CS2AS definition for *NameExpCS* using Gra2Mol

The usage of the curly brackets that follow the rule source *NameExpCS* (line 2) is clarified when explaining how Gra2Mol addresses Concern 6.

Concern 6: Multi-way mappings from LHS non-terminals to meta-classes

Concern 6 is supported by Gra2Mol by means of additional filters or guard conditions declared on the rule source. These conditions, which may be expressed as query expressions, are declared between curly brackets right after the name of the LHS non-terminal name. Listing 5.7 shows how the mOCL scenario that deals with multi-way mappings (see Section 3.4.7) is addressed using Gra2Mol.

Concern 7: Mapping properties from non-grammar terms

Concern 7 is supported by Gra2Mol by means of the bindings (declared in the mappings sub-clause) that assign value/s to properties. Arbitrary expressions can be used on the binding's RHS to provide the final value/s that are assigned to the binding's LHS property. Although the set of expressions is very limited, Gra2Mol provides an extension mechanism that allows the language designer to enhance CS2AS-TL, in this case, by introducing *extension* expressions that invoke custom behaviour in the form of parametrisable functions (i.e. an external black-box). When the Gra2Mol transformation is executed, an *extension* expression corresponds to the invocation of an *execute()* method of a Java class responsible for

```
rule 'mapPropCallExpWithImplicitSource'
  from NameExpCS { -- Check if NameExpCS maps to a propety call with implicit source
                 } nExp
  to PropertyCallExp
  queries -- queries ommitted
  mappings -- mappings omitted
end_rule
rule 'mapPropCallExpWithExplicitSource'
  from NameExpCS { -- Check if NameExpCS maps to a propety call with explicit source
                  } nExp
  to PropertyCallExp
  queries -- queries ommitted
  mappings -- mappings omitted
end_rule
rule 'mapVariableExp'
from NameExpCS { -- Check if NameExpCS maps to a variable expression
                } nExp
  to VariableExp
  queries -- queries ommitted
  mappings -- mappings omitted
end_rule
```

LISTING 5.7: Multi-way mapping definition for *NameExpCS* using Gra2Mol

executing the required operational behaviour. These *extension* expressions consist of the keyword **ext**, followed by the signature (name and arguments) that identifies the parametrisable function to invoke.

💡 More details about the extension mechanism can be found in Cánovas et al. [42]

Listing 5.5 shows an example in which the extension mechanism (line 14) is used for computing the type of the created implicit *self* variable.

Concern 8: Name resolution

Concern 8 is not directly supported by Gra2Mol. In principle, any lookup activity is described via query expressions that are performed on the CST. However, declaring more complex scenarios to support nested scopes, qualified name lookups, lookups in external models etc. require the use of the language extension mechanism.

5.2.3 Discussion

This subsection provides a comparative discussion between Gra2Mol and CS2AS-TL.

Parsing Technology Dependency

Gra2Mol create bridges between CS grammars and AS meta-models. Although, conceptually, the DSTL simply maps grammar terms (e.g. terminals and non-terminals), the implementation only works with ANTLR grammars. Therefore, the tool is dependent on a particular parser technology.

CS2AS-TL creates bridges between CS and AS meta-models. Therefore, it is parser-technology agnostic. It may be used with classic parser generators (e.g. ANTLR, LPG) or modern language workbenches (Xtext, Spoofax). The main restriction of the approach is that the output of the parser must be a model conforming to a meta-model (the CS one).

Modeling Technology Dependency

Both Gra2Mol and the presented prototype depend on the Eclipse Modeling Framework (EMF). Although, conceptually, both approaches simply map meta-model terms (e.g. meta-classes and properties), the corresponding implementations only work with Ecore meta-models. Therefore, the tools are dependent on a particular modeling technology.

Language Nature

The DSTL of Gra2Mol and CS2AS-TL are declarative. The declarations consist of mappings between CS and AS terms, and there is no imperative exposition of how the transformation is executed. On the one hand, CS2AS declarations tend to be concise. On the other hand, declarative languages require well designed execution engines; otherwise, they may negatively impact performance during the transformation execution.

💡 A quantitative study about performance is presented in Section 5.5.

Query Language

Gra2Mol is based on a tailored structure-shy XPath-like query language, whereas CS2AS-TL is based on the statically typed OCL.

On the one hand, the Gra2Mol query language is less verbose and more concise than OCL; thus, Gra2Mol DSTL instances tend to be smaller. In particular, the difference is significant when long OCL expressions are required to look up model elements that are not located near the contextual element that performs the lookup. For instance, Cánovas et al. [42] show an example that compares two equivalent expressions between the Gra2Mol query language and OCL. Figure 5.4 shows both excerpts side by side.

However, long expressions that traverse models are not usually required with the proposed solution. These long expressions are required to find model elements that are distantly located (somewhere else in the model). For instance, the example exposed by Cánovas et al. [42] is used for looking up *VariableDeclaration*s. This concern has been abstracted away in CS2AS-TL by means of the declarative name resolution section: providers declare which targets they contribute without knowing where the actual consumer is located in the

```
/program//#varDeclaration
```

```
Locals(p : program) : Sequence(varDeclaration)
post result =
  if (p.pgramBlock.block.declSection =
       oclIsUndefined())
  then Sequence {}
  else p.programBlock.block.declSection−>
    select (pd | ds.oclIsKindOf(
         procFuncDeclaration))−>
    collect (e | e.block.declSection)−>flatten()
         −>
    select (vd | ds.oclIsKindOf(varDeclaration))
  endif
```

FIGURE 5.4: Two equivalent queries in Gra2Mol (left) and OCL (right) (Figure 6 from [42])

model. Likewise, consumers do not declare where the targets that they need are located. They just provide the information required to match the targets (lookup criteria).

Listing 5.8 shows how that particular scenario is declared with CS2AS-TL. According to the example, *VariableDeclaration* model elements are contained by *Block* model elements. The latter (line 2) provides the former (line 4) for all its children in the current scope (line 3).

```
1 providers {
2   Block {
3     in current_scope
4       provides occluding subStatements−>selectByKind(VariableDeclaration);
5   }
6 }
```

LISTING 5.8: Name resolution declaration: *Block*s contribute *VariableDeclaration*s to the current scope

A quantitative study about the size of artefacts is presented in Section 5.6.

On the other hand, built-in navigation operators of the Gra2Mol query language are based on accessing a model element's children. This means that, whenever the model elements to access are not contained by a given CS element, the query has to declare deep navigations from the root model element. This may lead to performance penalties. For instance, in a case where a query involves access to the parent of the rule source, a deep model traversal is not as fast as a simple *oclContainer()* call. This Gra2Mol shortcoming can be mitigated by means of their language extension mechanism (further discussed).

Another difference to highlight is that the Gra2Mol query language is designed to work strictly on *CST*s. Navigating AS models (graphs) rather than CST ones (trees) may lead to less expensive navigations to retrieve some particular AS information (e.g. querying the type – a cross-reference – of a particular mOCL expression). More importantly, focusing on CS navigations prevents CS2AS transformations from working with external AS models (e.g. a library model with no CS).

That said, many of the Gra2Mol query language shortcomings may be alleviated by their powerful language extension mechanism, at the cost of having to produce the Java implementation required to support any additional enhancement.

Name resolution

Name-based cross-references are declared in Gra2Mol as in any other model query. These queries are described as direct searches that take into account where the target element is located in the model. The required queries get significantly complicated when simulating lookup scopes. In fact, they require the use of the language extension mechanism to solve them.

In CS2AS-TL, there is a dedicated declarative name resolution description that isolates this particular concern in its own section. Lookup scopes can be declared in the language and there is no elaboration about how to find a particular target. Instead, the focus is on how they are contributed to the lookup scopes. These contributions are usually easy to identify because the general case is that a named element is contributed to the lookup scope by its owner (e.g. properties and operations are contributed by the owning class. Classes are contributed by the owning package. A let variable is contributed by the owning let expression etc.). For abnormal cases, more complex OCL expressions can be used to configure the scope contributions.

Disambiguation

CS disambiguation scenarios can be similarly described with Gra2Mol by means of rule filtering expressions. These expressions are applied to different rules that work on the same grammar term. The filtering expressions act as traditional M2M transformation mapping guards.

Rule Inheritance

Gra2Mol provides a basic rule inheritance mechanism, so that specific rules that declare queries and mappings can be reused by other rules. These are called *mixin* rules, and the inheriting rules declare the name of the *mixin* rules they inherit. The only drawback of their mechanism is that the source of the inheriting and the *mixin* rules need to be the same. Listing 5.9 shows an example.

CS2AS-TL does not currently support rule inheritance. Despite offering a reusing mechanism by means of the helpers section, all the property assignment declarations have to be currently specified. Chapter 6 introduces some future work related to this topic.

```
rule 'myNormalRule'
  from declSection dec
  to Declaration
  mixin myMixinRule
  queries
    ...
  mappings
    ...
end_rule

```

```
mixin_rule 'myMixinRule'
  from declSection dec
  queries
    q1: /dec//#varDeclaration;
  mappings
    vars = q1;
end_rule
```

LISTING 5.9: *Mixin* rule example (Fig. 11 from [42])

Language Extension Mechanism

Despite the limited amount of expressions that can be used in Gra2Mol declarations, the language provides a flexible extension mechanism that cures this limitation. In essence, any arbitrary computation or domain-specific navigation (e.g. to increase performance) can be achieved by creating an extension to the language. The extension mechanism consists of the ability to contribute a Java class that encapsulates the particular behaviour of the extension, and specific syntax to bind the CS2AS declaration with the contributed Java class. Figure 5.5 shows an example of how to use the extension mechanism.

```
rule 'extensionExample'
  from varDeclaration vd
  to ValuedElement
  queries
  mappings
    value = ext toUpperCase(
        vd.VALUE);
end_value
```

```
public class UpperCaseExtension extends
    MAppingExtension {
  public ExtensionValueReturn execute() {
    String value = (String) getParam(0);
    return value.toUpperCase();
  }
  public String[] getKeywords() {
    return new String[] { "toUpperCase" };
  }
}
```

FIGURE 5.5: Extension mechanism in Gra2Mol (Fig. 16 from [42])

CS2AS-TL does not currently provide any extension mechanism. As long as all the required information to produce AS models is in the CS model, the expressiveness of CS2AS-TL (in particular OCL expressions) is sufficient. Otherwise, the particular CS2AS scenario cannot be supported using the proposed solution.

Specification Generation

As stated in Section 1.2.1, one of the interests of the industrial partner is having the means to model all these CS2AS concerns, so that parts of the OMG specification (e.g. Clause 9.3 from [39]) can be automatically generated rather than manually written and maintained. CS2AS bridges – conforming to the CS2AS-TL meta-model – constitute these desired models, and further MDE techniques can be used to generate parts of the OMG specifications.

Gra2Mol DSTL is also based on EMF. However, since different CS2AS scenarios need to be addressed by means of its language extension mechanism (i.e. complementary Java code is required), having CS2AS bridges written in Gra2Mol DSTL is less convenient for generating parts of the language specification.

Gra2Mol DSTL is not an "OMG-like" language, which may limit its adoption by OMG specification producers and maintainers.

5.3 Qualitative Study: Spoofax

The second qualitative study compares the proposed solution with Spoofax. Spoofax is a language workbench for textual languages. Although it was not originally designed to work on modeling languages, there is previous work [63] that demonstrates that this use case is possible within Spoofax. However, the support to modeling languages is discontinued[2]. This qualitative study shows how Spoofax can address the identified concerns explained in Chapter 3 and contains a comparative discussion on Spoofax's features and capabilities.

5.3.1 Spoofax Introduction

For a more detailed presentation of Spoofax, the reader is referred to its main source [46], website [72] and the work related to modeling languages [63].

This subsection presents a brief introduction about how the CS and AS of CTMLs can be bridged in Spoofax. This language workbench creates abstract syntax representations of a textual input using its own AST representation. Therefore, specifying CS2AS bridges within Spoofax does not involve a shift to the modelware technological space. However, the tool provides additional facilities to produce models that conform to an AS meta-model.

In order to explain how the CS2AS specification is done within Spoofax, a very simple example is introduced for convenience. The rationale of introducing this different example is having a minimal (smaller and simpler than mOCL) but complete and self-contained example which shows the Spoofax languages, the support to modeling languages and the language editor. Figure 5.6 shows the CS (Spoofax-based) grammar of a simple *Families* language (on the left) and the corresponding AS (Ecore-based) meta-model (on the right). From the point of view of the CS, a family consists of an identifier to define its name, preceded by the keyword *family* (line 8). Then, the name of the family mother and father is also declared with an identifier, respectively preceded by the keywords *mum* and *dad* (lines 9–10). Finally, an arbitrary number of children can be defined for the family (line 11). Each children declaration consists of an identifier preceded by the keyword *child* (line 13).

Figure 5.7 shows an example of the introduced modeling language. On the left hand side, the family *SBB* is defined with a *mum*, a *dad* and two *children*. On the right hand side, we can visualise the internal tree-based representation of the parsed textual input. In this case,

[2]The last Spoofax version in which you can produce EMF-based models is 1.5.0

```
module Families
imports
  Common
context−free start−symbols
  Start
context−free syntax
  Start . FamilyCS = <
    family <ID> {
      mum <ID>
      dad <ID>
      <ChildrenCS∗>
    } >
  ChildrenCS.ChildrenCS = <child <ID> >
```

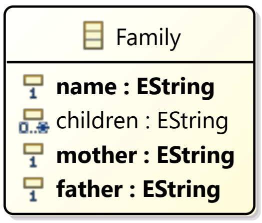
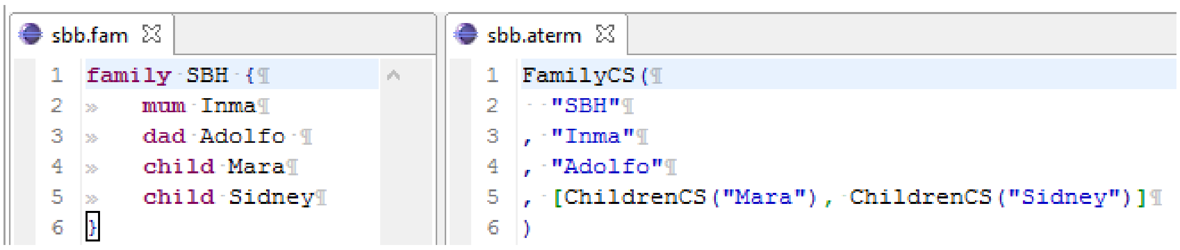


FIGURE 5.6: Left: CS grammar definition of a simple Families language. Right: the corresponding AS meta-model

.

the syntax tree node (also known as term) *FamilyCS* consists of four subterms: three *String* values corresponding to the grammar terminal *ID*, and a list – between square brackets – of *ChildrenCS* tree nodes. Each *ChildrenCS* term comprises another *String* value.



```
family SBH {
    mum Inma
    dad Adolfo
    child Mara
    child Sidney
}
```

```
FamilyCS(
  "SBH"
, "Inma"
, "Adolfo"
, [ChildrenCS("Mara"), ChildrenCS("Sidney")]
)
```

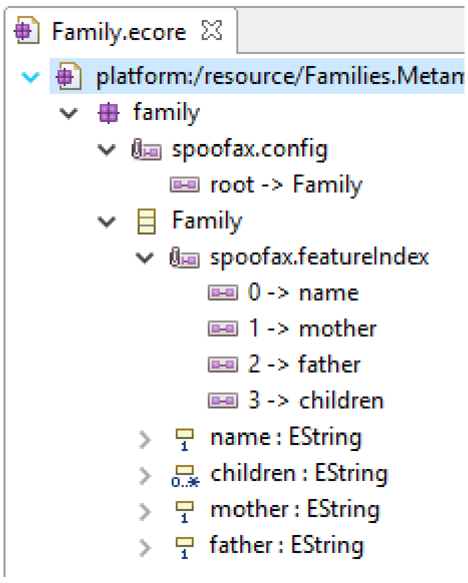
FIGURE 5.7: Textual instance of the *Families* language and the corresponding internal AST representation

.

This internal tree-based representation needs to be further transformed into a model that conforms to the AS meta-model that was shown in Figure 5.6. This means that a mapping between the concepts of the CS grammar and the concepts of the AS meta-model has to be declared. To achieve this in Spoofax, two different tasks need to be performed. Firstly, a Stratego/XT [75] transformation is required to refine the obtained syntax trees so that there is an 1-to-1 mapping between the syntax tree and the final AS model. The name of the syntax tree nodes (terms) after the transformation needs to coincide with the name of the target AS meta-classes. Secondly, a modification of the AS meta-model is required. The problem is that Spoofax tree-form representations do not include names that relate a term with the (children) subterms. Therefore, in order to specify how the different subterms of a term are mapped to a specific property of a meta-class, some *EAnnotation*s are added to the target meta-model. In this way, every meta-class specifies how its properties are mapped from a specific subterm position (or index).

To clarify these two tasks, Figure 5.8 shows the required specification artefacts for the *Families* example. On the left hand side, a Stratego-based definition declares the CS2AS bridge[3]. The relevant part consists of two *CS2AS* rules (lines 10–15). The first rewrites the *FamilyCS* term (line 12) to a new one called *Family* (line 13). The second rule is required for removing the *ChildrenCS* term from the syntax tree. The reason is that in the *Family* meta-model, there is no intermediate *Child* meta-class. Instead, the children names are held as *String* values in *Family* model elements. Therefore, every *ChildrenCS* term is rewritten as the *String*-valued name that it holds. On the right hand side of Figure 5.8, the modified *Family* meta-model is shown with additional *EAnnotation*s. Firstly, an *EAnnotation* on the *EPackage* specifies that *Family* is the meta-class of the root *AS* model elements. Secondly, for the *Family* meta-class, the corresponding *EAnnotation* declares the mapping derived from syntax tree subterm positions (0, 1, 2 and 3) to each meta-class property (name, mother, father, children).

```
1   module cs2as
2
3   signature
4     constructors
5       Family : ID ∗ ID ∗ ID ∗ LIST(ID) −>
            Family
6
7   imports
8     include/Families
9
10  rules
11    CS2AS:
12      FamilyCS(name, mum, dad, children)
13        −> Family(name, mum, dad, children)
14    CS2AS:
15      ChildrenCS(name) −> name
```



```
Family.ecore ☒
⌄ platform:/resource/Families.Metan
    ⌄ family
        ⌄ spoofax.config
            root -> Family
        ⌄ Family
            ⌄ spoofax.featureIndex
                0 -> name
                1 -> mother
                2 -> father
                3 -> children
            > name : EString
            > children : EString
            > mother : EString
            > father : EString
```

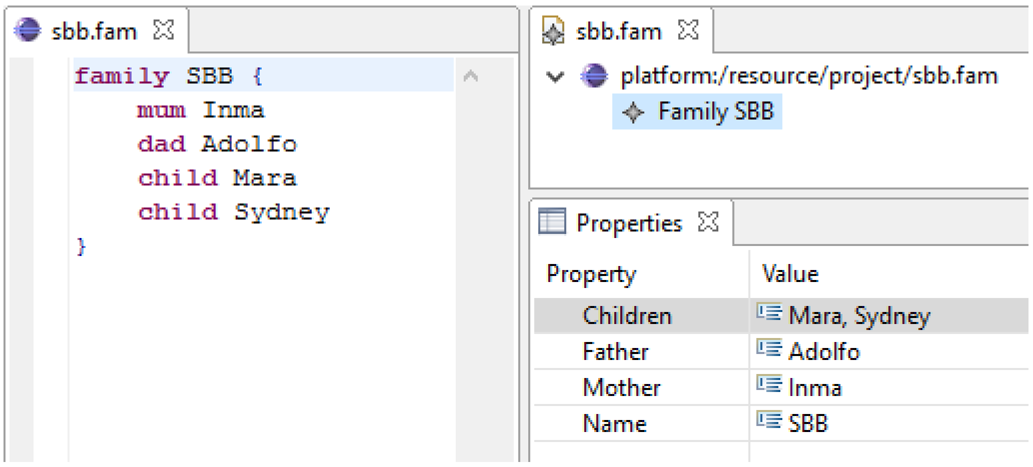
FIGURE 5.8: Left: CS grammar definition of a simple Families language. Right: the corresponding AS meta-model

To conclude how this *Family* example is supported via Spoofax, Figure 5.9 shows the AS model (on the right) corresponding to a textual instance of the *Family* language (on the left).

5.3.2 How Does Spoofax Address the Identified Concerns?

Following the introduction to Spoofax, this subsection explains how the different concerns explained in Chapter 3 are addressed by the tool.

[3] Note that the CS2AS bridge is specified in the *treeware* technological space.



```
family SBB {
    mum Inma
    dad Adolfo
    child Mara
    child Sydney
}
```

Property	Value
Children	Mara, Sydney
Father	Adolfo
Mother	Inma
Name	SBB

FIGURE 5.9: Left: instance of the *Family* textual modeling language. Right: the corresponding AS model, including the properties of the *SBB* family

The CS2AS bridge is performed within Spoofax by bridging the *grammarware* and *treeware* technological spaces. Therefore, the concepts of meta-class and attribute do not exist in Spoofax. The following sub-subsections show, by means of examples, how the different CS2AS scenarios can be addressed by Spoofax.

Concern 1: Mapping an LHS non-terminal to a meta-class

Concern 1 is supported by Spoofax by means of its Stratego-based rule definitions that declare – on the left hand side – the constructor corresponding to the LHS non-terminal and – on the right hand side – the new constructor corresponding to the meta-class.

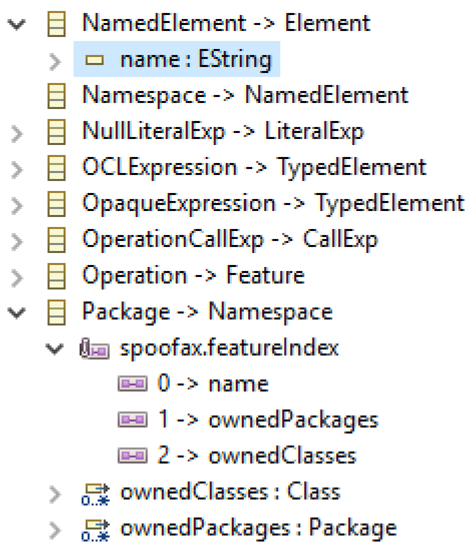
```
signature
  constructors
     Package : ID * List(Package) * List(Class) * -> Package

rules
  CS2AS:
    PackageCS(name, subPackages, classes) ->
      Package(name, subPackages, classes)
```

LISTING 5.10: CS2AS definition for *PackageCS* using Spoofax

Concern 2: Mapping an RHS non-terminal to a reference

Concern 2 is supported by Spoofax by adding annotations to the meta-model. A grammar RHS non-terminal corresponds to a subterm of the term corresponding to a grammar LHS terminal. The practical solution consists of specifying, in the form of meta-model annotations, a mapping from the position (or index) of the subterm to the name of the corresponding AS reference. Figure 5.10 shows an example, in which the references *ownedPackages* and *ownedClasses* are mapped from the subterm indexes 1 and 2. Note that in Listing 5.10, a *Package* term (line 3) consists of an *ID* (for the name), a list of *Package* subterms (for the owned packages) and a list of *Class* subterms (for the owned classes).



```
NamedElement -> Element
    name : EString
Namespace -> NamedElement
NullLiteralExp -> LiteralExp
OCLExpression -> TypedElement
OpaqueExpression -> TypedElement
OperationCallExp -> CallExp
Operation -> Feature
Package -> Namespace
    spoofax.featureIndex
        0 -> name
        1 -> ownedPackages
        2 -> ownedClasses
    ownedClasses : Class
    ownedPackages : Package
```

FIGURE 5.10: Mapping the position/index of a subterm (corresponding to an RHS non-terminal) to the name of a reference

The indexes of subterms within a term start on 0.

Concern 3: Mapping a terminal to an attribute

Concern 3 is supported by Spoofax by adding annotations to the meta-model. In Figure 5.10, the attribute *name* is mapped from the subterm index 0. Note that the attribute *name* belongs to an abstract super meta-class (*NamedElement*). However, the mapping needs to be specified in the specific meta-class because the index of the subterm is not necessarily the same for every meta-class that requires a mapping for its attributes.

Concern 4: Mapping an RHS non-terminal to a reference and additional meta-classes

Concern 4 is supported by Spoofax by means of Stratego rule definitions that can include additional constructor invocations. These constructors must have the same name as the additional meta-classes. Listing 5.11 shows an example, in which some arguments of the constructors are new constructor invocations, such as *ExpressionInOCL* (line 9) and *Variable* (line 10).

```
1  signature
2    constructors
3      Operation : ID * List(Parameter) * ExpressionInOCL * Type -> Operation
4      ExpressionInOCL : OCLExpression * Variable           -> ExpressionInOCL
5      Variable   : ID * Type                               -> Variable
6  rules
```

```
    CS2AS :
      OperationCS(name, parameters, resultType, body) −>
        Operation(name, parameters, ExpressionInOCL(body,
          Variable (' self ',  computeContext())), computeType(resultType))
```

LISTING 5.11: CS2AS definition for *OperationBody* using Spoofax

Concern 5: Mapping an LHS non-terminal to many meta-classes

Similarly, Concern 5 is supported by Spoofax by means of Stratego rule definitions. Listing 5.12 shows how the scenario is addressed. In this case, when the *NameExpCS* term (line 7) is rewritten into a *PropertyCallExp* term, an additional *VariableExp* subterm is created.

```
1 signature
2   constructors
3     PropertyCallExp : OCLExpression ∗ Property −> PropertyCallExp
4     VariableExp :  Variable                    −> VariableExp
5 rules
6   CS2AS:
7     NameExpCS(name, _ ) −>
8       PropertyCallExp(VariableExp(lookupVariable('self')), lookupProperty(name)))
9     where // Check  if  NameExpCS maps to a propety call  with  implicit  source
```

LISTING 5.12: CS2AS definition for *NameExpCS* using Spoofax

Note that the *where* keyword (line 9) is related to the multi-way mappings concern explained in the following sub-subsection.

