
ulm university universität
**u**ulm

**Ulm University** | 89069 Ulm | Germany

**Fakultät für Ingenieurwissenschaften, Informatik und Psychologie**
Institut für Softwaretechnik und Programmiersprachen
Institutsdirektor:
Prof. Dr. Matthias Tichy

# Semantics-Driven Translation of UML-Models into Object-Oriented Programming Languages

---

**Aligning the Semantics of UML Static Structures and Dynamic Behavior in an Approach for Model-Driven Development**

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
der Universität Ulm

Dominik Gessenharter
aus Kempten

2018

| | |
|---|---|
| Amtierender Dekan: | Prof. Dr.-Ing. Maurits Ortmanns |
| Gutachter: | Prof. Dr. Helmuth Partsch<br>Prof. Dr. Manfred Reichert<br>Prof. Dr. Matthias Tichy |
| Tag der Promotion: | 05.07.2019 |

# Copyright Notice



Typesetting: PDF-LaTeX 2ε

# Acknowledgement

# Danksagung

Zu Beginn meiner Ausarbeitung möchte ich Herrn Prof. Dr. Helmuth Partsch sehr herzlich danken für die Möglichkeit, begleitend zu meiner Arbeit in Forschung und Lehre diese Dissertation anzufertigen. Die angenehme Arbeitsatmosphäre und die großzügigen Freiräume in der Arbeitsgestaltung machen meine Zeit als Mitarbeiter an seinem Institut zu einem sehr schönen und unvergesslichen Teil meines Berufslebens.

Besonderer Dank gilt auch meinen weiteren Gutachtern, Herrn Prof. Dr. Manfred Reichert und Herrn Prof. Dr. Matthias Tichy, sowie den Professoren der Prüfungskommission, die mir in der Endphase die nötige Unterstützung gegeben und Möglichkeiten zur Verbesserung der Arbeit aufgezeigt haben.

Danken möchte ich auch meinen Kollegen, die mich in der Entwicklung meiner Ideen begleitet haben. Besonders hervorzuheben sind an dieser Stelle Prof. Dr. Stefan Sarstedt, der als Betreuer im Rahmen meiner Diplomarbeit dieser Arbeit den Weg bereitet hat, sowie Marcel Dausend, Dr. Jens Kohlmeyer und Dr. Alexander Raschke, die mir im Alltag am Institut nicht nur gute Kritiker sondern auch vertraute Freunde geworden sind. Ebenso zu nennen ist Dr. Ralf Gerlich, der besonders während der frühen Phase meines Schaffens in vielen Gesprächen durch wertvolle Anregungen sehr bereichernd gewirkt und mir Orientierung gegeben hat. Auch Dr. Walter Guttmann hat durch seine Unterstützung insbesondere im "handwerklichen Bereich" wertvolle Hilfestellungen gegeben.
Weiterhin nennen möchte ich die von mir betreuten Diplomanden Tobias Schlecht, Dennis Knotz, Martin Rauscher, Andreas Bulach und Christian Waniek. Ihre gründliche und engagierte Arbeit war eine große Hilfe, die von ihnen erarbeiteten Ergebnisse sind teilweise in diese Dissertation eingeflossen.

Mit besonderem Dank möchte ich auch meine Eltern bedenken, deren Unterstützung ich mir jederzeit sicher sein konnte.

Ganz entscheidend beigetragen zum Gelingen dieser Arbeit hat auch meine Frau Claudia. Ihren unerschütterlichen Glauben an mich und meine Fähigkeiten hat sie unzählige Male unter Beweis gestellt. Über all die Jahre hat sie mir den Rücken frei gehalten und mir ermöglicht, neben meiner Rolle als Vater und neben dem Beruf dennoch ohne Einschränkungen an meiner wissenschaftlichen Fortbildung weiterzuarbeiten. Ohne ihre Unterstützung wäre die vorliegende Arbeit niemals fertig gestellt worden.

Dominik Gessenharter

# Contents

**Abstract** **xxiii**

**Summary** **xxv**

**1 Introduction** **1**
1.1 Problem . . . 2
1.2 Research Approach . . . 2
1.3 Thesis Statement and Scope of Research . . . 3
1.4 Basic Results . . . 4
1.5 Organization of Thesis . . . 5

**2 Background** **7**
2.1 Model-Driven Software Engineering . . . 7
2.2 Modeling Using UML2 . . . 8
2.2.1 General Characteristics of UML . . . 8
2.2.2 UML Semantics Architecture . . . 8
2.2.3 Interplay of Structures, Actions, and Behaviors . . . 9
2.3 ActiveCharts — The Project Pioneering this Thesis . . . 10
2.4 Advanced Code Generation with ActiveCharts . . . 10

**3 Semantics of Static Structures** **11**
3.1 Object-Oriented Concepts of Structural Modeling . . . 11
3.1.1 Class, Property, and Operation at a Glance . . . 11
3.1.2 Discrepancies between UML and Programming Languages . . . 14
3.2 Structural Modeling beyond Object-Oriented Concepts . . . 14
3.2.1 Constituent Parts of Association . . . 15
3.2.2 Ownership of Association Ends . . . 15
3.2.3 Symmetry of Associations . . . 16
3.2.4 Multiplicities of Association Ends . . . 16
3.2.5 Navigability of Association Ends . . . 17
3.2.6 Visibility of Association Ends . . . 17
3.2.7 Aggregation Types of Association Ends . . . 17
3.3 Implications of Combining Concepts . . . 18
3.3.1 Orthogonality of Modeling Concepts . . . 18
3.3.2 Impact of Arity . . . 19
3.3.3 Replaceability of Modeling Concepts . . . 20
3.4 Implementation of Attributes . . . 22
3.5 Implementation of Associations . . . 24
3.5.1 Association Implementation — Attribute Pattern . . . 24
3.5.2 Association Implementation — Relationship Object Pattern . . . 26
3.6 Enhancements to State-of-the-Art Code Generation . . . 28
3.6.1 Handles . . . 29
3.6.2 Implementing Association Classes . . . 32
3.6.3 Implementing Aggregation and Composition . . . 33
3.6.4 Implementing Ordered Association Ends . . . 33
3.6.5 Avoiding Memory Leaking . . . 33
3.7 Summary . . . 35

**4 Semantics of Behaviors** **37**
4.1 Actions and Activities at a Glance . . . 37
4.2 Activities . . . 38
4.2.1 Basic Token Flow Concept . . . 39
4.2.2 Guarded Flows . . . 40
4.2.3 Code Generation Based on Token Flow Semantics . . . 40
4.2.4 Token Flow at Control Nodes . . . 41
4.2.5 Guard Propagation . . . 45
4.2.6 Black Box Implementation of Complex Flows . . . 50
4.2.7 Further Considerations on Control Nodes . . . 55
4.2.8 Implementing Object Flows . . . 56
4.2.9 Executing Activities . . . 61
4.2.10 Interruptible Activity Region . . . 62
4.2.11 Preparing Activities by Model Transformations . . . 64
4.2.12 Semantical Resolution of Behavioral Concepts at a Glance . . . 65
4.3 Actions . . . 66
4.3.1 Semantics of Actions and Implementations . . . 67
4.3.2 Action Execution and Static Structure . . . 80
4.4 Considering Constraints Defined by Static Structures . . . 81
4.4.1 Modeling Multiplicity Bounds Checks . . . 81
4.4.2 Avoiding Structural Feature Action and Link Action . . . 82
4.5 A More Convenient Semantics for UML Actions . . . 82
4.5.1 Reporting about Success by Outputs . . . 82
4.5.2 Reporting about Failures by Exceptions . . . 82
4.5.3 Facing Temporarily Invalid System States . . . 83
4.6 Summary . . . 84

**5 Model Transformation** **85**
5.1 Formalisms . . . 85
5.2 QVT . . . 87
5.3 Transformations by QVT Relations . . . 90
5.3.1 Limitations Caused by *medini QVT* . . . 90
5.3.2 Arbitrary Limitations for Decreasing Complexity . . . 90
5.3.3 General Aspects . . . 93
5.4 Transforming Models of Static Structure . . . 95
5.4.1 Transformation of Classes . . . 95
5.4.2 Transformation of Associations . . . 103
5.4.3 Modifying Member End Classes . . . 110
5.4.4 Transformation of Compositions . . . 114
5.5 Transforming Models of Dynamic Behavior . . . 115
5.5.1 Extraction of Action Sequences . . . 116
5.5.2 Limitations of Transforming Behaviors . . . 121
5.6 Summary . . . 122

**6 Code Generation** **123**
6.1 The Role of Code Generation in MDD Approaches . . . 123
6.2 Formalisms . . . 123
6.3 Transformation Strategies . . . 124
6.4 Mof2Text . . . 125
6.5 Generating Code for Static Structures . . . 127
6.5.1 Generating Code for Classes and Associations . . . 128
6.5.2 Generating Code for Owned Attributes . . . 128
6.5.3 Generating Code for Owned Operations . . . 129
6.5.4 Generating Code for Link Members . . . 130
6.5.5 Generating Code for Link Managers . . . 130
6.5.6 Generating Code for Method Bodies . . . 131
6.5.7 Including Code for Owned Behavior . . . 140
6.6 Generating Code for Dynamic Behavior . . . 142
6.6.1 Processing Classifier Behavior . . . 142

6.6.2 Generating Activity Class Content . . . 142
6.6.3 Generating Code for Control Flow . . . 143
6.6.4 Generating Code for Control Nodes . . . 145
6.6.5 Generating Code for Join Nodes . . . 145
6.6.6 Generating Code for Decision Nodes . . . 147
6.6.7 Generating Code for Fork Nodes . . . 148
6.6.8 Generating Code for Sequences of Actions in Subactivities . . . 148
6.7 Summary . . . 151

**7 Tool Support** **153**
7.1 Evolution of ActiveCharts . . . 153
7.1.1 ActiveChartsIDE . . . 153
7.1.2 Migration to Eclipse . . . 154
7.2 Approaching a Complete Model Translation . . . 155
7.2.1 ActiveCharts First Generation Prototype . . . 155
7.2.2 ActiveCharts Second Generation Prototype . . . 155
7.3 ActiveCharts Third Generation Prototype . . . 156
7.4 Discussion . . . 157
7.4.1 Level of Compliance Compared to Related Tools . . . 157
7.4.2 Architecture . . . 157
7.4.3 Comparison of Runtime Characteristics . . . 157
7.4.4 Current Limitations . . . 158
7.4.5 Possible Extensions . . . 159
7.4.6 Concluding Remarks . . . 161

**8 Discussion** **163**
8.1 Characteristics of Our Approach . . . 163
8.1.1 Association and Interfaces . . . 164
8.2 Related Work . . . 166
8.2.1 Related Research on Structural Modeling . . . 166
8.2.2 Related Tools for Structural Modeling . . . 167
8.2.3 Alternative Approaches of Implementing Structural Models . . . 167
8.2.4 Related Research on Behavioral Modeling . . . 168
8.3 Feasibility and Validity of Our Approach . . . 169
8.4 Advancing Towards Workflow Execution . . . 172
8.4.1 Implementing Recovery in ActiveCharts . . . 173
8.4.2 Implementing Persistence in ActiveCharts . . . 176
8.5 UML Simplified . . . 176
8.6 Current Trends in Model-Driven Sofware Engineering . . . 177

**9 Conclusion** **179**
9.1 Contributions . . . 179
9.2 Outlook . . . 182

**Zusammenfassung** **183**

**APPENDIX** **186**

**A Macros for Action Implementation** **187**

**B QVT Relations and Queries** **189**
B.1 Structural Transformation . . . 189
B.1.1 Transforming Attributes . . . 189
B.1.2 Transformations of Associations and Association Classes . . . 194
B.1.3 Copying Model Elements . . . 197
B.1.4 Marking Model Elements by Comments . . . 201
B.1.5 Queries . . . 204
B.2 Behavioral Transformation . . . 208
B.2.1 Copying Model Elements . . . 208

B.2.2 Considering Object Flows . . . . . 210
B.2.3 Queries . . . . . 212

**C Code Generation** **213**
C.1 Considering Guarded Flows . . . . . 213
C.2 Considering Object Flows . . . . . 214
C.3 Considering Interruptible Activity Regions . . . . . 218
C.4 Queries . . . . . 220
C.5 Snippets Processing Structures . . . . . 225
C.6 Snippets Processing Behavior . . . . . 226

**D Evaluation** **227**
D.1 Testing Control Flows and Activity Termination . . . . . 228
D.2 Testing Control Flows with Control Nodes . . . . . 230
D.3 Testing Interruptible Activity Regions . . . . . 237
D.4 Testing Object Flows . . . . . 238

**Biblography** **241**

# Figures

**Abstract** **xxiii**

**Summary** **xxv**
1 Blueprint of the 1903 Wright Flyer. . . . xxv
2 Real and virtual aerodynamic models. . . . xxvi

**1 Introduction** **1**

**2 Background** **7**
2.1 Evolution of languages and abstraction levels. . . . 7
2.2 The UML three-layer semantics architecture. . . . 8
2.3 Relationships between structural and behavioral modeling concepts in UML. . . . 9
2.4 Relationships between meta-classes and concerning modeling elements. . . . 9

**3 Semantics of Static Structures** **11**
3.1 Excerpt of the UML Classes meta-model. . . . 12
3.2 Classes related by an attribute. . . . 13
3.3 UML Association. . . . 15
3.4 Association end ownership notation. . . . 15
3.5 The Cable Pattern . . . . 16
3.6 Multiplicities at association ends. . . . 16
3.7 Navigability symbols at association ends. . . . 17
3.8 Visibilities of association ends. . . . 17
3.9 Invalid combination of association end characteristics. . . . 18
3.10 Transformation of a ternary association into a set of binary associations. . . . 20
3.11 Association notation vs. attribute notation. . . . 21
3.12 Structural comparison of attribute and association notation. . . . 21
3.13 Association navigability and end multiplicities. . . . 22
3.14 Implementation of private attributes. . . . 23
3.15 Shapes of the Relationship Object Pattern. . . . 26
3.16 Consequence of the equality of lower and upper bounds of multiplicities . . . . 28
3.17 Access to an *AssociationClass* by associated classes and member end classes. . . . 33
3.18 Memory leaking resistant implementation of the Relationship Object Pattern. . . . 34

**4 Semantics of Behaviors** **37**
4.1 Notation of activity and its owned elements. . . . 37
4.2 Notation of object flow. . . . 38
4.3 Notation of *InterruptibleActivityRegion*. . . . 38
4.4 Implicitly and explicitly forks and joins. . . . 39
4.5 Sequences of actions and implementations. . . . 40
4.6 Guarded flow and its implementation. . . . 40
4.7 UML activity with sequences of actions. . . . 41
4.8 UML control nodes. . . . 42
4.9 Control flow semantics for *Action* without incoming flows. . . . 42

4.10 Determinism and decidability at DecisionNode. . . . . 43
4.11 Application of *ForkNode*. . . . . 44
4.12 Joining of control flows. . . . . 45
4.13 Guard propagation at control nodes. . . . . 47
4.14 Abstracting from control flow nets to black boxes. . . . . 47
4.15 Simplifications of 1-N black boxes. . . . . 48
4.16 Simplifications of N-1 black boxes. . . . . 48
4.17 Pushing *MergeNode* downstream. . . . . 49
4.18 Assembling *Selection Guards* and *Activation Guards* by guard propagation. . . . . 50
4.19 Implementing volatile token offers. . . . . 51
4.20 Implementing uncertain token origins. . . . . 51
4.21 Explicification of an implicit fork. . . . . 52
4.22 Application of fork node buffering in an example activity. . . . . 53
4.23 Explicit modeling of fork node buffering. . . . . 54
4.24 Explicit modeling of multiple fork node buffering. . . . . 55
4.25 Abstracting from black boxes. . . . . 56
4.26 Concurrent object flows. . . . . 58
4.27 Alternative object flows. . . . . 59
4.28 Control flow semantics for *AcceptEventAction* inside *InterruptibleActivityRegion* . 62
4.29 *InterruptibleActivityRegion* with an interrupted and an interrupting flow. . . . . 63
4.30 Transformation of an activity. . . . . 65
4.31 Graphical vs. textual representation of an activity. . . . . 66
4.32 UML meta-model of Action. . . . . 67
4.33 Abstract syntax of *LinkAction*. . . . . 72
4.34 Modeling checks of multiplicity bounds. . . . . 81
4.35 Consequence of the equality of lower and upper bounds of multiplicities . . . . . 83
4.36 Proposal for transactions in UML activities. . . . . 83

**5 Model Transformation** **85**
5.1 Subsetting and Generalization applied to associations. . . . . 92
5.2 Transformation of classes. . . . . 96
5.3 Transformation of attributes. . . . . 98
5.4 Transformation of a ternary association applying the Relationship Object Pattern. 103
5.5 Extraction of sequences of actions from an activity. . . . . 116
5.6 Avoiding unintended concurrency by additional control flows. . . . . 122

**6 Code Generation** **123**

**7 Tool Support** **153**
7.1 Architecture of ActiveChartsIDE. . . . . 154
7.2 Architecture of ActiveCharts First Generation Prototype. . . . . 155
7.3 Architecture of ActiveCharts Second Generation Prototype. . . . . 156
7.4 Architecture of ActiveCharts Third Generation Prototype. . . . . 156
7.5 Activity adding semantics to *AddStructuralFeatureValueAction*. . . . . 159
7.6 Three tier architecture for considering structural integrity. . . . . 160
7.7 Adapting reaction to multiplicity bounds violations. . . . . 161

**8 Discussion** **163**
8.1 Association between interfaces. . . . . 164
8.2 Associations between implementations. . . . . 165
8.3 Implementation of associations between interfaces in ActiveCharts. . . . . 165
8.4 Activity modeling an online order workflow. . . . . 174
8.5 Support for activity recovery. . . . . 175

**9 Conclusion** **179**

**Zusammenfassung (Summary)** **183**
10.1 Blaupause des Wright Flyer von 1903. . . . . 183
10.2 Reales und virtuelles aerodynamisches Modell. . . . . 184

**APPENDIX** **186**

**A Macros for Action Implementation** **187**

**B QVT Relations and Queries** **189**

**C Code Generation** **213**
C.1 Composition of snippets for class code generation. . . . . 225
C.2 Composition of snippets for code generation of dynamic behavior. . . . . 226

**D Evaluation** **227**
D.1 Test activity for guards and flow final nodes. . . . . 228
D.2 Test activity for guards and activity final nodes. . . . . 229
D.3 Test activity for merge semantics without guards. . . . . 230
D.4 Test activity for merge semantics with guarded flows. . . . . 231
D.5 Test activity for join semantics without guards. . . . . 232
D.6 Test activity for join semantics with guarded flows. . . . . 233
D.7 Test activity for decision node. . . . . 234
D.8 Test activity for fork semantics without guards. . . . . 235
D.9 Test activity for fork semantics with guarded flows. . . . . 236
D.10 Test activity for interruptible activity region. . . . . 237
D.11 Data model used for test activities with object flows. . . . . 238
D.12 Simple test activity for object flows. . . . . 238
D.13 Activity with object flows and decision node. . . . . 239
D.14 Activity of Fig. D.13 with object flows and additional control flows. . . . . 239
D.15 Activity with object flows and fork node. . . . . 240
D.16 Activity of Fig. D.15 with object flows and additional control flows. . . . . 240

# Listings

**Abstract** **xxiii**

**Summary** **xxv**

**1 Introduction** **1**

**2 Background** **7**

**3 Semantics of Static Structures** **11**
3.1 Implementation of a private attribute. . . . . 23
3.2 Accessing private attributes in user code. . . . . 23
3.3 Implementation of hiding accessor methods to specializations. . . . . 23
3.4 Implementation of a fully navigable binary association and its links. . . . . 30
3.5 Implementation of a class participating an association. . . . . 31
3.6 Implementation of a binary association with one private end. . . . . 31
3.7 Implementation of access to a private association end with a method handle. . . . 31
3.8 Implementation of ordered association ends. . . . . 34
3.9 Implementation of memory leak prevention in member end classes. . . . . 35
3.10 Implementation of memory leak prevention in associations. . . . . 35

**4 Semantics of Behaviors** **37**
4.1 Implementation of sequences of actions in a single class. . . . . 41
4.2 Implementations of decision nodes. . . . . 44
4.3 Implementation of *ForkNode*. . . . . 44
4.4 Implementation of *JoinNode*. . . . . 45
4.5 Implementation of an output pin . . . . . 56
4.6 Implementation of an object flow considering input and output multiplicities. . . . 57
4.7 General implementation of an object flow containing a fork node. . . . . 58
4.8 General implementation of a join of two object flows. . . . . 58
4.9 Joining object flows. . . . . 59
4.10 Implementation of an object flow containing a decision node. . . . . 60
4.11 Implementation of a merged object flow. . . . . 60
4.12 Implementation of an *Activity*. . . . . 61
4.13 Initialisation of *InterruptibleActivityRegion*. . . . . 63
4.14 Implementation pattern for an *InterruptibleActivityRegion*. . . . . 63
4.15 Implementation pattern for an interruptible action sequence. . . . . 64
4.16 Implementation pattern for an interrupting flow. . . . . 64
4.17 Implementation of *CreateObjectAction*. . . . . 68
4.18 Implementation of *DestroyObjectAction*. . . . . 68
4.19 Implementation of object destruction by dereferencing. . . . . 69
4.20 Implementation of *ReadSelfAction*. . . . . 69
4.21 Implementation of *CallOperationAction* (*isSynchronous* = true). . . . . 70
4.22 Implementation of *CallOperationAction* (*isSynchronous* = false). . . . . 70
4.23 Implementation of *ClearStructuralFeatureAction*. . . . . 70
4.24 Implementation of *ReadStructuralFeatureAction*. . . . . 71
4.25 Implementation of *AddStructuralFeatureValueAction*. . . . . 71
4.26 Implementation of *RemoveStructuralFeatureValueAction*. . . . . 72

4.27 Implementation of *ReadLinkAction*. . . . . 73
4.28 Implementation of *CreateLinkAction*. . . . . 73
4.29 Implementation of *isReplaceAll*. . . . . 73
4.30 Implementation of *CreateLinkObjectAction*. . . . . 74
4.31 Adapted implementation of link creation for link object creation. . . . . 74
4.32 Implementation of *ReadLinkObjectEndAction*. . . . . 74
4.33 Implementation of *DestroyLinkAction*. . . . . 74
4.34 Implementation of `destroyLink` in an implementation class of an association. . . . . 75
4.35 Implementation of `destroyLink` in an implementation class of an association for an ordered, non-unique association. . . . . 75
4.36 Implementation of *ClearAssociationAction*. . . . . 75
4.37 Implementation of `Clear`. . . . . 75
4.38 Implementation of *SendSignalAction*. . . . . 76
4.39 Implementation of signal receipt. . . . . 76
4.40 Implementation of *AcceptEventAction*. . . . . 76
4.41 Implementation of *AcceptEventAction*. . . . . 77
4.42 Implementation of *BroadcastSignalAction*. . . . . 77
4.43 Implementation of *ReadExtentAction*. . . . . 78
4.44 Implementation of `readExtent`. . . . . 78
4.45 Implementation of *OpaqueAction*. . . . . 78
4.46 Implementation of *RaiseExceptionAction*. . . . . 79
4.47 Implementation of *StartClassifierBehaviorAction*. . . . . 79
4.48 Implementation of starting an owned behavior. . . . . 79
4.49 Implementation of *CallBehaviorAction*. . . . . 80

**5 Model Transformation** **85**
5.1 Excerpt of the concrete syntax of QVTR. . . . . 88
5.2 Concrete syntax of OCL Expressions. . . . . 88
5.3 Pseudo transformation. . . . . 89
5.4 QVTR transformation: model declarations. . . . . 93
5.5 QVTR transformation: Key declarations. . . . . 94
5.6 Relation for creating a package and element imports. . . . . 94
5.7 Transformation of classes. . . . . 96
5.8 Transformation of generalization relationships. . . . . 97
5.9 Transformation of non-private, multi-valued attributes. . . . . 99
5.10 Transformation of non-private, multi-valued attributes that are derived. . . . . 99
5.11 Creation of accessor operations for multi-valued attributes. . . . . 100
5.12 Completion of the operation for reading access to attributes. . . . . 100
5.13 Completion of operations for writing access to multi-valued attributes. . . . . 101
5.14 Transformation of operations. . . . . 101
5.15 Enforcing target operations for non-private source operations. . . . . 102
5.16 Enforcing target operations for non-private, abstract source operations. . . . . 102
5.17 General transformation of associations and association classes. . . . . 104
5.18 Linking member end classes to a link manager class. . . . . 106
5.19 Adapting a link class to the structure of an association. . . . . 107
5.20 Supplying a link class constructor with parameters. . . . . 107
5.21 Adapting a link manager class to the structure of an association. . . . . 108
5.22 Supplying an attribute representing a lower bound with a concrete value. . . . . 109
5.23 Supplying an attribute representing an upper bound with a concrete value.. . . . . 109
5.24 Specifying the return value of an operation for querying links. . . . . 110
5.25 Preparing operations for write access in member end classes. . . . . 110
5.26 Specification of parameters for accessor operations in member end classes. . . . . 111
5.27 Preparing an operation for reading ends of a higher order association. . . . . 112
5.28 Specifying an operation for reading association ends. . . . . 113
5.29 Supplying an operation for reading association ends with parameters. . . . . 113
5.30 Associating a manager class of a composition to manager classes of associations. . 114
5.31 Supplying ends of associations between manager classes with values. . . . . 115
5.32 Moving activities from the source model into the target model. . . . . 116

5.33 Checking actions for being part of a sequence. . . . 117
5.34 Preparing subactivities for sequences of actions. . . . 118
5.35 Moving flow body actions into subactivities. . . . 119
5.36 Updating the structure of a subactivity. . . . 119
5.37 Identifying flows to head actions. . . . 120
5.38 Updating the target of incoming edges of head actions to call behavior actions associated with subactivities. . . . 120
5.39 Identifying outgoing edges of the last flow body action of a sequence. . . . 121
5.40 Updating the source of an outgoing edge of the last flow body action of a sequence. 121