Concern 6: Multi-way mappings from LHS non-terminals to meta-classes

Concern 6 is supported by Spoofax by means of conditional rewrite rules, so that the term rewrite occurs as long as a condition holds. This condition follows the rule definition, along with a preceding *where* keyword. Listing 5.13 shows how the mOCL scenario that deals with multi-way mappings (see Section 3.4.7) can be addressed using Spoofax.

Concern 7: Mapping properties from non-grammar terms

Concern 7 is supported within Spoofax by using the Stratego language. When defining the constructors of every term, additional subterms can be declared to hold the value of additional properties. As occurs with Concerns 2 and 3, these additional subterms are mapped to the corresponding AS property by means of the *EAnnotation* mechanism described before.

Concern 8: Name resolution

With respect to name resolution, Spoofax provides its own language, called Names Binding Language (NaBL) [53], to specify declaratively how name resolution is performed throughout its internal AST representation. Instances of this NaBL language are further transformed

```
signature
  constructors
    // Constructors ommmitted
rules
  CS2AS:
     NameExpCS(name, _) −>
       PropertyCallExp ( /∗ constructor call ommitted ∗/ )
     where // Check if NameExpCS maps to a propety call with implicit source
  CS2AS:
     NameExpCS(name, _) −>
       PropertyCallExp ( /∗ constructor call ommitted ∗/ )
     where // Check if NameExpCS maps to a propety call with explicit source
  CS2AS:
     NameExpCS(name, _) −>
       VariableExp ( /∗ constructor call ommitted ∗/ )
     where // Check if NameExpCS maps to a variable expression
```

LISTING 5.13: CS2AS definition for *NameExpCS* using Spoofax

into Stratego transformations, which are responsible for relating the syntax tree terms of the transformed ASTs. The language is based on the concept of site definition (a term that can be referred by other terms) and site references (a term that can refer to other terms). Advanced name resolution concepts, such as scopes, imports etc. are supported by NaBL. Listing 5.14 shows the site definition of mOCL *Variables* when using the NaBL language.

```
namespaces variable
rules
   Variable(varName, _, _):
      defines variable varName
```

LISTING 5.14: Using NaBL to declare a site definition for mOCL *Variables*

⚠ Most of the name resolution concerns introduced in Section 3.5 are supported by NaBL. The following subsection discusses the particular features that are not supported.

5.3.3 Discussion

Previous subsections introduced Spoofax, including a brief description about how it deals with the concerns identified in Chapter 3. This subsection compares Spoofax and the proposed solution.

Parsing Technology Dependency

Spoofax is tied to its own parsing technology. It uses a Java library called JSGLR [23] that interprets the parse tables that are generated from its Syntax Definition Formalism (SDF) [47] grammars. On the contrary, as explained during the qualitative study above, CS2AS-TL is not dependent on a particular parsing technology.

Modeling Technology Dependency

Spoofax presents a practical solution to support textual modeling languages. The solution is tied to the Eclipse Modeling Framework (EMF). Therefore, Spoofax – like CS2AS-TL – is dependent on a particular modeling technology.

Language Nature

The Spoofax languages that are required to support complex textual languages (i.e. SDF, Stratego and NaBL) are declarative.

Query Language

Spoofax does not provide a query language per se. On the one hand, when declaring Stratego rules, the left hand side (that declares the type of term on which the rule applies) mandates to declare as many named variables as the subterms the particular term contains. These named variables can be used to access (navigate towards) each immediate subterm. On the other hand, for more complex navigations, the end user could declare additional reusable rules that act as query operations. Listing 5.15 shows an example.

```
signature
  constructors
    Variable   : ID * Type -> Variable
rules
  getVariableType:
    Variable(name, type) -> type
```

LISTING 5.15: Declaring Stratego rules to create navigation/query operations

Name resolution

With respect to name resolution, Spoofax provides a specific DSL to declare name resolution. Its DSL capabilities let Spoofax address most of the concerns explained in Section 3.5. This discussion focuses on the limitations found within Spoofax:

- The main difference with respect to Spoofax is that name resolution is performed in two different technical spaces. On the one hand, Spoofax name resolution operates in the *treeware* technological space, where the syntax trees are obtained from the parser. When performing name resolution, some syntax tree terms are rewritten to create additional subterms that resolve to another term located somewhere else in the syntax tree. In this way, all the information that is looked up needs to be in the syntax tree obtained from the parser. On the other hand, the approach proposed in this thesis operates in the *modelware* technological space. In particular, name resolution is performed on the AS models that are created during the CS2AS model transformation execution. This characteristic supports name resolution on AS models that do not even require to have been realised during a CS2AS transformation, as occurs with external models (see below).

- When working on textual modeling languages, the main concern that cannot be supported by Spoofax is lookup on external models (see Section 3.5.8). This concern is critical when supporting languages such as OCL because some AS model elements corresponding to expressions must refer to external models. For instance, *PropertyCallExp* model elements must refer to *Property* model elements that are normally located in an external model. The issue originates in the Spoofax approach, where the CS2AS gap is entirely bridged in the *treeware* technological space. This approach leaves the *modelware* shift to the very end, where name resolution has already taken place. Therefore, performing name resolution on external AS models is not supported.

Disambiguation

With respect to CS disambiguation, Stratego/XT [75] supports the so-called conditional rewrite rules that declaratively allow the disambiguation of ambiguous CS terms. In this way, these ambiguous terms are rewritten as the appropriate AS term.

Type resolution

One of the additions that Spoofax provides – and CS2AS-TL does not offer – is a DSL to declare type resolution. For those textual languages that offer a type system (such as OCL), Spoofax allows to specify declaratively the type of any term that requires a type declaration (e.g. expressions).

In CS2AS-TL, a type system can be declared, but it requires to be fully defined by means of OCL expressions. When discussing future work, Section 6.1 provides additional comments about this feature.

Specification Generation

Spoofax languages are not models [4] per se. However, they provide their own code generation techniques (e.g. using Stratego/XT), which could be used to produce parts of the OMG specification from the CS2AS descriptions.

⚠ Spoofax languages are not OMG-like languages. This fact may stall the adoption of Spoofax languages by OMG specification producers and maintainers.

[4] As the thesis scope presented, there is no underlying meta-model.

5.4 Quantitative Study: Introduction

The previous sections evaluated existing work in terms of their capabilities and how they address the different concerns explained in Chapter 3. This section focuses on a quantitative study that consists of evaluating the proposed solution in terms of measurable characteristics.

5.4.1 Goals

In order to evaluate quantitatively the contributions of this thesis, the following studies are presented with two goals in mind:

- Firstly, in the context of this research, to provide evidence of the generalisability of the proposed solution. In other words, to demonstrate that the presented solution can address varied CS2AS scenarios beyond the running example (mOCL).
- Secondly, by means of the empirical research method, to compare quantitatively the proposed solution's implementation to related work.

5.4.2 Metrics

This subsection presents the different metrics that are used for the quantitative study, including why they have been chosen and the instrumentation required to obtain the corresponding measurements.

Performance of the Implementation

The first metric to be used during the evaluation is performance, in terms of execution time, and in particular, the number of milliseconds that the implementation needs for obtaining AS models from CS models. Note that CS2AS-TL is agnostic of parsing technology. Therefore, any performance related to parsing activity has been deliberately excluded from the evaluation.

The instrumentation consists of using a Java system library to obtain a time stamp before the CS2AS transformation starts, and the time stamp right after the transformations ends.

Size of DSTL instances

The main contribution of this thesis is a CS2AS-TL to express bridges between the CS and the AS of textual modeling languages. When conducting a comparative study, an objective metric to evaluate CS2AS-TL consists of measuring the size of the instances of CS2AS bridges. There are different ways to measure this, including the size of the file and, taking into account that CS2AS-TL is textual, the number of Lines of Code (LoC) [64]. However, these metrics are subject to the length of identifiers, formatting style etc. Because of this, the main metric of interest is based on the number of words of CS2AS-TL instances, excluding those belonging to comments. The rationale for this is the following:

- Keywords and identifiers are the usual constructs that the end user has to write.
- Concerns about the length of identifiers to obtain better results are removed.

When showing the obtained results of the experiments related to the size of CS2AS-TL instances, measurements of the file size, the number of LoC and number of words are included. However, only the number of words is considered when performing hypothesis testing.

To measure the number of words, specific instrumentation[5] has been created. In this case, it consists of a Java class that processes input files and returns the number of comprising words (excluding comments).

Why these metrics?

This sub-subsection introduces the rationale behind using the proposed metrics.

With respect to the implementation performance, the main reason for choosing this metric is the fact that they provide an objective criterion according to which the evaluation is performed in an automated and agile way. Additionally, although there are other ways to evaluate implementation performance, such as memory footprint, execution time has been prioritised because the tools that are created in this context are designed to run in IDEs. Therefore, the time response (how much time the user has to wait to obtain the AS after textual editing happens) is a priority over how much memory is consumed by the IDE.

With respect to the size of CS2AS-TL instances, the same rationale applies. Doing experiments in an automated and efficient way is important because the solution is designed to deal with CTMLs in a broad sense, and the prototype has been subject to continuous improvement. This makes user-oriented experiments impractical: although there are other ways to evaluate the CS2AS-TL – such as usability and readability – this kind of evaluation requires designing and running expensive experiments with end users (not viable within the time frame set for this project. This concern is discussed in Section 6.1).

5.4.3 Subjects

Previous subsections focused on a qualitative study to compare relevant work: Gra2Mol and Spoofax. In the quantitative study, CS2AS-TL is one of the subjects of study. The other subject of study is Gra2Mol, whereas Spoofax has been discarded. The rationale behind this decision is the following:

- Although Spoofax is proven to give support to modeling languages, CS2AS bridges are addressed entirely in the *treeware* technological space. This prevents Spoofax from giving support to complex modeling languages such as mOCL because some language requirements cannot be fulfilled (e.g. external models cannot be referred by the target AS models).

[5] `https://github.com/adolfosbh/cs2as/tree/master/uk.ac.york.cs.cs2as.metrics`

- The technical solution to shift from syntax trees to AS models does not compare the solutions fairly. For instance, whereas the tool developed can systematically measure the size of artefacts specified within Spoofax, this is not the case when taking into account the meta-model modified with the additional *EAnnotations*.
- Similarly, implementation performance cannot be fairly compared. Its *modelware* layer, which enacts the transformation syntax trees into AS models, invokes the Spoofax APIs to obtain the syntax trees. This process includes the file parse stage, which is not part of the evaluation.

Including Spoofax in the quantitative study of the thesis evaluation gives rise to different concerns (e.g. threats to experimental validity) that may compromise the study results and conclusions. Therefore, it has been deliberately excluded from this chapter. However, Appendix E replicates one of the experiments with Spoofax, including some quantitative data.

With respect to Gra2Mol, the examples from the official Github repository[6] have been considered for conducting the experiments. The rationale relies on reusing existing examples that have been addressed by the tool's creator, who is the authoritative person knowing the pros and cons of the tool and how to address the particular examples better. In this way, some threats to the experiment's validity are mitigated. Additionally, there are various language examples in the repository (from simple languages to complex ones) to check whether CS2AS-TL transformations are producing the same outcomes as those of Gra2Mol.

5.4.4 Hypotheses

Once the metrics of the evaluation have been introduced, the hypotheses that are used for validating the contribution of the proposed solution are presented. Given **X**, the proposed solution explained in this thesis, and **Y**, an alternative solution for comparison purposes, the following hypotheses are formulated.

Hypothesis A.

1. **Null hypothesis (H_A0):** the execution time required to obtain AS models from CS ones using the implementation of approach X is the same as the execution time required by the implementation of approach Y.
2. **Alternative hypothesis (H_A1):** the execution time required to obtain AS models from CS ones using the implementation of approach X is different to the execution time required by the implementation of approach Y.

[6] https://github.com/jlcanovas/gra2mol

Hypothesis B.

1. **Null hypothesis (H_B0):** the size of the artefact/s to describe a CS2AS bridge of a textual modeling language using approach X is the same as the size of the artefacts required by approach Y.

2. **Alternative hypothesis (H_B1):** the size of the artefact/s to describe a CS2AS bridge of a textual modeling language using approach X is different to the size of the artefacts required by approach Y.

5.4.5 Methodology

This subsection presents the methodology that drives this quantitative study. Given the varied nature of the presented hypotheses, two different approaches are taken to conduct the corresponding experiments.

CS2AS transformations execution with different models

When executing CS2AS model transformations for a textual modeling language, textual inputs are processed. In order to reject the null hypothesis A (H_A0), it is necessary to perform the execution with different instances of the textual modeling language under study. Moreover, a representative sample of models for an experiment involves models with different size and topology. In order to provide this variety in input files, a tailored parametrisable generator has been developed for this purpose (see Section 5.5.1).

The experiment to test hypothesis A consists of the following:

1. Executing Gra2Mol and the prototype implementation with a variety of input files conforming to the textual modeling language.

2. Ensuring that the output AS models obtained from the proposed prototype are the same output models obtained from Gra2Mol.

3. Measuring the execution time (see Section 5.4.2) incurred by the M2M transformations.

4. With the collected data, performing the corresponding statistical test to evaluate hypothesis A.

CS2AS bridges for different textual modeling languages

When creating CS2AS bridges for a textual modeling language, a CS2AS bridge (i.e. a CS2AS-TL instance) needs to be created. In order to reject the null hypothesis B (H_B0), different instances of CS2AS-TL are required. Therefore, varied textual modeling examples are necessary to perform the evaluation. In addition, testing CS2AS-TL with different textual modeling languages shows whether the current set of features of CS2AS-TL is sufficient to declare CS2AS bridges for these languages.

The experiment to test hypothesis B consists of the following:

- For a textual modeling language, replicate the CS2AS bridge described with Gra2Mol, by using CS2AS-TL.
- Count the number of words (see Section 5.4.2) of both CS2AS bridges.
- Repeat the method for all the textual modeling language examples involved in the experiment.
- With the collected data, perform the corresponding statistical test to evaluate hypothesis *B*.

Experiment Reproducibility

One of the concerns that arise when doing empirical research is experiment reproducibility. Researchers should facilitate as far as possible access to the experiment environment that was used to conduct a set of experiments. In this way, other researchers can check and confirm the obtained conclusions, and even enhance the research with new experiments conducted within the same environment.

In order to ease reproducibility, an experiment environment has been set up in the SHARE [74] platform[7]. The reader just needs to log in the platform and request access to the prepared virtual machine[8]. When access is granted, the user will remotely connect to the virtual machine and access the system using *Ubuntu* as the user name and *reverse* as the password. Additional information (README) about how to repeat the experiments can be found in the user desktop.

Hypotheses Testing

In order to support the null hypotheses rejection with an accepted statistical significance (p-vale <= 0.1), statistical tests are performed as part of the experiments. In this case, the Wilcoxon (non-parametric) test is selected. Regardless of whether the statistical test is meant to reject the null hypothesis *A* or *B*, the following rationale applies:

- Selection of a non-parametric test, because there is no assumption about the normal distribution of data.
- The collected data is paired. In other words, for every subject in the sample we collect one data value for each tool (paired values).
- Differences between the (ordinal) data values for the selected metrics (milliseconds and number of words) can be ranked.

[7] `http://share20.eu`

[8] `http://is.ieis.tue.nl/staff/pvgorp/share/?page=ConfigureNewSession&vdi=Ubuntu12LTS_CS2AS-DSTL---Experiments.vdi`

5.4.6 Threats to Experiment Validity

When conducting empirical research, the different threats that can risk the validity of experiments and conclusions must be taken into account [83]. This subsection lists the different identified threats and the corresponding mitigation actions.

Representative Sample

The main threat that this quantitative study presents is related to the sample on which the experiments are conducted, and in particular, how representative the set of examples chosen to run the experiments is. Given the kind of studies that are conducted, the following is taken into account:

- **Size of Artefacts**. When measuring the size of artefacts, it is important to have a representative sample from the wide set of possible textual modeling languages.
- **Performance**. When measuring performance, it is important to have a representative sample from the wide set of possible textual inputs that conform to a particular textual modeling language.

Taking a representative sample of textual modeling languages is a hard task. It is not clear which are the characteristics of such representativeness. A random CTMLs generator is not sensible, as the example must be well known and easy to explain. In order to mitigate this issue, the decision to select examples already implemented by related work (i.e. Gra2Mol) was taken. This removes any suspicion of choosing a language example that is suitable for the proposed solution but cannot be implemented by the related work.

Likewise, taking a representative sample of textual inputs is also difficult. It is not clear which are the characteristics of this representativeness. A random textual inputs generator may be sensible, but it must be ensured that textual inputs are valid for the particular language. In order to mitigate this issue, a tailored textual inputs generator has been built for one of the examples, so that the textual inputs vary in size and topology.

Equivalent CS2AS bridges

When conducting the experiments, it is important that key artefacts such as CS2AS bridges are equivalent. This must hold even when there are mappings (or rules) in these CS2AS specification artefacts that are not executed during the transformation execution. In order to mitigate this threat, the following actions have been adopted:

- Create the instances of CS2AS-TL as a replica of the Gra2Mol instances. This replication implies, for instance, that if any mapping declaration or property computation exists in a CS2AS bridge expressed with Gra2Mol DSTL, an equivalent declaration must exist in the CS2AS bridge expressed with CS2AS-TL.
- Ensure that the output models are equivalent after the corresponding CS2AS transformation executions.

Same Experiment Environments

When conducting the experiment related to measuring performance, it is important to set up a common environment in which both prototypes are executed. The same environment should also ideally be accessible by third parties, so that the experiments can be replicated. With respect to this concern, the actions that have been taken are the following:

- The experiments are run in the same physical machine and same operating system.
- Since they run on Java Virtual Machine, the same amount of physical memory is provided.

💡 For details about the environment, see the experiment design from Section 5.5.1.

5.5 Quantitative Study: Hypothesis A

This section shows the quantitative study performed to test hypothesis A. The study consists of obtaining quantitative data when executing CS2AS transformations on two different examples, and performing statistical tests with all the data. First, Section 5.5.1 focuses on a detailed example where all the specification artefacts, as well as a scalability analysis, are shown. Second, Section 5.5.2 focuses on another example which is not explained in detail but, instead, the relevant CS2AS scenarios are discussed. Finally, Section 5.5.3 presents the statistical tests performed with all the collected data.

5.5.1 Detailed Experiment: 101 Companies Example

This subsection introduces the first experiment. The example is called *101 companies* and it is the only one explained in detail throughout the evaluation. This example is located in the Gra2Mol repository and complies to the *101companies project* [22]. The *101 companies* example is simple enough to be explained in detail, requires cross-reference resolution, and permits models of varied size and topology.

Example Introduction

Figure 5.11 shows the CS and AS definition of the textual modeling language, as defined in the Gra2Mol repository. On the left hand side, the CS is defined by means of an (ANTLR-based) grammar. On the right hand side, the diagram shows the AS (Ecore-based) metamodel of the language. The running example of the *101companies* project [22] describes an imaginary *Human Resource Management System*. The proposed textual modeling language is conceived to deal with the information required by this kind of systems and it is explained as follows: a company has a name and is divided into departments. Each department can be split into sub-departments. Each department must have a manager and, optionally, other employees. Employees have a name, address and salary. In this particular *101companies* example, every employee may have another employee as a mentor.

```
company:
  'company' STRING '{'
    department*
  '}' EOF ;
department :
  'department' STRING '{'
  department_manager
  department_employees
  department*
'}' ;
department_manager :
  'manager' employee ;
department_employees :
  ('employee' employee)* ;
employee :
  STRING '{'
    'address' STRING
    'salary' FLOAT
    ('mentor' STRING)?
  '}' ;
```

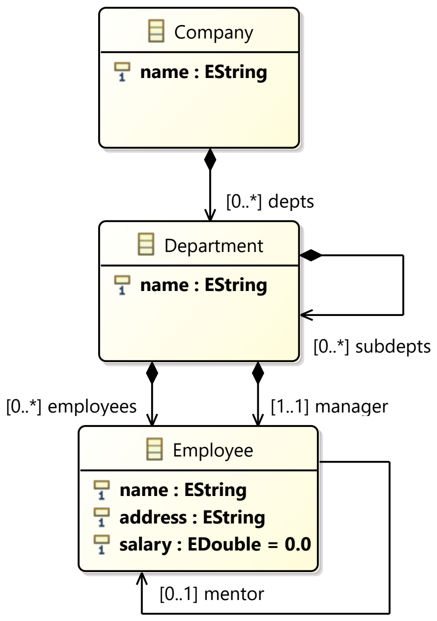


FIGURE 5.11: 101 Companies Example [22]. On the left, the CS (ANTLR-based) grammar of the language. On the right, the AS (Ecore-based) meta-model

In supporting this example with the proposed solution, the CS definition is specified by means of an analogous Xtext grammar. From this grammar specification, the CS meta-model can be automatically generated. Figure 5.12 shows the Xtext grammar definition on the left, and the corresponding CS meta-model on the right. Examining this generated CS meta-model and the final AS one, shown in Figure 5.11, the existing gap between the CS and AS in this particular example can be seen.

CS2AS Bridge

Once the example has been introduced, this subsection shows the specification of the CS2AS bridges defined with both tools.

Reminder. Instances of the Gra2Mol DSTL relate CS grammar terms to AS meta-model terms. Instances of CS2AS-TL relate CS meta-model terms to AS meta-model terms.

Figure 5.13 shows excerpts of the CS2AS bridges side by side. There are numerous similarities between the CS2AS descriptions, whereas the main differences are in the model queries. For instance, all the properties of the created AS meta-classes are initialised in Gra2Mol with a reference to a model query declared in the queries clause. In CS2AS-TL,

```
company :
  'company' name=STRING '{'
       deparment+=department*
  '}';
department :
  'department' name=STRING '{'
     department_manager=
          department_manager
     department_employees=
          department_employees
     deparment+= department*
  '}';
department_manager :
  'manager' employee=employee;
department_employees :
  {department_employees}
  ('employee' employee+=employee)*
 ;
employee :
  name=STRING '{'
   'address' address=STRING
   'salary' salary=FLOAT
   ('mentor' mentor=STRING)?
  '}';
```

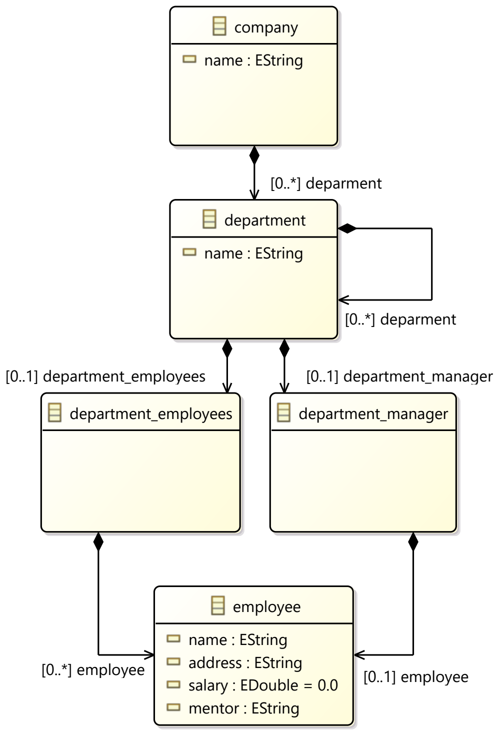


FIGURE 5.12: Equivalent Xtext grammar and corresponding CS meta-model

all these properties are initialised with the specific OCL expression that provides the result values for the properties.

Additionally, the CS2AS-TL isolates the name resolution concern in its own section. For instance, according to the Gra2Mol example (see line 27), an employee might be mentored by any other employee in the company. Therefore, in the name resolution section of the CS2AS-TL instance, it is declared (lines 25–30) that companies contribute to the current scope of all the employees spread across all departments – including subdepartments – of the company.

```
1  rule 'mapCompany'
2   from company c
3   to    Company
4   queries
5    dElem : /c//#department;
6   mappings
7    name = removeQuotes c.STRING;
8    depts = dElem;
9  end_rule
10 rule 'mapDepartment'
11  from department d
12  to    Department
13  queries
14   mElem : /d/department_manager/#
          employee;
15   eElem : /d/department_employees/#
          employee;
16   dElem : /d/#department;
17  mappings
18   name       = removeQuotes d.STRING;
19   manager   = mElem;
20   employees = eElem;
21   subdepts  = dElem;
22 end_rule
23 rule 'mapEmployee'
24  from employee e
25  to    Employee
26  queries
27   mElem : //#employee{STRING[0].eq(e
          .STRING[2])};
28  mappings
29   name    = removeQuotes e.STRING[0];
30   address = removeQuotes e.STRING[1];
31   salary   = e.FLOAT;
32   mentor  = mElem;
33 end_rule
```

```
1  source './gen/Companies.ecore#/'
2  target 'Company.ecore#/'
3  mappings {
4   create Company from company {
5    name := name;
6    depts := department.trace;
7   }
8   create Department from department {
9    name := name;
10   manager := department_manager.
          employee.trace;
11   employees := department_employees.
          employee.trace;
12   subdepts := deparment.trace;
13  }
14  create Employee from employee {
15   name := name;
16   address := address;
17   salary := salary;
18   mentor := lookup(Employee, mentor);
19  }}
20 name_resolution {
21  targets {
22   Employee using name;
23  }
24  providers {
25   Company {
26    in current_scope
27     provides depts−>closure(subdepts)
28                 −>collect(employees
29                 −>including(manager));
30   }}}
```

FIGURE 5.13: CS2AS specification in Gra2Mol (left) and CS2AS-TL (right)

Experiment Design

After introducing the example and how the CS2AS bridge is expressed using both tools, this subsection describes how the experiment is conducted. The goal of the experiment is testing hypothesis A.

With the aim of mitigating validity threats, a pseudo-random inputs generator for this particular experiment has been developed. The goal is to provide a set of varied inputs for the experiment, so that the obtained AS models are different in size and topology. The generator can be configured by the following parameters:

N_d : Number of (top level) departments in the company.

N_s : Number of subdepartments per department/subdepartment.

N_e : Number of employees per department/subdepartment.

D_s : Depth level of (sub)departments.

Attribute values are pseudo-randomly generated, whereas every employee is randomly mentored by another employee with a 0.5 probability. The inputs used in the experiment are characterised in Table 5.1. The samples vary from small inputs to big ones, and expected outputs vary from deep models to flat ones.

	Size (bytes)	N° of Elements	N_d	N_s	N_e	D_s
M_1	1,238	22	3	0	3	1
M_2	6,105	97	3	3	4	2
M_3	149,951	701	1	1	3	100
M_4	42,805	708	1	100	3	2
M_5	223,848	3061	4	4	5	4
M_6	1,018,254	11901	10	4	10	4
M_7	9,794,276	109341	10	5	10	5

TABLE 5.1: *101 Companies* example: models characterisation

The experiment consists of using both technologies to run the corresponding CS2AS transformations with each sample's input, and measuring the time required to perform the transformations. One of the features of the contributed CS2AS-TL is that it is parsing-technology agnostic. Therefore, the measurements do not include parsing time. In addition, any time on loading/storing models is not included either. In consequence, the required time stamps are taken just before the M2M transformation starts and right after it finishes.

With respect to the environment of the experiment in which the presented results were obtained, the following information is provided:

- It has been run in a desktop computer with the following characteristics:

  Architecture: x64 (64-bits)

  CPU: Inter(R) Core(TM) i7-3770 CPU @ 3.40 Ghz

  RAM: 8 GB

  OS: Windows 10

- The JVM is configured to use the default heap (2 GB).

- To warm up the JVM prior to taking measurements, the CS2AS transformation is run 3000 times with M_2.
- Garbage collection is forced prior to the transformation executions.

Experiment Results

All collected data from the experiments can be found in Appendix F.

This section presents the collected data, along with some observations and discussions. Figure 5.14 summarises the performance results.

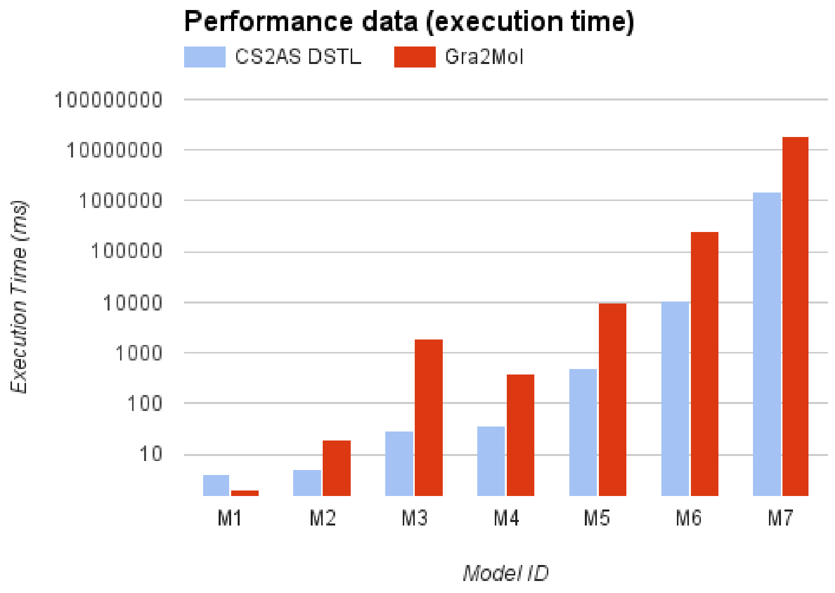


FIGURE 5.14: Experiment results: execution time

Gra2Mol took less time compared to our prototype when executing M_1 (the smallest one). As discussed further in the following scalability analysis, this reflects a start up cost which is greater in the proposed prototype. However, as soon as this setup time is negligible with respect to the model-size dependent time, it can be observed that Gra2Mol is over ten times slower than the prototype – in this particular example. There is an observed peak in performance when Gra2Mol deals with M_3 (701 model elements). This is unexpected, especially compared with M_4 (708 model elements), which has a similar number of model elements. If we look at the model parameters' characterisation from Table 5.1, we identify two main differences: M_3 is a deep model, whereas M_4 is a wide one. Despite the similar number of model elements, M_3 is bigger in terms of size (149,951 vs 42,805 bytes). This is explained by the logic used by the model generator to assign names to model elements: the deeper the named element is inside the model, the longer the string for the corresponding name. These topology differences between M_3 and M_4 make us conclude that model topology impacts on Gra2Mol performance – according to this particular example. In the case of the developed prototype, the difference in execution time is negligible.

Scalability Analysis

Aiming at a better interpretation of scalability, a new experiment is run with 25 models generated by the model generator. In this case, the independent variables are N_d, N_s and D_s (values set to 2), with N_e as the dependent variable. The dependent variable's value is gradually incremented to generate models whose amount of model elements varies between 10 and 10^6. The different values of N_e are conveniently calculated to obtain quasi-equidistant points in a log-scaled x-axis. Table 5.2 shows the values for certain involved models. With respect to the environment setup, we decreased the amount of the JVM memory heap[9], so that the scalability trends can be observed without having to wait unnecessarily for long executions with the largest models (i.e. the experiment tries to exhaust the memory).

	Size (bytes)	Elements	N_d	N_s	N_e	D_s
M_1	1,320	31	2	2	1	2
M_2	3,003	49	2	2	4	2
M_3	6,219	79	2	2	9	2
...	...	...	...	...	...	...
M_{21}	13,773,378	128473	2	2	21408	2
M_{22}	20,852,400	193663	2	2	32273	2
M_{23}	31,524,191	291931	2	2	48651	2
M_{24}	47,602,981	440071	2	2	73341	2
M_{25}	72,039,789	663379	2	2	110559	2

TABLE 5.2: New experiment models characterisation

Figure 5.15 shows the results that were obtained after running this new experiment. Overall, the prototype is still over ten times faster than Gra2Mol in this particular example. Likewise, the line slopes verify that the developed prototype and Gra2Mol do not present a linear performance. Additionally, we observe that the experiment involving the developed prototype consumed less memory, because model M_{23} was successfully transformed before an *out-of-memory* exception was obtained with model M_{24}, whereas the experiment involving Gra2Mol got an *out-of-memory* exception with model M_{23}.

All the data corresponding to this scalability analysis can be found in Table F.2 (Appendix F).

In order to address the scalability issue, the experiment was repeated with a modified transformation scenario; the computation of the*Employee::mentor* property from the Gra2Mol and CS2AS-TL instances (from Figure 5.13, lines 32 and 18 respectively) was removed. Additionally, the model query required to compute that property (line 27) was also removed from the Gra2Mol transformation.

Figure 5.16 shows the results that were obtained after running the new experiment. With respect to the proposed prototype the following is highlighted: in this particular experiment, the execution time remains constant overall when dealing with $M_1 - M_{13}$ models (roughly 7 milliseconds with models containing up to 4819 model elements). This shows that in

[9] Half of the amount for a default JVM heap size.

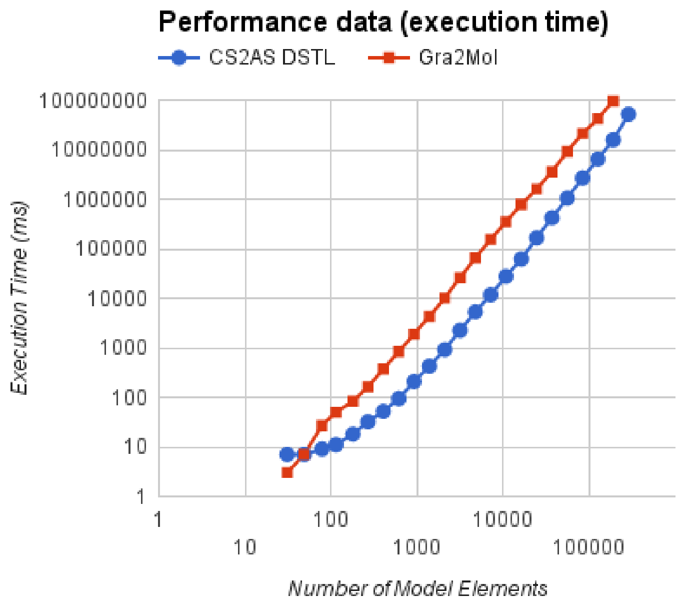


FIGURE 5.15: Scalability: execution times for 25 models

the developed prototype there is some required constant execution time that is model-size independent; that time is significantly greater than the model-size dependent time required for those specific models. With the subsequent models (from M_{14} to M_{20}), we can observe how the execution time increases along with model size. The important point is that the line slope observed shows that the increment is linear. With model M_{21}, an *out-of-memory* exception was thrown.

Looking at the results obtained for Gra2Mol, the model-size independent constant time is lower than the one required by the proposed prototype. However, once the model size increment starts to impact the execution time (from M_4), the line slope confirms that Gra2Mol performance is not linear with this simpler scenario.

With respect to the non-linear performance when the *Employee::mentor* property is computed, published work [65] has identified a need for an operation results caching mechanism to improve performance. The current prototype has improved the incorporation of this mechanism. Although performance results in big models have improved by a factor of two compared to the results obtained in the published paper, performance is still non-linear. Figure 5.17 shows in the same graph the results of running the scalability experiment (including the computation of the *Employee::mentor* property) with the three solutions: Gra2Mol, the prototype without operation results caches, and the prototype with operation results caches.

The conclusions that can be drawn from this scalability analysis is that further research can be done to improve the results. Some possible explanations about the current non-linear performance follow:

- The transformation engine execution has room for improvement in terms of performance (e.g. when working on collections).

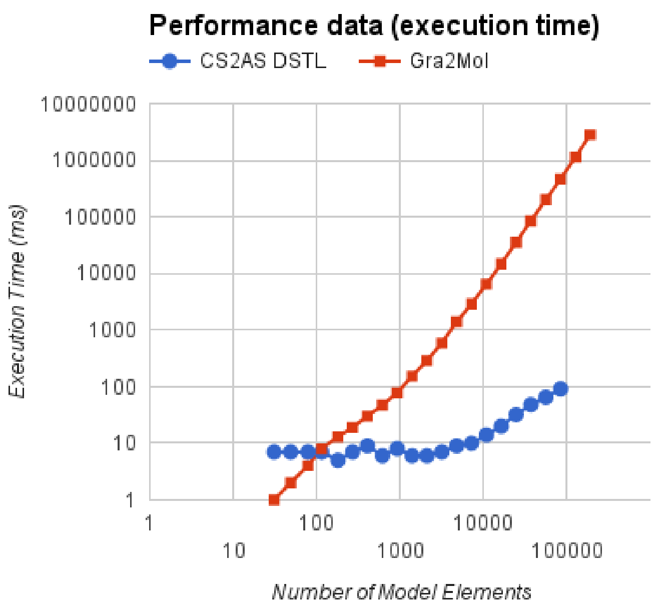


FIGURE 5.16: Scalability: performance when *Employee::mentor* is not computed

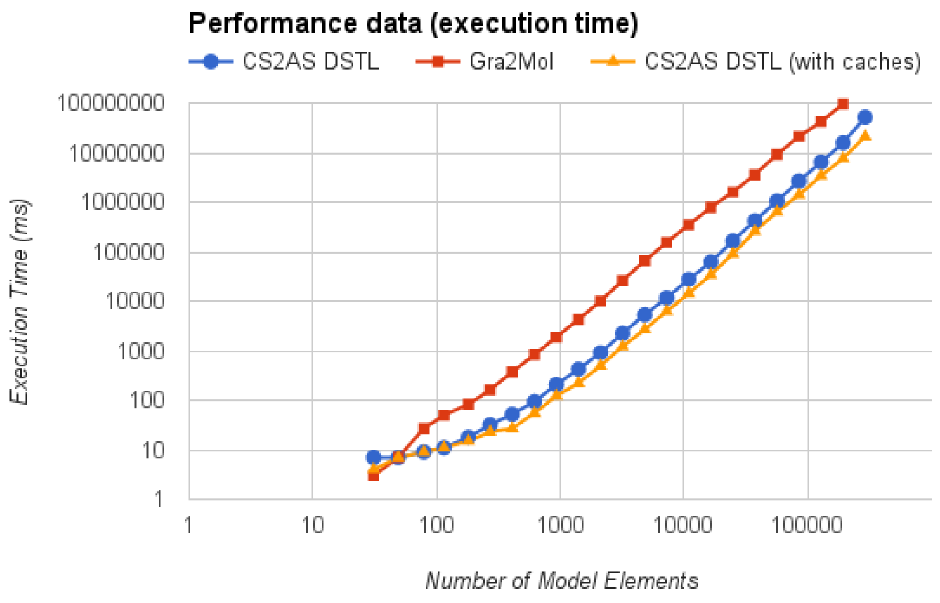


FIGURE 5.17: Scalability: performance when caching operation results

- The name-based lookup algorithms are non-linear in terms of performance (e.g. this scenario involved a closure).

5.5.2 Another Experiment: Delphi2ASTM Example

This subsection introduces the second experiment. The example is called *Delphi2ASTM* and it is proposed by Gra2Mol as a complex example, in which the textual files conforming to the CS of the Delphi language are transformed to AS models conforming to the Abstract Syntax Tree Metamodel (ASTM) [35]. The goal of this experiment is to obtain additional quantitative data, with a different language example, in order to test hypothesis A (see Section 5.5.3). Due to the size of this example, this experiment is not explained in as much detail as the *101 Companies* example.

The rationale for choosing this particular example is the following:

- It is one of the complex examples that can be found in the Gra2Mol repository. It requires creation and reference mappings, name resolution and disambiguation rules.
- It exposes some of the pros and cons (see Section 4.6) of CS2AS-TL.

Example Introduction

The *Delphi2ASTM* example targets the creation of AS models conforming to the ASTM from Delphi textual files. This meta-model is proposed by the OMG Architecture Driven Modernization (ADM) task force to provide a low-level software modeling language independent from any particular programming language. In this way, the source code of different programming languages can be translated into the same model representation.

On the one hand, due to the size of the example, no details about the CS grammars, AS meta-models or the entire CS2AS descriptions are provided. All the relevant artefacts are publicly available for both Gra2Mol[10] and CS2AS-TL [11]-based solutions. Table 5.3 shows the references to the different specification artefacts for the example.

Solution	Artefact path
Gra2Mol	
CS Definition	/ASTMExtraction/files/src/delphi.g
AS Definition	/ASTMExtraction/files/meta-models/astm.ecore
CS2AS Definition	/ASTMExtraction/files/src/delphi2astm.g2m
CS2AS-TL	
CS Definition	/src/org/xtext/example/delphi/Delphi.xtext
AS Definition	/model/astm.ecore
CS2AS Definition	/model/Delphi.cs2as

TABLE 5.3: *Delphi2ASTM* example: specification artefacts

[10] `https://github.com/adolfosbh/gra2mol/tree/master/examples/Grammar2Model.examples.Delphi2Model`

[11] `https://github.com/adolfosbh/cs2as/tree/master/org.xtext.example.delphi`

On the other hand, due to the discussion that certain CS2AS scenarios have provoked, the forthcoming subsections show how these specific CS2AS scenarios are addressed by both solutions.

Whilst the previous experiment did not require changing the CS2AS bridge defined in Gra2Mol, this was not the case for this experiment. All details of the changes made to accommodate the specification artefacts are explained in the forthcoming subsections.

CS2AS Bridge: Missing Rules/Mappings

When replicating the experiment with the proposed solution, the first problem that came up was the following: the CS2AS bridge of the *Delphi2ASTM* example defined in the Gra2Mol repository was incomplete. On the one hand, having a partial CS2AS bridge is not critical [12]. On the other hand, the proposed prototype has encountered different problems due to this incompleteness, and in particular with the absence of many rules (or mappings) definitions in the original Gra2Mol transformation.

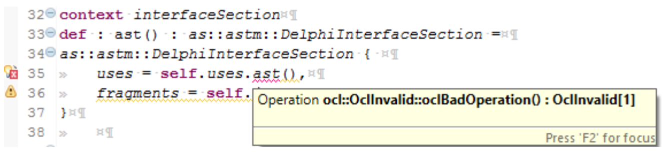
The problems that arise due to these missing rules (or mappings) can be categorised into the following:

- The RHS of Gra2Mol bindings usually require model elements to be returned from another rule. However, not all corresponding rules were defined. In Gra2Mol, when bindings are executed and there is no corresponding rule, nothing is returned. In CS2AS-TL, the problem arises during the compilation process. The Complete OCL documents will contain compilation errors, because the particular OCL expression (equivalent to the RHS of a Gra2Mol binding) will invoke an *ast()* operation which has not been defined (because of the missing mapping declaration). For instance, Figure 5.18 shows an excerpt of the Gra2Mol transformation in which a binding (line 11) involves a query (line 5) that requires a mapping for *usesClause* terms. However, in the original Gra2Mol transformation there was no mapping rule to transform these *usesClause* terms. The Gra2Mol engine can tolerate this incomplete transformation, but the developed prototype produces invalid Complete OCL documents that prevent the compilation process from finishing successfully.

- Some missing rules were supposed to create model elements designed to contain other model elements – created by existing rules. These missing rules produced different output models when using both tools. In the case of the developed prototype, all the models created by existing rules ended up as root model elements (their expected containers were not created due to the missing rules). One example of this situation is shown in Listing 5.16. In this case, a *skip* rule is declared for a *simpleExpression*. This rule skips to a *term* in case no *addOp* exists. However, in the Gra2Mol transformation, there is no rule to handle the case in which an *addOp* does exist. This missing rule,

[12] From the point of view of experiment validity, having incomplete CS2AS definitions is not important, as long as the corresponding CS2AS-TL instance is equivalent.

```
rule 'mapInterfaceSection'
  from interfaceSection is
  to    astm::sastm::
          DelphiInterfaceSection
  queries
    uElem : /is/#usesClause;
    cElem : /is/interfaceDecl/#
            constSection;
    tElem : /is/interfaceDecl/#
            typeSection;
    vElem : /is/interfaceDecl/#
            varSection;
    eElem : /is/interfaceDecl/#
            exportedHeading;
  mappings
    uses      = uElem;
    fragments   = cElem;
    fragments   = tElem;
    fragments   = vElem;
    fragments   = eElem;
end_rule
```



```
context interfaceSection
def : ast() : as::astm::DelphiInterfaceSection =
as::astm::DelphiInterfaceSection {
    uses = self.uses.ast(),
    fragments = self.
}

```

Operation **ocl::OclInvalid::oclBadOperation() : OclInvalid[1]**

Press 'F2' for focus

FIGURE 5.18: Left: Gra2Mol rule definition that specifies a binding that requires a missing rule. Right: the corresponding Complete OCL code with compilation errors

when the *addOp* does exist, results in the following: no proper container is created to accommodate the left and right (sub)expressions of the additive expression. For the CS2AS transformation of the developed prototype, all these (sub)expressions ended up as root model elements.

```
skip_rule 'skipAddOp'
  from  simpleExpression se
  to     astm::gastm::Expression
  queries
    existsAddOp : /se/#addOp[0];
    nextNode    :  /se/#term;
  mappings
    if (existsAddOp.hasResults) then
      skip existsAddOp;
    else
      skip nextNode;
    end_if
end_rule
```

LISTING 5.16: A Gra2Mol rule that *skips* to its (sub)term when there is no *addOp*

These missing mappings needed to be addressed: either they prevented the CS2AS transformation from being compiled by the prototype, or the transformation execution produced different output models (when comparing the outputs from the Gra2Mol transformation execution). In order to address these problems, two different kinds of actions were taken:

- Provided that many of the missing rules did not even need to be executed for the input examples, the action was to remove the bindings that required the results of the missing rules. For instance, in the left hand side listing from Figure 5.18, lines 5 and 11 were removed.
- All missing mappings that were required to have equivalent output models were added to the Gra2Mol transformation. For instance, Listing 5.17 shows a new mapping that was added.

When resolving this problem related to the missing mappings, the following discussion about the pros and cons of the tools can be made. Although the CS2AS transformation could not be executed by the developed prototype in the first instance, the statically typed OCL has helped to detect some deficiencies in the transformation defined by Gra2Mol. Additionally, having orphan model elements in the output models was an indication that there were additional missing rules in the transformation. Therefore, working on the CS2AS transformation using the developed prototype has helped to identify these missing mappings.

```
rule 'AddOp'
  from  simpleExpression{addOp[0].exists} se
  to     astm::gastm::BinaryExpression
  queries
    lTerm     :  /se/#term[0];
    rTerm     :  /se/#term[1];
  mappings
    leftOperand  = lTerm;
    operator      = new astm::gastm::Add;
    rightOperand = rTerm;
end_rule
```

LISTING 5.17: A Gra2Mol rule that creates a *BinaryExpression*

CS2AS Bridge: No information in CS

As stated in Section 4.6.1, one limitation of the proposed approach is that all the information that is needed by the AS model must be present in the CS model. Listing 5.18 shows this limitation. In this case, there are AS model elements designed to keep information about the source file (lines 8–10), for instance, the path of the parsed file. The internal CST models have been specifically designed not only to keep language information defined by the ANTLR grammar, but also to include additional information such as the source file path. Since this kind of information is not captured by the CS meta-model automatically generated by Xtext, it cannot be passed to the AS models via a CS2AS definition.

```
1  rule 'mapFileFromUnit'
2    from file//unit f
3    to astm::sastm::DelphiUnit
4    queries
5      -- queries ommitted
6    mappings
7      -- other mappings ommitted
8      locationInfo  = new astm::gastm::SourceLocation;
9      locationInfo.inSourceFile   = new astm::gastm::SourceFile;
10     locationInfo.inSourceFile.pathName = f.path;
11 end_rule
```

LISTING 5.18: A Gra2Mol rule which uses the path of the parsed file

In order to keep equivalent CS2AS transformations, the decision was taken to remove the Gra2Mol binding in which this source file path is assigned to a *SourceFile* model element (line 10).

CS2AS Bridge: Refactoring CS grammar

The last problem to discuss is related to a refactoring of the CS grammar defined by the Gra2Mol example. CS2AS-TL is parsing agnostic, but it needs a CS meta-model. Although there is no identified limitation with respect to the structure of the CS meta-model, Section 4.6.2 explained that the developed prototype imposes a limitation due to the chosen M2M transformation engine. This limitation prohibits CS meta-models that use references pointing to generic model elements (i.e. *EObject* in terms of EMF).

When using Xtext to generate the CS meta-model, a situation arises in which this kind of forbidden meta-models are generated. Listing 5.19 shows an excerpt of the ANTLR grammar defined by the Gra2Mol example. The problem appears when there is an alternative of non-terminals within a syntactic rule, in this case, *simpleStatement* and *structStmt*.

```
statement
  : (labelId ':')? (simpleStatement | structStmt)
;
```

LISTING 5.19: An ANTLR syntactic rule to declare statements

In this situation, Xtext cannot infer a common supertype from the two non-terminals and, therefore, the corresponding reference to hold the model element is typed as a generic *EObject*. To solve this situation, a refactoring of the Xtext grammar was performed, so there is a new non-terminal that defines the alternative. Listing 5.20 shows the refactored syntactic rule.

```
statement
  : (labelId=labelId ':')? statement=unlabelledStatement
;
unlabelledStatement
  : simpleStatement | structStmt
;
```

LISTING 5.20: The refactored syntactic rule to declare statements using Xtext

Experiment Design

This subsection introduces how the experiment was conducted. As with the previous experiment, it consists of executing the corresponding CS2AS transformation that consumes input CS models and produces the expected AS models.

In this case, a set of Delphi files that already existed in the Gra2Mol repository were chosen as the experiment sample. Table 5.4 shows a characterisation of the sample files, including the size of the input file in bytes and the corresponding lines of code.