**6 Code Generation 123**
6.1 Syntax of a Mof2Text template. . . . 125
6.2 Code generated by a simple template. . . . 126
6.3 Syntax of a Mof2Text template. . . . 126
6.4 Code generated by a simple template. . . . 126
6.5 Syntax of a Mof2Text template. . . . 127
6.6 Generation of class files for UML classes. . . . 127
6.7 Generation of class contents. . . . 128
6.8 Generation of code for the implementation of owned attributes. . . . 129
6.9 Generation of method signatures implementing owned operations. . . . 129
6.10 Generation of code implementing the participation in associations. . . . 130
6.11 Generation of code for link manager classes. . . . 131
6.12 Generation of method bodies for owned operations. . . . 132
6.13 Generation of a constructor with parameters. . . . 133
6.14 Generation of code for setting attribute values. . . . 133
6.15 Generation of code for adding attribute values. . . . 134
6.16 Generation of code for removing attribute values. . . . 134
6.17 Distinction of methods for accessing association ends from participating classes. . . . 135
6.18 Generation of code for creating association links. . . . 135
6.19 Generation of code for retrieving association links. . . . 136
6.20 Generation of code for destroying association links. . . . 136
6.21 Distinction between methods for accessing associations from member classes. . . . 137
6.22 Generation of code for linking objects. . . . 137
6.23 Generation of code for unlinking objects. . . . 138
6.24 Generation of code for retrieving linked objects. . . . 139
6.25 Generation of code for a classifier behavior. . . . 140
6.26 Generation of abstract method declarations for opaque actions. . . . 140
6.27 Generation of default implementations for opaque actions. . . . 141
6.28 Generation of a sourcefile for each activity. . . . 142
6.29 Generation of class contents for activities. . . . 143
6.30 Generation of code for the activity execution. . . . 143
6.31 Generation of code for the selection of sequences. . . . 144
6.32 Generation of code for control nodes. . . . 145
6.33 Generation code for join nodes. . . . 146
6.34 Generation of code for *JoinNode*. . . . 147
6.35 Generation of code for *DecisionNode*. . . . 147
6.36 Generation of code for *ForkNode*. . . . 148
6.37 Generation of code for sequences of actions. . . . 149
6.38 Generation of code for considering guarded flows. . . . 150
6.39 Generation of code for entering, leaving and aborting interruptible activity regions. 150

**7 Tool Support 153**

**8 Discussion 163**
8.1 Implementation of a recovery point between actions. . . . 175

**9 Conclusion 179**

**Zusammenfassung (Summary) 183**

**APPENDIX** **186**

**A Macros for Action Implementation** **187**
A.1 Macro building a string representation from input pins. . . . . 187
A.2 Macro building a string representation of input values. . . . . 187
A.3 Macro building a string representation of input and output values. . . . . 187

**B QVT Relations and Queries** **189**
B.1 Transforming non-private, non-derived, single-valued owned attributes. . . . . 189
B.2 Transforming non-private, derived, single-valued owned attributes. . . . . 190
B.3 Adding access operations for owned attributes. . . . . 190
B.4 Adding getter operations for accessing owned attributes. . . . . 190
B.5 Handling private attributes and private operations. . . . . 191
B.6 Hiding private operations in subclasses. . . . . 191
B.7 Hiding private, multi-valued attributes in subclasses. . . . . 192
B.8 Hiding private, single-valued attributes in subclasses. . . . . 193
B.9 Transformation of association excluding association classes. . . . . 194
B.10 Transformation of association classes. . . . . 195
B.11 Specification of operations for write access on an association class. . . . . 196
B.12 Preparing an operation for reading ends of a binary association. . . . . 196
B.13 Copying *ElementImport*s. . . . . 197
B.14 Copying attributes of *Property*. . . . . 197
B.15 Enforcing existence of types of owned attributes. . . . . 197
B.16 Copying primitive tapes of owned attributes. . . . . 198
B.17 Copying attributes of *Operation*. . . . . 198
B.18 Enforcing the existence of return types of operations. . . . . 198
B.19 Copying primitive return types of operations. . . . . 198
B.20 Copying attributes of operation parameters. . . . . 199
B.21 Enforcing the existence of parameter types. . . . . 199
B.22 Enforcing the existence of parameter types. . . . . 200
B.23 Copying primitive parameter types. . . . . 200
B.24 Marking a link implementation class. . . . . 201
B.25 Marking a link manager class. . . . . 201
B.26 Marking a class connected to an association end. . . . . 201
B.27 Marking the owned attribute of a member end class which references the link manager class. . . . . 201
B.28 Marking operations for accessing association ends within link class. . . . . 201
B.29 Marking operations for accessing association ends within a member end class. . . . 202
B.30 Marking operations reading association ends in member end classes. . . . . 202
B.31 Marking operations reading association ends of an association class in member end classes. . . . . 202
B.32 Marking a class being an implementation class. . . . . 202
B.33 Marking operations accessing owned attributes. . . . . 203
B.34 Query for obtaining a role name with lower first letter. . . . . 204
B.35 Query for obtaining a role name with upper first letter. . . . . 204
B.36 Query for obtaining an operation name. . . . . 204
B.37 Query for obtaining a concatenation of property names. . . . . 204
B.38 Query for obtaining the name of an association of deriving a name from its ends. . 204
B.39 Query for obtaining a concatenation of property names. . . . . 205
B.40 Query for making operation names distinct if multiple ends are of the same type. . 205
B.41 Query returning a properties lower multiplicity bound. . . . . 205
B.42 Query returning a properties upper multiplicity bound. . . . . 205
B.43 Query for deriving a visibility from visibilities of a set of properties. . . . . 206
B.44 Query returning the most restrictive visibility of a set of properties. . . . . 206
B.45 Query returning the visibility for the implementation of an owned element. . . . . 206
B.46 Query implementing a *modulo* operator. . . . . 206
B.47 Query implementing a mapping from Integer to String. . . . . 206
B.48 Query translating an Integer into its string representation. . . . . 207
B.49 Realtion enforcing the existence of all typed elements in the target model. . . . . 208

B.50 Relation enforcing the existence of a type. . . . 208
B.51 Relation for copying activity parameters. . . . 208
B.52 Relation enforcing the existence of the types of activity parameters. . . . 209
B.53 Relation enforcing the existence of primitive types of activity parameters. . . . 209
B.54 Updating the target of incoming object flows of head actions to call behavior actions associated with subactivities. . . . 210
B.55 Updating the source of an outgoing object flow of the last flow body action of a sequence. . . . 211
B.56 Query checking whether an action occurs after another action in the same sequence of actions. . . . 212

**C Code Generation** **213**
C.1 Consideration of guards at merge nodes. . . . 213
C.2 Generation of a wrapper for multiple action outputs. . . . 214
C.3 Preparing local variables representing inputs and outputs of sequences of actions. . . 215
C.4 Providing methods for setting inputs of action sequences. . . . 215
C.5 Passing inputs of action sequences as parameters. . . . 216
C.6 Considering object flows declarations of methods implementing action sequences. . 216
C.7 Providing output of actions as input for other actions. . . . 217
C.8 Generating methods for handling interruptible activity regions. . . . 218
C.9 Generating code for a fork node considering distribution of inputs to concurrently executing sequences of actions. . . . 219
C.10 Generating constructors supporting to set inputs before running the constructed thread. . . . 219
C.11 Queries for obtaining a qualified name for a given classifier. . . . 220
C.12 Query returning a qualified file name for a given class. . . . 220
C.13 Query returning a qualified file name for a given activity. . . . 220
C.14 Query mapping UML visibilities to Java visibility modifiers. . . . 220
C.15 Query returning the general classifier of a class. . . . 220
C.16 Query returning a return type of an operation. . . . 221
C.17 Query returning input parameters of an operation. . . . 221
C.18 Query returning a subset of a set of parameters. . . . 221
C.19 Query returning a string containing type and name for a set of parameters. . . . 222
C.20 Query translating a default value into s string representation. . . . 222
C.21 Query returning a return statement with a default return value. . . . 222
C.22 Query returning a string of comma separated pairs of input pin types and names. . 222
C.23 Query building a camel case string concatenation of two given strings. . . . 223
C.24 Query returning a sequence of actions starting at an initial node. . . . 223
C.25 Query returning a sequence of actions starting at a given action. . . . 223
C.26 Query checking whether an activity node is an Action. . . . 223
C.27 Query returning the string *if* or *else if*. . . . 223
C.28 Query returning true if a given activity contains a *CallBehaviorAction*. . . . 223
C.29 Query returning output parameters of a given activity. . . . 223
C.30 Query returning input parameters of a given activity. . . . 223
C.31 Query returning all typed input pins of a given Action. . . . 224
C.32 Query returning typed output pins of a given action. . . . 224
C.33 Query checking whether a class participates in an association. . . . 224
C.34 Query checking whether a class is a link manager class of an association. . . . 224
C.35 Query checking whether a property is an owned attribute referencing a link manager class. . . . 224
C.36 Query returning the rest of a comment starting with a given string. . . . 224

**D Evaluation** **227**
D.1 Modified code generation for actions in test activities. . . . 227
D.2 Output produced by executing test activity *Flows* of Fig. D.1 . . . 228
D.3 Output produced by executing test activity *Flows* of Fig. D.2 . . . 229
D.4 Output produced by executing test activity *Merge* of Fig. D.3 . . . 230
D.5 Output produced by executing test activity *Merge* of Fig. D.4 . . . 231
D.6 Output produced by executing test activity *Join* of Fig. D.5 . . . 232

D.7 Output produced by executing test activity *Join* of Fig. D.6 . . . . . 233
D.8 Output produced by executing test activity *Decision* of Fig. D.7 . . . . . 234
D.9 Output produced by executing test activity *Fork* of Fig. D.8 . . . . . 235
D.10 Output produced by executing test activity *Fork* of Fig. D.9 . . . . . 236
D.11 Output produced by executing test activity *IR* of Fig. D.10 . . . . . 237
D.12 Modified action implementation for test activities with data flows. . . . . 238
D.13 Output produced by executing test activity *DataFlow* of Fig. D.12 . . . . . 238
D.14 Output produced by executing test activity *DataFlow* of Fig. D.13 . . . . . 239
D.15 Output produced by executing test activity *DataFork* of Fig. D.16 . . . . . 240

# Abstract

Unified Modeling Language (UML) offers extensive possibilities to model static structure as well as dynamic behavior, both key domains of a complete system specification. UML, however, fails to clearly specify its semantics, in particular to coherently and consistently align the semantics of structure and behavior.

The key idea of Model-Driven Development (MDD), which is to generate implementations from models, is hardly feasible due to the informally and therefore ambiguously defined semantics. The misalignment between the semantics of static structure and dynamic behavior, which both are required for a complete specification of a system, makes things even worse.

This thesis is particularly concerned with the semantics of UML regarding the modeling of static structure and dynamic behavior. It identifies design flaws, inconsistencies, and shortcomings and presents suggestions on how to overcome semantical deficiencies in order to succeed in generating software implementations from models. The provided solutions are founded on an adapted semantics which tightly integrate modeling of static structure and dynamic behavior. Based on this adapted semantics, it is detailed how the high abstraction levels provided by the modeling language are transformed into the lower abstraction levels of programming language concepts.

With the profound study of the semantics and the presented modifications to it, a reliable foundation for better tool support is given. The mapping of modeling concepts to programming language constructs is specified using executable transformation formalisms. From these transformation specifications, a prototypical implementation of a code generator is derived, which translates UML models into the Java programming language.

# Summary

## Extended Abstract

Modeling is a concept applied in many fields of engineering. Accordingly, there are various forms of models which, however, always refer to a common basic idea: a model is a representative of an item of the real world or of imagination. Models may vary in the level of abstraction as well as in their intrinsic nature and their purpose.

For example, a blueprint is on the one hand a graphic specification of the shape and dimensions of components, on the other hand an illustration of how several individual parts together form assemblies that finally together form the overall structure. It is a 2D abstraction of the real 3D object, as e. g. the constructional drawing of the 1903 Wright Flyer shown in Fig 1.

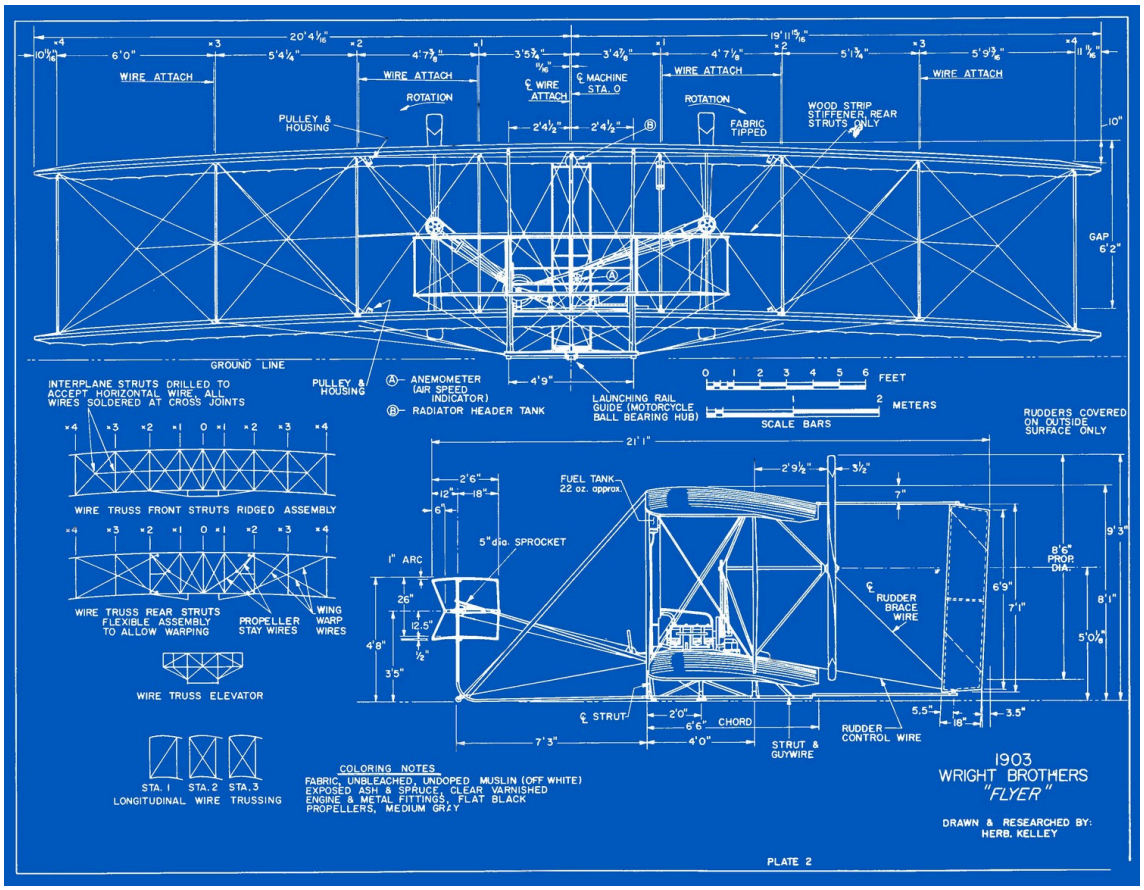


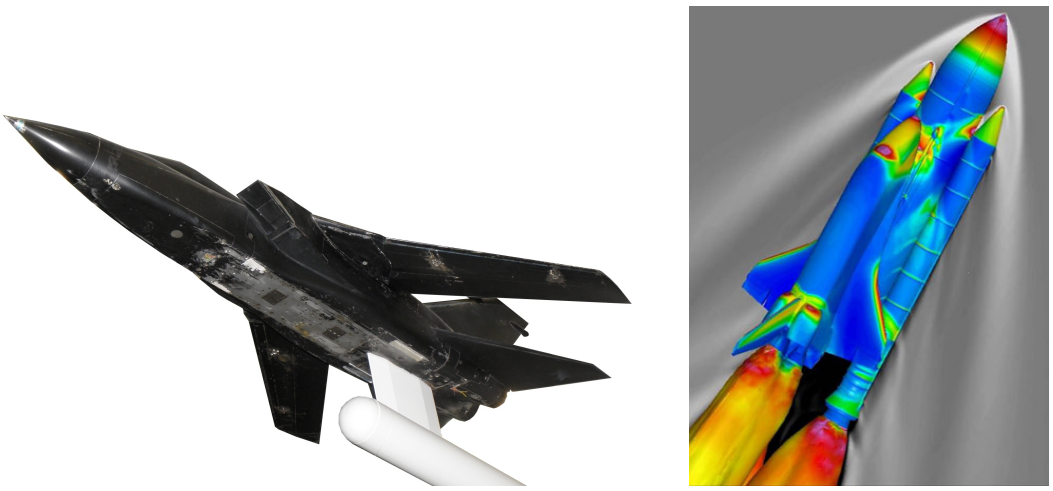

Figure 1: Blueprint of the 1903 Wright Flyer[1].

With regard to an objects shape, a 3D model created as a true-to-scaled mock-up is less abstract than a blueprint because it does not reduce the modeled entity to a 2D view. A mock-up may represent an item that itself possibly does not yet exist. For instance, the aerodynamics of a car or

[1]Based on the Christman drawings, ©http://wright-brothers.org [104].

an aircraft can be verified in a wind tunnel by means of a true-to-scale aerodynamic model. Figure 2(a) shows such a model of the Panavia PA200 Tornado[2] aircraft. The benefit of such models is that design flaws - here in the field of aerodynamics - can be identified before cost intensive prototypes are built, and the risks of driving or flight tests is reduced.

As computing power increases, modeling shifts continuously from real physical models towards computer simulations. The benefit of computer models is that such models are capable of exhibiting design flaws that occur in environments that are difficult or even impossible to simulate, such as e. g. zero gravity or vacuum, both conditions under which spacecraft and satellites are operated. Figure 2(b)[3] shows the pressure coefficient on the Space Shuttle Launch Vehicle and in the near-body flowfield at mach 2.46 at an altitude of 66.000 feet [101].



(a) Wind tunnel model of the Panavia PA200 aircraft[2].

(b) Aerodynamic model of the Space Shuttle Launch Vehicle[3].

Figure 2: Real and virtual aerodynamic models.

With regard to software, models can be beneficial in the same way: a model may represent a complex software system with respect to various characteristics prior to its implementation. For instance, software components of an embedded system can be implemented, tested, and integrated before other components are available, if the missing parts can be simulated by prototypes or executable models.

The relation between virtual models and modeled artifacts is essentially different when modeling software rather than modeling real world entities: software is immaterial and therefore, the model and the modeled entity are of the same domain — they both are artifacts that are processed by the same type of machines. For this reason, in principle it is possible to translate a design model into an implementation[4]. Model-Driven Development is an approach focusing on this idea.

The main benefit is considered to be achieved by a higher level of abstraction, i. e. a modeling language usually is designed to fit closely to the problem domain. The translation to runnable machine code is to be covered by tool support. A prominent approach to face this task is Model-Driven Architecture, an open standard providing a set of transformations from platform independent models, i. e. models completely independent from technical details induced by the underlying system software, to the target implementation which may depend on the technology of target machines.

This thesis targets such a translation from a highly abstract modeling language to a general purpose object-oriented programming language. In particular, it addresses two issues of modeling and implementation characteristics:

- the architecture of the system, i. e. static structures
- the functionality of the system, i. e. dynamic behaviour.

[2]Author's own work, picture taken at the Deutsches Museum, courtesy of the Deutsches Museum, Munich.
[3]The image is originally published by NASA and is in the public domain [101].
[4]To create the modeled entity from a model is not limited to software, e. g. automated production using metal cutting tools creates the modeled entities as well. But it requires a machining tool which must be operated and maintained and, with regard to complex systems such as a car or an aircraft, the complete assembling currently cannot be automated completely.

The general purpose modeling language, which is the specification language for input models is Unified Modeling Language (UML), a standard released by Object Management Group (OMG). The target object-oriented programming language in which the system is to be implemented is Java. This is an arbitrary choice and we could have chosen other languages for both, input and output artifacts. However, both languages are popular in business environment as well as in academic science and widely used by researchers as well as practitioners. In particular, UML currently is the de-facto standard regarding modeling of software systems.

A peculiarity of UML is, that its semantics is not formally specified. It is ambiguous and accordingly, different interpretations exist. Furthermore, the specifications regarding static structure on the one hand and dynamic behavior on the other hand partly contradict each other. Approaches focusing on only one of both aspects, either on static structure or on dynamic behavior, hence cannot cope with contradictions between the both fields of modeling. In contrast, this thesis aims to identify the design flaws of UML regarding the interplay between structural and behavioral models:

- In a first step, semantics of modeling static structure is investigated in detail with a main focus on classes with their attributes and relationships between them. Here, a central issue is the question how the semantics of associations, in particular symmetry, navigability, association end visibility and association end ownership can be implemented in object-oriented language concepts.
- In a second step, the semantics of activity is investigated and a mapping from the token flow semantics to threads of the target language is elaborated. Most challenging in this regard is supporting all kinds of control nodes while adequately considering the effects of guarded flows.
- With activity, the execution environment for actions is given. In a third step, the semantics of a subset of actions is discussed. This set mainly comprises those actions which are semantically related to static structures, e. g. by reading or updating class attributes or by creating or reading association links.
- Founded on these three steps, we provide a formal and executable description of how to translate models into code, taking into account the semantic details of input models.
- We discuss modifications of UML in order to properly align semantics of static structures with semantics of dynamic behavior and vice versa.

In the course of addressing the issues listed above, this work contributes to the understanding and development of model-driven development in the following ways:

- a better and deeper understanding of semantics and a semantic-preserving implementation of associations, especially of n-ary associations and association classes
- a general mapping from the token flow semantics of activity to Java threads, including the semantics of control nodes and the combination of control nodes with guarded flows
- suggestions on how to improve UML semantics in order to bring semantics of structural features into line with the semantics of actions that access structural features
- a transformation from highly abstract modeling concepts to less abstract programming language concepts, as well as a transition from models to code, both formally given by executable declarative OMG standards.

Based on these contributions, the efficiency of Model-Driven Development could be increased by improving the tools' ability to handle models using the full variety of UML modeling concepts. Development tools need to be able to translate the full semantic content of the input models into a modern target language as described in our approach, so that the advantage of the modeling languages, namely the higher level of abstraction, can be profitably exploited in the development process.

# Chapter 1

# Introduction

Abstraction is a key technique for handling complexity and therefore a fundamental concept in software engineering [100]. Todays programming languages are on almost the same level of abstraction as the first high-level programming languages were, even though there have been developments like the object-oriented paradigm. In contrast, complexity of todays software systems has significantly changed compared to systems for which to implement first high-level programming languages were designed. In this regard, we use mid-$20^{th}$ century technology to face $21^{st}$ century problems [89].

To improve the situation, more abstract languages are desired. Todays evolution of languages does not produce more abstract textual programming languages but (most often graphical) modeling languages, a very famous one of which is Unified Modeling Language (UML)[1].

Modeling is applying abstraction to any entity, be it of the real world or imagined. The result is a model that reduces complexity of the original entity but may represent it in order to make complex entities accessible to human comprehension and increase understandability or appreciation of it [91]. Similar to other engineering disciplines, in software engineering models are used for

- better understanding of the represented system
- easier and more accurate communication
- prediction of characteristics of a system built on the basis of a specific model
- design specification of the system to build.

In software engineering, models as well as the software they represent both are immaterial and — since both are processed by the same devices — it is possible to create the software through a transformation of the model [91].

Model-driven Development (MDD) is an approach based on this idea: models resulting from the analysis and design phases of the development process are automatically transformed to software. Changes to the software are applied by changing the models and re-generating the software. Key techniques a MDD approach must cope with is either interpretation of models or code generation, both possibly preceded by model transformations.

The de-facto standard for modeling software is UML which has been released by Object Management Group (OMG) in 2005 in its current version 2 with some modifications up to now. Model-Driven Architecture (MDA) is a MDD approach of OMG proposing to apply UML for modeling [47] and defining an architecture of models and transformations. A key feature of MDA is platform independence. All information contained in a model is reusable for implementations of different techniques thus providing a clear distinction between logical and technological aspects of a system. A representative of MDA based on UML is Executable UML [59].

This thesis summarizes our research results about code generation from UML models and how tool support can further improve software engineering. Although the presented approach shares commonalities with MDA, it is not completely aligned with it and can be categorized as a MDD approach slightly related or inspired by MDA.

In this chapter, we first properly describe some problems regarding the application of abstraction in modern software engineering. We state the three theses which formulate the core issues faced by our research, define the scope of our research work and survey our basic results. Finally, an outline of this thesis follows.

[1] in this thesis, we refer to UML 2.4, [69]

## 1.1 Problem

Abstraction is necessary to focus on relevant details of a system while omitting all the others. By this, complexity is reduced and an understanding of the overall problem is obtainable. But when generating software, this abstraction has to be resolved to concrete elements of the language running on the target machine. We term this process *Semantical Resolution*. First part of the problem is that current UML-based MDD approaches **lack a proper description of a semantical resolution**. This also prevents a higher degree of automation in tool support for UML [11].

Semantical resolution is a mapping from modeling language concepts to appropriate concepts of the target language or model. Elaborating such a mapping inevitably requires a clear specification of the semantics of input and target concepts. For UML semantics, only a description by natural language but no formal specification is given. Accordingly, semantics of some concepts are not clearly identifiable and thus, semantical equivalence of a model and the results of its transformation is not definitively determinable. Therefore, the second part of the problem is an **insufficient specification of the semantics** of UML.

A modeling language must support modeling static structure as well as dynamic behavior, since internal structure and interaction between components are integral parts of each system. UML complies to this claim, however, the semantics of some behavioral concepts is not well aligned with the semantics of concerned structural elements. Another problem is that the semantics at some points is not strong enough to only allow the intended interpretation, or a statement about the intended semantics is even completely absent. These issues make the third part of the problem, which generally speaking are **inconsistency and incompleteness of the semantics** of UML.

Current MDD approaches often do not generate a whole application. Code generation for static structure is far more supported than code generation for behavior. If behavior is covered, support of state machines is very common. In UML, a fundamental concept of behavioral modeling is *Action* which can only be applied within *Activity* [10]. Consequently, another part of the problem is the **unsatisfactory tool support of activities**.

In the following, we address two minor issues related to our problem statement. First, the question arises whether semantical resolution should target common general purpose languages like Java, C, or C# or whether to address languages comprising more abstract elements, e. g. relation. The design of languages directly supporting relations as a semantic construct was demanded by Rumbaugh already in 1987 [79].