Experiment Results

This section presents the collected data, along with some observations and discussions. Figure 5.19 summarises the performance results.

From the obtained results, it can be observed that for small models there is no significant difference in the obtained results; however, as soon as the size of the input files grows, the

	Size (bytes)	Lines of Code
M_1	1,304	58
M_2	1,858	81
M_3	7,246	263
M_4	66,048	2,238
M_5	241,188	8,118
M_6	708,228	23,798

TABLE 5.4: *Delphi2ASTM* example: models characterisation

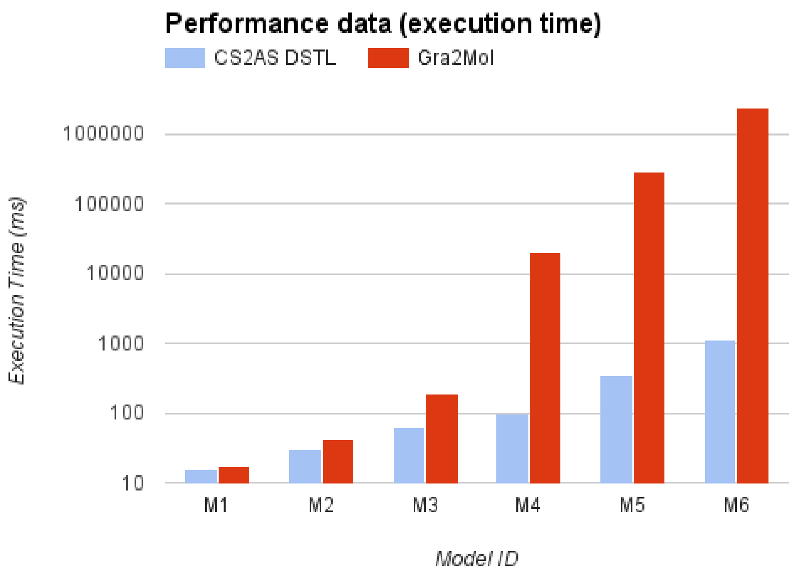


FIGURE 5.19: *Delphi2ASTM* experiment results: execution time

differences are much larger. For the largest input file, the Gra2Mol implementation is 2,215 times slower than the developed prototype.

5.5.3 Testing Hypothesis *A*

To conclude this first quantitative study, hypothesis *A* is formally tested. Recall that the null hypothesis (H_A0) we want to reject states that "the execution time required to obtain AS models from CS ones using the implementation of the proposed approach is the same as the execution time required by the implementation of the Gra2Mol approach".

Since the experiments have been conducted with different language examples, the hypothesis is tested with the data of each language individually.

101 Companies Example

According to the data shown in Table F.1 and Table F.2[13], and by performing the Wilcoxon statistical test, the null hypothesis (H_A0) can be rejected with a p-value = 0.000005549.

[13] For the developed prototype, the data used corresponds to a better solution based on cached operation results.

⚠ Since there are too many data points to be shown in one table, the reader is referred to Appendix F in which all the data is shown.

According to this example and the obtained data, the statistical test concludes that the execution time of CS2AS transformations for the developed prototype is lower than the execution time of CS2AS transformations for Gra2Mol. The rationale is based on the following statements:

- The statistical test concludes that the execution times are significantly different.
- In 26 out of 29 samples, the developed prototype was faster (this was not the case in 3 small models).

Delphi2ASTM Companies Example

According to the data shown in Table 5.5, and by performing the Wilcoxon statistical test, the null hypothesis (H_A0) can be rejected with a p-value = 0.03125.

Gra2Mol	17	43	187	20,400	287,633	245048783	
CS2AS-TL	16	31	63	98	355	1106	
							p-value=0.03125

TABLE 5.5: Wilcoxon statistical tests for H_A0 (Delphi Example). Paired data corresponds to the execution time obtained from the experiment

According to this example and the obtained data, the statistical test concludes that the execution time of CS2AS transformations for the developed prototype is lower than the execution time of CS2AS transformations for Gra2Mol. The rationale is based on the following statements:

- The statistical test concludes that the execution times are significantly different.
- In 6 out of 6 samples, the developed prototype was faster.

5.6 Quantitative Study: Hypothesis B

This section shows the quantitative study performed to test hypothesis B. The study consists of obtaining quantitative data about the size of artefacts (see Section 5.4.2) and performing a statistical test with all the collected data. In addition to the two examples already presented, the study comprises three new examples. Firstly, Section 5.6.1 introduces the three additional examples. Then, Section 5.6.2 shows the results obtained when measuring the size of the CS2AS bridges. Finally, Section 5.6.3 presents the statistical test performed with the collected data.

5.6.1 Examples Introduction

This subsection introduces the new language examples for which CS2AS bridges are defined using Gra2Mol and CS2AS-TL. As with the *Delphi2ASTM* example, no details about the different artefacts are given. Instead, pointers to the public locations where the artefacts can be checked are provided.

Example: DDL2ASTM

The third language example is called *DDL2ASTM*. Data Description Language (DDL) is a textual language designed to describe database schemas for relational database management systems. As with the *Delphi2ASTM* example, the goal is to produce AS models conforming to the ASTM. In this case, the textual inputs are DDL files.

The experiment consisted of replicating the Gra2Mol[14] transformation using CS2AS-TL[15]. Table 5.6 shows the references to the different specification artefacts for the example.

Solution	Artefact path
Gra2Mol	
CS Definition	/ASTMextraction/src/DDL.g
AS Definition	/metamodels/astm.ecore
CS2AS Definition	/ASTMextraction/src/extractASTMFromDDL.g2m
CS2AS-TL	
CS Definition	/src/org/xtext/example/plsql/DDL.xtext
AS Definition	/model/astm.ecore
CS2AS Definition	/model/plsql.cs2as

TABLE 5.6: *DDL2ASTM* example: specification artefacts

Example: iMacros

The fourth language example is called *iMacros* [16]. Internet Macros (iMac) provides a textual language designed to automate tasks on the web, for instance, data extraction. In this case, the Gra2Mol example provides a specific iMacros AS meta-model.

The experiment consisted of replicating the Gra2Mol[17] transformation using CS2AS-TL[18]. Table 5.7 shows the references to the different specification artefacts for the example.

Example: ABNF2Ecore

The fifth language example is called *ABNF2Ecore*. The Augmented Backus Naur Form (ABNF) is used in OMG specifications – such as the ASTM – to define meta-models textually. In this example, the target AS meta-model is the Ecore (EMF) meta-model. The goal

[14] `https://github.com/adolfosbh/gra2mol/tree/master/examples/Grammar2Model.examples.PLSQL2ASTMModel`

[15] `https://github.com/adolfosbh/cs2as/tree/master/org.xtext.example.plsql`

[16] `http://imacros.net/`

[17] `https://github.com/adolfosbh/gra2mol/tree/master/examples/Grammar2Model.examples.macros`

[18] `https://github.com/adolfosbh/cs2as/tree/master/org.xtext.example.macros`

Solution	Artefact path
Gra2Mol	
CS Definition	/files/src/iMacros.g
AS Definition	/files/metamodels/iMacros.ecore
CS2AS Definition	/files/src/transformation.g2m
CS2AS-TL	
CS Definition	/src/org/xtext/example/macros/Macros.xtext
AS Definition	/model/iMacros.ecore
CS2AS Definition	/model/imacros.cs2as

TABLE 5.7: *iMacros* example: specification artefacts

in this Gra2Mol example is creating Ecore meta-models from the corresponding textual inputs.

As with previous examples, the experiment consisted of replicating the Gra2Mol[19] transformation using CS2AS-TL[20]. Table 5.7 shows the references to the different specification artefacts for the example.

Solution	Artefact path
Gra2Mol	
CS Definition	/files/src/ABNF.g
AS Definition	/files/metamodels/Ecore.ecore
CS2AS Definition	/files/src/extractABNF.g2m
CS2AS-TL	
CS Definition	/src/org/xtext/example/abnf/ABNF.xtext
AS Definition	/model/Ecore.ecore
CS2AS Definition	/model/ABNF.cs2as

TABLE 5.8: *ABNF2Ecore* example: specification artefacts

5.6.2 Obtained Results

Once the Gra2Mol examples have been replicated using CS2AS-TL, the corresponding CS2AS bridges can be measured. Table 5.9 shows the obtained measurements. It can be observed that for all the examples there is a decrement in the number of words (up to 40% in some cases), the number of LoC and the size of the file.

Although the number of words is the measure of interest, it can be observed that for all examples and all measures, the size of the CS2AS specification artefacts using CS2AS-TL decreases compared to the measurements obtained for the Gra2Mol DSTL.

5.6.3 Testing Hypothesis *B*

To conclude this second quantitative study, hypothesis B is formally tested. Recall that the null hypothesis (H_B0) we want to reject states that "the size of the artefact/s to describe a

[19] `https://github.com/adolfosbh/gra2mol/tree/master/examples/Grammar2Model.examples.ABNF2MModel`
[20] `https://github.com/adolfosbh/cs2as/tree/master/org.xtext.example.abnf`

	Gra2Mol			CS2AS-TL			
Example	**Words**	LoC	Bytes	**Words**	LoC	Bytes	**Words Decrement**
101 Companies	83	35	757	73	33	738	10 (-21.04%)
Delphi2ASTM	1151	565	13,414	704	289	7,898	443 (-38.49%)
DDL2ASTM	376	150	5,071	224	86	2,411	152 (-40.42%)
iMacros	47	28	420	32	15	306	15 (-31.91%)
ABNF	541	225	6,086	353	151	4212	188 (-34.75%)

TABLE 5.9: Measurements of size of specification artefacts

CS2AS bridge of a textual modeling language using the proposed approach is the same as the size of the artefacts required by the Gra2Mol approach". According to the data shown in Table 5.10, and by performing the Wilcoxon statistical test, the null hypothesis (H_B0) can be rejected with a p-value = 0.0625.

Gra2Mol	83	1157	376	47	541	
CS2AS-TL	73	537	224	32	353	
						p-value=0.0625

TABLE 5.10: Wilcoxon statistical tests for H_B0. Paired data corresponds to the size of artefacts obtained from the experiment

According to these examples and the obtained data, the statistical test lets us conclude that the size of CS2AS-TL instances is lower than the size of the Gra2Mol DSTL instances when describing CS2AS bridges. The rationale is based on the following statements:

- The statistical test concludes that the sizes of CS2AS-TL instances are significantly different.
- In 5 out of 5 samples, the size of CS2AS-TL instances are lower.

5.7 Summary

This chapter presented the evaluation and obtained results of this thesis. Firstly, Section 5.1 introduced a comparative discussion between CS2AS-TL and general purpose M2M transformation languages. This discussion highlighted the benefits of CS2AS-TL when there is a need for looking up named elements throughout the AS models.

Section 5.2 and Section 5.3 presented two qualitative studies focused on comparing the proposed approach and related work: Gra2Mol and Spoofax. With respect to the former, Gra2Mol provides a useful language extension mechanism: many limitations of its short domain specific language can be worked around by plugging in custom functionality written in Java. Conversely, not having useful declarative constructs, such as those provided by the name resolution section of CS2AS-TL, underline the need for using this less convenient external language mechanism. With respect to the latter, Spoofax provides a specific set of languages to work on complex textual languages. However, its support for work on modeling languages is limited, and its specification artefacts cannot address certain requirements, such as making the AS models refer to external models.

The remainder of the chapter focused on evaluating this thesis' contribution with quantitative studies. Several examples were introduced with a different degree of detail, and the collected data was used to test statistically whether: a) Gra2Mol and CS2AS-TL had the same performance results (in terms of execution time); b) the amount of specification artefacts required to declare CS2AS bridges was the same within Gra2Mol and CS2AS-TL. According to the experiments and hypothesis tests, the benefits and contributions of CS2AS-TL were demonstrated.

Chapter 6

Future Work & Conclusions

This chapter describes future work and concludes the thesis. Section 6.1 discusses future work from two different points of view: CS2AS-TL to declare CS2AS bridges (the main contribution of this thesis) and the current prototype. Section 6.2 presents the conclusions of this thesis.

6.1 Future work

The work presented in this thesis indicates the need for further research and development. This section focuses on future work from two different perspectives: firstly, in terms of CS2AS-TL design and capabilities, and secondly, in terms of the development of the current prototype, which supports all the work presented in this thesis.

6.1.1 CS2AS-TL Improvements

CS2AS-TL Extension Mechanism

Gra2Mol avoids supporting a rich set of expressions and domain-specific constructs thanks to its flexible language extension mechanism. In this way, any limitation that the language has can be worked around via this black-boxing mechanism.

On the one hand, in this kind of specification language (to define CS2AS bridges), it is not ideal to have some *"undefined"* declarations that are separately fulfilled (implemented) in some lower-level programming language.

On the other hand, such a feature may be welcomed by pragmatic users in a particular CS2AS scenario. For instance, they may need some information in the AS model that is not present in the CS one, and they may prefer to provide this information during this CS2AS step rather than in a separate AS model post-processing step. The provision of a black-boxing mechanism to hook in custom functionality is a feature to consider in the future.

Type Resolution

One of CS2AS-TL capabilities consists of a declarative name resolution section (Section 4.2.5). It allows the end user to avoid dealing with name-based lookup algorithms to compute cross-references between AS model elements. CS2AS-TL could be similarly enhanced with a type resolution section. Such a section would alleviate the need for the end user to deal with type-based lookup algorithms that compute cross-references between AS model elements. Spoofax [72] already provides a DSL to declare type resolution, called *Type Specification Language*. Bettini [4] also presents another DSL, called XSemantics, to declare type system definitions separate from Xtext grammars. The goal of XSemantics is different to the one pursued in this research: it targets the generation of additional tools that validate the textual instances, rather than integrating the computed types as part of the AS models.

Disambiguation Rules Inheritance

Section 4.6.1 mentioned a limitation of the disambiguation section when working on a family of languages. Provided that there existed a set of disambiguation rules *A, B* and *C* for a base CS meta-class *X*, and a new derived CS meta-class *Y* of a derived (extending) language required an additional CS disambiguation rule *D*, CS2AS-TL forces you to (re-)declare all four *A, B, C* and *D* rules for the new derived CS meta-class *Y*. As a hypothetical requirement, the additional rule *D* may need to have a higher precedence than, for instance, rules *B* and *C*. Although during this research no real CS2AS scenario related to this need has been found, the syntax to deal with disambiguation rules could be enhanced to add an inheritance mechanism, including order roll-out (priority) modification.

Bi-directional CS2AS Bridges

One of the features that has not been considered is bi-directional CS2AS transformations. This activity is important to support advanced features, such as AS model refactoring, so that whenever the AS model is modified (e.g. in a further M2M transformation), the corresponding changes can be reflected back to the original textual input. Currently, to support this reverse AS2CS transformation with the current CS2AS-TL, a new instance in the opposite direction would need to be fully declared. However, this solution is far from ideal. Further study on bi-directional model transformations would be required.

6.1.2 Prototype Improvements

In addition to further research on the CS2AS-TL design, future work on the developed prototype can accrue a variety of benefits. Some topics focus on improving the internal functionality of the developed prototype (e.g. compilation process). Other topics focus on improvements for the end-user tools (e.g. better error reporting or a debugger) for the CS2AS-TL.

Compiling to QVT Relations

A compilation of instances of CS2AS-TL currently targets QVTm, a low-level M2M transformation language hosted in the Eclipse QVTd project. Additionally, a set of Complete OCL documents is produced as an intermediate compilation step. In this way, when specifying the CS2AS bridges for the OCL and QVT languages, the corresponding sets of Complete OCL documents can be incorporated into the respective OMG specification, so that the CS2AS bridge can be formalised [66].

Part of the proposed future work consists of compiling to QVT Relations (QVTr) rather than QVTm. Recent work [77] shows that QVTr transformations can be efficiently executed within the Eclipse Modeling Framework (EMF). Since QVTr can be a bi-directional M2M transformation language, compiling to QVTr proves to be a sensible step when considering the support for bi-directional execution. Additionally, since QVTr belongs to an OMG specification, these QVTr-based transformations could be a new way of specifying and formalising CS2AS bridges within any OMG specification.

Improved Error Reporting

One of the drawbacks of the current prototype is that the error reporting is poor, in the sense that the end user is not precisely informed about what is the cause of any problem that has occurred, either during the compilation process or during execution of the Java transformation. For future work, the following improvements are proposed:

- At compilation time, there are currently error reports within the generated Complete OCL documents. These reports come in the form of warning and error markers that are provided by the Eclipse OCL tooling. Since OCL is a statically typed language, different kinds of problems (e.g. a missing mapping) can be notified before executing any transformation. A further improvement consists of implementing a similar approach so that this type of markers appears in the Xtex-based editor of the CS2AS-TL.
- At execution time, additional work on the reused M2M transformation engine is required, so that intelligible errors are propagated to the final end-user tools (e.g. the editor that is editing the textual file).

Debugger

Currently, there is no debugger that works on instances of the CS2AS-TL. This is partly caused by the choice of the solution back-end. There is a need for a tracing mechanism between the generated Java transformation back to the original edited file to debug. This mechanism is not currently supported by the Java code generator hosted in the Eclipse QVTd project. Besides, having different compilation steps between the original textual file (the CS2AS-TL instance) and the final executable one (the code-generated Java transformation) makes the creation of a debugger even harder. According to the Eclipse QVTd project[1], there are plans for a debugger for QVTr and QVTc. Since the compilation and execution of these languages rely on the same multiple compilation steps approach, the debugger implemented in Eclipse QVTd may be easy to adopt by the current prototype implementation.

Incremental Execution

Part of the prototype evaluation focused on the execution time required to obtain AS models from textual input files; the larger the file is, the greater the execution time. Taking into

[1] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=487075`

account that the underlying transformation may be executed along with some editing facilities (i.e. textual editor), high execution times may produce a negative impact to the end user when working on large input files. In this context, although it is important to have an efficient CS2AS transformation that produces the entire AS model, incremental transformations are needed to improve user experience. This way, after the user makes a small change to the input file (delta), an equally small part of the output AS model should be modified in a short amount of time.

From the point of view of the proposed approach, exploiting M2M incremental transformations is a matter of targeting the appropriate M2M transformation technology that supports them. Although the current target M2M transformation engine does not properly execute incrementally yet, there are plans (and ongoing work) to support incremental execution[2].

⚠ For a complete incremental execution solution, it is also necessary to have an incremental front-end (i.e. the parser). Xtext does not currently support incremental parsing. However, recall that the proposed solution is independent of parser technology.

Xtext Integration

This thesis has presented work aimed at integrating tools that complement those generated by default by Xtext. However, there is still room for improvement in these integration activities:

- Although the Eclipse outline view shows the expected AS model, traceability back to the text is not implemented. Therefore, clicking on the outline view elements does not allow navigating throughout the textual file.
- Cross-references take place in the AS rather than in the CS model. Additional source code would be required for having proper hyper-linking across the textual editor.
- The generated lookup infrastructure (i.e. Java source code) that provides the set of candidates during name resolution can be reused by the generated editor (e.g. content assistant). Actually, for the mOCL example, the generated editor currently reuses such infrastructure (see Section 4.5.2). However, there is a manually written invocation of the lookup infrastructure to make the enhanced content assistant appear in the expected places of the editor. These manual invocations could also be auto-generated.

Bootstrapping CS2AS-TL

One of the pending improvements of the prototype is the creation of the CS2AS bridge of CS2AS-TL itself. Since the language extends and reuses the Eclipse OCL, and the CS2AS bridges have not been implemented in the project yet, the CS2AS bridge of CS2AS-TL is not currently implemented.

[2] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=500962`

6.1.3 Further Research

To conclude the exposition of future work, this subsection introduces additional further research that can be derived from this thesis.

Additional Experiments

Although the results obtained throughout the experiments were positive for the CS2AS-TL, the prototype could be used with more examples (both languages and input models) to provide additional evidence, reduce further experiment validity threats and, therefore, strengthen the obtained conclusions.

Additionally, other kinds of experiments can be conducted. In particular, experiments related to the usability of the tool when compared to related work or even to a whole solution written in a traditional programming language (e.g. Java). Prior to performing any user-oriented experiment, the maturity of the developed tools is required, including some important additions, such as better error reporting and debugging (see Section 6.1.2).

CS2AS Bridges for Graphical Languages

Although the target of this research project is complex textual modeling languages, there are still open questions related to whether the CS2AS-TL could be used on languages with a different CS, for instance, graphical languages. In this type of languages, the concrete syntax is based on graphical elements rather than text. Although there is no current work to provide the required evidence, it seems that CS2AS-TL could be used for describing CS2AS bridges for complex graphical modeling languages. This would only be possible as long as the graphical language CS was defined by meta-models.

Can Some CS2AS-TL Features be Adopted by General Purpose M2M Transformation Languages?

The main contribution of this thesis has been presented as a **domain-specific** transformation language. The rationale is that some features are inherent to bridging the gap between the CS and AS of a language (the specific domain). However, it may be argued that some features are not domain-specific and the ideas behind these features can be adopted by general purpose M2M transformation languages. In particular, the following two research lines are proposed:

- **Disambiguation section**. When compared to general purpose M2M transformation languages, the idea of separating disambiguation rules from mapping definitions and keep them in their own section with their own semantics is novel. However, these disambiguation rules align with the well-known concept of mapping/rule guard in M2M transformations. Different research questions arise: can the same ideas be adopted by existing M2M transformation languages? Otherwise, why does this particular domain make them useful to CS2AS-TL, but not at all to general purpose M2M transformation languages?

- **Name resolution section**. When compared to general purpose M2M transformation languages, Section 5.1 showed that the name resolution section provides significant benefits to the end user. However, the section conceptually comprises a more general operational behaviour for setting up cross-references by means of named element lookups. Different research questions arise: can only a name (*String* typed value) be used to perform these lookups? Is the scope propagation mechanism enough for the general case? Is there any other flexible but concise way to provide additional lookup mechanisms?

6.2 Conclusions

This thesis targets the problem of bridging the gap between the CS and AS of CTMLs. Although existing approaches suggest that these bridges can be accomplished merely by defining specification artefacts, such as grammars and meta-models (that shift from *grammarware* to *modelware*), the thesis showed that there are certain languages (e.g. specification-based languages, such as OCL and QVT) for which either these bridges cannot be specified or the artefacts required to express them can be improved in different ways.

Supported by the research engineering method, this doctoral thesis proposed a new approach for providing support for CTMLs in order to overcome the limitations of related work. Partly driven by this EngD project scope (Section 1.3) and partly driven by the convenience of the technology when working on textual modeling languages (Section 2.3.5), Xtext is reused as part of the solution. However, Xtext grammars cannot address several CS2AS scenarios (Section 3.6) and, therefore, the new approach is divided into two steps (Section 4.1.1):

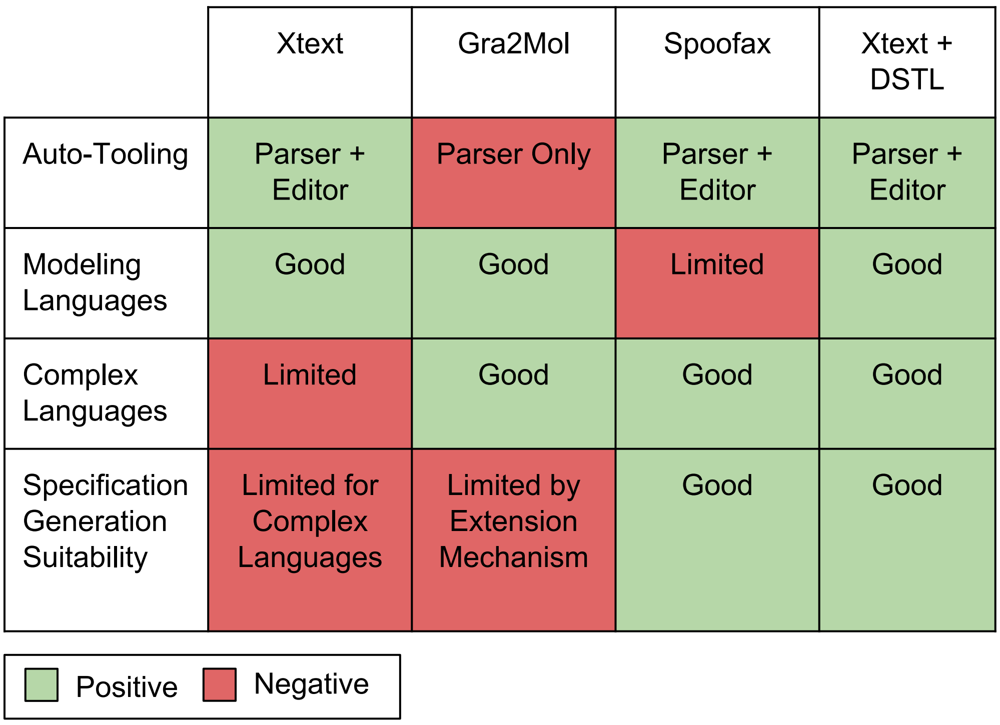
- Xtext is reused to achieve the *grammarware-to-modelware* technological space shift. In this way, a CS meta-model is automatically derived from any Xtext grammar, and the generated parser is responsible for producing CS models from textual inputs.
- Following the Gra2Mol approach, the solution provides an external DSTL to declare CS2AS bridges for CTMLs. However, this CS2AS-TL focuses on CS meta-models as the transformation source, rather than ANTLR grammars. This CS2AS-TL constitutes the main contribution of this thesis.