Another question is which formalisms to use for implementation of a semantical resolution. Depending on the applied techniques, it might be easier or more straightforward to split the resolution into several steps. Each of these steps then can be specified by the best fitting formalism or maybe is just better readable if specified apart from other aspects of the semantical resolution.

All these deficiencies of which — or at least, of part of which — most current UML-based MDD approaches suffer can be summarized as follows:

> Today's UML-based MDD approaches offer unsatisfactory support for activities and only rudimentary support for static structures, both very likely caused by UML's insufficient and inconsistent specification, and furthermore, they lack a proper description of the semantical resolution of modeling concepts to target language elements formalized in suitable formalisms.

## 1.2 Research Approach

This thesis has been triggered by the idea to model applications by means of class diagrams and activities. Whereas tools for generating code for static structures as well as tools that execute state machines have been available, support for translating or executing activities has been missing when our research in this field has started. With ActiveCharts[85, 84], Sarstedt [83] provided a foundation for obtaining runnable software from structural and behavioral models of an application.

In practical courses at Ulm University, the limitations of ActiveCharts have become visible. These are in particular a not completely UML compliant implementation of associations and a limitation to binary associations as well as bad scaling characteristics of the activity interpreter if used in larger application scenarios. Overcoming these limitations and supporting a larger set of

modeling elements, such as n-ary associations or more types of actions has influenced the approach how ActiveCharts should be advanced.

Our research therefore covers two separate objectives which are

- an internal technical redesign and
- functional extensions.

Concerning the internal redesign in order to speed up activity executions, two kinds of prototypes have been sketched:

- a high-performance execution engine (interpreter)
- implementation patterns for translating activities to code.

While trying to enhance the scaling characteristics of our tool, a better level of UML compliance has been targeted by concurrently analysing the UML semantics. The inclusion of other types of actions for compliantly accessing structural features has placed the semantical foundations of our research upon a sound basis. At this point, the weak semantics of actions possibly causing invalid system states affected our approach for the first time.

By continuously applying ActiveCharts in practical courses, a set of small sized systems for testing feasibility is available. TravelPlaner is another sample project launched in order to design a system making use of those modeling elements that should be included in an enhanced version of ActiveCharts. Finally, the expertise of some years of applying ActiveCharts in smaller projects and a deeper understanding of UML semantics made it possible to build a tool that completely translates UML class models and activities into runnable code while supporting a larger portion of behavioral modeling features.

## 1.3 Thesis Statement and Scope of Research

From our experiences in advancing ActiveCharts, we derive our thesis statement which comprises three essential claims:

- The semantics of UML is on a higher level of abstraction than typical object-oriented languages (like Java) thus requiring a semantical resolution.
- The higher level constructs of structural models can be feasibly translated to object-oriented languages without loss of information as well as this can be achieved for constructs of activities.
- Constraints specified upon the structure of a system, such as association multiplicities, impact behavioral modeling. This impact is not yet considered by the UML semantics specification, although it must be considered by the implementation of a modeled system.

In the scope of our research work we accurately analyze the semantics of UML and present code patterns suitable to implement UML concepts in Java. In contrast to other research in this field, we delve deeper into the complexity of UML. We do not define a subset of UML elements dropping those ones difficult to implement, like it is done by OMG defining the *Semantics of a Foundational Subset for Executable UML Models (fUML)* [72]. Furthermore, we do not consider elements independently from each other, but we focus on the interplay between concepts, e. g. we consider the impact of structural constraints like association multiplicities when executing behavior trying to link instances, or we consider guards constraining the execution flow in activities together with the semantics of control nodes.

Our analysis of UML semantics is based on the textual specification of UML. Since numerous ambiguities and deficiencies of this specification often cause misunderstandings or uncertainty about the intended meaning, the design of code patterns at some points is backed by consulting formalizations of UML semantics.

The semantic resolution from UML to Java is formalized by providing a model to model transformation covering most of the semantic resolution and a model to text transformation mainly for transcending the gap between model and code.

Even though thoroughly comparing semantics of models and code, consulting formalizations when necessary and formally defining our approach, the correctness of transformations and code

generation is not proved. The reason for this is that the basis for such a prove, a formal semantics of UML, is missing. Providing a formal specification of UML is only valuable, if the compliance to UML is somehow approved, i. e. it must be accepted by the UML vendor OMG. Otherwise, we could prove the correctness to a formal specification that is not necessarily a valid UML specification. Since for such a formalization and validation, remarkable effort is required, we decided to base on the informal specification and leave a formal validation for future work.

Our research is furthermore constrained by the following:
Considering structural and behavioral modeling in its completeness goes beyond the scope of this thesis. Those issues which are excluded from this work are not disregarded because of obvious difficulties in including them but rather because the scope of our work cannot cover the whole UML specification of structures and behavior. It cannot even cover the part of UML related to class diagrams and activity diagrams. For the modeling of classes with attributes and methods, we support *Class*, *Property*, and *Operation*, for the modeling of relationships between classes, we additionally include *Generalization* and *Association* as well as *AssociationClass*, all specified within the language unit *Classes*. These elements are specializations of other modeling elements such as *Classifier* or *Feature* which therefore are implicitly covered as well. Finally, we support *MultiplicityElement* and *VisibilityKind* which are basic concepts needed for modeling class features. Other elements of this language unit are not essential and therefore not included in our approach. Such elements are for example *Package* for providing mechanisms to group elements in packages, other kinds of relationships like *Realization* or *Usage* dependencies, the element *Type* for defining types or *ValueSpecification* for modeling values, e. g. for supplying attributes with default values. Another concept relating to association not considered in detail is *qualified association*.

Regarding behavior, we include the basic mechanisms needed for activity execution specified in language Unit *Activities*. These are *ActivityNode*, *ActivityEdge*, and *ActivityGroup*. Concerning nodes, we consider all kinds of *ControlNode*, *Pin* on the level needed for specifying object flows, and *Action*. Regarding flows, the effect of guards at activity edges is considered. Relating to *ActivityGroup*, we limit our approach to *InterruptibleActivityRegion*, which is covered in detail and include *ActivityPartition* on a very abstract level only. However, more advanced and special elements like *StructuredNode*, concepts for more sophisticated data processing like *ParameterSet* or *CentralBufferNode* are not addressed by this thesis.

The set of actions which are supported, is constrained to those actions which access the underlying static structure by reading or updating class properties or by creating or navigating association links. Some other actions which are needed to obtain context information, e. g. on which object to access a structural feature, are covered as well. Actions referring to elements of the static structure not covered in our approach as well as actions related to qualified associations, structured activity nodes, or modeling of variables are not considered. Thus, the language Unit *Actions* is not completely covered in our approach either. More precise hints on limitations of our work are contained in the sections where concerned parts of UML are discussed.

Parts of this work have already been published in [33, 34, 35, 36, 37, 39].

## 1.4 Basic Results

At this point, we glance over our research work and foreclose the main outcome.

The foundation of this thesis is a very detailed analysis of UML semantics of static structures and dynamic behaviors. This analysis reveals, that the semantics of UML cannot be easily mapped to programming languages for two reasons:

- object-oriented languages lack equivalents to some modeling elements, e. g. such as association
- even if adequate mechanisms exist in programming languages, the concrete mapping still may be very difficult, like it is when trying to implement control and data flows of activities.

Based on our analysis of UML semantics, the shortcomings of semantics itself are identified and solutions are proposed. The problem of mapping models to code is targeted by an evaluation of current implementation patterns. To overcome shortcomings of state-of-the-art code generation approaches, rich featured implementation patterns for *Classes* with *Attributes* and *Associations* are contrived. These pattens provide complete integration of multiplicity, navigability, and visibility of association ends and are applicable for the implementation of association classes.

Concerning behavior, Java implementations for *Actions* which have impact on static structures are developed. Beyond the semantics of UML, our approach offers a mechanism for cross checking multiplicity constraints implied in the static model in order to assure that no execution of any action that changes the set of values of a class attribute or the set of existing links of an association results in an invalid system state.

In order to support *Activities* which built the context of action executions, patterns for implementing control and object flows between actions are engineered, which are significantly closer to the intended semantics than state-of-the-art approaches. With regard to flows, an important contribution is the consideration of control nodes and the effects of guarded flows. Furthermore, the complex semantics of interruptible activity regions is properly considered.

Since UML is not specified in a formal way but informally by natural language, the semantic conformity of our proposed implementation patterns to the UML semantics is backed up by a very detailed analysis of UML semantics on the one hand as well as on the implementation patterns for static structure and for dynamic behavior on the other hand. Whether or not the implementations are fully UML compliant could be proved if a UML formalization was available. However, a formalization was not targeted in the scope of our research.

In order to not only give a catalogue of mappings between input models and target implementations, we decided to link our implementation patterns to UML by executable formalizations in *QVT-R* and *MOFM2T* which describe how to generate the source code as proposed by us for any input model. The *QVT-R* relations specify model transformations which show how to replace modeling concepts not available in the target implementation language by elements, which are offered in the implementation language as well. A transition from models to text, i. e. Java source code, is specified using *MOFM2T*, likewise. Both formalizations are prototypically implemented as eclipse plugins.

## 1.5 Organization of Thesis

In Chapter 2 we provide some basic background knowledge supporting a better understanding of the problems discussed and the solutions proposed. Chapter 3 gives a detailed discussion of the semantics of structural modeling in UML and appropriate code patterns for implementing static structure. A corresponding disquisition on the semantics of UML *Activity* and *Action* including suitable implementations follows in Chapter 4. The semantical resolution of modeling concepts is achieved by a model to model transformation. A formalization of this transformation using the OMG standard *QVT* (Query/View/Transformation) is presented in Chapter 5. The translation of the transformed model into code, which is given in Chapter 6, is implemented using another OMG standard, namely *MOF Model to Text Transformation Language* (*MOFM2T*). For evaluating our approach, we implemented a prototype. In Chapter 7, we discuss this implementation and compare it to other tools by means of benchmarks and a case study for proving feasibility. We discuss the keynotes of this thesis, survey related work and summarize our contributions in Chapter 8 and finally conclude in Chapter 9.

# Chapter 2

# Background

In this chapter, we provide some background of our work. We point out our view of model-driven software engineering, motivate the decision to use UML and sum up our previous work which pioneered this thesis.

## 2.1 Model-Driven Software Engineering

The desire for higher levels of abstraction caused the introduction of assembly language as an abstraction over machine code. Third-generation languages abstract from low-level, machine-specific instructions by introducing higher level abstractions (such as named variables and structured programming constructs). The translation to the underlying machine is achieved by compilers. With abstract data types and objects, the Object-Paradigm provides additional abstractions. [43]

MDD is the next step to higher levels of abstraction. Its objective is to increase productivity and quality by using modeling languages which provide semantically more advanced constructs and thus are closer to the problem domain and further away from platform specific implementation details. Fig. 2.1 shows the evolution of abstraction over time.

MDD's key challenge is to transform its abstractions in runnable code, i. e. transforming models to concrete representations [92, 43]. To this end, a promising perspective also associated with MDD is a higher level of automation for bridging the semantic gap between models and code [89].

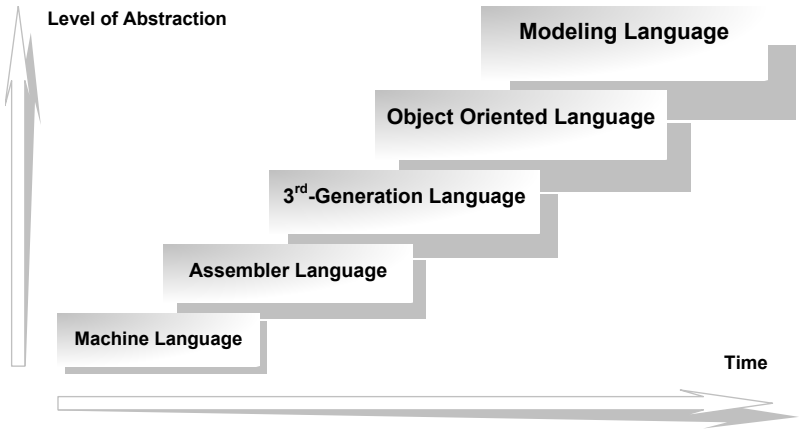



Figure 2.1: Evolution of languages and abstraction levels, based on [41].

MDD is one approach to face the continuously rising complexity of problems and solutions in software engineering. In this thesis, we do not mind about the question whether it is the best nor in which contexts MDD is or is not applicable. Provided that MDD is valuable in any concrete context, this thesis delivers insight into problems and possible solutions of using UML in MDD — and according to our and other's expertise, MDD is a valuable approach for at least some domains.

From our point of view, the phrase *model-driven* in contrast to *code-centric* clearly indicates that models are the central artifact in an approach based on MDD, not code. Code is derived from

models which themselves are a result of analysis and design phases of the development process. Although reverse and roundtrip engineering are quite popular and contained in the feature list of many tools, with our approach and implementation we solely target the model-to-code direction since we strongly belief conceptual work should be done on the highest possible level of abstraction. According to this claim, generated code is not the place where bug fixing or maintenance should be done.

A great benefit of modeling is that before a system is implemented, a formal representation of it exists which affords analysis of various characteristics by means of model checking or tests applied to the model. By accurately modeling and thoroughly analyzing, testing, and checking it, a model may attain to a level of best quality, preferably causing the implementation to be as well.

For generation of the full runnable code, models must cover a system's structure as well as its behavior [44]. UML supports both aspects but current UML tools implement both parts only rudimentary with little advance of structural aspects. Behavior is rarely supported and if, only the very basic constructs are considered.

## 2.2 Modeling Using UML2

Using UML is not only a promising option because OMG has established its MDA based on this language. Other considerations, too, suggest to apply UML.

### 2.2.1 General Characteristics of UML

Although suffering from numerous flaws, UML is popular, widely-used, and often, it is referred to it as the *de facto standard*, e. g. by Crane [18], Avgeriou et al. [2], or Broy and Cengarle [11].

Its specification is freely available for everyone and may be used for any purpose at no charge.

Being widely-used and supported by many tools, UML is of some relevance for developers as well as for researchers. Many approaches base on UML, a lot of research attends to it, altogether making sure that UML has a good chance to compete with other general purpose modeling languages in the long run. Furthermore, when basing research work on UML, there is a great chance to benefit from results of other researchers as well as from feedback of practitioners.

However, having no formal specification of its semantics, work with UML is challenging and sometimes even questionable. Nevertheless, we decided to base on UML, in particular on activities, in the tradition of our ActiveCharts project, introduced in Sect. 2.3.

### 2.2.2 UML Semantics Architecture

With his formalization of the semantics of activities, Sarstedt [83] is one of the first researchers addressing this formalism, which has been fundamentally revised with the release of UML2. UML semantics project is another attempt to base UML's semantics on a formal foundation [12]. Unfortunately, the project has been discontinued only having finished a fraction of the aspired extent.

Crane contributes to the field by a formalization of UML's three-layer semantics architecture of behavior [18], which is shown in Fig. 2.2.

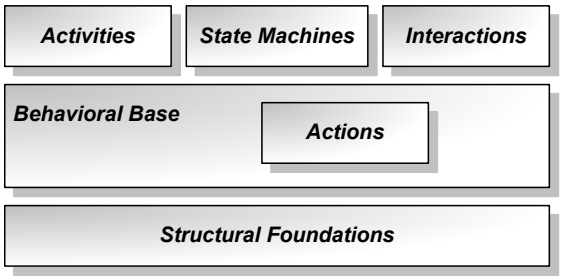



Figure 2.2: The UML three-layer semantics architecture, based on [69, 18, 39].

The basic idea of this three-layer architecture is to define actions, which access the underlying structural model, e. g. by reading or updating values of class attributes or by creating or navigating along association links. Higher level behavioral formalisms are built on the basis of actions.

Here, activities are of particular interest since actions can only be contained by them [10, 69]. Consequently, other formalisms can access the structural foundation only by invoking activities providing the desired functionality by using appropriate actions.

The UML specification is divided into different language units. From these, the units *Activities*, *Actions*, and *Classes* directly relate to the three-layers semantics. Figure 2.3 shows the three language units as boxes with round corners. The class *Person* with an attribute and two operations builds the structural foundation, specified in the language unit *Classes*. The action *writeName* is an element of language unit *Actions*. It is associated with the attribute name of class *Person* indicated by a dashed arrow, i. e. the action accesses a structural feature by setting an attribute to a given value. Actions are contained in activities, which might serve as specifications of operations, like activity *SetName* specifying a class operation, indicated by the dotted lines stretching from the operation to the activity frame. The parameter of the operation corresponds to the parameter node of the activity, indicated by a dashed arrow. The given activity with all of its containing elements except of the actions is specified by the language unit *Activities*.

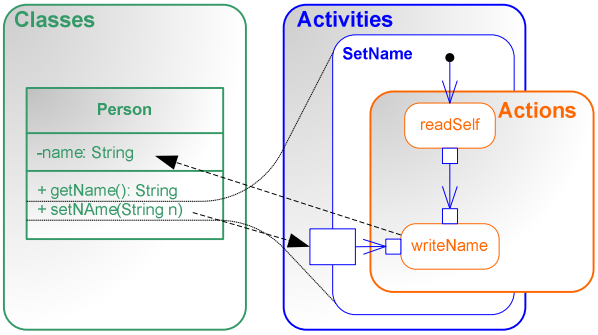



Figure 2.3: Relationships between structural and behavioral modeling concepts in UML.

### 2.2.3 Interplay of Structures, Actions, and Behaviors

Behavioral models depend on structural models if actions access structural features. Behavioral models may also serve as a specification of operations contained in structural models. This is illustrated in Fig. 2.4: An excerpt of the UML meta-model is given on the left, a concrete instance on the right. Gray shaded beams connect meta classes with their instances in the model. The bold framed actions *read* and *write* are instances of specializations of the bold framed *StructuralFeatureAction*. Dashed arrows pointing from actions to attributes and from operations to activities represent instances of the bold printed meta associations between the corresponding meta classes [39].

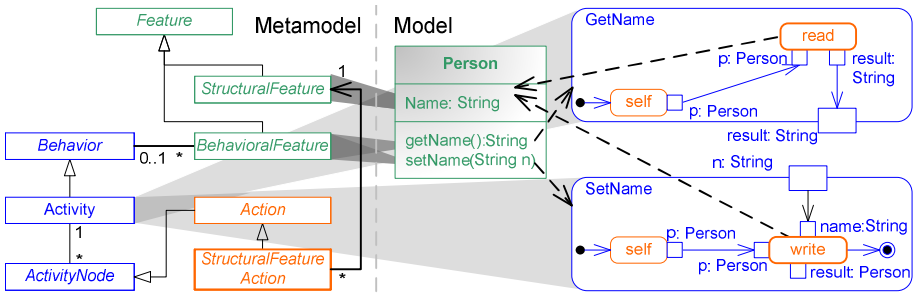



Figure 2.4: Excerpt of the UML metamodel showing dependencies and relationships between some basic modeling elements (left) and notation of these elements (right), based on [39].

## 2.3 ActiveCharts — The Project Pioneering this Thesis

With ActiveCharts [84, 85, 86] an approach has been introduced that combines modeling using UML2 and coding in a new way. The basic idea is to completely model static structures by means of class diagrams, whereas dynamic aspects of the system are partially modeled as activity diagrams and to some extent are expressed in code. Initially, ActiveCharts has been based on an activity interpreter that executes activities according to the token flow semantics defined by UML.

Whereas UML specifies numerous kinds of actions for different purposes, ActiveCharts only supports *CallBehaviorAction*. The specification intends to associate a *CallBehaviorAction* with a behavior which is executed when the control flow reaches the action. In ActiveCharts, *CallBehaviorAction* may be used complying to UML specification to invoke a behavior associated with the action. If not associated with another behavior, *CallBehaviorAction* builds the connection from models to code: for each *CallBehaviorAction* contained in an activity, a method of the same name with conforming parameters must be present in the class implementation of the owning classifier that represents the context of the *CallBehaviorAction*.

By generating code for static structures, implementations of actions get access to features of the context classifier.

In ActiveCharts, activities are used to model behavior to an arbitrary degree. It is not assumed that covering all behavioral details of a system by models is feasible. For example, sort or search algorithms have been elaborated and implementations exist in many languages. It is much more convenient to embed such implementations in code than to translate them into activities.

For this reason, ActiveCharts seamlessly combines user code with models. The code, e. g. a search or sort algorithm, makes up the body of a method which is represented as a *CallOperationAction* in an activity. By that, details like when to invoke the method, where to get inputs from and which action to provide outputs to are all captured by activity diagrams.

If applied adequately, models are clear and readable as well as the code, which comprises methods providing solutions to special, self-contained tasks by implementing possibly commonly known, highly sophisticated and efficiently designed algorithms.

Although activities are processed by an interpreter, code generation is part of ActiveCharts with respect to static structures. Beyond generating classes with members for attributes and association ends, methods are generated to access members as well as to manage associations. Depending on the upper bound of a feature, methods are generated for setting its value or adding a new value, for reading its value or its set of values, and for un-setting its value or removing one (or all) value(s).

The names for these methods are derived from the names of attributes and association ends by prepending the appropriate prefix *set*, *add*, *unset*, *remove*, or *get*. In user code, structural features may be accessed by invoking these generated methods.

## 2.4 Advanced Code Generation with ActiveCharts

Improving code generation for ActiveCharts is causative for the research work presented in this thesis. As a result of this, code generation from models of the static structure has been revised in particular with regard to associations and multiplicity specifications of structural features. With a newly elaborated pattern which supports the implementation of n-ary and binary associations as well as association classes considering the detailed semantics of visibility and navigability of association ends, the results of this thesis excel the capabilities of code generation for static structures offered by ActiveCharts.

Concerning Activity, ActiveCharts is based on an interpreter approach with a special focus on *difficult concepts* which are often excluded when dealing with activities, such like concurrency or *InterruptibleActivityRegion*. We now offer translation of activities into code covering in particular concurrency, complex compositions of control and object flows and *InterruptibleActivityRegion*. By this, the presented translation approach is as mighty as the interpreter of ActiveCharts, however, it is noticeably faster and scales better if applied to larger systems.

# Chapter 3

# Semantics of Static Structures

UML offers concepts for modeling static structure on different levels. The lowest level comprises the basic modeling concepts of UML which relate to class diagrams. In particular, these are classes and their relationships. Upon them, more complex structures which are called *Components* are defined. *Composite Structures* support the composition of interconnected elements which are collaborating to achieve some common objectives. The execution architecture of a system, i. e. the assignment of software artifacts to hardware devices or software execution environments, is covered by *Deployments* [69].

In this chapter, we discuss the semantics of the basic UML concepts for modeling static structure, as specified in the language unit *Classes*. In particular, we deal with *Class*, *Property* considering multiplicities, *Association* including higher order associations as well as aggregation and composition, and *AssociationClass*. The specification of these elements is shown by an excerpt of the UML metamodel in Fig. 3.1.

First, we introduce the semantics of typical object-oriented concepts as is informally given by the specification of UML in Sect. 3.1. In Sect. 3.2, we concentrate on the semantics of association, a concept going beyond the core of the object paradigm. We regard shortcomings of UML concerning structural modeling concepts in Sect. 3.3. Afterward, we survey state-of-the-art code generation approaches for attributes in Sect. 3.4 and for associations in Sect. 3.5, where we additionally outline how implementations of associations suffer from the inappropriate specification. Finally, we present solutions to those problems of code generation which are not covered by existing approaches by defining a mapping from modeling elements to code which — for all supported concepts — is fully complying to the semantics of UML. In Sect. 3.6, we provide some technical, implementation specific details to unsolved problems of association implementation in Java. We summarize this chapter in Sect. 3.7.

Some of the concepts and ideas discussed here were also examined and analyzed by Waniek in his diploma thesis [99], which I supervised. The aim of Waniek's work is to prove the applicability of code generation patterns. Some of its findings have been further refined as part of this work, others are beyond the scope of this thesis and therefore are not included here.

## 3.1 Object-Oriented Concepts of Structural Modeling

*"Built upon fundamental OO concepts including class and operation, UML is a natural fit for object-oriented languages and environments such as C++, Java, and the recent C#"* [67]. In fact, the semantics of class, attribute and method as known from object-oriented programming languages comply with *Class*, *Property*, and *Operation* of UML, except for some discrepancies probably induced by the complexity of the UML meta-model. Regarding these concepts, the level of abstraction of UML equals to that of object-oriented languages.

### 3.1.1 Class, Property, and Operation at a Glance

In UML, a class describes a set of objects which share the same features. Such a feature is either a *StructuralFeature*, i. e. an attribute of a class represented by an instance of *Property* or a *BehavioralFeature*, i. e. an *Operation* [69, § 7.3.7] which corresponds to a method.

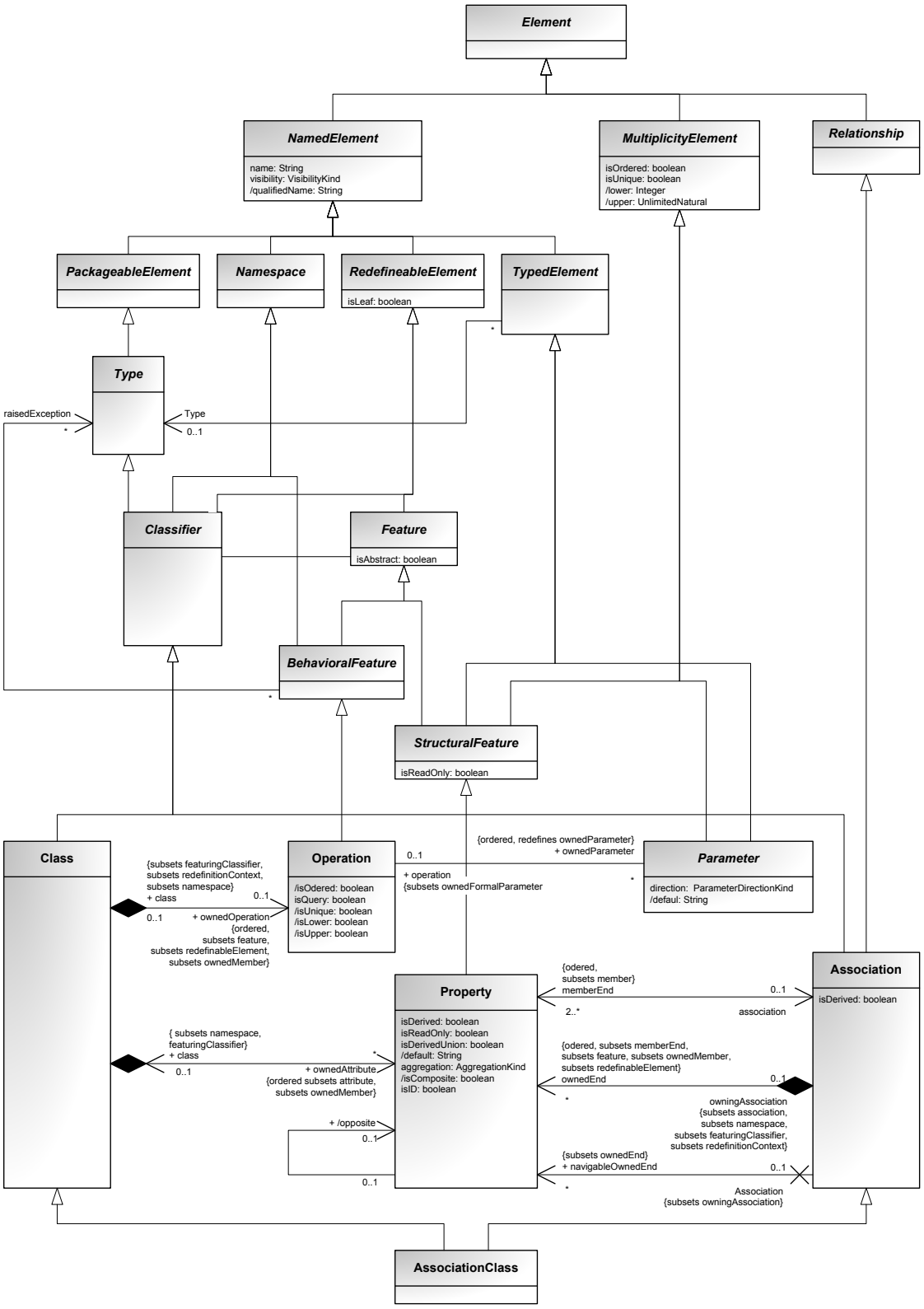



Figure 3.1: Excerpt of the UML Classes meta-model.

Being a *Classifier*, a class can be related to another class by a directed relationship specifying a classifier to be a specialization of another, more general one. This relates to the same concept in programming languages where subclasses refer to their superclasses whereas from a superclass, navigation to subclasses is not provided.

*Class* and *Property* are typed elements meaning that they can only represent instances of the specified type. Furthermore, both as well as *Operation* are *NamedElements*. Apart from having a name, *NamedElements* have a visibility. In UML, visibility is defined as follows and is almost identical to visibility modifiers in object-oriented languages [69, §7.3.56]:

- a *public* element is visible to all elements that can access the contents of the namespace that owns it.
- a *private* element is only visible inside the namespace that owns it.
- a *protected* element is visible to elements that have a generalization relationship to the namespace that owns it.[1]
- a *package* element is owned by a namespace that is not a package, and is visible to elements that are in the same package as its owning namespace. Only named elements that are not owned by packages can be marked as having package visibility. Any element marked as having package visibility is visible to all elements within the nearest enclosing package (given that other owning elements have proper visibility). Outside the nearest enclosing package, an element marked as having package visibility is not visible. [69, p. 141]

*Property* is an indirect subclass of *MultiplicityElement*. Therefore, each attribute of a class has a multiplicity. *"A multiplicity is a definition of an inclusive interval of non-negative integers beginning with a lower bound and ending with a (possibly infinite) upper bound"* [69, §7.3.33].Multiplicities are used to determine how many values an attribute must at least and may at most hold at a time.

### Semi-Formal Definition of an Attribute

Diskin and Dingel [22] give an intuitive, semi-formal understanding of association from which a semi-formal definition of attributes is derivable. An attribute $a$ of type $Y$ owned by class $X$ as in Fig. 3.2 is a mapping defined by the function

$$a_X : X \to \mathcal{P}(Y).$$


Figure 3.2: Classes related by an attribute.

For $x \in X$, $a_X(x) = x^{\to a}$ where $x^{\to a} \in \mathcal{P}(Y)$ denotes the set of instances which are values of attribute $a$. Let $l$ be the lower bound of $a$ and $u$ its upper bound, then the size of $x^{\to a}$ is

$$l \leq |x^{\to a}| \leq u.$$

If the upper bound is 1, attribute $a$ is a function

$$a_X : X \to Y.$$

Such an attribute is also called a *single-valued* attribute, since at most one value for it may exist.

If a lower bound is 1, attribute $a$ is a mapping

$$a_X : X \to x^{\to a} \in \mathcal{P}(Y), x^{\to a} \neq \emptyset.$$

Such an attribute is also called a *mandatory* attribute, since it must always hold at least one value.

If upper and lower bounds are 1, a mandatory, single-valued attribute is specified, which always must hold exactly one value. Attributes which are not single-valued are referred to as *multi-valued* attributes.

[1] In Java, a member with protected visibility is accessible in subclasses and in elements located in the same package.

### 3.1.2 Discrepancies between UML and Programming Languages

On the face of it, *Class*, *Property*, and *Operation* of UML exactly fit to classes, attributes and methods in object-oriented languages. On closer inspection, some discrepancies become visible, which we will shortly explain in the following:

- Parameters of *Operation* may be featured with characteristics not applicable to method parameters
- For attributes, a minimum amount of values may be specified
- Attributes may be featured with a special aggregation semantics.

Operations may have parameters. *Parameter* is a specialization of *NamedElement*. Hence, it has a visibility, but the semantics of parameter visibility, in particular if parameter visibility and operation visibility are different, is not specified. We suspect that this is an effect of being caught in the *overgeneralization trap* [91] — the inadvertent outcome of unanticipated conflicts between multiple generalizations in deep and complex generalization hierarchies by which parameter inherits visibility, an attribute actually not needed for parameters.

A class property in UML as well as an attribute in an implementation class may consist of a set of values. A lower bound greater than 0, say 1, forces an attribute to always hold at least one value, which is similar to the effect of a *not null* statement in *SQL*[2] column definitions. Upper bounds of an attribute relate to the size of a set or the length of an array implementing that attribute, however, lower bounds greater than 1 cannot directly be mapped to characteristics of member declarations in object-oriented languages.

Another meta-attribute of *Property* is aggregation. This attribute defines whether values assigned to the property are parts of the owning classifier. If they are, the owning classifier has responsibility for the lifetime of its owned parts, i. e. if the owner of parts is destroyed, the parts must either be removed before or be destroyed, too. This responsibility relates to lower bounds of attributes: if an object which is included in the set of values of an attribute owned by another classifier is destroyed, the lower bound of this attribute may be violated.

In either cases, composition as well as lower bound violation, structural integrity is preserved by a cascading destruction or by rejection of a requested destruction. Unfortunately, in order to come to the point, cascading destruction or rejection of a destruction request is not pursued by semantics of action, as it is discussed in Chapter 4.

These three examples show that although UML is a close match to object-oriented languages, it cannot be seen as an exact image thereof.

## 3.2 Structural Modeling beyond Object-Oriented Concepts

In contrast to the close match between classes and their features in UML and those in programming languages, other concepts of UML do not have any equivalents provided by object-oriented programming languages at all. A foundational concept of UML structural modeling is *Association*, which is not directly supported in mainstream languages. For this reason, a detailed examination of the semantics of *Association* is the basis for supporting adequate code generation.

The presentation of UML modeling concepts in this section is based on and extends the introduction of association as provided in our previous work [33].

Many patterns for implementing associations have been elaborated in recent years, however, we are not aware of any covering all the aspects which a UML association may be featured with. Probably, the hardest problems with regard to implementation are caused by UML offering the ability to specify associations between more than two classes. In contrast to binary associations, i. e. association between two classes, *"higher-order (e.g., ternary) associations are difficult to implement and are generally avoided"* [93].

Since higher order associations indeed are difficult to understand as well as to implement, we concentrate on binary associations in the following sections unless otherwise stated. But in order to be able to make use of higher order associations in system engineering, we fully support higher order associations and provide the necessary details for sufficient implementation.

[2]Structured Query Language, ISO/IEC 9075

### 3.2.1 Constituent Parts of Association

> "An association specifies a semantic relationship that can occur between typed instances. It has at least two ends represented by properties, each of which is connected to the type of the end. (...) Each end represents participation of instances of the classifier connected to the end in links of the association." [69, §7.3.3]

The lower part of the class diagram of Fig. 3.1 shows the abstract syntax of *Association*. Since association ends are properties, they are named, have a multiplicity and a visibility. They are owned either by the association itself or by the classifier connected at the opposite end of the association. Further, some special aggregation semantics may be specified. Figure 3.3 gives a general view on the notation of association. Optional items are printed gray.


Figure 3.3: UML Association.

Associations are defined between classes, the relation between instances of classes is represented by links. An association can be seen as a table whose rows are tuples in the association and columns are the association ends [22]. Another view of an association is to consider it as a set of mappings, e. g. for the binary association between classes $X$ and $Y$:

$$f_1 : X \to \mathcal{P}(Y),$$
$$f_2 : Y \to \mathcal{P}(X)$$

For higher order association, such mappings become more complex. A ternary association between classes $X_1, X_2, X_3$ may be considered as a triple of binary mappings [22]:

$$f_1 : X_2 \times X_3 \to \mathcal{P}(X_1),$$
$$f_2 : X_1 \times X_3 \to \mathcal{P}(X_2),$$
$$f_3 : X_1 \times X_2 \to \mathcal{P}(X_3).$$

The specification of *Association* is highly inconsistent. Effects of combining characteristics like e. g. navigability and ownership of association ends are defined contradictory to the actual implications of ownership on navigability. In order to crave out such inconsistencies, we first present the intended semantics of associations before we oppose the intentions to the real facts.

### 3.2.2 Ownership of Association Ends

An association end may be owned by the association itself as an *ownedEnd* or by a member end class as an *ownedAttribute*. If an end is owned by a class, it is a structural feature of it representing an *ownedAttribute* of this class. As such, it becomes part of the object's state of class instances on which adding or removing a link will have impact. For ends owned by associations, the same rules hold, but the impact is on the association and therefore not emergent in linked instances [33]. Figure 3.4 shows an association where the end connected to class $B$ is owned by class $A$, indicated by the dot (•) at end $b$. Since this end is owned by class $A$, it may by displayed in the attribute compartment of $A$ as well.

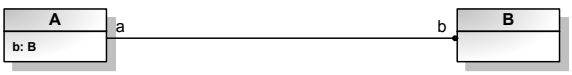



Figure 3.4: Association end ownership notation.

For a binary association between classes $X$ and $Y$, the mappings $f_1, f_2$ defining this association can be attributed to participating classes, such that $f_1 : X \to \mathcal{P}(Y)$ is related to $X$ and $f_2 : Y \to \mathcal{P}(X)$ is related to $Y$. By doing so, $f_1, f_2$ also can represent an owned attribute of $X, Y$. Therefore, the specification considers ends of binary associations to possibly be owned by participating classes.

In either case, the relation between the objects remains unaffected, however, implementation of links does not: as suggested by Diskin and Dingel [22], ownership of association ends should be considered by the implementation of links, in particular where links are to be stored.

Note that for higher order associations, all ends must be owned by the association. Mappings defining higher order associations are of the form

$$f_1 : X_2 \times \ldots \times X_n \to \mathcal{P}(X_1), \ldots, f_n : X_1 \times \ldots \times X_{n-1} \to \mathcal{P}(X_n) \text{ [22].}$$

Function $f_1$ can be attributed to any or all classes except $X_1$, say $X_2$, however, $X_3, \ldots, X_n$ then must be provided as parameters — or in the terminology of UML, as qualifiers. Since UML does not define qualified properties, association ends of higher order association cannot be owned by participating classes.

### 3.2.3 Symmetry of Associations

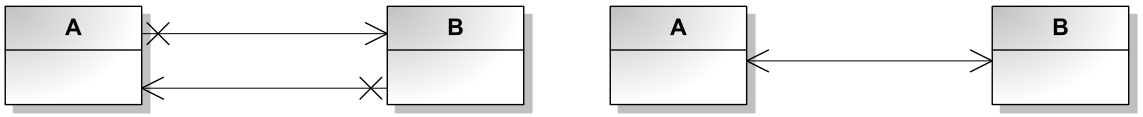
An association is a constraint given to two or more properties which are the association ends. While a property only representing an attribute of a class may hold any value of the specified type, properties additionally representing association ends cannot.

For an association between classes $X, Y$, and mappings

$$f_1 : X \to x^{\to} \in \mathcal{P}(Y), x^{\to} = \{y_1, \ldots, y_k\},$$
$$f_2 : Y \to y^{\to} \in \mathcal{P}(X), y^{\to} = \{x_1, \ldots, x_m\}$$

the condition $y \in x^{\to} \Leftrightarrow x \in y^{\to}$ must always hold [22].

Since association is not an object-oriented concept, its implementation must be based on attributes, resulting in a set of uni-directional links as shown in Fig. 3.5(a). Here, the links are shown as connectors with a UML-like navigability indication. Association is an abstraction of such a set of uni-directional links to a single bi-directional link, depicted as a single connector in Fig. 3.5(b). Such an abstraction of a collection of connectors into a single connector at the higher abstraction level is called a cable [90].




(a) Two uni-directional relations between two classes.

(b) A single bi-directional relation between two classes.

Figure 3.5: Cable Pattern: abstraction of a collection of connectors into a single connector.

### 3.2.4 Multiplicities of Association Ends

An essential feature of associations are multiplicities at association ends. According to the semantics of multiplicities of attributes, multiplicities at association ends determine how many instances of the associated classes can or must be linked at a time [33].

For the sets $x^{\to}$ and $y^{\to}$ and multiplicities [k..l] at the X-end and [n..m] at the Y-end,

$$k \leq |y^{\to}| \leq l \text{ and } n \leq |x^{\to}| \leq m$$

must hold.

Figure 3.6 shows the notation for an association according to the given generic example.

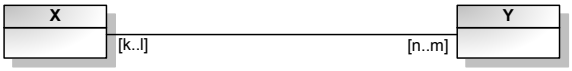



Figure 3.6: Multiplicities at association ends.

### 3.2.5 Navigability of Association Ends

> "Navigability means instances participating in links at runtime (instance of an association) can be accessed efficiently from instances participating in links at the other ends of the association." [69, §7.3.3]

An association end is navigable when it is either a *navigableOwnedEnd* of the association or an *ownedAttribute* of an end class. Otherwise, it is not navigable [69, §7.3.3].The specification does not claim that a non-navigable end must not be accessed by opposite ends. It rather states that it may or may not. Accordingly it is not necessary to store links directed in a non-navigable direction of an association, however, it is not invalid to store them either. An association between $X$ and $Y$ navigable from $X$ to $Y$ as in Fig. 3.7 is sufficiently represented by a single mapping $f : X \to \mathcal{P}(Y)$ [33].


Figure 3.7: Navigability symbols at association ends.

An informal convention whereby non-navigable ends are assumed to be owned by the association whereas navigable ends are assumed to be owned by the classifier at the opposite end is deprecated [69, §7.3.3], but we find a dependency when having a look at the inversion. A non-navigable end must be owned by the association. If owned by the class via *ownedAttribute*, it is navigable [33].

### 3.2.6 Visibility of Association Ends

The visibility *"determines where the NamedElement appears within different Namespaces within the overall model, and its accessibility."* [69, §7.3.34]

The specification allows to define a non-navigable end as public. Such an end must be owned by the association (because it is non-navigable) but being public, it is accessible. It can be used to create links, however, the effect of accessing a non-navigable end for reading values is unclear [33].

An example for visibilities of association ends is given by Fig. 3.8. In the scope of class $B$, all ends $w$, $x$, $y$, and $z$ are visible. From class $A$, the end $w$ is accessible through $b.w$, since it is public. The end $x$ is accessible through $b.x$, if classes $A$ and $B$ are members of the same package, the end $y$ is accessible through $b.y$ if $A$ is a (possibly transitive) specialization of $B$, and finally, the end $z$ is never visible to $A$ since it is of private visibility.

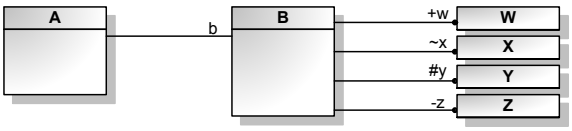



Figure 3.8: Visibilities of association ends.

### 3.2.7 Aggregation Types of Association Ends

An association end has an *aggregationType* which may have one of the values

- *none*
- *shared*
- *composite*.

The values *shared* and *composite* may only appear in binary associations and indicate a whole/part relationship. Aggregation is transitive and may not be modeled in a cyclic way. An association end with *aggregationType composite*

> "indicates that the property is aggregated compositely, i. e. the composite object has responsibility for the existence and storage of the composed objects (parts)." [69, §7.3.2]

This implies that the deletion of the composite entails the deletion of all of its current parts unless these parts have been removed from the composite before deletion [33].

## 3.3 Implications of Combining Concepts

Concerning the presented modeling concepts, the specified semantics seems to fail implementing the intended semantics. This certainly is the result of a non-formal specification containing examples based on the graphical notation of UML diagrams which is an abstraction over the underlying model.

We discuss the stated orthogonality as well as replaceability of modeling concepts which both are proof of either flawed specifications or inopportuneness of varying the level of abstraction between explained facts and examples given to ease perceivability.

### 3.3.1 Orthogonality of Modeling Concepts

The specification states that "aggregation type, navigability, and end ownership are orthogonal concepts," [69, §7.3.3].

We agree to this with regard to an end's aggregation type.

Concerning navigability and end ownership, it is not clear whether or not both concepts are orthogonal since *navigability* denotes two different things, depending on the context:

- navigability may denote the concept of association ends to be navigable or not (e. g. depending on the navigability of an end, . . . )
- navigability may denote the fact, that an association end is navigable (e. g. the navigability of end A guarantees the access from end B, the non-navigability of the B end prevents from any access in the opposite direction.)

Considering the latter, more special case, both concepts are orthogonal since a navigable end may be owned by both, an end class or the association. However, in the context of the former, more general meaning of navigability, both concepts are not orthogonal:

- an association end owned by a class via *ownedAttribute* implies that this association end is navigable
- being not navigable implies that the association end is owned by the association via *ownedEnd* and must not be included in the subset *navigableOwnedEnd* [33].

We suspect that in the citation above, the general meaning of navigability applies, because Fig. 3.9[3], which is taken from the specification, shows the combination of ownership with navigability — in the sense of defining an end to be navigable or non-navigable — on the right end of an association. This figure gives an example for combining line path graphics, however, it unintendedly proves that orthogonality of ownership and navigability is assumed although ownership has impact on navigability: the right end is owned by class *A*, indicated by the black dot (•) at the association end, and therefore cannot be non-navigable as indicated by the crossed lines (×) at the same end.


Figure 3.9: Invalid combination of association end ownership and navigability, according to [69, §7.3.3, p. 41].

A similar dependency exists between navigability and visibility. Consider a *navigableOwnedEnd* of an association with *private* visibility:

- being navigable, it is supposed to be efficiently accessed by instances of classifiers at the opposite end,
- being private and not owned by opposite end classifiers, it is supposed to be not accessed by instances of them at all [33].

[3] This is Figure 7.20 of the UML Superstructure [69]

### 3.3.2 Impact of Arity

Higher order associations differ from binary associations not only in the number of association ends. In the following, we discuss the three most relevant differences between binary and higher order associations which affect the semantics of three features of association ends:

- visibility
- navigability
- lower bounds.

The impact on visibility is caused by an additional constraint over end ownership. For binary associations, ownership of association ends offers two arbitrary options: an end may be owned by a member end class or by the association. If owned by a member end class, a private end is accessible to the member end class exclusively. If a private end is owned by the association, it is not accessible to any class participating in the association or any other class of the system.

If an association has more than two ends, all these ends must be owned by the association [69, §7.3.3]. A consideration of feasibility of this constraint is given in Sect. 3.2.2. If an end is private, it is not accessible from classes participating in the association, if it is public, it is visible to all classes, regardless of whether they participate in the association or not.

Therefore, we identify an impact of the arity of an association to visibility and consequently to accessibility of association ends. Whereas in binary associations, participating classes may be privileged, this is not possible for higher order associations. In consequence, it is impossible to conceal the relation of objects from other objects which are not parts of a link. Links of higher order associations are always publicly[4] visible and no secret matter of concerned classes and their instances.

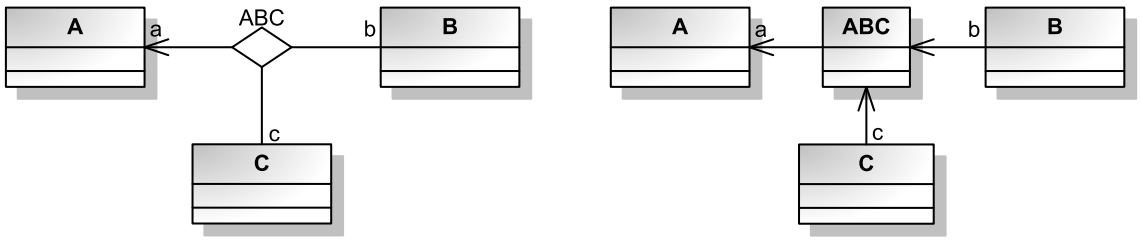
Another problem of higher oder associations is an impact on navigability. Navigability is defined as the efficient access to instances referenced by a navigable end. The problem here is, that for accessing a navigable end, $n-1$ values for all the other ends — the so called *context* — must be provided by the instance navigating across the association. In binary associations, the context is atomic, comprising only the instance for which related instances are queried. In higher order associations with $n$ ends, the context is a composite comprising $n-1$ ends, i. e. to retrieve a link, a $n-1$ tuple must be already known.

Based on the consideration of mappings implementing an association, the design fault of navigability is that for an association between classes $A, B, C$ with ends $a, b, c$ as in Fig. 3.10(a), navigability of end $a$ is associated with a mapping $f_a : B \times C \to \mathcal{P}(A)$. But how can the context $B \times C$ be obtained at runtime, if either one or both ends $b, c$ are non-navigable?

A solution could be to define navigability of end $a$ by a set of mappings — for each element of the context — $f_{a_b} : B \to \mathcal{P}(A \times C)$ and $f_{a_c} : C \to \mathcal{P}(A \times B)$. This view is motivated by the question, what higher order associations are good for, if obtaining information from it requires to already know most of the desired information. If one is interested in all tuples in which one specific instance is referenced, it is necessary to query the association with all possible contexts. We assume that handling higher order associations is easier to understand as well as to implement if it is possible to obtain all tuples in which a given instance is included and to filter from this set of tuples. This view furthermore is closer to the result of replacing higher order associations by a set of binary associations as in Fig. 3.10(b) and might countervail the avoidance of higher order associations [93]. In the modified model, it is possible to access end $a$ only providing one element of the former context e. g. from an instance of $B$ via the mappings $f_{ABC_b} : B \to \mathcal{P}(ABC)$ and $f_{a_{ABC}} : ABC \to \mathcal{P}(A)$.

Finally, arity of associations also has impact on the semantics of lower bounds. In a binary association, a lower bound of 1 at one end demands that each instance of a classifier connected to the opposite end participates in at least one link of the association. Relating to the association between $X_1, X_2, X_3$, a lower bound of 1 at the $x_1$-end demands a link for all possible pairs $X_2 \times X_3$ with an instance of $X_1$. A probably more intuitive view is that for each existing element of $X_2 \times X_3$, a link must exist which associates this tuple to an instance of $X_1$, as proposed by Génova et al. [31]. The fact that UML expressly defines the former interpretation of minimum multiplicities in higher order associations causes binary and multi-ary associations to be qualitatively different [22].

[4] We subsume package visibility here as well, which is public visibility constrained to the publicity of a package.




(a) An association between three classes.

(b) A transformation of the association from (a) into a set of binary associations and an additional class.

Figure 3.10: Transformation of a ternary association into a set of binary associations.

### 3.3.3 Replaceability of Modeling Concepts

A serious shortcoming of the UML specification is that it is suggestive of a replaceability of association ends and attributes, if ends are owned by a binary association.

#### Owned Attributes versus Association Ends

The UML specification postulates a replaceability of attribute and association notation. The transformation from attribute to association notation (which we term $\mathbf{T_1}$) is described as follows:

> "The attribute notation can be used for an association end owned by a class, because an association end owned by a class is also an attribute. This notation may be used in conjunction with the line-arrow notation to make it perfectly clear that the attribute is also an association end." [69, §7.3.3]

Instead of *"may be used"*, it should rather state *"can only be used* in conjunction with (...)". A proper application of this transformation is discussed by Crane et al. [19], where classes are enriched by attributes instead of replacing associations.

An analogous transformation from the attribute to the association notation (which we term $\mathbf{T_2}$) is stated as follows:

> "An attribute may also be shown using association notation, with no adornments at the tail of the arrow (...)" [69, §7.3.8].

Since an association at least has two ends but only one attribute is intended to be shown, some characteristics of the second end must be specified correctly. The item *"with no adornments at the tail of the arrow"* inhibits the explicit definition of multiplicity and navigability for this end. By default, upper and lower bounds of a multiplicity are 1 and according to no specification, these values are to be applied. We assume that the owner of the second end is the association since the structure of the referenced classifier should remain unchanged, but navigability is undefined since ends owned by an association may be either navigable or non-navigable. $\mathbf{T_2}$ is supposed to be the inverse transformation to $\mathbf{T_1}$.

#### Replaceability of Attribute and Association Notation

In general, an association end owned by an end class is not equal to an attribute which is not representing an association end.

Figure 3.11 shows two class diagrams. $\mathbf{T_1}$ allows the transformation from the left to the right diagram, $\mathbf{T_2}$ allows the transformation in the opposite direction. For the end $b$, no multiplicity is specified, since this is prohibited by $\mathbf{T_2}$, and by $\mathbf{T_1}$, a multiplicity specification is not considered, as it cannot be modeled using the attribute notation anyway. Because of the missing specification, the multiplicity of the end $b$ is 1, which is the default multiplicity.