The proposed approach can address various CS2AS scenarios; for instance, in mOCL (Section 4.2.8) it was proven that CS2AS-TL can provide CS2AS bridges for a number of textual modeling languages (Sections 5.5 and 5.6).

Compared to related work, the following conclusions are stated. Firstly, taking into account the context (Section 1.1), motivation (Section 1.2) and scope (Section 1.3) of this research project, Figure 6.1 shows a high-level comparison of the relevant alternatives that support CTMLs:

- Xtext is an appropriate technology to use when producing tooling for textual modeling languages. However, it is not the most appropriate when working on complex languages: when the gap between the CS and AS is significant, Xtext specification artefacts (i.e. grammars) cannot directly support them.

- Spoofax is an appropriate technology to use when producing tooling for complex textual languages. However, it is not the most appropriate when working on modeling languages: several CS2AS scenarios are not supported.
- Gra2Mol is an appropriate technology to use when creating parsers for complex textual modeling languages. However, it is not the most appropriate when producing additional tooling such as editors: they are not supported. Besides, since several CS2AS scenarios require the use of their language extension mechanism (Java black-boxing), this solution is not suited to generate specification parts automatically.
- Xtext, complemented by CS2AS-TL, is a more appropriate solution to work on CTMLs. Since AS meta-models, CS grammars and CS2AS bridges are models (e.g. there is an underlying meta-model), MDE technologies can be used largely to generate end-user tools and parts of the specification automatically.



	Xtext	Gra2Mol	Spoofax	Xtext + DSTL
Auto-Tooling	Parser + Editor	Parser Only	Parser + Editor	Parser + Editor
Modeling Languages	Good	Good	Limited	Good
Complex Languages	Limited	Good	Good	Good
Specification Generation Suitability	Limited for Complex Languages	Limited by Extension Mechanism	Good	Good

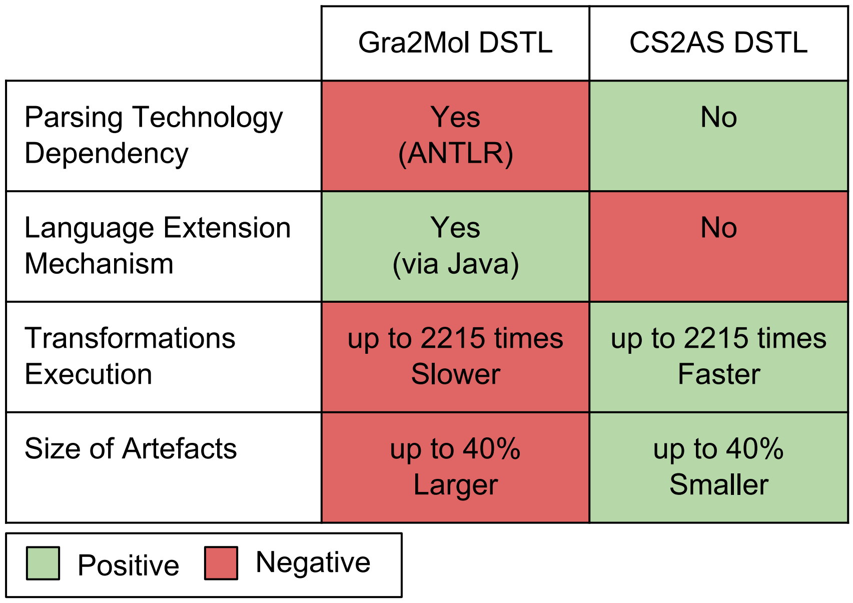
🟩 Positive 🟥 Negative

FIGURE 6.1: High-level Comparison of alternatives to support CTMLs

Figure 6.2 shows the relevant differences between the DSTLs of Gra2Mol[3] and the proposed solution. These differences are obtained from the qualitative (Section 5.2) and quantitative (Sections 5.5 and 5.6) studies:

[3] When comparing CS2AS-TL with other solutions, Gra2Mol is state-of-the-art with respect to bridging textual CS and AS meta-models.

- CS2AS-TL is independent of parsing technology, whereas Gra2Mol only works on ANTLR grammars. That said, with the proposed solution, it is mandatory that the chosen parser produces models conforming to a CS meta-model.
- Although the syntax of Gra2Mol is limited, its language extension mechanism can address various CS2AS scenarios, by invoking Java code. CS2AS-TL does not support this feature, which is a limitation when the information that the AS model requires is not present in (or computed from) the CS model.
- As shown in Section 5.5, the proposed solution improves Gra2Mol up to 40% when measuring the size of artefacts required to solve CS2AS scenarios.
- As shown in Section 5.6, the current prototype is up to 2215 faster than Gra2Mol when measuring the execution time of CS2AS transformations.



	Gra2Mol DSTL	CS2AS DSTL
Parsing Technology Dependency	Yes (ANTLR)	No
Language Extension Mechanism	Yes (via Java)	No
Transformations Execution	up to 2215 times Slower	up to 2215 times Faster
Size of Artefacts	up to 40% Larger	up to 40% Smaller

🟩 Positive 🟥 Negative

FIGURE 6.2: Comparison between the DSTLs of Gra2Mol and the proposed solution

To conclude, the following subsections discuss the research questions that were proposed in the introduction (see Section 1.4.2).

6.2.1 Industrial Perspective

From the point of view of the industrial partner, the following research questions are discussed:

Can a new alternative solution decrease the amount of hand-written code required to support CTMLs within Xtext?

The alternative solution consists of the proposed DSTL to define the CS2AS bridges needed for supporting CTMLs. The automatically generated Java classes producing AS models from CS models render Xtext-based tooling capable of supporting CTMLs. Given the declarative nature of the language and domain-specific constructs (e.g. the name resolution section), the amount of required hand-written artefacts is decreased. For instance, for the *101 Companies* example[4], the CS2AS bridge[5] consists of 34 lines of code, whereas the corresponding generated Java class[6] consists of 1114 lines of Java code.

If so, can this alternative solution also be used for automatically producing parts of OMG specifications?

The OMG OCL specification describes how the CS should be mapped to the AS (see clause 9.3 from [39]). The alternative solution consists of a new DSTL to declare CS2AS bridges. These bridges constitute a model that conforms to its underlying meta-model and, therefore, MDE techniques can be used to generate the CS2AS mappings declared in OMG specifications. For instance, the set of Complete OCL operations generated from instances of CS2AS-TL (see Section 4.3) could be used to define these CS2AS bridges in an OMG-like language.

6.2.2 Academic Perspective

From an academic point of view, the following research questions are discussed:

Do existing approaches address all the identified concerns that are required to support CTMLs?

Depending on the particular existing approach, the discussion varies:

- **Xtext** is used on textual modeling languages, but its specification artefacts (i.e. grammars) cannot address more complex CS2AS scenarios (e.g. multi-way CS2AS mappings).
- **Spoofax** is used on complex textual languages, but some CS2AS scenarios that arise from working on modeling languages cannot be addressed (e.g. AS models may refer to external models).
- **Gra2Mol** is used on complex textual modeling languages, but Gra2Mol needs its language extension mechanism (additional Java code) to support some CS2AS scenarios.

Can a new approach improve the performance (in terms of execution time) of existing approaches to produce AS models from textual files of CTMLs?

Yes, according to the quantitative study performed in this thesis (see Section 5.5), the new proposed approach performs better than Gra2Mol.

Can a new approach reduce the size of specification artefacts required by existing approaches to bridge the gap between the CS and AS of CTMLs?

Yes, according to the quantitative study performed in this thesis (see Section 5.6), the new proposed approach requires fewer manually written artefacts than Gra2Mol.

[4] `https://github.com/adolfosbh/cs2as`

[5] `/org.xtext.example.companies/model/companies.cs2as`

[6] `/org.xtext.example.companies/src-gen/org/xtext/example/mydsl/_companies_qvtp_qvtcas/companies_qvtp_qvtcas.java`

Appendix A

Mini-OCL CS grammar

```
grammar org.eclipse.qvtd.doc.MiniOCLCS with org.eclipse.xtext.common.Terminals
generate minioclcs "http://www.eclipse.org/qvtd/doc/MiniOCLCS"

RootCS:
    imports+=ImportCS∗
  (packages+=PackageCS
  | contraints+=ConstraintsDefCS)∗
;
ImportCS:
  'import' (alias=ID ':')? uri=STRING ';'
PackageCS:
  'package' name=ID '{'
    ( packages+=PackageCS
    | classes+=ClassCS)∗
  '}'
;
ClassCS:
  'class' name=ID ('extends' extends=PathNameCS)? '{'
    ( properties+=PropertyCS
    | operations+=OperationCS )∗
  '}'
;
PropertyCS:
  'prop'
  name=ID ':' typeRef=PathNameCS
  (multiplicity=MultiplicityCS)? ';'
;
MultiplicityCS:
  '['
  (opt?='?' | mult?='*' | (lowerInt=INT '..' (upperInt=INT | upperMult?='*'))
      )
  ']'
;
OperationCS:
  'op' name=ID
  '(' (params+=ParameterCS (','params+=ParameterCS)∗)? ')'
  ':' resultRef=PathNameCS
  '=' body=ExpCS
  ';'
;
ParameterCS:
  name=ID ':' typeRef=PathNameCS
;
ConstraintsDefCS:
  'context' typeRef=PathNameCS '{'
  (invariants+=InvariantCS)∗
  '}'
;
InvariantCS:
  'inv' ':' exp=ExpCS ';'
;
// Expressions
ExpCS:
  EqualityExpCS
;
EqualityExpCS:
  CallExpCS ({EqualityExpCS.left=current} opName=('=' | '<>') right=CallExpCS)∗
;
CallExpCS:
  PrimaryExpCS ({CallExpCS.source=current} opName=('.' | '->') navExp=
      NavigationExpCS)∗
;
PrimaryExpCS:
  SelfExpCS | NameExpCS | LiteralExpCS | LetExpCS
;
SelfExpCS:
   {SelfExpCS} 'self'
;
NavigationExpCS:
  LoopExpCS | NameExpCS
;
LoopExpCS:
  CollectExpCS | IterateExpCS
;
CollectExpCS:
  'collect' '(' (itVar=IteratorVarCS '|')? exp=ExpCS')'
;
IteratorVarCS:
  itName=ID (':' itType=PathNameCS)?
;
IterateExpCS:
  'iterate' '(' itVar=IteratorVarCS ';' accVar=AccVarCS '|' exp=ExpCS ')'
;
AccVarCS:
  accName=ID (':' accType=PathNameCS)? '=' accInitExp=ExpCS
;
NameExpCS:
  expName=PathNameCS
  (roundedBrackets=RoundedBracketClauseCS)?
```

```
;
RoundedBracketClauseCS:
  {RoundedBracketClauseCS}
  '('
  (args+=ExpCS (','args+=ExpCS)* )?
  ')'
;
LiteralExpCS:
  IntLiteralExpCS | BooleanLiteralExpCS | NullLiteralExpCS |
      CollectionLiteralExpCS
;
IntLiteralExpCS :
  intSymbol=INT
;
BooleanLiteralExpCS:
  {BooleanExpCS}
  (boolSymbol?='true'
  | 'false')
;
NullLiteralExpCS:
  {NullLiteralExpCS}
  'null'
;
enum CollectionKindCS:
  Collection='Collection'
;
CollectionLiteralExpCS:
  kind=CollectionKindCS '{'
  (parts+=CollectionLiteralPartCS)*
  '}'
;
CollectionLiteralPartCS:
   first =ExpCS
  ('..' last=ExpCS)?
;
LetExpCS:
  'let' letVars+=LetVarCS (',' letVars+=LetVarCS)*
  'in' inExp=ExpCS
;
LetVarCS:
  name=ID (':' typeRef=PathNameCS)? '=' initExp=ExpCS
;
PathNameCS :
  pathElements+=PathElementCS
  ('::'pathElements+=PathElementCS)*
;
PathElementCS:
  elementName=ID
;
```

LISTING A.1: Mini-OCL CS grammar

Appendix B

Mini-OCL AS meta-model

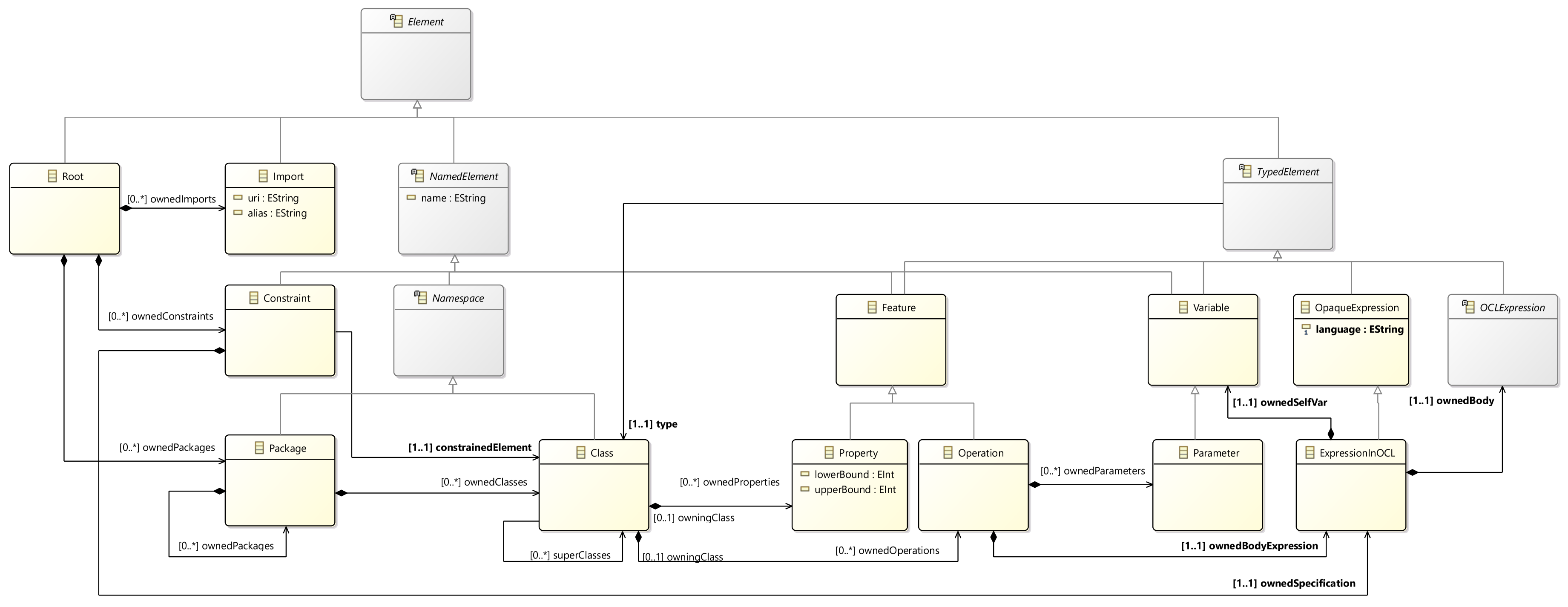


FIGURE B.1: MiniOCL AS Meta-Model: Main Structure

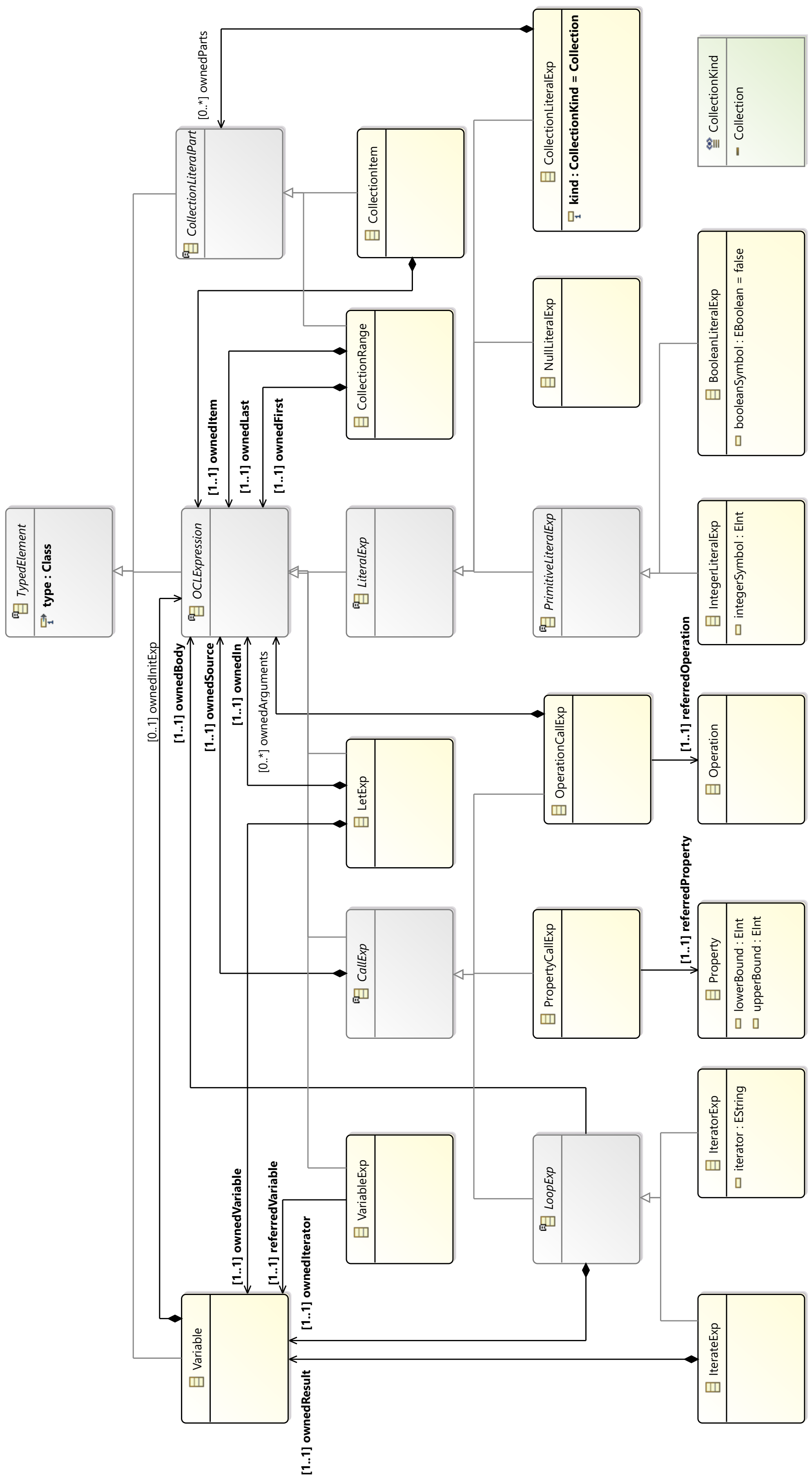


FIGURE B.2: MiniOCL AS Meta-Model: OCL Expressions

Appendix C

CS2AS bridge for Mini-OCL

```
source cs : 'generated/MiniOCLCS.ecore#/'
target as : '/resource/org.eclipse.qvtd.doc.miniocl/model/MiniOCL.
    ecore#/'

helpers {
  as::Class {
    commonSupertype(another : Class) : Class :=
      let allSupertypes = self->asOrderedSet()->closure(superClasses),
          allOtherSupertypes = another->asOrderedSet()->closure(superClasses)
      in allSupertypes->intersection(allOtherSupertypes)->any(true);
    conformsTo(another : Class) : Boolean :=
      self = another or superClasses->exists(conformsTo(another));
  }
  cs::NavigationExpCS {
    parentAsCallExpCS() : CallExpCS :=
      let container = self.oclContainer()
      in if container.oclIsKindOf(CallExpCS)
         then container.oclAsType(CallExpCS)
         else null
       endif;
  }
  cs::NameExpCS {
    isNavExpOfACallExpCS() : Boolean :=
      let parentCallExpCS = parentAsCallExpCS()
      in parentCallExpCS <> null and parentCallExpCS.navExp = self;
  }
  cs::PropertyCS {
    computeLowerBound() : Integer :=
      if multiplicity = null then 0
      else if multiplicity.opt then 0
           else if multiplicity.mult then 0
           else if multiplicity.mandatory <> 0 then multiplicity.mandatory
           else multiplicity.lowerInt
           endif endif endif
      endif;
    computeUpperBound() : Integer :=
      if multiplicity = null then 1
      else if multiplicity.opt then 1
```

```
                else if multiplicity.mult then −1
                else if multiplicity.mandatory <> 0 then multiplicity.mandatory
                else if multiplicity.upperMult then −1
                else multiplicity.upperInt
                endif endif endif endif
            endif;
    }
}
disambiguation {
    CollectionLiteralPartCS   {
        withoutLastExpression := last = null;
        withLastExpression := last <> null;
    }
    NameExpCS { −− Note: order of the disambiguation rules matters
        isOpCallExpWithExplicitSource :=
            roundedBrackets <> null and isNavExpOfACallExpCS();
        isOpCallExpWithImplicitSource :=
            roundedBrackets <> null and not isNavExpOfACallExpCS();
        isPropCallExpWithExplicitSource :=
            roundedBrackets = null and isNavExpOfACallExpCS();
        isVariableExp :=
            roundedBrackets = null and not isNavExpOfACallExpCS()
            and lookup(Variable, expName.pathElements−>first()) <> null;
        isPropCallExpWithImplicitSource :=
            roundedBrackets = null and not isNavExpOfACallExpCS()
            and lookup(Property, expName) <> null;
    }
    LetExpCS {
        singleVarDecl := letVars−>size() = 1;
        multipleVarDecls := letVars−>size() > 1;
    }
}
mappings {
    create Root from RootCS {
        ownedImports := imports.trace;
        ownedPackages := packages.trace;
        ownedConstraints := constraints.invariants.trace;
    }
    create Import from ImportCS {
        alias := if alias = null then null else alias endif;
        uri := uri;
    }
    create Constraint from InvariantCS {
        ownedSpecification := as::ExpressionInOCL {
                                    language = 'OCL',
                                    ownedBody = exp.trace,
                                    ownedSelfVar = as::Variable { name = 'self',
                                                        type = trace.constrainedElement }
                                  };
        constrainedElement := lookup(Class, self.ConstraintsDefCS.typeRef);
    }
```

```
create Package from PackageCS {
  name := name;
  ownedPackages := packages.trace;
  ownedClasses := classes . trace ;
}
create Class from ClassCS {
  name := name;
  ownedProperties := properties.trace;
  ownedOperations := operations.trace;
  superClasses := if _extends −>notEmpty()
                  then _extends−>collect(x | lookup(Class,x))
                  else OrderedSet{}
                  endif;
}
create Operation from OperationCS {
  name := name;
  type := lookup(Class, resultRef);
  ownedParameters := params.trace;
  ownedBodyExpression := as::ExpressionInOCL {
                           language = 'OCL',
                           ownedBody = body.trace,
                           ownedSelfVar = as::Variable {name = 'self',
                                                        type = trace .owningClass }
                         };
}
create Variable from ParameterCS {
  name := name;
  type := lookup(Class,typeRef);
}
create Property from PropertyCS {
  name := name;
  lowerBound := computeLowerBound();
  upperBound := computeUpperBound();
  type := lookup(Class, typeRef);
}
−− Expressions
refer CallExp from CallExpCS :=
  self .navExp.trace
;
create OperationCallExp from EqualityExpCS {
  ownedSource := left.trace;
  ownedArguments := right.trace;
  referredOperation := lookupExported(Operation, trace.ownedSource.type,
                                     opName, trace.ownedArguments);
  type := lookup(Class, 'Boolean');
}
create VariableExp from NameExpCS
when fall_back {
  referredVariable := null;
  type := lookup(Class, 'OclVoid');
}
```

```
create  VariableExp from NameExpCS
when isVariableExp {
  referredVariable  := lookup(Variable, expName.pathElements−>first());
  type :=  trace.referredVariable.type;
}
create  PropertyCallExp from NameExpCS
when isPropCallExpWithExplicitSource {
  ownedSource := parentAsCallExpCS()._source.trace;
  referredProperty := lookupExported(Property,trace.ownedSource.type,expName);
  type :=  trace.referredProperty?.type;
}
create  PropertyCallExp from NameExpCS
when isPropCallExpWithImplicitSource {
  ownedSource := let referredVar = lookup(Variable, 'self')
                 in  as :: VariableExp {
                        referredVariable  = referredVar,
                        type = referredVar.type
                     };
  referredProperty := lookupExported(Property,trace.ownedSource.type,expName);
  type :=  trace.referredProperty?.type;
}
create  OperationCallExp from NameExpCS
when isOpCallExpWithExplicitSource {
  ownedSource := parentAsCallExpCS()._source.trace;
  ownedArguments := roundedBrackets.args.trace;
  referredOperation := lookupExported(Operation,trace.ownedSource.type,
                                     expName, trace.ownedArguments);
  type :=  trace.referredOperation?.type;
}
create  OperationCallExp from NameExpCS
when isOpCallExpWithImplicitSource {
  ownedSource := let referredVar = lookup(Variable, 'self')
                 in  as :: VariableExp {
                        referredVariable  = referredVar,
                        type = referredVar.type
                     };
  ownedArguments := roundedBrackets.args.trace;
  referredOperation:= lookupExported(Operation,trace.ownedSource.type,
                                    expName, trace.ownedArguments);
  type :=  trace.referredOperation?.type;
}
create  LetExp from LetExpCS
when singleVarDecl {
  ownedVariable := letVars−>at(1).trace;
  ownedIn := inExp.trace;
  type :=  inExp.trace.type;
}
create  LetExp from LetExpCS
when multipleVarDecls {
  ownedVariable := letVars−>first().trace;
  ownedIn := letVars−>excluding(letVars−>first())−>reverse()
```

```
                          ->iterate(x:LetVarCS; acc: as::OCLExpression=inExp.trace |
                                          as::LetExp {
                                             ownedVariable = x.trace,
                                             ownedIn = acc,
                                             type = acc.type
                                          });
   type := inExp.trace.type;
 }
 create Variable from LetVarCS {
   name := name;
   ownedInitExp := initExp.trace;
   type := if typeRef <> null then lookup(Class,typeRef) else trace.ownedInitExp.
       type endif;
 }
 create IterateExp from IterateExpCS {
   ownedIterator := itVar.trace;
   ownedResult := accVar.trace;
   ownedBody := exp.trace;
   ownedSource := parentAsCallExpCS()._source.trace;
   type := trace.ownedResult.type;
 }
 create IteratorExp from CollectExpCS {
   iterator := 'collect';
   ownedIterator := if itVar = null
                   then as::Variable { name='self', type=lookup(Class,'
                       OclAny') }
                   else itVar.trace
                   endif;
   ownedBody := exp.trace;
   ownedSource := parentAsCallExpCS()._source.trace;
   type := lookup(Class,'Collection');
 }
 create Variable from IteratorVarCS {
   name := itName;
   type := if itType <> null then lookup(Class,itType) else lookup(Class,'OclAny
       ') endif;
 }
 create Variable from AccVarCS {
   name := accName;
   ownedInitExp := accInitExp.trace;
   type := if accType <> null then lookup(Class,accType) else trace.ownedInitExp.
       type endif;
 }
 create CollectionLiteralExp from CollectionLiteralExpCS {
   kind := kind;
   ownedParts := parts.trace;
   type := lookup(Class,'Collection');
 }
 create CollectionItem from CollectionLiteralPartCS
 when withoutLastExpression {
   ownedItem := first.trace;
```

```
    type := trace.ownedItem.type;
  }
  create CollectionRange from CollectionLiteralPartCS
  when withLastExpression {
    ownedFirst := first.trace;
    ownedLast := last.trace;
    type := trace.ownedFirst.type;
  }
}
name_resolution {
  targets {
    NamedElement using name escaped_with '_';
    Package qualifies Package using ownedPackages,
                      Class using ownedClasses;
    Class qualifies Operation using ownedOperations,
                    Property using ownedProperties;
    Operation filtered_by arguments : OrderedSet(OCLExpression)
              when ownedParameters->size() = arguments->size() and
                   arguments->forAll(x |
                      let argIdx = arguments->indexOf(x)
                      in x.type.conformsTo(ownedParameters->at(argIdx).type));
    Property;
    Variable;
  }
  inputs {
    PathElementCS using elementName;
    qualifier PathNameCS using pathElements;
  }
  providers {
    Root {
      in current_scope provides adding
        Package using ownedPackages, exported ownedImports;
    }
    Import {
      in exported_scope provides
        Package using loaded Package from uri;
    }
    Package {
      in current_scope provides occluding
        Package using ownedPackages
        Class using ownedClasses;
    }
    Class {
      vars allSuperClasses = self->closure(superClasses);
      in current_scope provides occluding
        Operation using ownedOperations occluding allSuperClasses.
            ownedOperations
        Property using ownedProperties occluding allSuperClasses.ownedProperties;
      in exported_scope provides
        Operation using ownedOperations occluding allSuperClasses.
            ownedOperations
            Property using ownedProperties occluding allSuperClasses.ownedOperations
                ;
        }
        Operation {
          in current_scope provides occluding
            Variable using ownedParameters;
        }
        ExpressionInOCL {
          in current_scope provides occluding
            Variable using ownedSelfVar;
        }
        LetExp {
          in current_scope
            for all excepting ownedVariable provides occluding
              Variable using ownedVariable;
        }
        IteratorExp {
          in current_scope provides occluding
            Variable using ownedIterator;
        }
        IterateExp {
          in current_scope
            for ownedBody provides occluding
               Variable using ownedIterator, ownedResult;
        }
      }
    }
```