The implication of this default multiplicity is, that each instance $a : A$ must be linked (or in terminology of Java: must be referenced) by exactly one instance $b : B$. If $a : B \to \mathcal{P}(A)$ denotes the mapping from an instance of $B$ to values of its attribute $a$, the following condition must hold: $\forall a_i : A \; \exists \; b_j : B$ such that $a_i \in a(b_j)$ and $\forall j_1, j_2 : j_1 \neq j_2 \Rightarrow a(b_{j_1}) \cap a(b_{j_2}) = \emptyset$.
This is much more restrictive than the common view of attributes and the way all code generators we evaluated generate code or implementors would write it.

According to the meta-model, the underlying models of the two diagrams are not identical. An association has at least two ends, each of which is a *Property*, a subtype of *MultiplicityElement*. An attribute is a *Property*, too, but when replacing an association by an attribute, there is no second *MultiplicityElement* contained in the model. This can be seen when comparing the tree views of Fig. 3.12 representing the models given in Fig. 3.11. While the tree in Fig. 3.12(a) contains a property named *b*, this property is missing in Fig. 3.12(b).


(a) Association notation of Fig. 3.11(b).

(b) Attribute notation of Fig. 3.11(a).

Figure 3.11: Association notation vs. attribute notation.

```
Class
  Name=A
Class
  Name=B
  ownedAttributes
    Property
      Name=a
      Type=<class::A>
      visibility=public
      navigability=true
      lower=...
      upper=...
Association
  Name=AB
  memberEnd
    Property <class::B:a>
    Property
      Name=b
      Type=<class::B>
      visibility=public
      navigability=false
      lower=...
      upper=...
  ownedEnd
    Property <Property::AB.b>
  navigableOwnedEnd
```

(a) Tree view representation of association notation as in Fig 3.11(a)

```
Class
  Name=A
Class
  Name=B
  ownedAttributes
    Property
      Name=a
      Type=<class::A>
      visibility=public
      navigability=true
      lower=...
      upper=...
```

(b) Tree view representation of attribute notation as in Fig 3.11(b)

Figure 3.12: referring to Fig. 3.11.

Although association notation and attribute notation are exchangeable, the examples given in the UML specification do not reflect the differences between the concepts on the level of the meta-model.

Modeling a reference from $A$ to $B$ with no restriction about how many references to an instance of $B$ may exist at a time can be achieved by an attribute or by means of an association with appropriate multiplicities. The replacement rules as stated in the specification lack a proper consideration of multiplicities and thus entail an unnatural semantics that obviously is not intended.

The specification does not define where links are to be stored. This implies that parts of links may not be stored at all if these parts are derivable from other information. Diskin and Dingel [22] give a detailed discussion about that. An example is given in Fig. 3.13 showing an association that

is navigable from $A$ to $B$. Instances of $B$ do not need to know to which instances of $A$ they are related. Therefore, links may be implemented by a single multi-valued attribute e. g. by a static `Map<A,Set<B»`, accessible through the class $A$. The multiplicity constraint of the $A$-end can be satisfied by checking the size of sets before inserting new values thus assuring that each instance of $B$ is not contained in more than 10 sets, i. e. is not referenced by more than 10 instances of $A$. This example shows, that an association may be implemented by means of attributes. But the replaceability as stated in the specification is to general and needs to consider the constraints under which both variants are semantically identical.

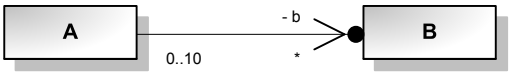



Figure 3.13: Association navigability and end multiplicities.

## 3.4 Implementation of Attributes

Implementation of attributes comprises a structural and a functional part. The structural part is to add a class member that can hold attribute values. The functional part is to provide means for reading and updating attribute values.

Typically, implementation of attributes focuses on the structural part, i. e. it is associated with a member declaration for storing the attribute value or a collection of values, if the attribute is multi-valued. Upper bounds may be considered if using arrays of appropriate size for multi-valued attributes. If using other kinds of data sets which dynamically grow with the number of actually stored values, upper bounds possibly can only be considered if the data set supports the specification of a maximum size. Thus, if applying data sets of an appropriate fixed size, the structural specification of an attribute copes with upper bounds. For lower bounds, this is not possible in standard object-oriented languages, since data sets lack a minimum size constraint.

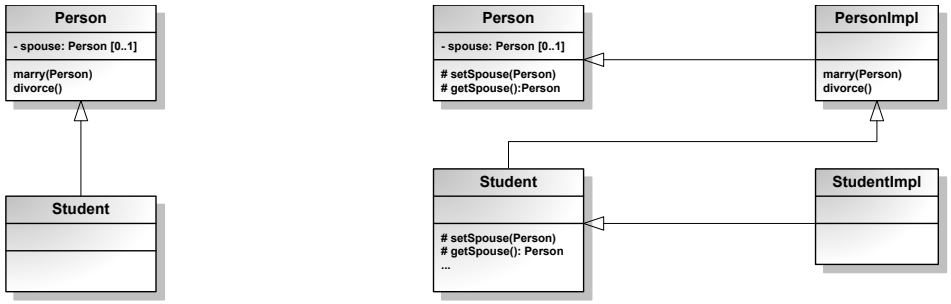
Apart from the structural issue, our intention is to include also the functional dimension of attributes, i. e. providing means for accessing attributes. In the course of implementing access to attributes, additional semantics of attributes, in particular multiplicity bounds, are taken into account.

An upper bound of an attribute must be checked before adding a new value, in particular if applying dynamically growing data structures without a specified maximum capacity for attribute implementation. A lower bound must be checked before removing a value. This is easily achieved by applying encapsulation: by hiding the attribute and providing methods for accessing it, the code for checking bounds that is inserted into accessor methods is always executed when updating attribute values. The visibility of accessor methods corresponds to the visibility of the implemented structural feature.

A problem of this approach is, that accessor methods may be bypassed if attribute values are updated by the owning classifier. If behavior is implemented by the developer directly by adding methods to the code of generated classes, such access to attributes very likely could occur. If behavior is modeled as well, code for behavior is generated and preferably should be correct. Nevertheless, a mechanism for ensuring the structural integrity of a system is aspired in our approach regardless of modeling or coding behavioral aspects.

The basic idea is to force the use of accessor methods, which contain bound checks. For some technical reasons, which are discussed in the following section, generated code is separated from user implementations. This separation entails a solution for forcing the use of accessor methods even to private attributes. An example is given in Fig. 3.14: methods `marry(Person)` and `divorce()` access the private attribute `spouse` of class `Person` given in Fig. 3.14(a).

Using accessor methods is forced by making the attribute private and the accessor methods protected, thus visible in specializations of `Person`. In user code, which is included in an implementation class such as `PersonImpl` in Fig. 3.14(b) for class `Person`, the private attribute is accessible only via the accessor methods. Since visibility of these methods cannot be reduced in specializations, access to a private attribute is granted to all specializations, unless accessor methods are redefined in specializations. For this purpose, accessor methods for attribute `spouse` of class `Person` are redefined in class `Student`. Redefinition may remove side effects and return default values such




(a) Generalization with private attribute in general classifier.

(b) Implementation of a private attribute in the context of generalization.

Figure 3.14: Implementation of private attributes.

as *null*, or raise exceptions in order to indicate the discouraged use of accessor methods outside the intended visibility of the concerned attribute. By marking such methods as deprecated, compilers and code editors could give warnings at compile time.

With regard to an implementation in Java, another problem arises by this approach: protected visibility also permits access to classes belonging to the same package. Hence, there is a choice of either forcing the use of accessor methods but widening visibility or to strictly consider visibility and therefore moving private attributes into implementation classes, resulting in the possibility of bypassing accessor methods. We provide details to this issue in Chapter 5, Sect. 5.4.1. Note that in other languages, e. g. C#, protected visibility strictly restricts access to the class owning the feature and its subclasses. Implementations for the classes *Person*, *PersonImpl*, and *Student* are given in Listing 3.1, Listing 3.2, and Listing 3.3.

```
1  class Person {
2    private Person spouse;
3    protected Person getSpouse(){ return spouse;  }
4    protected void setSpouse(Person spouse){  this.spouse = spouse; }
5  }
```

Listing 3.1: Implementation of a class with private attribute (see Fig. 3.14(b)).

```
1    class PersonImpl extends Person{
2    public void marry(Person p){  setSpouse(p); }
3    public void divorce(){  setSpouse(null);  }
4  }
```

Listing 3.2: Accessing private attributes in user code (see Fig. 3.14(b)).

```
1    class Student extends PersonImpl{
2    protected Person getSpouse(){ return null;  }
3    protected void setSpouse (Person spouse){ }
4  }
```

Listing 3.3: Implementation of hiding accessor methods to specializations (see Fig. 3.14(b))

Note that as a consequence of dynamic method binding in Java, this pattern for implementing private attributes does not allow for having a private attribute in a specialization with the same name as a private attribute of a general classifier.

Whereas introducing additional generalization as shown in Fig. 3.14(b) is often done for separating generated code from user code what can be done by partial classes [57] if supported by the target language as well, partial classes cannot be used to force the use of accessor methods.

## 3.5 Implementation of Associations

Generally, two main alternative code patterns for implementing associations are considered:

- *Attribute Pattern*: binary associations are implemented as a pair of mutual references
- *Relationship Object Pattern*: an association is implemented as a class in its own right.

The first strategy is quite straightforward and closely related to the idea of replaceability of class attributes and associations. We already discussed the problem of multiplicities for this approach. If they are adequately considered, an implementation by mutual references is practicable. However, this approach suffers some limitations which definitely require a more complex implementation based on the Relationship Object Pattern: a dedicated implementation class for associations.

We will first explain the implementation by mutual references and point out its limitations. Afterward, we introduce the second strategy as is state-of-the-art, compare it to mutual references, exhibit its limitations and present our enhancements to achieve full compliance to the UML specification.

### 3.5.1 Association Implementation — Attribute Pattern

This way of implementing associations is often referred to, e. g. by Akehurst et al. [1] and Génova et al. [30]. It is also the most common implementation of tools providing code generation from UML class diagrams. However, implementations of tool vendors vary considerably with regard to their compliance to the UML specification. Most of the tools only rudimentarily support code generation at all.

The following tools which are listed at the UML vendor directory[5] use the Attribute Pattern for association implementation: Altova umodel [105], ARTiSAN Studio [107], Micro Focus Together [113], ChangeVision Jude Professional [108], Gentleware Apollo [106], IBM Rational Software Architect [117], No Magic MagicDraw [112], Sparx System Enterprise Architect [110] and Visual Paradigm for UML [118], as well as EMF [23], UMPLE [3], Fujaba [111], Gentleware Poseidon [115], and Telelogic Rhapsody [116] which are not included there [33].

#### General Implementation of the Attribute Pattern

The implementation by using the Attribute Pattern bases on the decomposition of a binary association into its two mappings $f_1 : X \to \mathcal{P}(Y), f_2 : Y \to \mathcal{P}(X)$ (see 3.2.1). Attributes are used to implement the set of instances to which an instance is related. Class $X$ owns an attribute $f$ so that the values assigned to $x.f, x \in X$ implement $f_1(x) = \{y_1, \ldots, y_k\}$. Accordingly, the type of attribute $f$ in general must be a suitably typed collection.

In principle, higher order associations could be managed as well, but depending on characteristics of association ends such as multiplicity or navigability, necessary adaptations complicate the attribute pattern. The attribute pattern hence is not a well-fitting implementation for higher oder association. We agree to France [93] that due to the difficulties in implementing higher order associations, they are generally avoided.

#### Multiplicities at Association Ends

Upper bounds of multiplicities can easily be considered by implementing an attribute as a typed collection of appropriate size, e. g. an array of corresponding length. If the upper bound of an end is 1, the class connected to the concerning end may serve as the type of the attribute instead of a typed collection. Thus, the upper bound can be implemented exactly according to the input model [33].

However, tool support most often is on a lower level only identifying two states, namely an upper bound that is either 1 or infinite. For the latter case, attributes are implemented by collection types which are not limited in size, such as sets or lists [33].

Lower bounds cannot be considered on the level of attribute specification. Generally, they are fully omitted although they could be checked when changing the set of references by removing values as well as upper bounds can be checked when adding new values [1].

[5] http://uml-directory.omg.org/vendor/list.htm

### Ownership of Association Ends

Ownership of association ends is not considered by the Attribute Pattern. Since the attributes are located within the associated classes, ends are always treated as if they were owned attributes of end classes. This implies that non-navigable ends become navigable [33]. The specification does not claim for non-navigable ends to be not accessible. It rather states that accessibility of opposite ends is not guaranteed. Consequently, to this end, the attribute pattern is not contradictory to UML semantics.

Many tools do not support association end ownership on the modeling level. Therefore, developers will not mind about modifications of the ownership relation of association ends. Very likely, they will not even miss this modeling feature since it rarely is addressed in literature dealing with UML modeling [45, 80][33].

The sole consideration of association ends owned by member end classes gives rise to the attribute pattern. It facilitates the association implementation without a dedicated implementation class by completely managing the association by means of attributes of member end classes. Between classes of the model and the code, a bijective mapping exists.

### Navigability of Association Ends

The lack of ownership representation in code leads to associations that are navigable in both ways. It is common to omit the generation of an attribute in a member end class if the opposite end is non-navigable. It is assumed that an association which ends are all non-navigable cannot appear in a model. This assumption is problematic because non-navigable associations might be useful in combination with association classes and therefore should be allowed and translated to code [33].

By omitting the generation of an attribute, the ownership of a non-navigable association end is considered (see Sect 3.2.5) as it is not a feature of a classifier participating in the association [33]. However, the end becomes not only inaccessible for reading but also for writing, i. e. for creating new links by relating instances with each other. It is not clear whether this is intended by the specification or not.

### Visibility of Association Ends

Visibility of an association end is taken over to the generated attribute representing that end. As non-navigable ends are not implemented in code, visibility for those ends is not taken into account. Thus, all implemented association ends are supplied with the intended visibility [33].

### Symmetry of Associations

Many tools just provide the static structure for associations, i. e. the attribute generation in the member end classes. When using those tools, managing the association remains in the responsibility of developers [33].

However, Fujaba [111], Rhapsody [116], and EMF [23] generate code that allows to add or remove instances from associations. These tools even implement the consistency of associations: Whenever an instance $x$ is referenced by an instance $y$, the instance $y$ calls a generated method in $x$ passing itself as a parameter. The instance $x$ then obtains a reference to $y$ [14, 28, 64][33].

### Aggregation and Composition

Aggregation is a relationship in which one participant is considered to be a part of another one. The only implication of this is that aggregation may not be modeled so that an element is a part of its own, i. e. cyclic aggregations must not exist in a model.

Composition is stronger in two ways: an element may only be part of at most one other element and when an element is destroyed, all parts of it are destroyed, too, unless they are removed before.

Cyclic aggregations can be statically detected by model validation. Therefore, it is not necessary to prevent cycles by the implementation of associations. The constraint that an element may only be part of at most one other element as well as the dependency of lifetime of elements and their parts must be considered by the association implementation at runtime.

If applying the attribute pattern for association implementation, a cascading destruction of objects can easily be achieved if the target language supports destructors. However, destroying instances may violate lower bounds of other associations in which the classifier of the destroyed

instance participates. A sound implementation, possibly even with avoiding dangling references requires comprehensive enhancements of the attribute pattern.

For languages using garbage collection, destruction of objects is the result of removing all references to this object. An implementation for removing references based on the attribute pattern cannot be done efficiently if objects are referenced along associations which are not navigable in the opposite direction.

Insofar, it is not surprising that the situation regarding code generation for aggregate associations is staggering. The evaluated tools do not consider the difference between "plain" associations and aggregations or compositions. Akehurst et al. [1] evaluated a selection of more tools describing nearly the same situation in 2006. As far as we know, the situation has not substantially changed. In this regard, the modeling facilities of the current tools allow the violation of UML semantics like having a part included in multiple composites at a time [33].

#### Why Does the Attribute Pattern Fail?

Mutual updates of references require each association end to be visible to its opposite end [33]. Associations with both ends defined private cannot be managed because synchronizing references is not possible [30].

Another problem is that if for non-navigable ends no attribute is generated in a member class, information of multiplicities are not considered [33]. Instead, the attribute pattern implements a lower bound of 0 and an infinite upper bound unless additional code for counting references of the omitted end is inserted.

A minor defect of the attribute pattern is that associations of which both ends are non-navigable cannot be managed at all unless ends are made navigable.

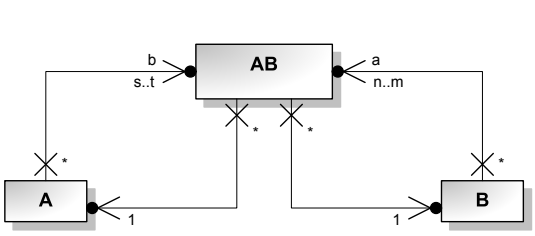
Therefore, the attribute pattern is only well suited for some cases of binary associations. For higher order associations, to our best knowledge, it has not yet been adapted. Although it probably is possible to generalize this pattern for higher order associations, this kind of associations generally is associated with the Relationship Object Pattern. The same holds for *AssociationClass* which can be implemented by the Attribute Pattern but is inherently closer related to the Relationship Object Pattern.

### 3.5.2 Association Implementation — Relationship Object Pattern

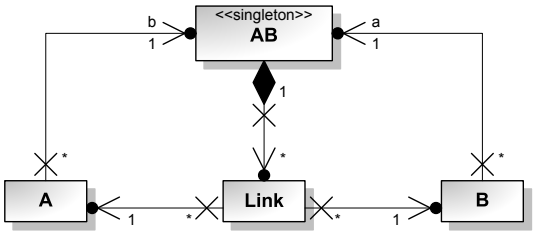
The Relationship Object Pattern makes the association a class in its own right. The implementation of the association is no longer dispersed among its participating classes hampering readability and maintainability of the code [65], rather it is merged into a single class.

Generating a class for an association is a straight-forward implementation of a link that is defined as a tuple of values by the UML specification. Accordingly, we term the class representing instances of the association *Link* and the references to participating class instances the values of a link [33]

In Fig. 3.15, two alternative implementations of the Relationship Object Pattern are given. Which alternative is used may have impact on the implementation of classes $A$ and $B$, but it should not be visible through their interfaces. These interfaces comprise accessor methods for creating links, querying links and destroying links.




(a) Association implementation by means of a dedicated class representing links.




(b) Association implementation by means of a dedicated link class and a manager class, based on [34].

Figure 3.15: Shapes of the Relationship Object Pattern.

We prefer the pattern including a dedicated class for managing links as shown in Fig. 3.15(b). On the one hand, a drawback of this variant is that the multiplicities of the association it implements cannot be expressed by multiplicities of associations which are part of this pattern. Furthermore, having a manager class introduces an additional indirection. On the other hand, handling links is more straightforward with respect to multiplicities and implementation of compositions.

The pattern as presented here may cause a memory leak: if objects are referenced by a link only, they cannot be removed by the garbage collector, because they are indirectly linked by the manager class. But if all classes are not referenced elsewhere, they should be removed from the heap. A solution to this problem is given at the end of this chapter in Sect. 3.6.5.

In the following, we refer to the Relationship Object Pattern with a manager class.

#### Implementing Consistency of Associations

Consistency of associations is assured by dedicated link instances [34]. Referring to the tabular view of associations [22], each link object represents a row in a table of links.

Requests to the association are delegated to the association implementation which manages link objects. Linking instances to each other is achieved by creating new link objects, unlinking instances is done by deleting link objects. Finding instances linked to a given context is achieved in two steps: first, all links fitting the given context must be identified and, afterwards, for each such link, the value of the inquired end must be added to the result which is returned.

#### Implementing Ownership

Diskin and Dingel [22] associate ownership with the location where to store links or references. But when applying the Relationship Object Pattern, links are represented by dedicated instances and ownership has no effect on the location of links.

An association end is a structural feature of its owner. We assume, that the intended implication of that is to specify the element offering access to association ends [34]. This is either the association itself or classes connected to association ends. Considering ownership therefore is achieved by creating accessor methods to an association end within the implementation of this end's owner.

If an end is owned by the association, the visibility of accessor methods determines which elements can access this end. Accessor methods for an end not owned by the association are part of the implementation of the opposed class which delegates the request to the association implementation. Such a delegation is only valid if initiated by the end's owning class. Thus, implementation of ownership requires to offer access to association ends only to those classes participating in the association while excluding all the other classes. However, without additional effort, granting access to a method to one class excluding other classes cannot be achieved in most object-oriented languages. How to solve this problem is presented in Sect. 3.6.

#### Implementing Navigability

Navigability is correctly implemented, if values of a tuple are only returned to the requester if the end to which the values refer is navigable. Additionally, if the end is not owned by the association, the requester must be an instance of a class participating in links at the opposite end. By that, we assume that navigability entails access granted to member end classes even for those ends privately owned by the association.

#### Implementing Visibility

For a privately owned association end, access methods to the corresponding set of links must be defined private [33]. If a private end is owned by a member end class, only this class can access the opposite end. If an end is owned by the association, private visibility restricts access to this end to the association itself. Such a restriction is reasonable in the case of association classes to restrict access to the link object. But as stated above, in general we assume it is intended that navigable ends are accessible from linked instances regardless of visibility.

Non-private ends owned by the association may be accessed by those classes to which the association is visible. If an end is owned by a member end class, it must be assured that this end is only accessible to instances of the class owning this end, as mentioned in the context of implementing ownership. According to the end's visibility, the owning class grants access to this end by providing accessor methods with appropriate visibility.

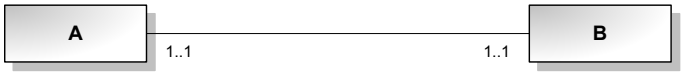
### Implementing Multiplicities

Checking upper and lower bounds can only be done by the implementation class created for each association by counting links before creating or destroying a link. Violations of bounds can be reported by raising exceptions or by return parameters of accessor methods.

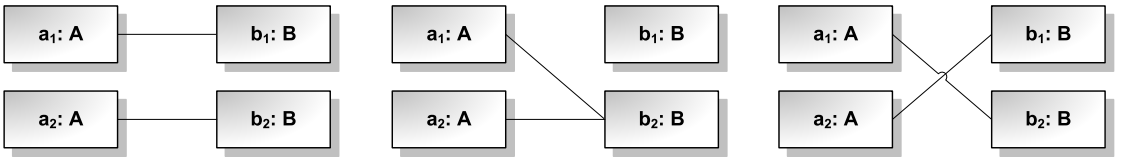
Technically speaking, a lower bound never should be underrun by the actual number of links, but to lower bounds greater 0 it is not always possible to comply with. Imagine a 1-to-1 association where objects at either ends must be linked to objects at the opposite end. After creation of the first object, its lower bound is violated until the second object needed for establishing a link is created.

Applying the factory pattern [29] can be used for the creation of instances. A factory makes complex object structures unavailable before all links are set and thus ensures that no inconsistent state is revealed to other instances.

If lower and upper bounds of a multiplicity are equal, changing links is not possible, as can be seen in Fig. 3.16. Classes $A$ and $B$ are associated with multiplicity 1..1 on both ends (Fig. 3.16(a)). The corresponding instances $a_1$, $a_2$, $b_1$, and $b_2$ are associated by two links, one between $a_1, b_1$, another between $a_2, b_2$ (Fig 3.16(b)). The desired final state is to relate $a_1$ with $b_2$ and $a_2$ with $b_1$ (Fig. 3.16(d)). From the initial state to the final state, an invalid state with violation of multiplicity bounds occurs (Fig. 3.16(c)), unless changing the links can be done by means of an atomic switch implemented in a single operation [30]. To come to the point, neither such an operation exists in UML nor any kind of transactions supporting temporarily but isolated invalid system states.




(a) Class diagram with a 1-to-1 association.




(b) Initial system state.

(c) Intermediate wrong system state.

(d) Desired valid final system state.

Figure 3.16: Consequence of the equality of lower and upper bounds of multiplicities, based on [30].

### Implementing Ordered Association Ends

When implementing associations by means of member end attributes, ordering is covered by implementing such attributes as ordered sets. If applying the Relationship Object Pattern, links are managed from a manager class. If only one end of an association is ordered, than aggregation of links must be ordered accordingly. But if two or more ends are ordered, links must be included in multiple sets, each of which is ordered according to one ordered end.

## 3.6 Enhancements to State-of-the-Art Code Generation

A problem occurring regardless of whether applying the Attribute Pattern or Relationship Object Pattern is how to handle associations if the visibility of all ends is `private`. Concerning the Attribute Pattern , the problem is that mutual updates of association ends are not possible. Génova et al. [30] state that the Relationship Object Pattern as shown in Fig. 3.15(b) does not solve this problem either, since the class AB cannot provide access to methods for classes A and B excluding all others. In general, this statement is right, however, we developed a mechanism to solve that problem. We term this mechanism *Adornment with a Handle* or briefly a *Handle* [34].

### 3.6.1 Handles

The idea of *Handles* is, that an instance having a valid handle can use it for the invocation of methods that cannot be invoked by instances that do not have a valid handle. We term methods that require a handle *semi-private* methods. The scope in which such methods are accessible is not limited to the owning class but extended to those classes, which have a valid handle. A handle is a runtime feature and could be applied to single instances. In our approach, it is used as a static class member and therefore is applicable on the class level rather than for individual instances.

We consider three kinds of handles:

- method handles
- global handles
- distinguishable handles [34].

A *method handle* is implemented as a parameter of a semi-private method. The implementation of the invoked method checks the handle and executes only if the handle is valid.

A *global handle* may be used, when all methods of a class are semi-private. In this case, it is more efficient to use the class itself as the handle. Not the handle is passed as a parameter when invoking a method but the methods are invoked on the handle.

Another issue is whether handles should be *distinguishable*. If they are, semi-private methods can have different effects for callers with different handles. Instances can be grouped by providing the same handle to them. If handles are distinguishable, as a matter of fact a global handle is not applicable.

#### Implementing Global Handles

The implementation of a global handle is related to the application of the visitor pattern. Listing 3.4 shows the implementation of the visitor pattern by the method `getHandle()` (lines 8–13). This method does not return a reference of the association object to the caller rather than it calls the method `takeHandle(AB)` (Listing 3.5, lines 5–7) of the caller's class passing the reference as a parameter. The invocation of `getHandle()` is located inside the initialization block of participating classes (Listing 3.5, line 3). Consequently, the method is only called once when a class participating in the association is loaded. The overhead of obtaining a handle thus is reduced to a single method invocation for each associated class when creating the first instance of it. Afterward, all instances share the same handle [34].

#### Implementing Method Handles

A method handle is implemented similarly to a global handle. Listing 3.6 implements a binary association of which only accessing one end requires a handle. An implementation of a class accessing this end is given in Listing 3.7.

The singleton instance representing the association is public and so are the methods for accessing association ends. For that end owned by the member end class, access methods must not be invoked by other classes than the owner of this end.

A dedicated handle object is requested by end classes (Listing 3.7, line 3). The handle is passed to the end's owner by the same mechanism as it is used for global handles (Listing 3.6, lines 8–10) and stored in the end class (Listing 3.7, lines 8–10). When attempting to access the end not owned by the association, this handle must be passed as a parameter (Listing 3.7, line 12). Before accessing the association end, the handle is validated (Listing 3.6, line 12). Since the handle is only provided to the end's owning class, access to this end is limited to the owning class as well.

#### Implementing Distinguishable Handles

A distinguishable handle is a method handle. The `getHandle()` method must take some kind of discriminator as a parameter in order to be able to return that kind of handle that is intended for the actual discriminator value.

When invoked, semi-private methods do not compare the handle to a single reference object like it is when checking a method handle. Here, the handle is used to decide what the effect of the invoked method is depending on the kind of caller, which is identified by the handle.

```
1  public class AB {
2
3    /* singleton pattern */
4    private static final AB fInstance = new AB();
5    private AB(){}
6
7    /* providing handles */
8    public static void getHandle(Class<?> end){
9      if (end == A.class)
10       A.takeHandle(fInstance);
11     if (end == B.class)
12       B.takeHandle(fInstance);
13   }
14
15   /* methods for managing the association */
16   private List<Link> tuples = new LinkedList<Link>();
17   public void createLink(A a, B b){
18     if (a != null && b != null)
19       tuples.add(new Link(a, b));
20   }
21   public void destroyLink(A a, B b){
22     for(Link l : tuples){
23       if(l.a == a && l.b == b){
24         tuples.remove(l);
25         break;
26       }
27     }
28   }
29   public List<A> getA(B context){
30     List<A> result = new LinkedList<A>();
31     for(Link l : tuples){
32       if (l.b == context)
33         result.add(l.a);
34     }
35     return result;
36   }
37   public List<B> getB(A context){
38     List<B> result = new LinkedList<B>();
39     for(Link l : tuples){
40       if (l.a == context)
41         result.add(l.b);
42     }
43     return result;
44   }
45
46   /* implementation of links */
47   private class Link{
48     private A a;
49     private B b;
50
51     public Link(A a, B b){
52       this.a = a;
53       this.b = b;
54     }
55   }
56 }
```

Listing 3.4: Implementation of a fully navigable binary association and its links, based on [34].

```
public class A {
  /* association implementation code */
  static { AB.getHandle(A.class); }
  private static AB handle;
  public static void takeHandle(AB ab){
    handle = ab;
  }

  /* methods for owned end */
  public void addB(B b){
    handle.createLink(this, b);
  }
  public void removeB(B b){
    handle.destroyLink(this, b);
  }
  public List<B> getB(){
    return handle.getB(this);
  }
}
```

Listing 3.5: Implementation of a class participating an association, based on [34].

```
public class AB{
  /* singleton pattern */
  public static final AB fInstance = new AB();
  private AB(){};
  /* handle object */
  private Object handle = new Object();

  public void getHandle(){
    A.takeHandle(handle);
  }
  public List<B> getB(A a, Object handle){
        if(this.handle != handle) return null;
    // original implementation here
  }
  ...
}
```

Listing 3.6: Implementation of a binary association with one private end, based on [34].

```
public class A {
  /* singleton pattern */
  static { AB.fInstance.getHandle(); }
  private static AB fAB = AB.fInstance;
  /* handle object */
  private static Object handle;

  public static void takeHandle(Object h){
    handle = h;
  }
  public List<B> getB(){
    fAB.getB(this, handle);
  }
  ...
}
```

Listing 3.7: Implementation of access to a private association end with a method handle, based on [34].

#### Hiding Details from the User

Note that classes presented in Listings 3.4, 3.5, 3.6, and 3.7 can be generated automatically. In order to keep code generators easy, re-generating code typically replaces the complete contents of created source files. Therefore, developer code must be located in subclasses of A and B to prevent loss of modifications after model changes and subsequent repeated code generation.

Since the handle is defined as a `private static` field, it is not accessible in subclasses. The implementation of handles, how to get a handle and how to use it is completely hidden from the developer, thus excluding any kind of unintended as well as willful misuse [34].

#### Using Handles for Accessing Associations

If both ends of a binary association are owned by the associated classes, methods for managing the association must not be accessible to other than those classes. As the association itself has no features accessible to other classes, using a global handle like implemented in Listings 3.4 and 3.5 is appropriate.

If one end is owned by the association and the other one owned by an end class, methods for the end owned by the association are accessible in the same way to all classes according to the end's visibility whereas methods for accessing the other end must be restricted to the owner of that end. In this case, method handles are needed for methods accessing the end not owned by the association. An implementation is given by Listing 3.6.

For higher order associations, all ends must be owned by the association and a handle is not required because classes participating the association are not privileged. But if desired, privileged access to association ends can be granted to classes participating in the association by an implementation based on handles.

Note that before adding new links, upper bounds must be checked, before removing links, lower bounds must be checked. Both checks can be implemented by the implementation of an association, but are omitted in Listing 3.4 for brevity.

Handles can be used for implementing associations by means of the Attribute Pattern. Because method handles allow for the restriction of access on attributes representing association ends to classes participating in the association, applying the handle pattern facilitates the attribute pattern to successfully implement binary associations.

However, since the Relationship Object Pattern additionally offers some advantages in terms of association class implementation or implementation of composition semantics, we do not concentrate on the application of handles in combination with the Attribute Pattern here [34].

### 3.6.2 Implementing Association Classes

For the implementation of an association class, the Relationship Object Pattern can be used to implement the association characteristics, the class characteristics are implemented by implementing attributes and operations as members of links of the association.

Since an association class is both, a class as well as an association, it can participate in other associations as a member end class. It is necessary to distinguish the different kinds of accessing an association class:

- accessing it from the role of a member end class
- accessing it as an associated class.

Figure 3.17 gives an example. Class $A$ and $B$ are member end classes which cannot access features of $AB$, but can navigate across $AB$ to the opposite ends. Class $C$ can access the public features of $AB$, i. e. the operation *toString()* and can access the instances at the $b$ end of the association class, which is public, but not instances of the $a$ end, which is private. But the access to an association end is different from a member end class like $A$ and another class associated to the association class like $C$[6]: an instance of $C$ is associated with an instance of $AB$ and can access the instance of $B$ which is the value of the link represented by the $AB$-instance. In contrast, accessing the $b$ end from an instance of $A$ results in a set of instances of $B$ representing all instances linked to the instance which is actually accessing the association.

[6]This is the result of the semantics of actions, not of AssociationClass itself (see *ReadLinkAction* and *ReadLinkObjectEndAction* in Chapter 4, Sect. 4.3.1).

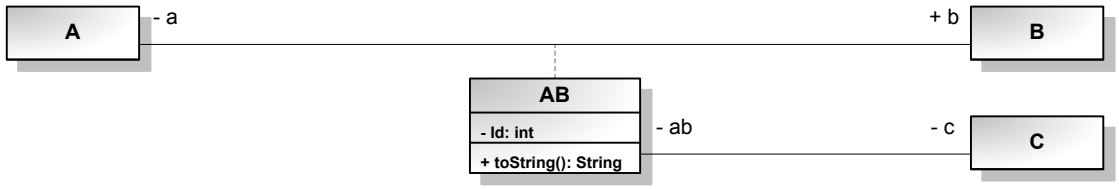



Figure 3.17: Access to an *AssociationClass* by associated classes and member end classes.

Therefore, accessing association ends must be implemented in multiple ways and it must be assured that the returned values are appropriate for the kind of request, more technically speaking to the kind of requester. The method for accessing the $b$ end of $AB$ could be implemented by means of a distinguishable handle to comply to the different kinds of accessing the end.

### 3.6.3 Implementing Aggregation and Composition

The semantic impact of aggregation is little. It just indicates, that an instance is part of another instance, representing the whole whereas a composition entails that the whole is responsible for the lifetime of its parts.

Java uses a garbage collector that removes unreachable objects. As long as a part is contained in the composite, both the part and the whole are referenced by a link and either both or none of them are removed. This effect runs contrary to lifetime control in the way that the parts may prevent the composite from being destroyed. If an element representing the whole of a composite is no longer referenced by an instance not participating in the composition, it should be removed by the garbage collector. Akehurst et al. [1] consider the use of Java weak references but discard it because this mechanism does not assuredly prevent access to deleted parts. An alternative implementation is given but a drawback of this is that objects are marked inaccessible while links persist.

For the proposed code pattern, a simple idea is applicable: Links are always accessible, either directly by the referenced instances or indirectly by the association. If the whole of a composition is to be deleted, the values of links referencing its parts must be set to `null` if the links are not an instance of the composition itself. If neither the whole nor its parts are referenced by an attribute of another class, both stay linked but become unreachable from anywhere else and may be removed by the garbage collector. An unsolved problem is that destroying links may violate multiplicities of associated classes [33].

### 3.6.4 Implementing Ordered Association Ends

For ordered association ends, the implementation of association (Listing 3.4) must be adapted as in Listing 3.8. The signature of `createLink` (line 9) contains additional parameters for specification of the insertion position of newly created links. For each ordered end, links are added into an ordered list at the specified position.

When reading an association end, the appropriate list of links must be used for iterating over all existing links. As an example, the method for reading end $a$ is given. In line 21, the list containing the links ordered according to and $a$ is used. Thus, the result is created in correct order.

### 3.6.5 Avoiding Memory Leaking

The Relationship Object Pattern as introduced above causes a memory leak, if instances are linked, but none of the instances is referenced from elsewhere. In this case, the garbage collector cannot remove the instances because they are indirectly referenced by the manager class through the link. Replacing the reference from the manager class to its links by weak references makes links and their values removable by the garbage collector, if the whole complex of a link and its values is not externally referenced. But since a link is not referenced by its values, the link object will be accessible for the garbage collector immediately after its creation. To prevent a link form being removed by the garbage collector, non-weak references to it must exist.

Figure 3.18 shows an extension to the Relationship Object Pattern. The manager references its links by weak references and by that does not prevent a link from being removed. In order to

```
1  public class AB {
2    ...
3    /* list implementing ordered association ends */
4    private List<Link> tuples_A = new LinkedList<Link>();
5    private List<Link> tuples_B = new LinkedList<Link>();
6    private List<Link> tuples_N = new LinkedList<Link>();
7
8    /*  methods for managing the association */
9    public void createLink(A a, int posA, B b, int posB, ..., N n, int posN){
10     if (a != null && b != null && ... && n != null){
11       Link link = new Link(a, b, ..., n);
12       tuples_A.insertAt(link, posA);
13       tuples_B.insertAt(link, posB);
14       ...
15       tuples_N.insertAt(link, posN);
16     }
17   }
18
19   public List<A> getA(B b, ..., N n){
20     List<A> result = new LinkedList<A>();
21     for(Link link : tuples_A){
22       if (link.b == b && ... && link.n == n)
23         result.add(link.a);
24     }
25     return result;
26   }
27   ...
28 }
```

Listing 3.8: Implementation of ordered association ends.

not loose all links immediately after creation, link values reference the link. If no instance which is a value of link $l$ is referenced from anywhere except from $l$, than the values of $l$ as well as $l$ itself are inaccessible and are removed.

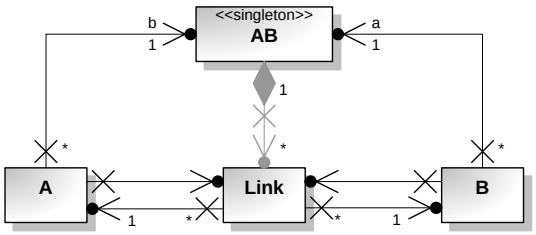



Figure 3.18: Memory leaking resistant implementation of the Relationship Object Pattern.

The implementation of member end classes (Listing 3.9) must include a set of links (line 3) as well as methods for adding (line 4) and removing (line 6) links. The list of links is not typed for a concrete association so it can be used for links of any association in which the class participates. Because methods for adding and removing links are public, it is possible to abuse the list for storing other objects than links. However, this has no effects on the management of association links.

In the association implementation class (Listing 3.10), weak references must be used to store links (line 4). Furthermore, creation and destruction of links must be extended: new links must be referenced by link values (line 9 and line 10) and references to destroyed links must be removed (line 16 and line 17). Additionally, from time to time, the list of weak references to links must be cleaned up by removing weak references which do not reference an object any more.

```
public class A {
  ...
  private List<Object> links = new LinkedList<Object>();
  public void associate(Object link){ links.add(link); }
  public void deAssociate(Object link){ links.remove(link); }
   }
```

Listing 3.9: Implementation of memory leak prevention in member end classes.

```
public class AB{
  ...
  private List<WeakReference<Link>> tuples =
    new LinkedList<WeakReference<Link>>();

  public void createLink(A a, B b){
    Link l = new Link(a, b);
    tuples.add(new WeakReference(l));
    a.associate(l);
    b.associate(l);
  }
  public void destroyLink(A a, B b){
    for(WeakReference<Link> l : tuples){
      if(l.get().a == a && l.get().b == b){
        tuples.remove(l);
        a.deAssociate(l.get());
        b.deAssociate(l.get());
        break;
      }
    }
  }
}
```

Listing 3.10: Implementation of memory leak prevention in associations.

## 3.7 Summary

In this chapter, we introduced some key concepts of modeling static structures in UML. With respect to attributes and associations, in particular association ends, models of static structures support the specification of constraints which must be held at runtime, e. g. the size of sets of values for attributes or association ends.

As a result of missing code generation for dynamic behavior, some tools include code for *using* the static structure at runtime by providing access to class attributes and associations. Technically speaking, this is an implementation of *Action*s which are not included in a behavioral modeling formalism but which can be accessed directly by the user when completing the implementation of generated classes by implementing behavior.

Additionally, we designed the implementation of association in a way that considers details like end ownership, navigability, visibility, or composition semantics, and keeps these characteristics of associations orthogonal to each other. Furthermore, we provide privileged access on private association ends to member end classes, i. e. regardless of ownership, member end classes can always access associations. Implementation of association classes can be based on this design of association implementation, too.

Since the ideas for implementing static structure as described in this chapter implicitly implement the semantics of some actions, the semantics of models of dynamic behavior and the semantics of actions should be cross checked and code generation for static structure should be aligned to be supportive for code generated from behavioral models as well.

For this purpose, the following chapter is concerned with the semantics of *Activities* and *Actions*.

# Chapter 4

# Semantics of Behaviors

In UML, behavior is based on *Action*. Actions are predefined foundational units of behavior. Activities contain actions and manage their execution. In this chapter, we introduce activities and their constituent parts as well as actions in general and the semantics of a subset of them concentrating on implications on the underlying static structure.

Furthermore, we present a mapping from *Activity* to Java and a suitable model transformation reflecting the implementation pattern back to the model. Finally, we propose some improvements to the UML specification.

## 4.1 Actions and Activities at a Glance

In this section, we give a short survey over activities and actions, introduce notations and outline the basic ideas of modeling concepts related to activity modeling.

Activities own (and in terms of graphical representation, they contain) actions, which might be executed during the execution of the owning activity. Activities are drawn as rectangles with round corners with its name attached to the upper left corner. We omit names if they do not contribute to a better understanding of a model or of given examples. Figure 4.1 shows an example of an activity.

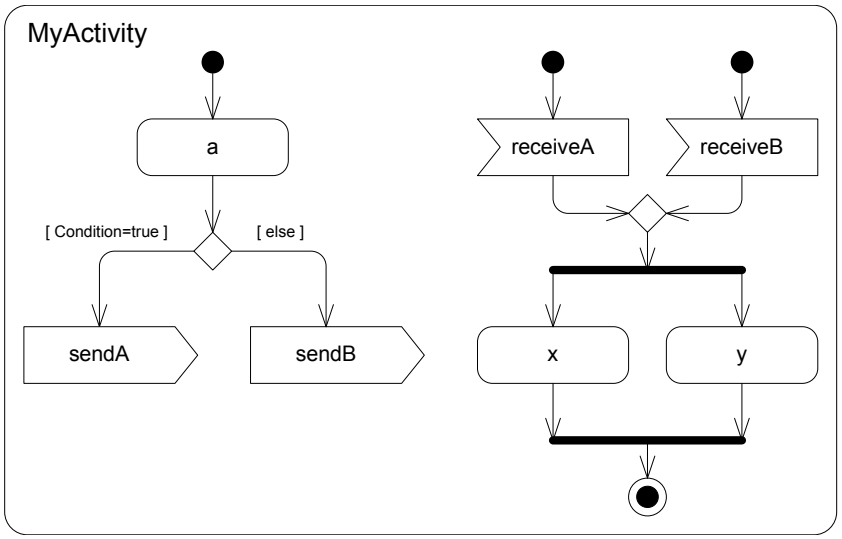



Figure 4.1: Notation of activity and its owned elements.

Actions are contained inside the activity frame as rectangles with round corners and a name centered inside the action shape ($a, x$, and $y$ in Fig. 4.1). Some kinds of action, such as *SendSignalAction* ($sendA, sendB$) or *AcceptEventAction* ($receiveA, receiveB$) have different shapes. By that, paths of communication given by a signal sender and a receiver are easily recognizable.

The order of action execution is given by the specification of a control flow. A control flow is a directed edge from a source action to a target action indicating that the target action is executed after the source action has been terminated.

If output of source actions shall be provided as input for target actions, object flows[1] can be used instead of or additionally to control flows. Object flows do not connect actions, but object nodes, such as input and output pins which are connected to actions. The notation of object flow and pins is schown in Fig. 4.2 by the flow from $a$ to $b$.

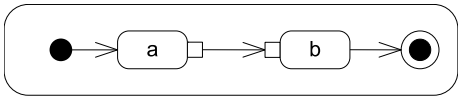



Figure 4.2: Notation of object flow [39].


Control nodes allow for the specification of starting and end points of flows as well as alternative or parallel branches:

- Control flows start at *InitialNode* ●.
- Control and object flows end at *FlowFinalNode* ⊗.
- Flows end and cause activity termination at *ActivityFinalNode* ◉.
- Alternative branches (**or** semantics) start at *DecisionNode* and end at *MergeNode*. ◇
- Parallel, concurrent branches (**and** semantics) start at *ForkNode* and end at *JoinNode* ━.[2]

Control and object flows may be guarded. A guard is a constraint attached to an edge and specifies a boolean expression. A guard must hold in order to cause the execution of a target action as a result of the termination of the source action. In Fig. 4.1, depending on the value of *Condition*, one of the two possible alternative flows will cause either *SendA* or *SendB* to be executed.

After either signal $a$ has been received by *receiveA* or signal $b$ has been recieved by *receiveB*, actions $x$ and $y$ are executed concurrently, since both actions are contained in parallel brachnes. The activity execution terminates when both actions have been terminated, i. e. when both parallel flows have been joined and reached the activity final node.

*InterruptibleActivityRegion* supports aborting executions of actions which are grouped inside the region. Figure 4.3 shows the notation of *InterruptibleActivityRegion* and an interrupting edge. Actions $a, b, c$, and $d$ are executed sequentially. If during execution of action $b$ or $c$ an *interrupt* signal is received, $b$, or $c$ respectively, is aborted and no downstream action of the aborted action is executed. Instead, action $x$ is executed.

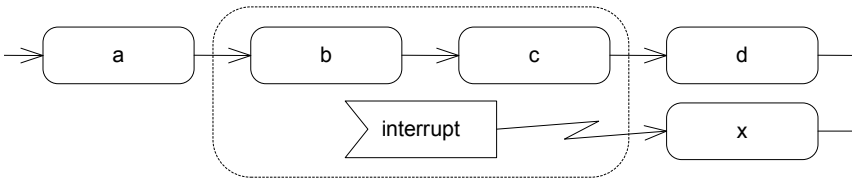



Figure 4.3: Notation of *InterruptibleActivityRegion*.

## 4.2 Activities

UML defines the semantics of a number of specialized actions which serve as fundamental units of behavior specification. These fundamental units are composed to complex behavior by means of *Activity*, which contains actions and specifies the sequence of action executions. Thus, an activity represents an algorithm whereas actions represent basic steps.

[1] We use the terms *data flow* and *object flow* interchangeably.
[2] This shape is drawn orthogonal to the flow direction, therefore it may be drawn vertically.

Our approach focuses on three concepts which make activities a very expressive formalism:

- *ObjectFlow*,
- *InterruptibleActivityRegion*,
- and concurrency.

Dedicated object flows are convenient as they clearly show locations of data creation and consumption and make activities well suited for modeling behaviors where sharing data between actions is extensively used [39].

Interruptible regions support more flexible non-local termination of flow. They are a mighty concept to handle interrupts, unexpected events, or invalid system states which need to end the default behavior and execute actions for recovering.

Concurrency supports parallelism in activities. It is either explicitly or implicitly modeled. Fig. 4.4 shows the use of *ForkNode* and *JoinNode* for explicitly modeling concurrency as well as an implicit fork (outgoing flows of $a$) and an implicit join (incoming flows of $d$) [39].

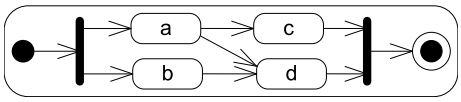



Figure 4.4: Implicitly and explicitly forks and joins [39].


The above mentioned aspects of activities are embedded in an implementation design that completely covers the basic semantics of activities, which is given by the execution semantics based on a token flow concept.

### 4.2.1 Basic Token Flow Concept

The semantics of the abstract meta-class *Action* from FundamentalActivities defines four steps of action execution [69, §12.3.2]:

- Creation of an execution, which requires all object flow and control flow prerequisites to be satisfied, i. e. tokens must be available at all incoming edges and input pins.
- Consumption of the offered tokens, which are removed from their original sources.
- Execution of the action until it terminates.
- Creation of token offers at all outgoing edges and output pins.

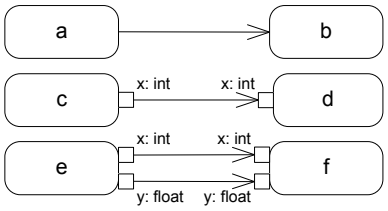
Tokens representing control flow are considered to be indistinguishable from each other whereas tokens of object flows represent concrete, distinct values.

A sequence of actions $a$ and $b$ with a single flow from $a$ to $b$ as shown in Fig. 4.5(a) corresponds to a sequence of statements such as the first line of the listing of Fig. 4.5(b) [39].

Object flows may be implemented by explicitly using a variable or by nesting calls. For the flow from pin $x$ at action $c$ to pin $x$ at action $d$ in Fig. 4.5, the former option is shown in Fig. 4.5(b), the latter one in Fig. 4.5(c) [39].

If multiple object flows exist between two actions, this still can be implemented as a sequence of statements. Depending on the implementation language, processing of the parameters requires appropriate techniques. Considering Java which only supports a single return value and no out parameters, a class is required to hold the return values as shown in Fig. 4.5(b). The class `XY` has two attributes representing the output pins $x$ and $y$ of action $e$ which must create an instance of that class and write its output values to the attributes. Fig. 4.5(c) shows an alternative implementation suitable for languages providing *out* parameters [39].

A basic flow, i. e. a flow from one to another action without any control nodes in between, sequences actions so that two sequential statements calling the according methods are a suitable implementation. Statements representing an action sequence are implemented in single threads. By doing so, parallel token flows of an activity are implemented by concurrently executing threads each representing one flow. For a proper implementation of the token flow semantics, it is necessary to properly implement guards and control nodes, both explained in the following, but for a start limited to control flows. Later, we generalize our considerations to suite to object flows as well.




(a) Control and data flows.

```
a(); b();

int x = c(); d(x);

XY xy = e();
f(xy.x, xy.y);
```

(b) Implementation of control and data flows.

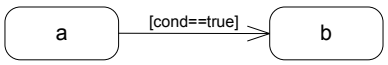
```
d(c());

int x; float y;
e(out x, out y); f(x, y);
```

(c) Alternative implementation of data flows.

Figure 4.5: Sequences of actions and implementations [39].


### 4.2.2 Guarded Flows

Guards are boolean conditions associated with an edge. If such a condition does not hold, it prevents tokens from traversing the concerned edge. Applied to Fig. 4.6(a), if the guard at the edge from *a* to *b* does not hold, the call of `b()` must be deferred as long as the guard does not hold.




(a) Guarded control flow.

```
a();
while (cond==false){ ; }
b();
```

(b) Implementation of a guarded flow.

Figure 4.6: Guarded flow and its implementation.

A proper implementation of guards requires to pause the execution of code as long as a guard prevents a token from flowing. This is important in particular if concurrency occurs in an activity: The evaluation of a guard may depend on other concurrently executed sequences of actions. An implementation based on `if else` statements as proposed e. g. by Bhattacharjee and Shyamasundar [7] does not cause the control flow of an application to pause. Therefore, it is insufficient in any situation, where no alternative flow exists or where guards of all alternative flows do not hold.

An implementation of guards based on `if else` statements is only valid at control nodes splitting a flow into alternative branches of which always at least one of all guards holds.

Regarding Fig. 4.6(b), the `while` loop pauses the activity execution. A more sophisticated approach is to let a thread wait until the guard holds. As a waiting thread cannot evaluate the guard, all waiting threads must be notified if a guard might evaluate to true. Therefore, the implementation of actions accessing attributes, variables, or associations must contain code to notify waiting threads which immediately re-evaluate guards. Depending on the result of this evaluation, threads must either continue execution or wait for another notification [39].