LISTING C.1: CS2AS bridge for Mini-OCL

Appendix D

Complete OCL documents of the Mini-OCL CS2AS bridge

```
import cs : 'generated/MiniOCLCS.ecore#/'
import as : '/resource/org.eclipse.qvtd.doc.miniocl/model/MiniOCL.
    ecore#/'
package cs

context cs::NavigationExpCS
def : parentAsCallExpCS() :CallExpCS =
  let container  = self.oclContainer()
  in if container.oclIsKindOf(CallExpCS) then container.oclAsType(CallExpCS) else
      null endif
context cs::NameExpCS
def : isNavExpOfACallExpCS() :Boolean =
  let parentCallExpCS = parentAsCallExpCS()
  in parentCallExpCS <> null and parentCallExpCS.navExp = self

context cs::PropertyCS
def : computeLowerBound() :Integer =
  if multiplicity = null then 0
  else if multiplicity.opt then 0
    else if multiplicity.mult then 0
      else if multiplicity.mandatory <> 0 then multiplicity.mandatory
        else multiplicity.lowerInt endif endif endif endif
def : computeUpperBound() :Integer =
  if multiplicity = null then 1
  else if multiplicity.opt then 1
    else if multiplicity.mult then -1
      else if multiplicity.mandatory <> 0 then multiplicity.mandatory
        else if multiplicity.upperMult then - 1
          else multiplicity.upperInt endif endif endif endif endif
endpackage
package as

context as::Class
def : commonSupertype(another : Class) :Class =
 let allSupertypes = self->asOrderedSet()->closure(superClasses)
```

```
        , allOtherSupertypes = another->asOrderedSet()->closure(superClasses)
  in allSupertypes->intersection(allOtherSupertypes)->any(true)
def : conformsTo(another : Class) :Boolean =
  self = another or superClasses->exists(conformsTo(another))

endpackage
```

LISTING D.1: MiniOCLHelpers.ocl

```
import cs : 'generated/MiniOCLCS.ecore#/'
import as : '/resource/org.eclipse.qvtd.doc.miniocl/model/MiniOCL.
    ecore#/'
import 'MiniOCLFullHelpers.ocl'
import 'MiniOCLFullLookup.ocl'
package cs

context CollectionLiteralPartCS
def : withoutLastExpression() : Boolean =
   last = null
def : withLastExpression() : Boolean =
   last <> null
context NameExpCS
def : isOpCallExpWithExplicitSource() : Boolean =
   roundedBrackets <> null and isNavExpOfACallExpCS()
def : isOpCallExpWithImplicitSource() : Boolean =
   roundedBrackets <> null and not isNavExpOfACallExpCS()
def : isPropCallExpWithExplicitSource() : Boolean =
   roundedBrackets = null and isNavExpOfACallExpCS()
def : isVariableExp() : Boolean =
   roundedBrackets = null and not isNavExpOfACallExpCS() and ast.lookupVariable(
        expName.pathElements->first()) <> null
def : isPropCallExpWithImplicitSource() : Boolean =
   roundedBrackets = null and not isNavExpOfACallExpCS() and ast.lookupProperty(
        expName) <> null
context LetExpCS
def : singleVarDecl() : Boolean =
   letVars->size() = 1
def : multipleVarDecls() : Boolean =
   letVars->size() > 1
endpackage
```

LISTING D.2: MiniOCLDisambiguation.ocl

```
import cs : 'generated/MiniOCLCS.ecore#/'
import as : '/resource/org.eclipse.qvtd.doc.miniocl/model/MiniOCL.
    ecore#/'
import '/resource/org.eclipse.qvtd.doc.miniocl/model/Lookup.ecore
    '
import 'MiniOCLFullHelpers.ocl'

package ocl
```

```

-- Domain specific default functionality
context OclElement
def : unqualified_env_Class() : lookup::LookupEnvironment[1] =
  _unqualified_env_Class(null)
def : unqualified_env_Package() : lookup::LookupEnvironment[1] =
  _unqualified_env_Package(null)
def : unqualified_env_Operation() : lookup::LookupEnvironment[1] =
  _unqualified_env_Operation(null)
def : unqualified_env_Variable() : lookup::LookupEnvironment[1] =
  _unqualified_env_Variable(null)
def : unqualified_env_NamedElement() : lookup::LookupEnvironment[1] =
  _unqualified_env_NamedElement(null)
def : unqualified_env_Property() : lookup::LookupEnvironment[1] =
  _unqualified_env_Property(null)

def : _unqualified_env_Class(child : OclElement) : lookup::LookupEnvironment[1] =
  parentEnv_Class()
def : _unqualified_env_Package(child : OclElement) : lookup::LookupEnvironment
    [1] =
  parentEnv_Package()
def : _unqualified_env_Operation(child : OclElement) : lookup::LookupEnvironment
    [1] =
  parentEnv_Operation()
def : _unqualified_env_Variable(child : OclElement) : lookup::LookupEnvironment
    [1] =
  parentEnv_Variable()
def : _unqualified_env_NamedElement(child : OclElement) : lookup::
    LookupEnvironment[1] =
  parentEnv_NamedElement()
def : _unqualified_env_Property(child : OclElement) : lookup::LookupEnvironment
    [1] =
  parentEnv_Property()

def : parentEnv_Class() : lookup::LookupEnvironment[1] =
  let parent = oclContainer() in if parent = null then lookup::LookupEnvironment {
      } else parent._unqualified_env_Class(self) endif
def : parentEnv_Package() : lookup::LookupEnvironment[1] =
  let parent = oclContainer() in if parent = null then lookup::LookupEnvironment {
      } else parent._unqualified_env_Package(self) endif
def : parentEnv_Operation() : lookup::LookupEnvironment[1] =
  let parent = oclContainer() in if parent = null then lookup::LookupEnvironment {
      } else parent._unqualified_env_Operation(self) endif
def : parentEnv_Variable() : lookup::LookupEnvironment[1] =
  let parent = oclContainer() in if parent = null then lookup::LookupEnvironment {
      } else parent._unqualified_env_Variable(self) endif
def : parentEnv_NamedElement() : lookup::LookupEnvironment[1] =
  let parent = oclContainer() in if parent = null then lookup::LookupEnvironment {
      } else parent._unqualified_env_NamedElement(self) endif
def : parentEnv_Property() : lookup::LookupEnvironment[1] =
  let parent = oclContainer() in if parent = null then lookup::LookupEnvironment {
      } else parent._unqualified_env_Property(self) endif

def : _exported_env_Property(importer : OclElement) : lookup::LookupEnvironment
    [1] =
  lookup::LookupEnvironment { }
def : _exported_env_Package(importer : OclElement) : lookup::LookupEnvironment
    [1] =
  lookup::LookupEnvironment { }
def : _exported_env_Operation(importer : OclElement) : lookup::
    LookupEnvironment[1] =
  lookup::LookupEnvironment { }

def : _qualified_env_Class(qualifier : OclElement) : lookup::LookupEnvironment[1]
    =
  lookup::LookupEnvironment { }
def : _qualified_env_Package(qualifier : OclElement) : lookup::LookupEnvironment
    [1] =
  lookup::LookupEnvironment { }
def : _qualified_env_Operation(qualifier : OclElement) : lookup::
    LookupEnvironment[1] =
  lookup::LookupEnvironment { }
def : _qualified_env_Property(qualifier : OclElement) : lookup::LookupEnvironment
    [1] =
  lookup::LookupEnvironment { }
-- End of domain specific default functionality
endpackage

package lookup
-- Some common lookup functionality
context LookupEnvironment
def : nestedEnv() : LookupEnvironment[1] =
  LookupEnvironment {
    parentEnv = self
  }
-- End of common lookup functionality
endpackage

package as
context Element
-- NamedElement unqualified lookup
def : _lookupNamedElement(env : lookup::LookupEnvironment, nName : String) :
    OrderedSet(NamedElement) =
  let foundNamedElement = env.namedElements->selectByKind(NamedElement)
      ->select(name = nName)
  in if foundNamedElement->isEmpty() and not (env.parentEnv = null)
     then _lookupNamedElement(env.parentEnv, nName)
     else foundNamedElement
     endif
def : _lookupUnqualifiedNamedElement(nName : String) : NamedElement[?] =
   let foundNamedElement = _lookupNamedElement(
        unqualified_env_NamedElement(), nName)
  in if foundNamedElement->isEmpty()
     then null
     else foundNamedElement->first() -- LookupVisitor will report ambiguous result
     endif

def : lookupNamedElement(nName : String) : NamedElement[?] =
  _lookupUnqualifiedNamedElement(nName)
def : lookupNamedElement(aPathElementCS : cs::PathElementCS) : NamedElement
     [?] =
  _lookupUnqualifiedNamedElement(aPathElementCS.elementName)
-- End of NamedElement unqualified lookup

context Package

def : _lookupQualifiedPackage(pName : String) : Package[?] =
   let foundPackage = _lookupPackage(_qualified_env_Package(), pName)
  in if foundPackage->isEmpty()
     then null
     else foundPackage->first()
     endif
def : _qualified_env_Package() : lookup::LookupEnvironment =
   let env = lookup::LookupEnvironment{}
  in env
        .addElements(ownedPackages)
def : lookupQualifiedPackage(aPathElementCS : cs::PathElementCS) : Package[?] =
  _lookupQualifiedPackage(aPathElementCS.elementName)

def : _lookupQualifiedClass(cName : String) : Class[?] =
   let foundClass = _lookupClass(_qualified_env_Class(), cName)
  in if foundClass->isEmpty()
     then null
     else foundClass->first()
     endif
def : _qualified_env_Class() : lookup::LookupEnvironment =
   let env = lookup::LookupEnvironment{}
  in env
        .addElements(ownedClasses)
def : lookupQualifiedClass(aPathElementCS : cs::PathElementCS) : Class[?] =
  _lookupQualifiedClass(aPathElementCS.elementName)
context Element
-- Package unqualified lookup
def : _lookupPackage(env : lookup::LookupEnvironment, pName : String) :
     OrderedSet(Package) =
   let foundPackage = env.namedElements->selectByKind(Package)->select(name =
        pName)
  in  if foundPackage->isEmpty() and not (env.parentEnv = null)
      then _lookupPackage(env.parentEnv, pName)
      else foundPackage
      endif
```

```
def : _lookupUnqualifiedPackage(pName : String) : Package[?] =
  let foundPackage = _lookupPackage(unqualified_env_Package(), pName)
  in if foundPackage−>isEmpty()
     then null
     else foundPackage−>first() −− LookupVisitor will report ambiguous result
     endif

def : lookupPackage(pName : String) : Package[?] =
  _lookupUnqualifiedPackage(pName)
def : lookupPackage(aPathElementCS : cs::PathElementCS) : Package[?] =
  _lookupUnqualifiedPackage(aPathElementCS.elementName)
−− End of Package unqualified lookup

−− Package qualified −name lookup
def : lookupPackage(aPathNameCS : cs::PathNameCS) : Package[?] =
  lookupPackage(aPathNameCS .pathElements)

def : lookupPackage(segments : OrderedSet(cs::PathElementCS)) : Package[?] =
  if segments−>size() = 1
  then lookupPackage(segments−>first())
  else let qualifierSegments = segments−>subOrderedSet(1,segments−>size()−1),
           qualifier = lookupPackage(qualifierSegments)
       in qualifier ?.lookupQualifiedPackage(segments−>last())
  endif

context Class

def : _lookupQualifiedOperation(oName : String, arguments : OrderedSet(
    OCLExpression)) : Operation[?] =
  let foundOperation = _lookupOperation(_qualified_env_Operation(), oName,
      arguments)
  in if foundOperation−>isEmpty()
     then null
     else foundOperation−>first()
     endif
def : _qualified_env_Operation() : lookup::LookupEnvironment =
  let env = lookup::LookupEnvironment{}
  in env
       .addElements(ownedOperations)
def : lookupQualifiedOperation(aPathElementCS : cs::PathElementCS, arguments :
    OrderedSet(OCLExpression)) : Operation[?] =
  _lookupQualifiedOperation(aPathElementCS.elementName, arguments)

def : _lookupQualifiedProperty(pName : String) : Property[?] =
  let foundProperty = _lookupProperty(_qualified_env_Property(), pName)
  in if foundProperty−>isEmpty()
     then null
     else foundProperty−>first()
     endif
def : _qualified_env_Property() : lookup::LookupEnvironment =
  let env = lookup::LookupEnvironment{}
  in env
        .addElements(ownedProperties)
def : lookupQualifiedProperty(aPathElementCS : cs::PathElementCS) : Property[?] =
  _lookupQualifiedProperty(aPathElementCS.elementName)
context Element
-- Class unqualified lookup
def : _lookupClass(env : lookup::LookupEnvironment, cName : String) : OrderedSet(
    Class) =
  let foundClass = env.namedElements->selectByKind(Class)->select(name =
      cName)
  in if foundClass->isEmpty() and not (env.parentEnv = null)
     then _lookupClass(env.parentEnv, cName)
     else foundClass
     endif
def : _lookupUnqualifiedClass(cName : String) : Class[?] =
  let foundClass = _lookupClass(unqualified_env_Class(), cName)
  in  if foundClass->isEmpty()
  then null
  else foundClass->first() -- LookupVisitor will report ambiguous result
  endif

def : lookupClass(cName : String) : Class[?]  =
  _lookupUnqualifiedClass(cName)
def : lookupClass(aPathElementCS : cs::PathElementCS) : Class[?] =
  _lookupUnqualifiedClass(aPathElementCS.elementName)
-- End of Class unqualified lookup

-- Class qualified -name lookup
def : lookupClass(aPathNameCS : cs::PathNameCS) : Class[?] =
  lookupClass(aPathNameCS .pathElements)

def : lookupClass(segments : OrderedSet(cs::PathElementCS)) : Class[?] =
  if segments->size() = 1
  then lookupClass(segments->first())
  else let qualifierSegments = segments->subOrderedSet(1,segments->size()-1),
           qualifier  = lookupPackage(qualifierSegments)
       in qualifier ?.lookupQualifiedClass(segments->last())
  endif

context Operation

def : _appliesFilter_Operation(arguments : OrderedSet(OCLExpression)) : Boolean =
  ownedParameters->size() = arguments->size() and arguments->forAll(x |
     let argIdx  = arguments->indexOf(x)
     in x.type.conformsTo(ownedParameters->at(argIdx).type)
)
context Element
-- Operation unqualified lookup
def : _lookupOperation(env : lookup::LookupEnvironment, oName : String,
    arguments : OrderedSet(OCLExpression)) : OrderedSet(Operation) =
  let foundOperation = env.namedElements->selectByKind(Operation)->select(
      name = oName)
                                      ->select(_appliesFilter_Operation(arguments))
  in if foundOperation->isEmpty() and not (env.parentEnv = null)
     then _lookupOperation(env.parentEnv, oName, arguments)
     else foundOperation
     endif
def : _lookupUnqualifiedOperation(oName : String, arguments : OrderedSet(
    OCLExpression)) : Operation[?] =
  let foundOperation = _lookupOperation(unqualified_env_Operation(), oName,
      arguments)
  in if foundOperation->isEmpty()
     then null
     else foundOperation->first() -- LookupVisitor will report ambiguous result
     endif

def : lookupOperation(oName : String, arguments : OrderedSet(OCLExpression)) :
    Operation[?] =
  _lookupUnqualifiedOperation(oName, arguments)
def : lookupOperation(aPathElementCS : cs::PathElementCS, arguments : OrderedSet
    (OCLExpression)) : Operation[?] =
  _lookupUnqualifiedOperation(aPathElementCS.elementName, arguments)
-- End of Operation unqualified lookup

-- Operation qualified-name lookup
def : lookupOperation(aPathNameCS : cs::PathNameCS, arguments : OrderedSet(
    OCLExpression)) : Operation[?] =
  lookupOperation(aPathNameCS .pathElements, arguments)

def : lookupOperation(segments : OrderedSet(cs::PathElementCS), arguments :
    OrderedSet(OCLExpression)) : Operation[?] =
  if segments->size() = 1
  then lookupOperation(segments->first(), arguments)
  else let qualifierSegments = segments->subOrderedSet(1,segments->size()-1),
           qualifier = lookupClass(qualifierSegments)
       in qualifier ?.lookupQualifiedOperation(segments->last(), arguments)
       endif
context Element
-- Property unqualified lookup
def : _lookupProperty(env : lookup::LookupEnvironment, pName : String) :
    OrderedSet(Property) =
  let foundProperty = env.namedElements->selectByKind(Property)->select(name
      = pName)
  in if foundProperty->isEmpty() and not (env.parentEnv = null)
     then _lookupProperty(env.parentEnv, pName)
     else foundProperty
     endif
def : _lookupUnqualifiedProperty(pName : String) : Property[?] =
  let foundProperty = _lookupProperty(unqualified_env_Property(), pName)
  in if foundProperty->isEmpty()
     then null
```

```
        else foundProperty->first() -- LookupVisitor will report ambiguous result
        endif

def : lookupProperty(pName : String) : Property[?] =
  _lookupUnqualifiedProperty(pName)
def : lookupProperty(aPathElementCS : cs::PathElementCS) : Property[?] =
  _lookupUnqualifiedProperty(aPathElementCS.elementName)
-- End of Property unqualified lookup

-- Property qualified -name lookup
def : lookupProperty(aPathNameCS : cs::PathNameCS) : Property[?] =
  lookupProperty(aPathNameCS .pathElements)

def : lookupProperty(segments : OrderedSet(cs::PathElementCS)) : Property[?] =
   if segments->size() = 1
  then lookupProperty(segments->first())
  else let qualifierSegments = segments->subOrderedSet(1,segments->size()-1),
           qualifier  = lookupClass(qualifierSegments)
       in qualifier ?.lookupQualifiedProperty(segments->last())
  endif
context Element
-- Variable unqualified lookup
def : _lookupVariable(env : lookup::LookupEnvironment, vName : String) :
    OrderedSet(Variable) =
   let foundVariable = env.namedElements->selectByKind(Variable)->select(name =
      vName)
  in if foundVariable->isEmpty() and not (env.parentEnv = null)
      then _lookupVariable(env.parentEnv, vName)
      else foundVariable
      endif
def : _lookupUnqualifiedVariable(vName : String) : Variable[?] =
   let foundVariable = _lookupVariable(unqualified_env_Variable(), vName)
  in if foundVariable->isEmpty()
      then null
      else foundVariable->first() -- LookupVisitor will report ambiguous result
      endif

def : lookupVariable(vName : String) : Variable[?]  =
  _lookupUnqualifiedVariable(vName)
def : lookupVariable(aPathElementCS : cs::PathElementCS) : Variable[?] =
  _lookupUnqualifiedVariable(aPathElementCS.elementName)
-- End of Variable unqualified lookup

context Root
def : _unqualified_env_Package(child : ocl::OclElement) : lookup::
    LookupEnvironment =
  parentEnv_Package()
     .addElements(ownedPackages)
     .addElements(ownedImports._exported_env_Package(self).namedElements)

context Import
```

```
def : _exported_env_Package(importer : ocl::OclElement) : lookup::
    LookupEnvironment =
  let env = lookup::LookupEnvironment {}
  in env
       .addElements(OrderedSet{/*AutoGenCode Will Load The Package*/})

def : _lookupExportedPackage(importer : ocl::OclElement, pName : String) : Package
    [?] =
  let foundPackage = _lookupPackage(_exported_env_Package(importer), pName)
   in if foundPackage->isEmpty()
      then null
      else foundPackage->first()
      endif
def : lookupExportedPackage(importer : ocl::OclElement, pName : String) : Package
    [?] =
  _lookupExportedPackage(importer, pName)
def : lookupExportedPackage(importer : ocl::OclElement, aPathElementCS : cs::
    PathElementCS) : Package[?] =
  _lookupExportedPackage(importer, aPathElementCS.elementName)

context Element
-- Import exports Package
def : lookupPackageFrom(exporter : Import , iName : String) : Package[?] =
  exporter.lookupExportedPackage(self, iName)
def : lookupPackageFrom(exporter : Import, aPathElementCS : cs::PathElementCS) :
    Package[?] =
  exporter.lookupExportedPackage(self, aPathElementCS)
def : lookupPackageFrom(exporter : Import, aPathNameCS : cs::PathNameCS) :
    Package[?] =
  lookupPackageFrom(exporter, aPathNameCS.pathElements)

def : lookupPackageFrom(exporter : Import, segments : OrderedSet(cs::
    PathElementCS)) : Package[?] =
  if segments->size() = 1
  then lookupPackageFrom(exporter, segments->first())
  else let qualifierSegments = segments->subOrderedSet(1,segments->size()-1),
             qualifier  = lookupPackage(qualifierSegments)
       in qualifier ?.lookupQualifiedPackage(segments->last())
  endif
context Package
def : _unqualified_env_Package(child : ocl::OclElement) : lookup::
    LookupEnvironment =
  parentEnv_Package().nestedEnv()
    .addElements(ownedPackages)

def : _unqualified_env_Class(child : ocl :: OclElement) : lookup::LookupEnvironment
     =
  parentEnv_Class().nestedEnv()
    .addElements(ownedClasses)

context Class
```

```
def : _unqualified_env_Operation(child : ocl::OclElement) : lookup::
    LookupEnvironment =
  let allSuperClasses = self->closure(superClasses)
  in
     parentEnv_Operation().nestedEnv()
        .addElements(allSuperClasses.ownedOperations)
        .nestedEnv()
        .addElements(ownedOperations)

def : _unqualified_env_Property(child : ocl :: OclElement) : lookup::
    LookupEnvironment =
  let allSuperClasses = self->closure(superClasses)
  in
     parentEnv_Property().nestedEnv()
        .addElements(allSuperClasses.ownedProperties)
        .nestedEnv()
        .addElements(ownedProperties)

def : _exported_env_Operation(importer : ocl::OclElement) : lookup::
    LookupEnvironment =
  let allSuperClasses = self->closure(superClasses)
  in
    let env = lookup::LookupEnvironment {}
    in env
          .addElements(allSuperClasses.ownedOperations)
          .nestedEnv()
          .addElements(ownedOperations)

def : _exported_env_Property(importer : ocl::OclElement) : lookup::
    LookupEnvironment =
  let allSuperClasses = self->closure(superClasses)
  in
    let env = lookup::LookupEnvironment {}
    in env
          .addElements(allSuperClasses.ownedOperations)
          .nestedEnv()
          .addElements(ownedProperties)

def : _lookupExportedOperation(importer : ocl::OclElement, oName : String,
    arguments : OrderedSet(OCLExpression)) : Operation[?] =
  let foundOperation = _lookupOperation(_exported_env_Operation(importer),
      oName, arguments)
  in if foundOperation->isEmpty()
     then null
     else foundOperation->first()
     endif
def : lookupExportedOperation(importer : ocl::OclElement, oName : String,
    arguments : OrderedSet(OCLExpression)) : Operation[?] =
  _lookupExportedOperation(importer, oName, arguments)
def : lookupExportedOperation(importer : ocl::OclElement, aPathElementCS : cs::
    PathElementCS, arguments : OrderedSet(OCLExpression)) : Operation[?] =
  _lookupExportedOperation(importer, aPathElementCS.elementName, arguments)

def : _lookupExportedProperty(importer : ocl::OclElement, pName : String) :
    Property[?] =
   let foundProperty = _lookupProperty(_exported_env_Property(importer), pName)
  in if foundProperty->isEmpty()
      then null
      else foundProperty->first()
      endif
def : lookupExportedProperty(importer : ocl::OclElement, pName : String) : Property
    [?] =
  _lookupExportedProperty(importer, pName)
def : lookupExportedProperty(importer : ocl::OclElement, aPathElementCS : cs::
    PathElementCS) : Property[?] =
  _lookupExportedProperty(importer, aPathElementCS.elementName)

context Element
-- Class exports Operation
def : lookupOperationFrom(exporter : Class , cName : String, arguments : OrderedSet
    (OCLExpression)) : Operation[?] =
  exporter.lookupExportedOperation(self, cName, arguments)
def : lookupOperationFrom(exporter : Class, aPathElementCS : cs::PathElementCS,
    arguments : OrderedSet(OCLExpression)) : Operation[?] =
  exporter.lookupExportedOperation(self, aPathElementCS, arguments)
def : lookupOperationFrom(exporter : Class, aPathNameCS : cs::PathNameCS,
    arguments : OrderedSet(OCLExpression)) : Operation[?] =
  lookupOperationFrom(exporter, aPathNameCS.pathElements, arguments)

def : lookupOperationFrom(exporter : Class, segments : OrderedSet(cs::
    PathElementCS), arguments : OrderedSet(OCLExpression)) : Operation[?] =
   if segments->size() = 1
  then lookupOperationFrom(exporter, segments->first(), arguments)
  else let qualifierSegments = segments->subOrderedSet(1,segments->size()-1),
             qualifier = lookupClass(qualifierSegments)
        in qualifier ?.lookupQualifiedOperation(segments->last(), arguments)
  endif
-- Class exports Property
def : lookupPropertyFrom(exporter : Class , cName : String) : Property[?] =
  exporter.lookupExportedProperty(self, cName)
def : lookupPropertyFrom(exporter : Class, aPathElementCS : cs::PathElementCS) :
    Property[?] =
  exporter.lookupExportedProperty(self, aPathElementCS)
def : lookupPropertyFrom(exporter : Class, aPathNameCS : cs::PathNameCS) :
    Property[?] =
  lookupPropertyFrom(exporter, aPathNameCS.pathElements)

def : lookupPropertyFrom(exporter : Class, segments : OrderedSet(cs::PathElementCS
    )) : Property[?] =
   if segments->size() = 1
  then lookupPropertyFrom(exporter, segments->first())
  else let qualifierSegments = segments->subOrderedSet(1,segments->size()-1),
                qualifier  = lookupClass(qualifierSegments)
          in  qualifier ?.lookupQualifiedProperty(segments−>last())
    endif
context Operation
def :  _unqualified_env_Variable(child : ocl :: OclElement) : lookup::
      LookupEnvironment =
   parentEnv_Variable().nestedEnv()
      .addElements(ownedParameters)

context ExpressionInOCL
def :  _unqualified_env_Variable(child : ocl :: OclElement) : lookup::
      LookupEnvironment =
   parentEnv_Variable().nestedEnv()
       .addElements(ownedSelfVar)

context LetExp
def :  _unqualified_env_Variable(child : ocl :: OclElement) : lookup::
      LookupEnvironment =
   if  not (ownedVariable−>includes(child))
   then parentEnv_Variable().nestedEnv()
      .addElements(ownedVariable)
   else  parentEnv_Variable()
   endif

context IteratorExp
def :  _unqualified_env_Variable(child : ocl :: OclElement) : lookup::
      LookupEnvironment =
   parentEnv_Variable().nestedEnv()
      .addElements(ownedIterator)

context IterateExp
def :  _unqualified_env_Variable(child : ocl :: OclElement) : lookup::
      LookupEnvironment =
   if  ownedBody−>includes(child)
   then parentEnv_Variable().nestedEnv()
      .addElements(ownedIterator)
      .addElements(ownedResult)
   else  parentEnv_Variable()
   endif

endpackage
```