Besides deferring the execution of an action in a sequence of actions, guards have a far more complex impact on the execution of actions if used in combination with control nodes. This is discussed in detail in Sect. 4.2.4.

### 4.2.3 Code Generation Based on Token Flow Semantics

Generating code for sequences of actions is trivial as can be seen from the examples given in Fig. 4.5 since one action is executed after another. When considering control nodes, the situation gets more complicated, depending on how complex flows are composed. We consider the following cases:

1. at most one control node is used in a flow from one action to another;
2. any number of control nodes may occur, but the flow from the source to the target is acyclic;
3. any number of control nodes and cycles may occur.

Here, we present code generation as designed for activities where flows contain at most one control node (1). However, it can be adapted to handle more complex flows (2, 3), but remarkable effort is required. We provide additional information to this issue in Sect. 4.2.5, Sect. 4.2.6, and Sect. 4.2.7.

Our prototype is implemented in Java, but the general idea can be applied to any other object oriented language supporting threads:

- sequences of actions are mapped to sequences of method calls, each method being the implementation of a corresponding action
- an activity is implemented as a switch statement with each case block containing a sequence of method calls
- the switch statement is executed repeatedly, an *id* is used to select the case block to execute
- control nodes are implemented by setting the id at the end of each sequence, so that — according to the semantics of the control node — the correct downstream sequence is selected to be executed in the next iteration
- concurrency is implemented by creating new threads with the appropriate id.

Fig. 4.7 gives an example of an activity implemented by Listing 4.1. Gray shaded boxes group actions which build a sequence. The number refers to the case block used for implementation. By setting the `id` to `-1`, no valid sequence to execute is selected and the thread terminates. When all threads have been terminated, the activity execution ends.

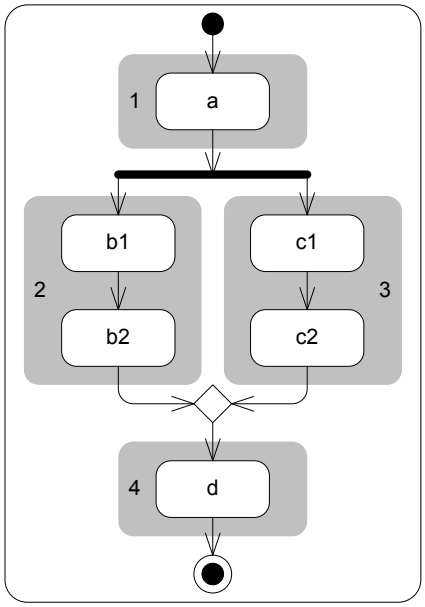



Figure 4.7: UML activity with sequences of actions, according to [39].

```
1  class ActThread extends Thread{
2   private int id;
3   public ActThread(int id){
4     this.id = id;
5   }
6   public void run(){
7     while(id >= 0){
8       switch(id){
9         case 1: a();
10                break;
11        case 2: b1();
12                b2();
13                break;
14        case 3: c1();
15                c2();
16                break;
17        case 4: d();
18      }
19    }
20  }
21 }
```

Listing 4.1: Implementation of sequences of actions in a single class.

Note that at the end of each case block in Listing 4.1, the value of `id` must be updated before the `break` statement in order to cause the thread to execute another sequence of actions during the next while-loop iteration. Which value to assign to `id` depends on the control node between ending and subsequent sequences. In Listing 4.1, code for setting appropriate values to `id` or create new threads is omitted because we introduce basic considerations to that issue in the following section before completing the code example.

### 4.2.4 Token Flow at Control Nodes

UML defines seven kinds of control nodes, which are specializations of the common ancestor *ControlNode* as shown in Fig. 4.8: *InitialNode*, *FlowFinalNode*, *ActivityFinalNode*, *DecisionNode*, *MergeNode*, *ForkNode*, and *JoinNode*.

Before going into detail, we roughly give the implication of control nodes. *InitialNode* defines starting points of flows when an activity execution is initiated. *FlowFinalNode* specifies flow ends, *ActivityFinalNode* specifies flow ends causing the activity execution to end as well. A *DecisionNode* is used to choose between multiple alternatives from which only one may be taken. Which alternative actually is taken depends on guards of outgoing edges. For complex decisions, a

behavior can be associated with the node and additional information for taking a decision may be provided via the decision input flow. A *ForkNode* splits an incoming flow into multiple concurrent outgoing flows. Multiple flows are combined to one single flow by a *MergeNode*. A token arriving at one incoming edge results in a token flow on the outgoing edge. When combining multiple flows with a *JoinNode*, a token must be available at each incoming edge to emit a single token at the outgoing edge [39].

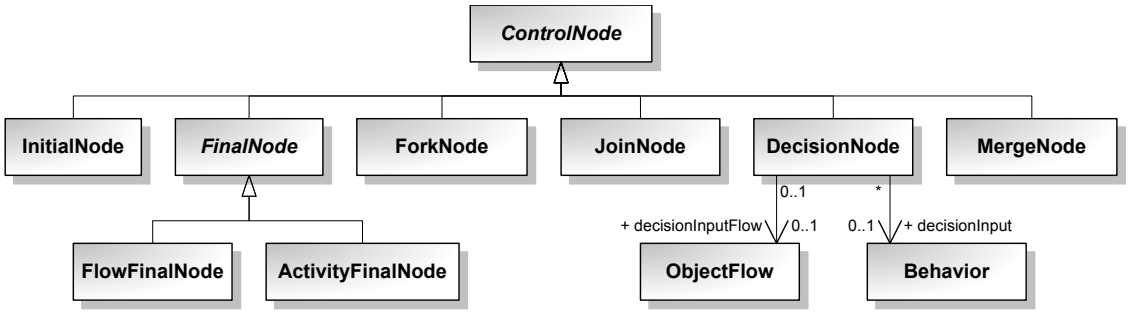



Figure 4.8: UML control nodes.

Tokens cannot rest at control nodes. Control tokens always flow from one action to another action, data tokens flow from one object node to another one. An exception to this rule is *ForkNode*, where tokens may be buffered. We discuss this in detail when looking closer at fork nodes later.

Note that two or more incoming edges of an action represent an implicit join, i. e. tokens must be offered to all incoming edges. This implicit join may be made explicit either by modeling so or by a model transformation. Therefore, we consider activities to be free of implicit joins. Analogously, two or more outgoing edges represent an implicit fork. For the same reason as for implicit joins, we consider all forks to be explicitly modeled [39].

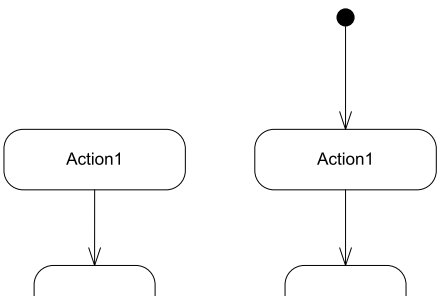
### InitialNode

When starting to execute an activity, flows are initialized by each *InitialNode* emitting a single control token. Actions with no incoming edges are executed immediately. If for each such action, an *InitialNode* and a flow from there to the action is inserted, these actions still are executed immediately after starting the activity execution (see Fig. 4.9(a)).

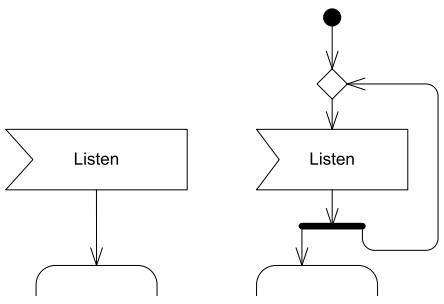
Concerning *AcceptEventAction* without incoming flows, two exceptions have to be considered:

- after accepting an event, the action is immediately executed again waiting to accept another event
- if placed inside an *InterruptibleActivityRegion*, it is not executed before a flow enters the region directly containing it.

The first exception can be made explicit by modeling as shown in Fig. 4.9(b). However, this modification to the model breaks the sequence of the *AcceptEventAction* to the following action. In




(a) *Action* without incoming flows and equivalent model with incoming flows.




(b) *AcceptEventAction* without incoming flows and equivalent model with incoming flows.

Figure 4.9: Control flow semantics for *Action* without incoming flows.

consequence, the code generated from such a modified model becomes more complex. With regard to code complexity and performance, it probably is preferable to address this special semantic issue by generating suitable, optimized code for restarting an *AcceptEventAction* without incoming flows than modifying the input model.

Even more complex is a model transformation in order to make the activation of *AcceptEventAction* contained in an *InterruptibleActivityRegion* explicit. We refer to this in the context of *InterruptibleActivityRegion* later in Sect. 4.2.10.

### FlowFinalNode and ActivityFinalNode

The effect of *FlowFinalNode* is the destruction of all arriving tokens without any further effect to the execution of an activity, in particular to other flows. It can easily be implemented by setting the `id` to -1 causing the thread of Listing 4.1 to terminate.

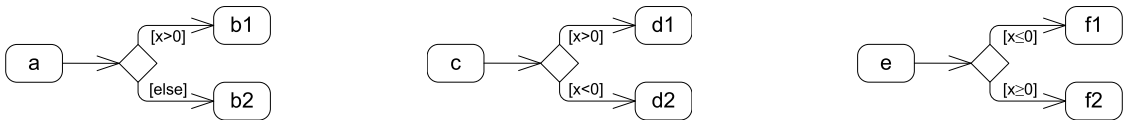
An *ActivityFinalNode* terminates an activity execution, i. e. all threads processing a sequence must terminate. Therefore, an activity execution must be aware of all threads processing action sequences of it and must interrupt all these threads. In order to actually stop the processing of action sequences, between two actions, the thread status must be checked. Within actions which are responsible for any kind of consistency, e. g. *CreateLinkAction*, if the processing thread is interrupted, already done work must be finished or undone in order to maintain consistency of the system. Aborting an activity execution is related to the issue of aborting an interruptible activity region. We provide a code example when discussing this in Sect. 4.2.10.

### MergeNode

Implementing a merge node is achieved by changing the value of `id` [39]. In Listing 4.1, before line 13 and line 16, the value 4 must be assigned to `id`. Thus, action *d* will be executed next. If incoming edges are guarded, the terminating sequence must wait until the guard holds, if outgoing edges are guarded, the new sequence must wait for the guard to hold before executing its first action. If incoming and outgoing edges are guarded, both guards must be checked at the same time, either by the ending sequence implementation or by the following one. Otherwise, the execution might continue even though both guards had never been true at the same time.

### DecisionNode

A token arriving at a decision node may traverse any of the outgoing edges, but only one of them. If guards are annotated to outgoing edges — what typically is the case as these guards define the decision to take — the token may traverse any edge the guard of which evaluates to true. If one of the outgoing edges is labeled with an `else` guard as is in Fig. 4.10(a), at least one edge may always be traversed [39]. A suitable implementation is given in Listing 4.2(a), in which $G_{incoming}$ denotes the guard at the incoming edge of the decision node. If the incoming edge is not guarded, $G_{incoming}$ may be replaced by the value `true` or the `if` statement (line 2) can be omitted.




(a) Deterministic decision.

(b) Partially undefined decision.

(c) Partially non-deterministic decision.

Figure 4.10: Determinism and decidability at *DecisionNode* [39].


Fig. 4.10(b) shows a situation in which none of the outgoing edges may be traversed if `x=0`. The decision shown in Fig. 4.10(c) is non-deterministic if `x=0`. Consideration of this while code generation demands for an analysis of guards which is difficult since guards may be any boolean expression evaluated within the context of the activity. We prefer to test guards of all edges for holding until a guard is found that is satisfied. If none of the guards holds, the thread waits until receiving a notification which is sent when the values of attributes or variables of the context object have changed [39]. Since the flow from a source to a target is considered to be an atomic event, guards of all edges of a path from the source to the target must hold at the time of token transition.

Therefore, the guard of the incoming edge has to be tested together with the evaluation of guards of outgoing edges.

Listing 4.2(b) implements the decision node of Fig. 4.10(b), Listing 4.2(c) implements the decision node of Fig. 4.10(c).

An else guard, as in Fig. 4.10(a), could also be considered by replacing *else* with *true* and ensuring that the affected edge is processed last. Then the last `if`-statement reads `else if (true)`, which causes this edge to be traversed if guards of all other edges fail.

```
1  while (true){
2    if (<G_incoming>){
3      if (x>0)
4        id = ...;
5        break;
6      else
7        id = ...;
8        break;
9    }
10   wait();
11 }
12
13
14
```

(a) Deterministic decision

```
1  while (true){
2    if (<G_incoming>){
3      if (x>0){
4        id = ...;
5        break;
6      }
7      else if (x<0)
8      {
9        id = ...;
10       break;
11     }
12   }
13   wait();
14 }
```

(b) Partially undefined decision

```
1  while (true){
2    if (<G_incoming>){
3      if (x<=0){
4        id = ...;
5        break;
6      }
7      else if (x>=0)
8      {
9        id = ...;
10       break;
11     }
12   }
13   wait();
14 }
```

(c) Non-deterministic decision

Listing 4.2: Implementations of decision nodes.

### ForkNode

The semantics of *ForkNode* is very complex. A token offered to its incoming edge is offered to the targets of its outgoing edges, if the corresponding guards hold. If at least one token is accepted by a target, a copy of this token is buffered at each outgoing edge which guard holds, but the target of which cannot accept the token. This can only happen if a target has more than one incoming edge. Such an implicit join can be made explicit, consequently, if we assume that all implicit joins of an activity have been eliminated, token buffering cannot occur in the context of a single fork node as shown in Fig. 4.11.

We address the issue of token buffering when discussing more complex control flows in Sect. 4.2.6. The basic implementation of *ForkNode* is similar to that of *DecisionNode* as can be seen in Listing 4.3. Since a flow can only happen when the guard of the incoming edge and the guard of at least one outgoing are true, the thread waits until this condition holds (lines 1–3).

Figure 4.11 gives an example: if, after the execution of action *a*, the value of *i* is an even number, either *x* or *y*, or *x* and *y*, or none of both actions may be executed, depending on the evaluation of guards at outgoing edges of the fork node. Note that if none of both actions is executed, the token rests at the incoming edge of the fork node and may cause the execution of one or both actions later.

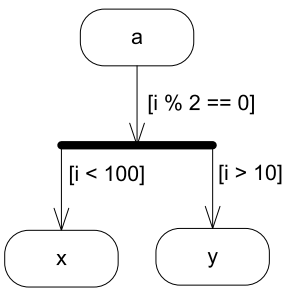



Figure 4.11: Application of *ForkNode*.

```
1  while (!(i%2==0 && (i<100 || i>10))){
2    wait();
3  }
4  if (i<100){
5    new ActThread(<id_x>).start();
6  }
7  if (i>10){
8    new ActThread(<id_y>).start();
9  }
10 ...
```

Listing 4.3: Implementation of *ForkNode*.

### JoinNode

*JoinNode* is the most complex node with regard to implementation. Although it is known which sequences must be executed before the join node is reached, it is unknown, which threads will be the first ones reaching the join node[39].

Our implementation of a join node is a class with lists of objects. An example is given in Fig. 4.12 and its implementation, Listing 4.4. Each incoming edge of the join node is represented by a list (line 2) [39].

Whenever a sequence ending at a join node terminates, all guards of incoming edges and that of the outgoing edge of the join node are evaluated. If all guards hold, a token is added to the appropriate list by calling one of the provided methods (line 3 or 6) and the following sequence is initiated. If at least one guard fails, the ending sequence waits to be notified of a system change before re-evaluating guards.

When started, the subsequent sequence checks whether the join node offers a token by calling the `canJoin` method (line 9). All lists are checked and if each list contains at least one token, tokens are removed from the lists and the initiated sequence may continue. A guard check is not necessary since `canJoin` is only called when a token is added at an incoming edge and before that, guards have been tested. If `canJoin` returns `false`, the sequence is aborted.

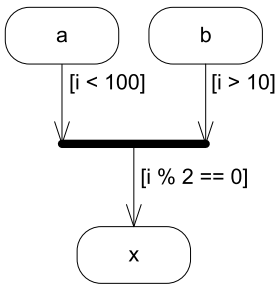



Figure 4.12: Joining of control flows.

```
1  class Join1 {
2    private List<Object> l1, l2;
3    public void add1(object token) {
4      l1.add(token);
5    }
6    public void add2(object token) {
7      l2.add(token);
8    }
9    public boolean canJoin() {
10     if(!(l1.isEmpty() || l2.isEmpty())){
11       l1.remove(0);
12       l2.remove(0);
13       return true;
14     }
15     else return false;
16   }
17 }
```

Listing 4.4: Implementation of *JoinNode*.

Some approaches like e. g. translating activities to Esterel [7] use an implementation based on the assumption that tokens to be joined are siblings created from the same token offer at a fork node. However, such an assumption is unjustifiable since tokens may be produced by distinct initial nodes or be the result of accepted events. Accordingly, in general tokens to be joined are not the result of a fork. Furthermore, since control tokens have no identity, they cannot be identified as siblings even if they are.

To this end, the presented code is a proper implementation of *JoinNode* accommodating the vast diversity of possible executions and resulting combinations of threads to be joined.

## 4.2.5 Guard Propagation

The implementations of control nodes discussed above are all subject to the restriction that sources of incoming flows and targets of outgoing flows are not control nodes. In order to be more general, we define some rules facilitating the replacement of a set of control nodes by a single control node. In preparation therefore, we first introduce *Guard Propagation Rules*.

The idea of guard propagation is, that guards of incoming edges can be combined with guards of outgoing edges and vice versa without changing the execution semantics of the affected flows. We term the prepending of guards of incoming edges to those of outgoing edges *forward propagation*, appending guards of outgoing edges to those of incoming ones, we term *backward propagation*.

The effect of backward propagation is to consider guards of outgoing edges of a control node while evaluating the guard of an incoming edge of this control node. If properly constructed,

the guard of incoming edges of a control node only hold, if a token can reach at least one target downstream the control node. Thus, backward propagation is a means of determining whether or not an offered token can reach any target at all.

The effect of forward propagation is to consider guards of incoming edges of a control node while evaluating guards of outgoing edges. If properly constructed, this guard only holds, if a token can traverse any incoming edge of the control node and, in the same moment, can traverse the outgoing edge. Thus, forward propagation is a means to determining which actions are to be executed downstream of a control node.

Backward and forward propagation are means for determining whether a token may traverse edges connected to a control node, and if so, which actions it may reach.

For the following definitions, let $E = \{e_1, \ldots, e_n\}$ be a set of edges such that $typeOf(e_i) \in \{ControlFlow, ObjectFlow\}$ for each $e_i \in E$. Let $G_e$ be the guard of edge $e$ and $G'_e$ the guard of the same edge $e$ resulting from guard propagation.

**DecisionNode**

Let $inc : DecisionNode \to E$ be a function returning the incoming edge of a decision node, $out : DecisionNode \to \mathcal{P}(E)$ a function returning the set of its outgoing edges. The following guard propagation rules hold for incoming and outgoing edges of *DecisionNode* $D$:

$$\forall e \in out(D) : G'_e = G_{inc(D)} \wedge G_e. \qquad \text{(forward propagation)}$$
$$G'_{inc(D)} = G_{inc(D)} \wedge (\bigvee_{e \in out(D)} G_e). \qquad \text{(backward propagation)}$$

**ForkNode**

Guard propagation very likely is not legal at a *ForkNode* since the token buffering may be affected if guards are changed. If the context in which a *ForkNode* is used cannot lead to token buffering, guard propagation is possible as for a *DecisionNode*.

Let $inc : ForkNode \to E$ be a function returning the incoming edge of a fork node, $out : ForkNode \to \mathcal{P}(E)$ a function returning the set of its outgoing edges. The following guard propagation rules hold for incoming and outgoing edges of *ForkNode* $F$:

$$\forall e \in out(F) : G'_e = G_{inc(F)} \wedge G_e. \qquad \text{(forward propagation)}$$
$$G'_{inc(F)} = G_{inc(F)} \wedge (\bigvee_{e \in out(F)} G_e). \qquad \text{(backward propagation)}$$

**JoinNode**

Let $inc : JoinNode \to \mathcal{P}(E)$ be a function returning the set of incoming edges of a join node, $out : JoinNode \to E$ a function returning its outgoing edge. For a *JoinNode* $J$, the following guard propagations hold:

$$G'_{out(J)} = (\bigwedge_{e \in inc(J)} G_e) \wedge G_{out(J)}. \qquad \text{(forward propagation)}$$
$$\forall e \in inc(J) : G'_e = G_e \wedge G_{out(J)}. \qquad \text{(backward propagation)}$$

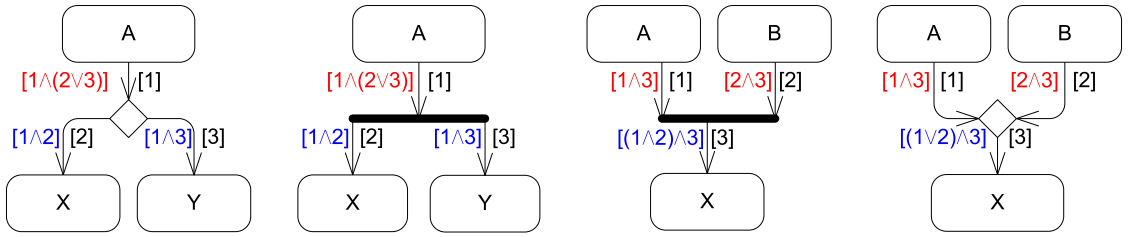
**MergeNode**

Let $inc : MergeNode \to \mathcal{P}(E)$ be a function returning the set of incoming edges of a merge node, $out : MergeNode \to E$ a function returning its outgoing edge. For a *MergeNode* $M$, the following guard propagations hold:

$$G'_{out(M)} = (\bigvee_{e \in inc(M)} G_e) \wedge G_{out(J)}. \qquad \text{(forward propagation)}$$
$$\forall e \in inc(M) : G'_e = G_e \wedge G_{out(M)}. \qquad \text{(backward propagation)}$$

Figure 4.13 abstractly shows guard propagation at *DecisionNode* (a), *ForkNode* (b), *JoinNode* (c), and *MergeNode* (d). Original guards are annotated to the right side of an edge, propagated guards are annotated to its left side. Furthermore, forward propagation is marked with blue color, backward propagation is marked red. Note that forward and backward propagation are applied on the original guards, i. e. none of both propagations is applied on the results of the other one.

Here and in the following, we use a generic notation for guards. By $[n]$, we denote a guard which is a numbered, but not further specified boolean expression. The guard $[n \wedge m]$ denotes a boolean expression resulting from the conjunction of the expressions specified by the guards $[n]$ and $[m]$. The $\wedge$-operator is granted a higher precedence than the $\vee$-operator.




(a) Guard propagation at *DecisionNode*

(b) Guard propagation at *ForkNode*

(c) Guard propagation at *JoinNode*

(d) Guard propagation at *MergeNode*

Figure 4.13: Guard propagation at control nodes.

### Application of Guard Propagation Rules

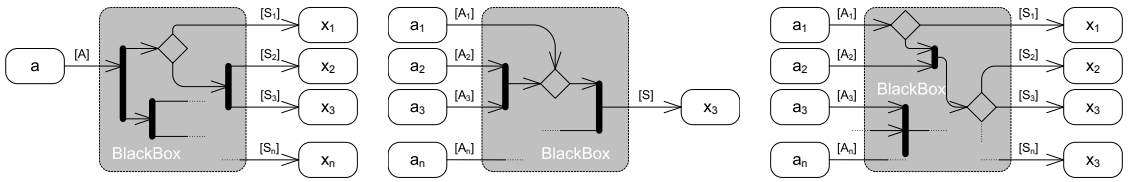
By propagating guards according to the introduced guard propagation rules, two kinds of characteristic guards can be constructed for each flow:

- *selection guard*, which is the result of forward propagation, denoted $S_i$ in Fig. 4.14. This guard determines where to continue activity execution.
- *activation guard* denoted $A_i$ in Fig. 4.14, which is the result of backward propagation of selection guards. This guard holds, if a token offer can reach any target, i. e. if the execution can be continued somewhere.

The backward propagation of selection guards is performed after forward propagation and after removing guards of all incoming edges of each control node which is part of the flow[3].

A net of control flows and control nodes thus can be abstracted to a black box which can distribute a token to its targets if an activation guard holds. To which target to direct a token — or insofar as fork nodes are concerned, to which targets to direct copies of a token — is determined by evaluating selection guards.

Activation and selection guards must be evaluated whenever a new token is offered to the black box and whenever the system state changes, e. g. when values of variables or attributes are changed or when links are created or destroyed.




(a) *1–N* Black Box

(b) *N–1* Black Box

(c) *N–M* Black Box

Figure 4.14: Abstracting from control flow nets to black boxes.

Code generation for activities targets to statically determine which action to execute next whenever an action is terminated. Implementations of control flow nets can become very complex ending in worst case with the generation of an interpreter such as described by Bulach [13].

By the limitation of at most one control node between two actions, generated code can be kept plain and clear by using the Java implementations of control nodes presented above. These implementations already make use of activation and selection guards.

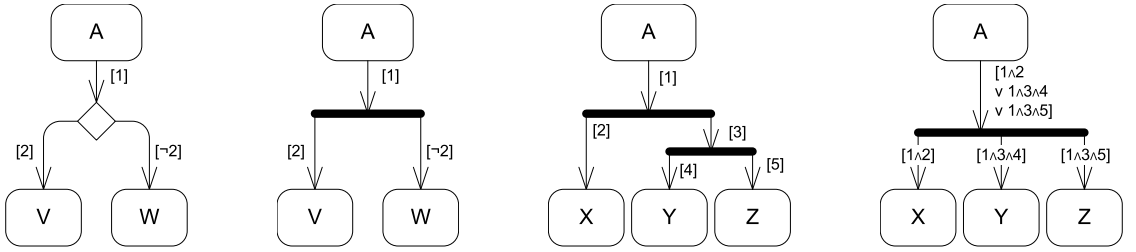
For nets containing only fork and decision nodes, i. e. *1–N* Black Boxes (Fig. 4.14(a)), assembling activation guards across multiple subsequent nodes still produces a valid result. A holding activation guard guarantees that tokens will be directed to some target without resting at any downstream node due to failing guards. The actual target of each token can be determined by checking selection guards, provided that there is no non-determinism at any decision node. If

[3]For merge nodes, an exception from this rule is needed, which will be discussed later.

non-deterministic decisions occur, selection guards of a set of actions may hold of which only a subset of actions may execute.