Listing D.3: MiniOCLNameResolution.ocl

```
import cs : 'generated/MiniOCLCS.ecore#/'
import as : '/resource/org.eclipse.qvtd.doc.miniocl/model/MiniOCL.
      ecore#/'
import 'MiniOCLFullHelpers.ocl'
import 'MiniOCLFullLookup.ocl'
import 'MiniOCLFullDisambiguation.ocl'

```

```
package cs
context RootCS
def : ast() : as::Root =
  as::Root {
    ownedImports = imports.ast(),
    ownedPackages = packages.ast(),
    ownedConstraints = constraints.invariants.ast()
  }
context ImportCS
def : ast() : as::Import =
  as::Import {
    alias = if alias = null then null else alias endif,
    uri = uri
  }
context InvariantCS
def : ast() : as::Constraint =
  as::Constraint {
    ownedSpecification = as::ExpressionInOCL {
        language = 'OCL' ,
        ownedBody = exp.ast(),
        ownedSelfVar = as::Variable {
          name = 'self' ,
          type = ast().constrainedElement
        }
    },
    constrainedElement = ast().lookupClass(self.ConstraintsDefCS.typeRef)
  }
context PackageCS
def : ast() : as::Package =
  as::Package {
    name = name,
    ownedPackages = packages.ast(),
    ownedClasses = classes.ast()
  }
context ClassCS
def : ast() : as::Class =
  as::Class {
    name = name,
    ownedProperties = properties.ast(),
    ownedOperations = operations.ast(),
    superClasses = if _extends->notEmpty() then _extends->collect(x | ast().
        lookupClass(x)) else OrderedSet { } endif
}
context OperationCS
def : ast() : as::Operation =
  as::Operation {
    name = name,
    type = ast().lookupClass(resultRef),
    ownedParameters = params.ast(),
    ownedBodyExpression = as::ExpressionInOCL {
        language = 'OCL' ,
        ownedBody = _body.ast(),
        ownedSelfVar = as::Variable {
          name = 'self' ,
          type = ast () .owningClass
        }
    }
}
context ParameterCS
def : ast () : as :: Variable =
  as :: Variable {
    name = name,
    type = ast () .lookupClass(typeRef)
  }
context PropertyCS
def : ast () : as :: Property =
  as :: Property {
    name = name,
    lowerBound = computeLowerBound(),
    upperBound = computeUpperBound(),
    type = ast () .lookupClass(typeRef)
  }
context CallExpCS
def : ast () : as :: CallExp =
  self .navExp.ast()
context ExpCS
def : ast () : as :: OCLExpression =
  null -- to be overriden
context EqualityExpCS
def : ast () : as :: OperationCallExp =
  as :: OperationCallExp {
    ownedSource = left.ast() ,
    ownedArguments = right.ast(),
    referredOperation = ast () .lookupOperationFrom(ast().ownedSource.type,
        opName, ast().ownedArguments),
    type = ast () .lookupClass('Boolean' )
  }
context NavigationExpCS
def : ast () : as :: CallExp =
  null -- to be overriden
context NameExpCS
def : ast () : as :: OCLExpression =
  if isOpCallExpWithExplicitSource()
  then as :: OperationCallExp {
    ownedSource = parentAsCallExpCS()._source.ast(),
    ownedArguments = roundedBrackets.args.ast(),
    referredOperation = ast () .lookupOperationFrom(ast().oclAsType(as::
        OperationCallExp).ownedSource.type, expName, ast().oclAsType(as::
        OperationCallExp).ownedArguments),
    type = ast () .oclAsType(as::OperationCallExp).referredOperation ?. type
  }
  else
```

```
    if isOpCallExpWithImplicitSource()
    then as::OperationCallExp {
       ownedSource = let referredVar = ast().lookupVariable('self')
           in as::VariableExp {
              referredVariable = referredVar,
              type = referredVar.type
           }
      ,
      ownedArguments = roundedBrackets.args.ast(),
      referredOperation = ast().lookupOperationFrom(ast().oclAsType(as::
          OperationCallExp).ownedSource.type, expName, ast().oclAsType(as::
          OperationCallExp).ownedArguments),
      type = ast().oclAsType(as::OperationCallExp).referredOperation?.type
    }
    else
      if isPropCallExpWithExplicitSource()
      then as::PropertyCallExp {
         ownedSource = parentAsCallExpCS()._source.ast(),
         referredProperty = ast().lookupPropertyFrom(ast().oclAsType(as::
             PropertyCallExp).ownedSource.type, expName),
         type = ast().oclAsType(as::PropertyCallExp).referredProperty?.type
      }
      else
         if isVariableExp()
         then as::VariableExp {
            referredVariable = ast().lookupVariable(expName.pathElements->first()
                ),
            type = ast().oclAsType(as::VariableExp).referredVariable.type
         }
         else
           if isPropCallExpWithImplicitSource()
           then as::PropertyCallExp {
              ownedSource = let referredVar = ast().lookupVariable('self')
                  in as::VariableExp {
                     referredVariable = referredVar,
                     type = referredVar.type
                  }
              ,
              referredProperty = ast().lookupPropertyFrom(ast().oclAsType(as::
                  PropertyCallExp).ownedSource.type, expName),
              type = ast().oclAsType(as::PropertyCallExp).referredProperty?.type
           }
           else
              as::VariableExp {
                 referredVariable = null,
                 type = ast().lookupClass('OclVoid')
              }
           endif
         endif
      endif
     endif
```

```
   endif
context LetExpCS
def : ast() : as::LetExp =
  if singleVarDecl()
  then as::LetExp {
     ownedVariable = letVars->at(1).ast(),
     ownedIn = inExp.ast(),
     type = inExp.ast().type
  }
  else
  if multipleVarDecls()
  then as::LetExp {
     ownedVariable = letVars->first().ast(),
     ownedIn = letVars->excluding(letVars->first())->reverse()->iterate(x :
         LetVarCS ; acc : as::OCLExpression = inExp.ast() |
        as::LetExp {
          ownedVariable = x.ast(),
          ownedIn = acc,
          type = acc.type
        }),
     type = inExp.ast().type
  }
  else
    invalid
  endif
endif
context LetVarCS
def : ast() : as::Variable =
  as::Variable {
    name = name,
    ownedInitExp = initExp.ast(),
    type = if typeRef <> null then ast().lookupClass(typeRef) else ast().
        ownedInitExp.type endif
  }
context IterateExpCS
def : ast() : as::IterateExp =
  as::IterateExp {
    ownedIterator = itVar.ast(),
    ownedResult = accVar.ast(),
    ownedBody = exp.ast(),
    ownedSource = parentAsCallExpCS()._source.ast(),
    type = ast().ownedResult.type
  }
context CollectExpCS
def : ast() : as::IteratorExp =
  as::IteratorExp {
    iterator = 'collect',
    ownedIterator = if itVar = null then as::Variable {
           name = 'self',
           type = ast().lookupClass('OclAny')
         } else itVar.ast() endif,
        ownedBody = exp.ast(),
        ownedSource = self.parentAsCallExpCS()._source.ast(),
        type = ast().lookupClass('Collection')
    }
context IteratorVarCS
def : ast() : as::Variable =
    as::Variable {
        name = itName,
        type = if itType <> null then ast().lookupClass(itType) else ast().lookupClass('
            OclAny') endif
    }
context AccVarCS
def : ast() : as::Variable =
    as::Variable {
        name = accName,
        ownedInitExp = accInitExp.ast(),
        type = if accType <> null then ast().lookupClass(accType) else ast().
            ownedInitExp.type endif
    }
context CollectionLiteralExpCS
def : ast() : as::CollectionLiteralExp =
    as::CollectionLiteralExp {
        kind = kind,
        ownedParts = parts.ast(),
        type = ast().lookupClass('Collection')
    }
context CollectionLiteralPartCS
def : ast() : as::CollectionLiteralPart =
    if withoutLastExpression()
    then as::CollectionItem {
        ownedItem = first.ast(),
        type = ast().oclAsType(as::CollectionItem).ownedItem.type
    }
    else
        if withLastExpression()
        then as::CollectionRange {
            ownedFirst = first.ast(),
            ownedLast = last.ast(),
            type = ast().oclAsType(as::CollectionRange).ownedFirst.type
        }
        else
            invalid
        endif
    endif
endpackage
```

LISTING D.4: MiniOCLCS2AS.ocl

Appendix E

Using Spoofax To Support The 101 Companies Example

The source code of the following example is publicly available in a GitHub repository[a], in particular, within the *org.spoofax.example.companies* and *org.spoofax.example.companies.as* projects.

[a] `https://github.com/adolfosbh/cs2as`

This appendix shows how one of the modeling language examples introduced in the evaluation (see Chapter 5) is supported by Spoofax [46].

As with CS2AS-TL and Gra2Mol tools, the following subsections focus on the different specification artefacts that are required to support this language example within Spoofax. These specification artefacts (shown in the forthcoming sections) are:

- A grammar of the textual modeling language, using their SDF language.
- A name resolution definition, using their NaBL language.
- A *treeware*-based CS2AS definition, using their Stratego/XT language.
- The modified AS meta-model to exploit the *EAnnotation*-based mechanism that allows Spoofax to do the *treeware-to-modelware* technological space shift.

The version of Spoofax that has been used is 1.5.0. There is a more recent version (2.0.0) under development for which the support to modeling languages has been discontinued (according to one of the project moderators[a]).

[a] `https://www.linkedin.com/in/guidowachsmuth`

E.1 Grammar Definition

Listing E.1 shows the example's grammar definition. Note that some grammar rules have been adequately modified (when compared to the original Gra2Mol grammar definition) to avoid bloating CS2AS mappings that would have been used for renaming a syntax tree term (e.g. rename employee as Employee). Since the specification of an employee mentor is optional (line 23), a *Mentor* specification has to be defined in its own grammar rule (lines 26–27).

```
1  Start .Company = <
2    company <STRING> {
3      <Dept∗>
4    }
5  >
6  Dept.Department = <
7    department <STRING> {
8      <DeptManager>
9      <DeptEmployee∗>
10     <Dept∗>
11   }
12 >
13 DeptManager.DeptManager = <
14   manager <Employee>
15 >
16 DeptEmployee.DeptEmployee = <
17   employee <Employee>
18 >
19 Employee.Employee = <
20   <STRING> {
21     address <STRING>
22     salary  <FLOAT>
23     <Mentor?>
24   }
25 >
26 Mentor.Mentor = <
27   mentor <STRING>
28 >
```

LISTING E.1: 101 Companies example grammar definition using SDF

E.2 Name Resolution Definition

Listing E.2 shows the example's name resolution definition. Note that *Employee* and *Mentor* are different terms, the former being the referenced term and the latter the referrer one. The name resolution is separately defined on them, and the further CS2AS Stratego definition (next section) is responsible for mixing the two terms in just one. In this way, *Employee* terms refer to other *Employee* terms.

```
module names
imports
  include/CompaniesTest
namespaces
  Employee
```

```
binding rules
  Employee(e,_,_,_) :
    defines Employee e
  Mentor(m):
    refers  to  Employee m
```

LISTING E.2: Name resolution definition for the 101 Companies Example

E.3 CS2AS Definition

In the context of Spoofax, the goal of a CS2AS transformation is to create syntax trees that can be directly transformed (1-to-1 mappings) into AS models. Given the CS grammar definition (Listing E.1) and the AS meta-model (presented in Subsection 5.5.1, Figure 5.11), the following two CS2AS scenarios need to be solved:

- According to the CS grammar, a *Department* term contains a *DeptManager* and a list of *DeptEmployee* sub-terms (lines 8–9). These two sub-terms likewise contain an *Employee* sub-term (lines 14 and 17). According to the AS meta-model, a *Department* contains (via two different containment references) many *Employees*. To solve this misalignment between the initial syntax trees and the target AS models, two rewrite rules are needed to remove these intermediate *DeptManager* and *DeptEmployee* terms from the tree.
- According to the CS, an *Employee* term contains a *Mentor* sub-term (line 23). According to the AS, *Employee* model elements refer to their employee mentors, via the *Employee::mentor* cross-reference. To solve this misalignment between the initial syntax trees and the target AS models, a rewrite rule is needed to remove the intermediate *Mentor* terms from the tree.

Listing E.3 shows the example's CS2AS definition written in Stratego. The listing shows the required rewrite rules that were mentioned above:

- The first rewrite rule (lines 5–6) rewrites *DeptManager* terms as the first sub-term that it comprises, i.e. an *Employee* sub-term.
- The second rewrite rule (lines 7–8) rewrites *DeptEmployee* terms as the first sub-term that it comprises, i.e. an *Employee* sub-term.
- The third rewrite rule (lines 9–10) rewrites *Mentor* terms as the first sub-term that it comprises, i.e. a reference to another *Employee* (this reference is computed via name resolution).

```
1 module cs2as
2 imports
3   include/CompaniesTest
4 rules
5   CS2AS:
    DeptManager(manager) −> manager
  CS2AS:
    DeptEmployee(employee) −> employee
  CS2AS:
    Mentor(mentor) −> mentor
```

LISTING E.3: CS2AS definition for the 101 Companies Example

⚠ Note that this CS2AS transformation is performed in the *treeware* technological space. The following section depicts the *treeware-to-modelware* technological space shift.

E.4 *Treeware-to-Modelware* Mapping Definition

Finally, Figure E.1 shows how the *EAnnotations*-based mechanism (explained in Subsection 5.3.1) is used to solve the *treeware-to-modelware* technological space shift.

E.5 Experiment Results

Although Spoofax has been removed from the quantitative study, this final section informally provides some overall data related to the metrics used in the quantitative study.

E.5.1 Size of artefacts

This subsection informally provides some information with hints about the size of the Spoofax specification artefacts. The same tool developed for the quantitative study was used to measure the instances of the Spoofax languages required to declare a CS2AS bridge. With respect to the modified AS meta-model, the following manual calculation has been conducted[1]:

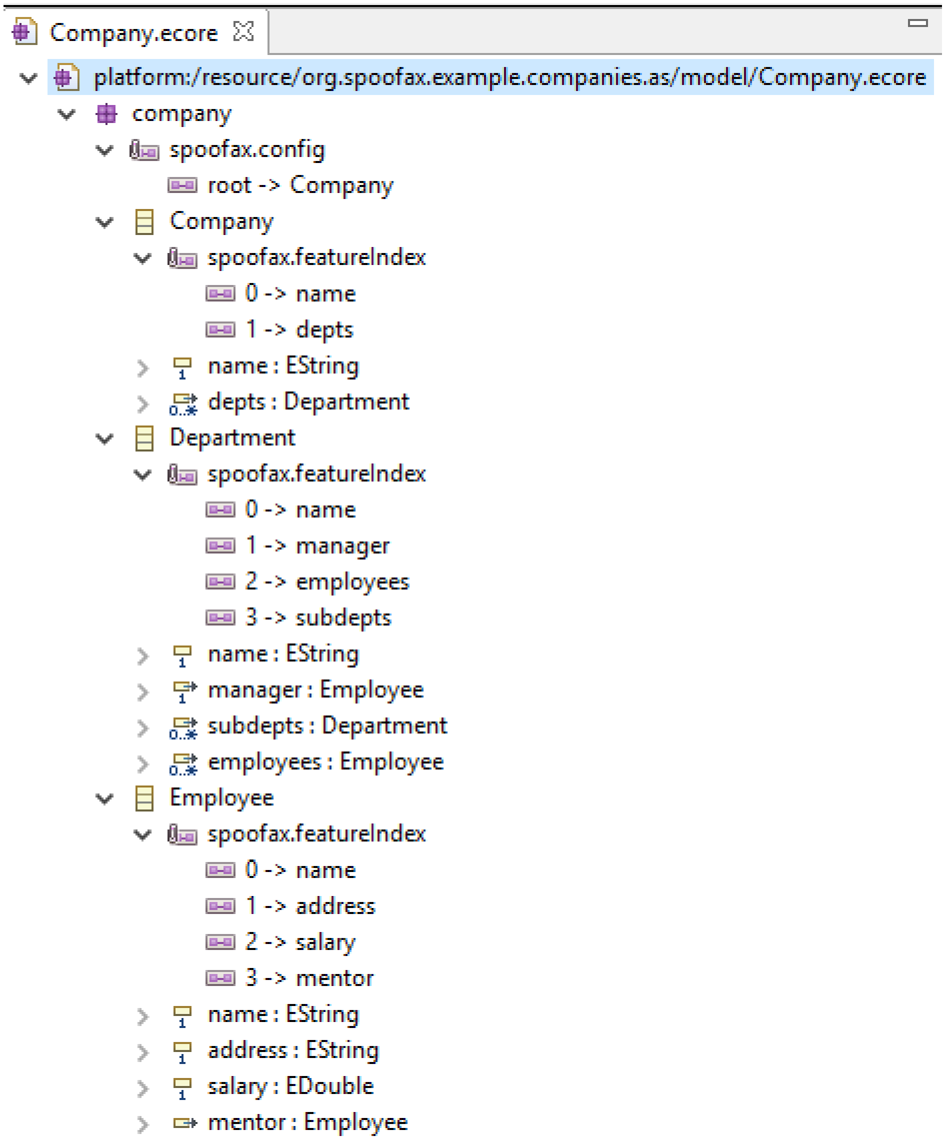
- Two words per meta-class, particularly for the required *EAnnotation*'s source.
- Two "words" per *EAnnotation*'s detail entry.

Table E.1 shows the size of the specification artefacts for the 101 Companies example that were required within Spoofax.

	Number of words	File bytes
Name Resolution in NaBL	23	267
CS2AS in Stratego	18	128
AS meta-model modifications	30	?
Total	71	?

TABLE E.1: Size (see Section 5.4.2) of the required artefacts specified within Spoofax

[1] No estimation has been attempted for the corresponding number of bytes of the involved Ecore file.



```
Company.ecore
platform:/resource/org.spoofax.example.companies.as/model/Company.ecore
  company
    spoofax.config
      root -> Company
    Company
      spoofax.featureIndex
        0 -> name
        1 -> depts
      name : EString
      depts : Department
    Department
      spoofax.featureIndex
        0 -> name
        1 -> manager
        2 -> employees
        3 -> subdepts
      name : EString
      manager : Employee
      subdepts : Department
      employees : Employee
    Employee
      spoofax.featureIndex
        0 -> name
        1 -> address
        2 -> salary
        3 -> mentor
      name : EString
      address : EString
      salary : EDouble
      mentor : Employee
```

FIGURE E.1: Mapping sub-term indexes to property names of the *Company* meta-model, through the *EAnnotations*-based mechanism

E.5.2 Implementation performance

This subsection informally provides some information with hints about Spoofax performance. In this case, the Spoofax modelware source code has been modified[2] in order to measure the amount of time required to load an *emf* model from the corresponding textual input.

The experiment environment and input files are the same as the ones that were used for the quantitative study experiment (see Subsection 5.5.1). Table E.2 shows the same information that characterises the experiment input models (see Subsection 5.5.1), with the addition of a column containing the execution time required to perform the AS model production.

	Size (bytes)	Elements	N_d	N_s	N_e	D_s	Execution time (ms)
M_1	1,238	22	3	0	3	1	643
M_2	6,105	97	3	3	4	2	1560
M_3	149,951	701	1	1	3	100	4033
M_4	42,805	708	1	100	3	2	3828
M_5	223,848	3061	4	4	5	4	16407
M_6	1,018,254	11901	10	4	10	4	44748
M_7	9,794,276	109341	10	5	10	5	-

TABLE E.2: Execution time (ms) required by Spoofax implementation to produce the AS models.

The execution time for the last M_7 model is not presented, because Spoofax failed to produce the expected output AS model.

[2] `https://github.com/adolfosbh/spoofax-modelware`

Appendix F

Experiments Data

	Size (bytes)	Elements	Gra2Mol (ms)	CS2AS-TL (ms)	Factor
M_1	1,238	22	2	4	-2
M_2	6,105	97	20	5	4
M_3	149,951	701	1856	30	61.86
M_4	42,805	708	396	36	11
M_5	223,848	3061	9876	482	20.48
M_6	1,018,254	11901	256346	10648	24.07
M_7	9,794,276	109341	19710767	1560768	12.63

TABLE F.1: Hypothesis A: *101 Companies* example (Experiment 1)

			Gra2Mol	CS2AS-TL	
	Size (bytes)	Elements	Exec. Time (ms)	No cached operations Exec. Time (ms)	Cached operations Exec. Time (ms)
M_1	1,320	31	3	7	4
M_2	3,003	49	7	7	7
M_3	6,219	79	27	9	9
M_4	9,648	115	50	11	11
M_5	16,309	181	83	18	15
M_6	25,082	271	162	32	23
M_7	39,539	409	372	52	27
M_8	60,132	619	831	94	55
M_9	93,307	931	1897	207	123
M_{10}	141,758	1405	4301	425	222
M_{11}	214,483	2119	10110	914	498
M_{12}	326,499	3193	25941	2242	1219
M_{13}	495,625	4819	65475	5308	2678
M_{14}	751,672	7261	154293	11714	6267
M_{15}	1,141,188	10945	349712	27656	14389
M_{16}	1,728,189	16501	784860	61942	34202
M_{17}	2,620,513	24877	1607842	165619	90753
M_{18}	3,956,386	37501	3583335	421534	258086
M_{19}	5,968,356	56533	9210046	1051139	637879
M_{20}	9,083,583	85225	21296415	2681990	1408042
M_{21}	13,773,378	128473	42248951	6451952	3419606
M_{22}	20,852,400	193663	96065259	15794135	7653003
M_{23}	31,524,191	291931	-	51963587	21170844
M_{24}	47,602,981	440071	-	-	-
M_{25}	72,039,789	663379	-	-	-

TABLE F.2: Hypothesis A: *101 Companies* example (Experiment 2)

	Size (bytes)	Lines of Code	Gra2Mol (ms)	CS2AS-TL (ms)	Factor
M_1	1,304	58	17	16	1.06
M_2	1,858	81	43	31	1.38
M_3	7,246	263	187	63	2.96
M_4	66,048	2,238	20400	98	108.16
M_5	241,188	8,118	287633	355	810.23
M_6	708,228	23,798	2450487	1106	2215.63

TABLE F.3: Hypothesis A: *Delphi2ASTM* example

	Gra2Mol			CS2AS-TL		
	Words	LoC	Bytes	Words	LoC	Bytes
$Lang_1$	83	35	757	73	33	738
$Lang_2$	1151	565	13,414	704	289	7,898
$Lang_3$	376	150	5,071	224	86	2,411
$Lang_4$	47	28	420	32	15	306
$Lang_5$	541	225	6,086	353	151	4212

TABLE F.4: Hypothesis B: Size of specification artefacts

Glossary

ABNF Augmented Backus Naur Form.

ADM Architecture Driven Modernization.

ANTLR Another Tool for Language Recognition.

AS Abstract Syntax.

ASG Abstract Syntax Graph.

AST Abstract Syntax Tree.

ASTM Abstract Syntax Tree Metamodel.

ATL ATL Transformation Language.

BNF Backus Naur Form.

CIM Computation-Independent Model.

CS Concrete Syntax.

CS2AS Concrete Syntax to Abstract Syntax.

CS2AS-TL Concrete Syntax to Abstract Syntax Transformation Language.

CST Concrete Syntax Tree.

CTML Complex Textual Modeling Language.

DDL Data Description Language.

DSL Domain Specific Language.

DSTL Domain Specific Transformation Language.

EBNF Extended Backus Naur Form.

Eclipse A platform that provide IDEs for varied languages.

EMF Eclipse Modeling Framework.

EngD Engineering Doctorate.

ETL Epsilon Transformation Language.

GPL General Purpose Language.

IDE Integrated Development Environment.

iMac Internet Macros.

JDT Java Development Tools.

KDM Knowledge Discovery Metamodel.

LHS Left Hand Side.

LoC Lines of Code.

LSCITS Large-Scale and Complex IT Systems.

LW Language Workbench.

M2M Model-to-Model.

M2T Model-to-Text.

MBE Model-Based Engineering.

MD* Model-Driven*.

MDA Model-Driven Architecture.

MDD Model-Driven Development.

MDE Model-Driven Engineering.

MDSD Model-Driven Software Development.

mOCL Mini OCL.

MOF Meta-Object Facility.

NaBL Names Binding Language.

OCL Object Constraint Language.

OMG Object Management Group.

OOP Object Oriented Programming.

PIM Platform Independent Model.

PSM Platform Specific Model.

QVT Query/View/Transformation.

QVTc QVT Core.

QVTd QVT Declarative.

QVTm QVT Minimal.

QVTo QVT Operational Mappings.

QVTr QVT Relations.

RHS Right Hand Side.

SDF Syntax Definition Formalism.

SE Software Engineering.

T2M Text-to-Model.

UML Unified Modeling Language.

URI Uniform Resource Identifier.

Bibliography

[1] Alfred V. Aho et al. *Compilers: principles, techniques, & tools*. Pearson Education Inc., 2007.

[2] Marcus Alanen and Ivan Porres. *A Relation Between Context-Free Grammars and Meta Object Facility Metamodels*. Technical Report. 2003.

[3] Victor R Basili. "The experimental paradigm in software engineering". In: *Experimental Software Engineering Issues: Critical Assessment and Future Directions*. Springer, 1993, pp. 1–12.

[4] Lorenzo Bettini. "Implementing Java-like languages in Xtext with Xsemantics". In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM. 2013, pp. 1559–1564.

[5] Jean Bézivin. "In search of a basic principle for model driven engineering". In: *Novatica Journal, Special Issue* 5.2 (2004), pp. 21–24.

[6] Jean Bézivin. "Model driven engineering: An emerging technical space". In: *Generative and transformational techniques in software engineering*. Springer, 2006, pp. 36–64.

[7] Jean Bézivin. "On the unification power of models". In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188.

[8] Jean Bézivin et al. "Model Transformations? Transformation Models!" In: *Model Driven Engineering Languages and Systems: 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006. Proceedings*. Ed. by Oscar Nierstrasz et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 440–453. ISBN: 978-3-540-45773-2. DOI: 10.1007/11880240_31. URL: http://dx.doi.org/10.1007/11880240_31.

[9] Harold Boley et al. "Declarative and procedural paradigms-do they really compete?" In: *Processing Declarative Knowledge*. Springer, 1991, pp. 383–398.

[10] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven Software Engineering in Practice*. Vol. 1. Morgan & Claypool Publishers, 2012.

[11] Achim D Brucker et al. "Report on the Aachen OCL meeting". In: *Proceedings of the MODELS 2013 OCL Workshop*. Vol. 1092. CEUR Workshop Proceedings. 2014. DOI: arXivpreprintarXiv:1408.5698. URL: http://ceur-ws.org/Vol-1092/aachen.pdf.

[12] Jordi Cabot and Martin Gogolla. “Object Constraint Language (OCL): A Definitive Guide”. In: *Formal Methods for Model-Driven Engineering: 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2012, Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*. Ed. by Marco Bernardo, Vittorio Cortellessa, and Alfonso Pierantonio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 58–90. ISBN: 978-3-642-30982-3. DOI: `10.1007/978-3-642-30982-3_3`. URL: `http://dx.doi.org/10.1007/978-3-642-30982-3_3`.

[13] Javier Luis Cánovas Izquierdo and Jesús García Molina. “A Domain Specific Language for Extracting Models in Software Modernization”. In: *Model Driven Architecture - Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 82–97. ISBN: 978-3-642-02674-4. DOI: `10.1007/978-3-642-02674-4_7`. URL: `http://dx.doi.org/10.1007/978-3-642-02674-4_7`.

[14] Philippe Charles et al. “Accelerating the Creation of Customized, language-Specific IDEs in Eclipse”. In: *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’09. Orlando, Florida, USA: ACM, 2009, pp. 191–206. ISBN: 978-1-60558-766-0. DOI: `10.1145/1640089.1640104`. URL: `http://doi.acm.org/10.1145/1640089.1640104`.

[15] James Clark, Steve DeRose, et al. *XML path language (XPath) version 1.0*. 1999.

[16] James Cordy. *Cross Language Transformations in TXL*. Tech. rep. 1995. URL: `http://txl.ca/docs/TXLcross.pdf`.

[17] Jesús Cuadrado Sánchez, Jesús Molina García, and Marcos Menarguez Tortosa. “RubyTL: A Practical, Extensible Transformation Language”. In: *Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006. Proceedings*. Ed. by Arend Rensink and Jos Warmer. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 158–172. ISBN: 978-3-540-35910-4. DOI: `10.1007/11787044_13`. URL: `http://dx.doi.org/10.1007/11787044_13`.

[18] Krzysztof Czarnecki and Simon Helsen. “Classification of model transformation approaches”. In: *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. Vol. 45. 3. 2003, pp. 1–17.

[19] *EMF Text*. On-Line. URL: `http://www.emftext.org/`.

[20] Sebastian Erdweg et al. “The state of the art in language workbenches”. In: *Software Language Engineering*. Springer, 2013, pp. 197–217.

[21] Moritz Eysholdt and Heiko Behrens. “Xtext: Implement Your Language Faster Than the Quick and Dirty Way”. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. SPLASH ’10. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 307–309. ISBN: 978-1-4503-0240-1. DOI: `10.1145/1869542.1869625`. URL: `http://doi.acm.org/10.1145/1869542.1869625`.

[22] Jean-Marie Favre et al. “101companies: a community project on software technologies and software languages”. In: *Objects, Models, Components, Patterns*. Springer, 2012, pp. 58–74.

[23] MetaBorg Software Foundation. *Java implementation of a scannerless generalised LR parser*. On-Line. URL: `http://strategoxt.org/Stratego/JSGLR/`.

[24] The Eclipse Foundation. *Eclipse OCL*. On-Line. 2005. URL: `http://projects.eclipse.org/projects/modeling.mdt.ocl`.

[25] The Eclipse Foundation. *Eclipse Platform*. On-Line. 2004. URL: `http://www.eclipse.org/`.

[26] The Eclipse Foundation. *Eclipse QVTd*. On-Line. 2007. URL: `https://projects.eclipse.org/projects/modeling.mmt.qvtd`.

[27] The Eclipse Foundation. *Eclipse QVTo*. On-Line. 2007. URL: `http://projects.eclipse.org/projects/modeling.mmt.qvt-oml`.

[28] The Eclipse Foundation. *Eclipse Simultaneous Relase Train*. On-Line. URL: `http://wiki.eclipse.org/Simultaneous_Release`.

[29] The Eclipse Foundation. *MoDisco*. On-Line. URL: `http://www.eclipse.org/MoDisco/`.

[30] The Eclipse Foundation. *Xtext*. On-Line. URL: `http://www.eclipse.org/Xtext/`.

[31] Martin Fowler. *Domain-Specific Languages*. Pearson Education, 2010.

[32] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Languange*. Addison-Wesley Professional, 2004.

[33] Mārtiņš Francis et al. “Adding Spreadsheets to the MDE Toolkit”. In: *Model-Driven Engineering Languages and Systems: 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 – October 4, 2013. Proceedings*. Ed. by Ana Moreira et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 35–51. ISBN: 978-3-642-41533-3. DOI: `10.1007/978-3-642-41533-3_3`. URL: `http://dx.doi.org/10.1007/978-3-642-41533-3_3`.

[34] Jesús García Molina et al. *Desarrollo de Software Dirigido por Modelos: Concepts, Métodos y Herramientas*. Ed. by Ra-Ma. Ra-Ma, 2013, p. 586.

[35] Object Management Group. *Abstract Syntax Tree Metamodel (ASTM), V1.0*. OMG Document: formal/2011-01-05.pdf (`http://www.omg.org/spec/ASTM/1.0/`). Jan. 2011.

[36] Object Management Group. *Knowledge Discovery Meta-Model (KDM), V1.4*. OMG Document: ptc/16-02-03 (`http://www.omg.org/spec/KDM/1.4/`). Jan. 2016. URL: `http://www.omg.org/spec/KDM/1.4/`.

[37] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation V1.1*. OMG Document: formal/2011-01-01 (`http://www.omg.org/spec/QVT/1.1`). Jan. 2011. URL: `http://www.omg.org/spec/QVT/1.1`.

[38] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation V1.2*. OMG Document: ptc/2014-03-38 (`http://www.omg.org/spec/QVT/1.2`). May 2014. URL: `http://www.omg.org/spec/QVT/1.1`.

[39] Object Management Group. *Object Constraint Language (OCL), V2.4*. OMG Document: ptc/2013-08-13 (`http://www.omg.org/spec/OCL/2.4`). Jan. 2013. URL: `http://www.omg.org/spec/OCL/2.4`.

[40] Object Management Group. *Unified Modeling Language (UML), V2.5*. OMG Document: ptc/2012-10-24 (`http://www.omg.org/spec/UML/2.5`). 2012. URL: `http://www.omg.org/spec/UML/2.5`.

[41] D. Harel and B. Rumpe. "Meaningful modeling: what's the semantics of "semantics"?" In: *Computer* 37.10 (Oct. 2004), pp. 64–72. ISSN: 0018-9162. DOI: `10.1109/MC.2004.172`.

[42] Javier Luis Cánovas Izquierdo and Jesús García Molina. "Extracting models from source code in software modernization". In: *Software & Systems Modeling* 13 (2012), pp. 1–22.

[43] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev. "TCS:: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering". In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: ACM, 2006, pp. 249–254. ISBN: 1-59593-237-2. DOI: `10.1145/1173706.1173744`. URL: `http://doi.acm.org/10.1145/1173706.1173744`.

[44] Frédéric Jouault et al. "ATL: a QVT-like transformation language". In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM. 2006, pp. 719–720.

[45] Frédéric Jouault et al. "Towards Functional Model Transformations with OCL". In: *Theory and Practice of Model Transformations*. Springer, 2015, pp. 111–120.

[46] Lennart CL Kats and Eelco Visser. "The spoofax language workbench: rules for declarative specification of languages and IDEs". In: *ACM Sigplan Notices*. Vol. 45. 10. ACM. 2010, pp. 444–463.

[47] Lennart C.L. Kats, Eelco Visser, and Guido Wachsmuth. "Pure and Declarative Syntax Definition: Paradise Lost and Regained". In: *Proceedings of the ACM International Conference OOPSLA '10*. Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 918–932. ISBN: 978-1-4503-0203-6. DOI: `10.1145/1869459.1869535`. URL: `http://doi.acm.org/10.1145/1869459.1869535`.

[48] Anneke G Kleppe, Jos Warmer, and Wim Bast. *MDA Explained. The model driven architecture: practice and promise*. Addison Wesley Reading, 2003.

[49] Paul Klint, Ralf Lämmel, and Chris Verhoef. "Toward an engineering discipline for grammarware". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14.3 (2005), pp. 331–380.

[50] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. "Eclipse development tools for epsilon". In: *Eclipse Summit Europe, Eclipse Modeling Symposium*. Vol. 20062. 2006, p. 200.

[51] Dimitrios S Kolovos, Richard F Paige, and Fiona AC Polack. "The epsilon transformation language". In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2008, pp. 46–60.

[52] Dimitrios S. Kolovos et al. "Programmatic Muddle Management". In: *Proceedings of the Workshop on Extreme Modeling at MoDELS'13*. Ed. by Juan de Lara, Davide Di Ruscio, and Alfonso Pierantonio. Vol. 1089. urn:nbn:de:0074-1089-2. CEUR Workshop Proceedings. 2013. URL: http://ceur-ws.org/Vol-1089/1.pdf.

[53] Gabriël Konat et al. "Declarative Name Binding and Scope Rules". In: *Software Language Engineering*. Vol. 7745. Springer Berlin Heidelberg, 2013, pp. 311–331.

[54] I. Kurtev, J. Bézivin, and M. Akşit. "Technological Spaces: An Initial Appraisal". In: *International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, Irvine, USA*. Irvine, USA, Oct. 2002, pp. 1–6. ISBN: not assigned.

[55] *LALR Parser Generator (LPG)*. On-Line. URL: http://sourceforge.net/projects/lpg/.

[56] Joaquin Miller, Jishnu Mukerji, et al. "MDA Guide Version 1.0.1". In: *Object Management Group* 234 (2003), p. 51.

[57] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. "Weaving executability into object-oriented meta-languages". In: *Model Driven Engineering Languages and Systems*. Springer, 2005, pp. 264–278.

[58] Pierre-Alain Muller et al. "Model-driven analysis and synthesis of concrete syntax". In: *Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 98–110.

[59] Pierre-Alain Muller et al. "Model-driven analysis and synthesis of textual concrete syntax". In: *Software & Systems Modeling* 7.4 (2008), pp. 423–441. ISSN: 1619-1374. DOI: 10.1007/s10270-008-0088-x. URL: http://dx.doi.org/10.1007/s10270-008-0088-x.

[60] Peter Naur and Brian Randell. *Software Engineering: Report of a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO*. Ed. by Peter Naur and Brian Randell. NATO, 1969.

[61] Terence Parr. *ANTLR*. On-Line. URL: http://www.antlr.org/.

[62] Marian Petre. "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming". In: *Commun. ACM* 38.6 (June 1995), pp. 33–44. ISSN: 0001-0782. DOI: 10.1145/203241.203251. URL: http://doi.acm.org/10.1145/203241.203251.

[63] Oskar van Rest et al. "Robust Real-Time Synchronization between Textual and Graphical Editors". In: *Theory and Practice of Model Transformations*. Ed. by Keith Duddy and Gerti Kappel. Vol. 7909. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 92–107. ISBN: 978-3-642-38882-8. DOI: `10.1007/978-3-642-38883-5_11`. URL: `http://dx.doi.org/10.1007/978-3-642-38883-5_11`.

[64] Jarrett Rosenberg. "Some misconceptions about lines of code". In: *Software Metrics Symposium, 1997. Proceedings., Fourth International*. IEEE. 1997, pp. 137–142.

[65] Adolfo Sánchez-Barbudo Herrera, Edward D. Willink, and Richard F. Paige. "A Domain Specific Transformation Language to Bridge Concrete and Abstract Syntax". In: *Theory and Practice of Model Transformations: 9th International Conference, ICMT 2016, Held as Part of STAF 2016, Vienna, Austria, July 4-5, 2016, Proceedings*. Ed. by Pieter Van Gorp and Gregor Engels. Cham: Springer International Publishing, 2016, pp. 3–18. ISBN: 978-3-319-42064-6. DOI: `10.1007/978-3-319-42064-6_1`. URL: `http://dx.doi.org/10.1007/978-3-319-42064-6_1`.

[66] Adolfo Sánchez-Barbudo Herrera, Edward D. Willink, and Richard F. Paige. "An OCL-based Bridge from Concrete to Abstract Syntax". In: *Proceedings of the 15th OCL Workshop*. Ed. by Frédéric Tuong et al. Vol. 1512. CEUR. 2015, pp. 19–34. URL: `http://ceur-ws.org/Vol-1512/paper02.pdf`.

[67] Jesús Sánchez Cuadrado. "Towards a Family of Model Transformation Languages". In: *Theory and Practice of Model Transformations: 5th International Conference, ICMT 2012, Prague, Czech Republic, May 28-29, 2012. Proceedings*. Ed. by Zhenjiang Hu and Juan de Lara. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 176–191. ISBN: 978-3-642-30476-7. DOI: `10.1007/978-3-642-30476-7_12`. URL: `http://dx.doi.org/10.1007/978-3-642-30476-7_12`.

[68] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. "Towards the Systematic Construction of Domain-Specific Transformation Languages". In: *Modelling Foundations and Applications: 10th European Conference, ECMFA 2014, Held as Part of STAF 2014, York, UK, July 21-25, 2014. Proceedings*. Ed. by Jordi Cabot and Julia Rubin. Cham: Springer International Publishing, 2014, pp. 196–212. ISBN: 978-3-319-09195-2. DOI: `10.1007/978-3-319-09195-2_13`. URL: `http://dx.doi.org/10.1007/978-3-319-09195-2_13`.

[69] Edwin Seidewitz. "What models mean". In: *Software, IEEE* 20.5 (2003), pp. 26–32.

[70] Dave Steinberg et al. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2008.

[71] Tijs van der Storm. *Language Workbench Challenge 2013*. On-Line. 2013. URL: `http://www.languageworkbenches.net/wp-content/uploads/2013/11/Ql.pdf`.

[72] Delft University of Technology. *Spoofax*. On-Line. URL: `http://metaborg.github.io/spoofax/`.

[73] Massimo Tisi et al. “On the Use of Higher-Order Model Transformations”. In: *Model Driven Architecture - Foundations and Applications: 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings*. Ed. by Richard F. Paige, Alan Hartman, and Arend Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 18–33. ISBN: 978-3-642-02674-4. DOI: 10.1007/978-3-642-02674-4_3. URL: http://dx.doi.org/10.1007/978-3-642-02674-4_3.

[74] Pieter Van Gorp and Paul Grefen. “Supporting the internet-based evaluation of research software with cloud infrastructure”. In: *Software & Systems Modeling* 11.1 (2012), pp. 11–28.

[75] Eelco Visser. “Program Transformation with Stratego/XT”. English. In: *Domain-Specific Program Generation*. Ed. by Christian et. al Lengauer. Vol. 3016. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 216–238. ISBN: 978-3-540-22119-7. DOI: 10.1007/978-3-540-25935-0_13. URL: http://dx.doi.org/10.1007/978-3-540-25935-0_13.

[76] Edward Willink, Horacio Hoyos, and Dimitris Kolovos. “Yet another three QVT languages”. In: *International Conference on Theory and Practice of Model Transformations*. Springer. 2013, pp. 58–59.

[77] Edward D. Willink. “Local Optimizations in Eclipse QVTc and QVTr Using the Micro-Mapping Model of Computation”. In: *Proceedings of the 2nd International Workshop on Executable Modeling co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. Ed. by Tanja Mayerhofer et al. 2016. URL: Toappear.

[78] Edward D. Willink. “Optimized Declarative Transformation: First Eclipse QVTc Results ”. In: *Proceedings of the 4th Workshop on Scalable Model Driven Engineering part of the Software Technologies: Applications and Foundations (STAF 2016) federation of conferences*. Ed. by Dimitris Kolovos et al. Vol. 1652. CEUR. 2016, pp. 47–56. URL: http://ceur-ws.org/Vol-1652/paper6.pdf.

[79] Edward D. Willink and Nicholas Matragkas. *QVT Traceability: What does it really mean?* Technical Report. 2015. URL: https://www.eclipse.org/mmt/qvt/docs/ICMT2014/QVTtraceability.pdf.

[80] Edward Daniel Willink. “Re-engineering Eclipse MDT/OCL for Xtext”. In: *Electronic Communications of the EASST* 36 (2010), p. 1.

[81] Manuel Wimmer and Gerhard Kramler. “Bridging grammarware and modelware”. In: *Satellite Events at the MoDELS 2005 Conference*. Springer. 2006, pp. 159–168.

[82] Manuel Wimmer et al. “A Comparison of Rule Inheritance in Model-to-Model Transformation Languages”. In: *Theory and Practice of Model Transformations: 4th International Conference, ICMT 2011, Zurich, Switzerland, June 27-28, 2011. Proceedings*. Ed. by Jordi Cabot and Eelco Visser. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 31–46. ISBN: 978-3-642-21732-6. DOI: 10.1007/978-3-642-21732-6_3. URL: http://dx.doi.org/10.1007/978-3-642-21732-6_3.

[83] Claes Wohlin et al. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012.

[84] *Xtend*. On-Line. URL: `https://www.eclipse.org/xtend/`.

[85] *Xtext Documentation*. On-Line. URL: `http://www.eclipse.org/Xtext/documentation.html`.

[86] *Xtext Forum*. On-Line. URL: `http://www.eclipse.org/forums/index.php?t=thread&frm_id=27`.

[87] Athanasios Zolotas et al. "Flexible Modelling for Requirements Engineering". In: *Proceedings of the Workshop on Flexible Model Driven Engineering at MoDELS'15*. Ed. by Davide Di Ruscio, Juan de Lara, and Alfonso Pierantonio. Vol. 1470. urn:nbn:de:0074-1470-7. CEUR Workshop Proceedings. 2015. URL: `http://ceur-ws.org/Vol-1470/FlexMDE15_paper_6.pdf`.