If we abandon non-deterministic decisions, evaluation of activation guards and starting sequences for which selection guards hold, make up a semantically correct implementation of this type of control flow nets. Furthermore, abandonment of non-determinism affords replaceability of decision nodes by fork nodes as shown in Fig. 4.15(a) and Fig. 4.15(b). Note that both activities have the same execution semantics.

This replacement can be advanced to a replacement of cascading fork nodes as shown in Fig. 4.15(c) by one single fork node (Fig. 4.15(d)).




(a) Definite choice at a *DecisionNode*.

(b) Definite choice at a *ForkNode*.

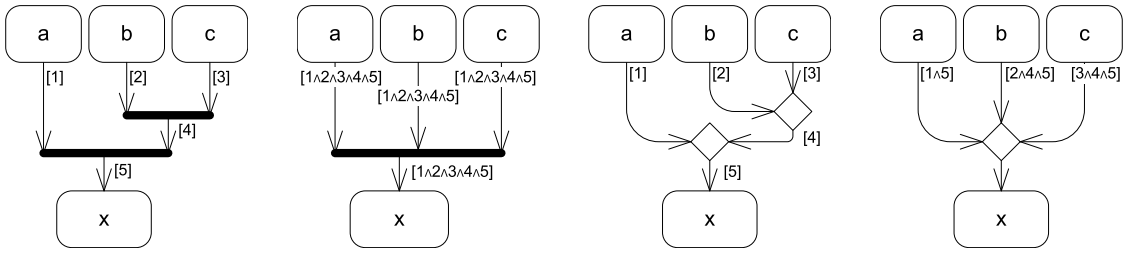
(c) Cascade of fork nodes.

(d) Fusion of the cascade from (c) to a single fork node.

Figure 4.15: Simplifications of 1-N black boxes.

A net of decision nodes can be translated to a single decision analogously to Fig. 4.15(c) and Fig. 4.15(d). The implementation of *1–N* black boxes therefore is possible if only decision and fork nodes are involved by either replacing all decision nodes by fork nodes or — if non-determinism occurs — by propagating guards and translating each node separately.

Analogously, subsequent join nodes like those of Fig. 4.16(a) can be combined to a single join (Fig. 4.16(b). Subsequent merge nodes like such of Fig. 4.16(c) may be combined to a single merge node as well, however, valid guards are obtained only if solely applying backward propagation (Fig. 4.16(d)). Both of these rules contribute to a reduction of complexity of *N–1* Black Boxes (Fig. 4.14(b)) if both types of nodes do not occur strictly alternately downstream.




(a) Cascade of join nodes.

(b) Single join node implementing the cascade of joins from (a).

(c) Cascade of merge nodes.

(d) Single merge node implementing the cascade of merge nodes from (c).

Figure 4.16: Simplifications of N-1 black boxes.

By applying the above defined rules, control flow nets can be processed if all control nodes are of the same type. Flows containing both decision and fork nodes may be handled if decisions are deterministic. In the following, we assume all decisions to be deterministic. Non-determinism could be handled by manually implementing the non-deterministic decision to be made, as outlined in Sect. 4.2.7.

Although the rule for forward propagation at merge node is defined correctly, it has to be modified in order to correctly assemble valid selection and activation guards. It is the only rule that adds conditions by conjunction. In this way, the result of the propagation contains fragments

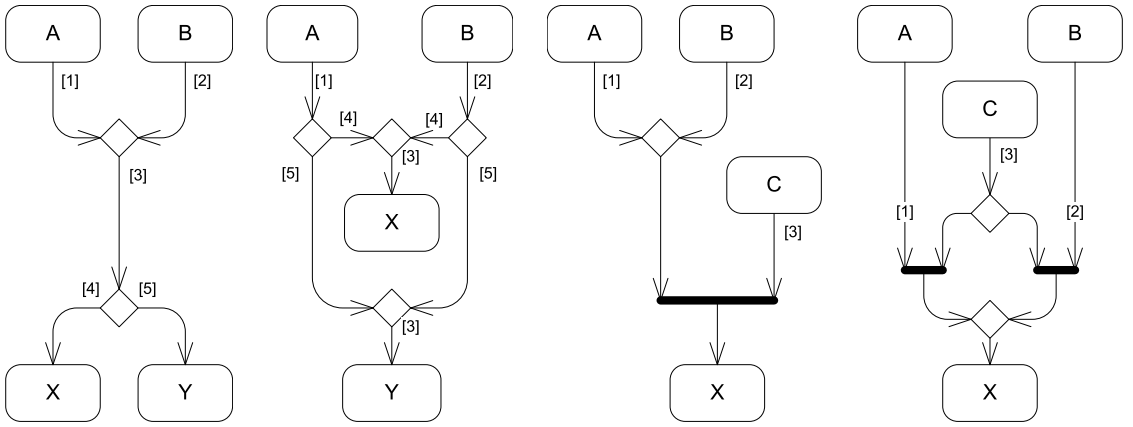
that refer to guards of edges that are not part of the actual path from the source to the target. The selection guard for action $X$ of the model shown in Fig. 4.17(a) obtained by forward propagation is $((1 \vee 2) \wedge 3) \wedge 4$ the selection guard for action $Y$ is $((1 \vee 2) \wedge 3) \wedge 5$. By backward propagation of these guards, the activation guard of both actions $A$ and $B$ is $((1 \vee 2) \wedge 3) \wedge 4) \vee ((1 \vee 2) \wedge 3) \wedge 5)$.

If action $A$ offers a token, the activation guard might be true due to conditions 2, 3, and 4 holding while conditions 1 and 5 do not hold. In this case, the selection guard of $X$ succeeds, too. But this is caused by those parts of the selection and activation guard referring to edges that are not part of the path from the actually considered source to the intended target. Indeed, the fragments derived from guards actually located at traversed edges fail (1 and 5). Therefore, the selection guard for paths containing merge nodes must be determined for each source separately.

This can be achieved by a transformation which pushes merge nodes to the end of each flow as shown in Fig. 4.17. Merge nodes occurring ahead of a decision node can be pushed downstream of the decision as shown in Fig. 4.17(a) and Fig. 4.17(b). This rule can be applied in the same way for contexts similar to that of Fig. 4.17(a) but with a *ForkNode* occurring at the position of the *DecisionNode*. Analogously, the result of a transformation equals to that of Fig. 4.17(b) with each decision node being replaced by a fork node.

Figure 4.17(c) and Figure 4.17(d) show the same rule for a *JoinNode* downstream to a *MergeNode*. Note that this transformation is only semantic preserving if for each incoming edge of the join, the weight is defined to be 1. Otherwise, this transformation conflicts with the following statement of the UML specification: *"Multiple control tokens offered on the same incoming edge [of a join node] are combined into one (. . . )"* [69, §12.3.34].

If a token is offered at $A$, another one at $B$, and two tokens are offered at $C$ but guard 3 fails, action $X$ will be executed once as soon as guard 3 succeeds. When executing the transformed activity, $X$ will be executed once, too, but the offer of either the token at $A$ or $B$ will continue standing if all tokens offered at incoming edges of the join are consumed. With a weight of 1 defined for all incoming edges of the join node, action $X$ is executed twice because only one token of each offer at incoming edges is consumed.




(a) *MergeNode* ahead of a *DecisionNode*.
(b) Result of pushing a *MergeNode* after a *DecisionNode*.
(c) *MergeNode* ahead of a *JoinNode*.
(d) Result of pushing a *MergeNode* after a *JoinNode*.

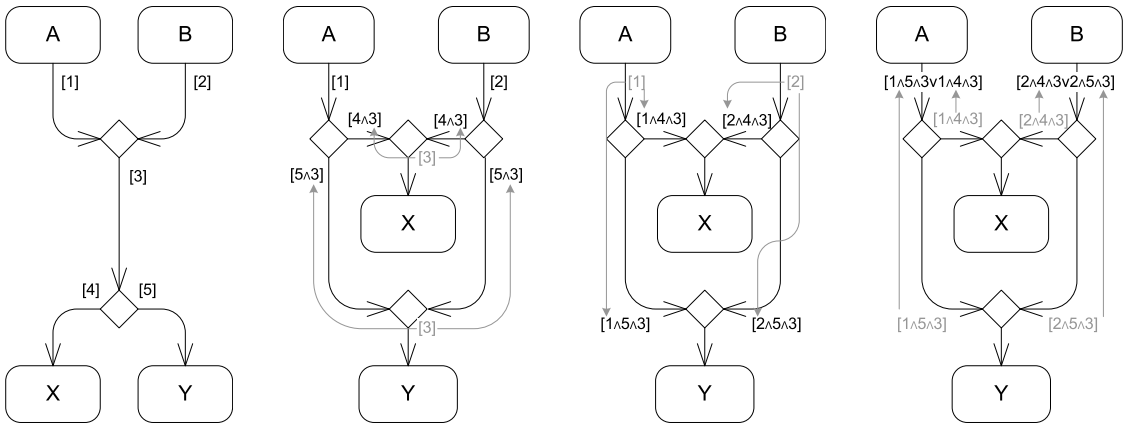
Figure 4.17: Pushing *MergeNode* downstream.

If after applying rules for pushing merge nodes downstream, these nodes build a cascade like shown in Fig. 4.16(c). This cascade is replaced by a single merge node as illustrated in Fig. 4.16(d).

Note that the application of forward and following backward guard propagation on an activity as in Fig. 4.18(a) into one as in Fig. 4.18(b) results in flawed activation and selection guards. Forward propagation returns $(1 \wedge 4 \vee 2 \wedge 4) \wedge 3$ as selection guard for action $X$ and $(1 \wedge 5 \vee 2 \wedge 5) \wedge 3$ as selection guard for action $Y$. Backward propagation leads to the activation guard $((1 \wedge 4 \vee 2 \wedge 4) \wedge 3) \vee ((1 \wedge 5 \vee 2 \wedge 5) \wedge 3)$ for actions $A$ and $B$. With these guards, a token offered at the outgoing edge of $A$ could activate $X$ if $(2 \wedge 4 \wedge 3)$ is true. But Fig. 4.18(a) shows that for a token flow from $A$ to $X$, guard 1 necessarily must hold.

Valid results are achieved by modifying the algorithm for the construction of activation and selection guards: First, guards from outgoing edges of the last merge node are propagated to

incoming edges of this node(see also Fig. 4.18(b)). Following this transformation, forward propagation is applied starting at all outgoing edges of actions until incoming edges of the merge node are reached (Fig. 4.18(c)). Since by traversing incoming edges of this merge node can only activate one action (the *target action*), the guard at an incoming edge of the merge node is the selection guard for this target action valid for the source action from where the forward guard propagation started. By now applying backward propagation to these selection guards, valid activation guards are constructed (Fig. 4.18(d)).




(a) Initial situation.

(b) Backward propagation of guard to incoming edges of a *MergeNode*.

(c) Assembling selection guards by forward propagation.

(d) Assembling activation guards by backward propagation.

Figure 4.18: Assembling *Selection Guards* and *Activation Guards* by guard propagation.

### 4.2.6 Black Box Implementation of Complex Flows

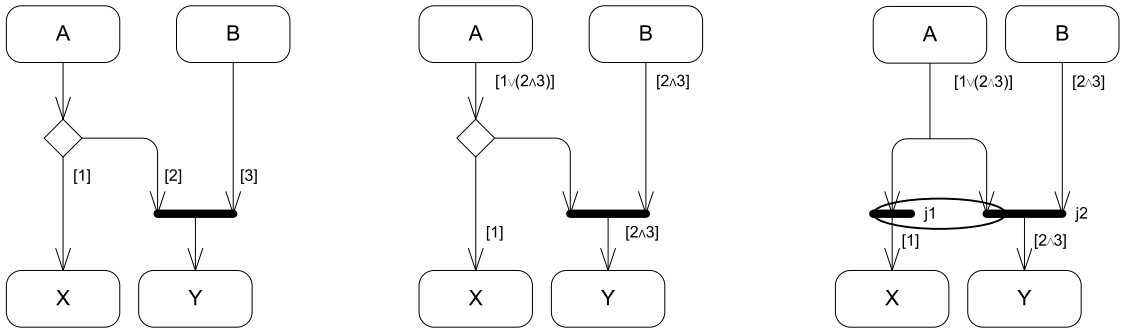
After having defined some rules for the transformation of *1–N* Black boxes not containing join or merge nodes and *N–1* Black Boxes not containing decision or fork nodes, we now discuss some patterns occurring when using control nodes for splitting flows and those for combining flows together. *N–M* Black Boxes consisting of a combination of the following patterns can be implemented by combining the proposed transformations and implementations.

#### Volatile Offers

The first case we consider is a combination of decision node and join node, as shown in Fig. 4.19. The initial situation is depicted in Fig. 4.19(a). Fig. 4.19(b) shows the same situation after guard propagation. The problem here is what to do if after executing action $A$, guard 1 fails, but guard 2 and 3 hold. If no token is offered to the outgoing edge of action $B$, the join cannot emit a token. If a token is offered, then this token together with the token offered at action $A$ would cause a token to be offered to action $Y$.

The implementation of a join node given in Sect. 4.2.4 considers offers by adding tokens to lists representing incoming edges. As long as such an offer cannot expire since the token cannot be consumed by another flow target, this implementation suffices. However, the combination with a previous decision node invalidates the token offer if guard 2 does not hold. This fact is considered by the outgoing edge which is guarded by condition 2 and 3. But if guard 1 holds, the offer definitely has to be removed permanently from the list representing incoming edges. For this reason, we consider the token offered to the outgoing edge of $A$ to be a volatile offer to the join node.

How this can be achieved is illustrated in Fig. 4.19(c), which is not conforming UML although using its notation. It is a visualization of an implementation based on the insertion of another join node between the decision node and action $X$ and the deletion of the decision node itself. The join nodes $j1$ and $j2$ share a common list for representing the split edge from $A$ to $j1$ and $j2$. If one of both nodes can join, it consumes the token, which also is offered to the other join node, from the list. As an indication for a common representation of incoming edges of different join nodes, we use an ellipse surrounding the ends of concerned edges.




(a) Initial situation.

(b) Result of applying guard propagation rules.

(c) Implementation model (not UML).

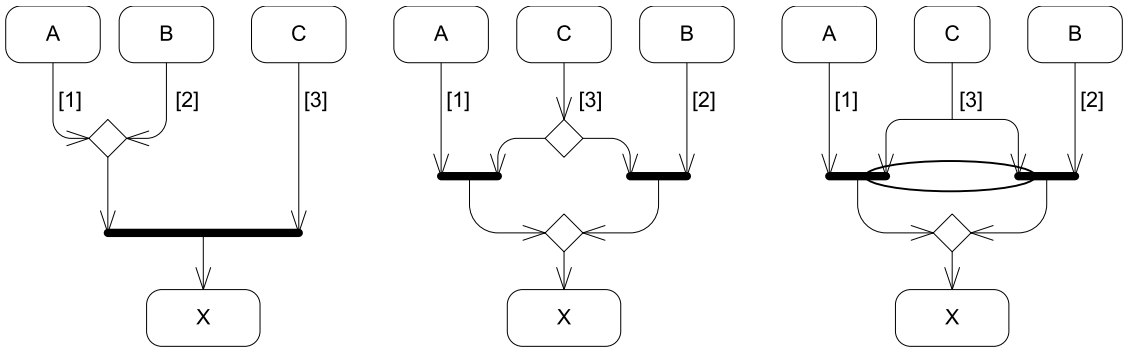
Figure 4.19: Implementing volatile token offers.

A modification to the implementation of join node is necessary as well since it now is possible that tokens are offered even though the guard of the outgoing edge fails. Therefore, the implementation of `canJoin()` must return false, if not all incoming edges have token offers, it must wait when tokens are offered but guards fail and it must return true, when tokens are offered and guards hold.

### Unknown Origins

Based on the implementation for volatile offers, we can support another combination of control nodes which is a merge node preceding a join node (Fig. 4.20(a)). This combination is problematic if one or both incoming edges of the merge node are guarded. If a token is offered by actions $A$ or $B$, this token is added to the list of tokens representing the offers of the outgoing edge of the merge node. When another token is offered by action $C$, it is necessary to know the actual origin of a token emitted by the merge node in order to be able to evaluate the right guards. But if the offered token has been added to the list representing an incoming edge of the join node, the token origin is unknown. This is no problem if all edges upstream to the merge node are not guarded because then, there is no guard to evaluate.

The pattern shown in Fig. 4.20(a) can be transformed to the pattern given in Fig. 4.20(b) by pushing the merge node downstream. The problem then has turned into a volatile offer (Fig. 4.20(c). A token provided by $C$ is either merged with a token of $A$ or $B$, thus being a volatile offer to both join nodes.




(a) Initial situation.

(b) Model transformation to uncertain token offers.

(c) Implementation model of transformation result (not UML)

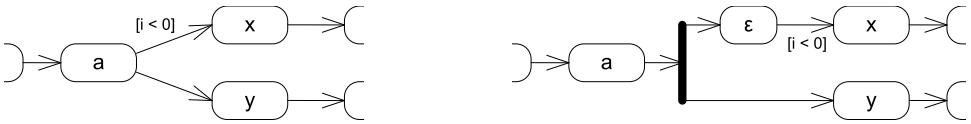
Figure 4.20: Implementing uncertain token origins.

### Fork Node Buffering

Another semantic feature of UML that is difficult to handle by statical analysis is buffering of tokens at fork nodes. It is specified as follows:

> Any offer that was not accepted on an outgoing edge [of a fork node] due to the failure of the target to accept it remains pending on that edge and may be accepted by the target at a later time. These edges effectively accept a separate copy of the offered tokens, and offers made to the edges stand to their targets in the order in which they were accepted by the edge (first in, first out). This is an exception to the rule that control nodes cannot hold tokens if they are blocked from moving downstream (see "Activity (from BasicActivities, CompleteActivities, FundamentalActivities, StructuredActivities)" on page 324).
>
> Note that any outgoing edges that fail to accept an offer due to the failure of a guard do not receive copies of those tokens. [69, §12.3.30]

Because of token buffering, making implicit fork nodes explicit cannot be done by just inserting a fork node like shown in Fig. 4.21. Such a transformation is not semantics preserving if guards are applied to the outgoing edges of an action (Fig. 4.21(a)). Tokens at outgoing edges of a fork node are only buffered if the guard of this outgoing edge is true, but token offers to outgoing edges of actions are independent from the evaluation of guards. For this reason, it is necessary to additionally insert empty actions (i. e. actions with no effects) at each outgoing edge of the inserted fork node which is guarded (action $\epsilon$ in Fig. 4.21(b)). By this, all outgoing edges can accept a token and the inserted actions just act as token buffers.




(a) Implicit fork between action *a* and actions *x*, *y*.

(b) Explicification of the implicit fork of (a).

Figure 4.21: Explicification of an implicit fork.

Figure 4.22 illustrates the effects of token buffering by means of a simple example which is an assistance system for decelerating an aircraft after touch down. For deceleration, wheel brakes and thrust reversers can be used. Both options can be preselected and are automatically applied considering the following constraints:

- thrust reversers must not be used in flight, i. e. the signal weight-on-wheels (WoW) indicating that the aircraft hit the ground must be received before
- wheel brakes must not be used if the speed of 120 knots is exceeded
- wheel brakes must not be applied before wheels hit the ground in order to avoid immoderate abrasion of tires or even tire failures.

If only one of both options is preselected, only this option will be applied. This scenario is shown for using thrust reversers in Fig. 4.22(b). Green filled circles indicate tokens, green arrows indicate a token flow. The sequence of single execution steps following Sarstedt [83] is indicated by the number next to the arrows indicating token flows. **T** (true) and **F** (false) indicate the actual values of guards in the activity.

In a first step, the accept event action listening for the *WoW* signal is activated. After receiving *WoW*, one token is offered to the outgoing edge of the accept event action. This token is offered to action *Reverse* whereas neither a token is offered to *Brake* nor is a token buffered (discarding of the token is indicated by the red circle filled with a $\times$) due to the failing guard *auto_break* representing the missing preselection of wheel brakes. The evaluation of the guard *[speed < 120]* is irrelevant for this activity execution.

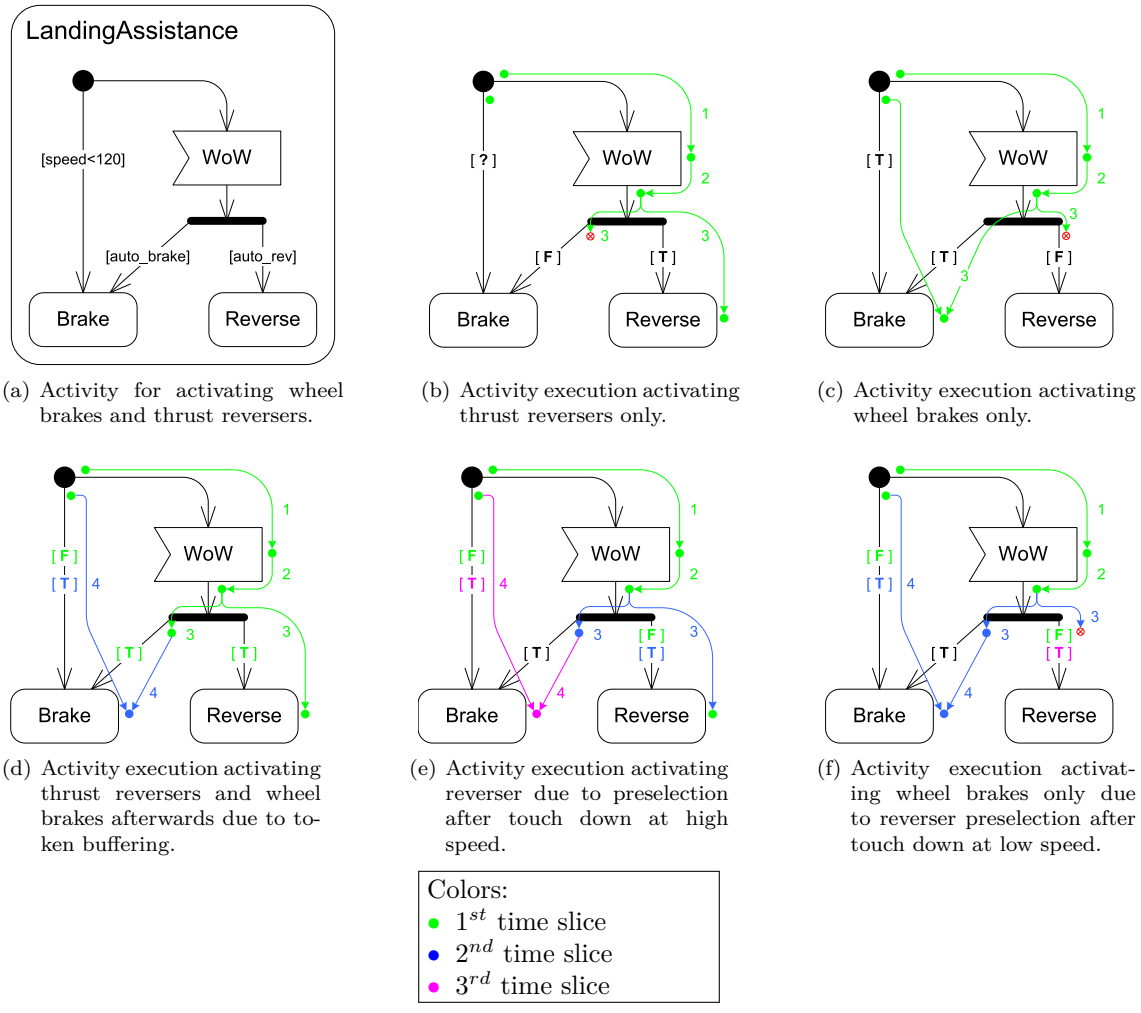
If only wheel brakes are preselected, after receiving *WoW*, a token is offered to *Brake*. Since if the speed limit of 120 knots is not exceeded another token is offered to *Brake* and thus all token flow prerequisites are satisfied, *Brake* executes and the token offered to the edge connected to *Reverse* is discarded due to the failing guard (Fig. 4.22(c)).

A situation where a token actually is buffered at an outgoing edge of the fork node is shown in Fig. 4.22(d). The first three steps of the activity execution lead to the execution of *Reverse* while brakes cannot be applied because the speed constraint is not satisfied (steps and guards are drawn

in green color). Since guards of all outgoing edges of the fork node are true, a token is buffered and contributes to the execution of *Brake* as soon as the speed constraint is satisfied (indicated by guards and steps in blue color).

A similar situation is illustrated by Fig. 4.22(e). During this execution, after performing the first steps (green), neither breaks nor reversers are used because speed is to high for wheel breaks, which already are selected, reversers are not preselected. If now selecting reversers (indicated by blue color), *Reverse* is executed and a token is buffered. Wheel brakes are applied as soon as the speed falls under 120 knots (indicated by cyan color).

A last scenario is depicted by Fig. 4.22(f). In the same initial situation as before (indicated by green color), the speed constraint is satisfied (blue) before selecting the reversers (cyan). Wheel brakes will be used whereas reversers are not activated due to the discarding of the token during step 3.




(a) Activity for activating wheel brakes and thrust reversers.

(b) Activity execution activating thrust reversers only.

(c) Activity execution activating wheel brakes only.

(d) Activity execution activating thrust reversers and wheel brakes afterwards due to token buffering.

(e) Activity execution activating reverser due to preselection after touch down at high speed.

(f) Activity execution activating wheel brakes only due to reverser preselection after touch down at low speed.

Colors:
- $1^{st}$ time slice
- $2^{nd}$ time slice
- $3^{rd}$ time slice

Figure 4.22: Application of fork node buffering in an example activity.

At a first glance, token buffering is a nice feature to support concurrency where the execution of one branch is delayed until an additional condition holds which is not relevant for other branches. In the example given above, wheel brakes and thrust reversers shall be used together. The first condition affecting both reversers and brakes is that the aircraft must not be in flight. The second condition only affecting brakes is that speed must not exceed 120 knots. However, by using guards, two more conditions appear which are the preselection of brakes and reversers. Preselection is intended to be done before touchdown, the actual activation is effected after touchdown.

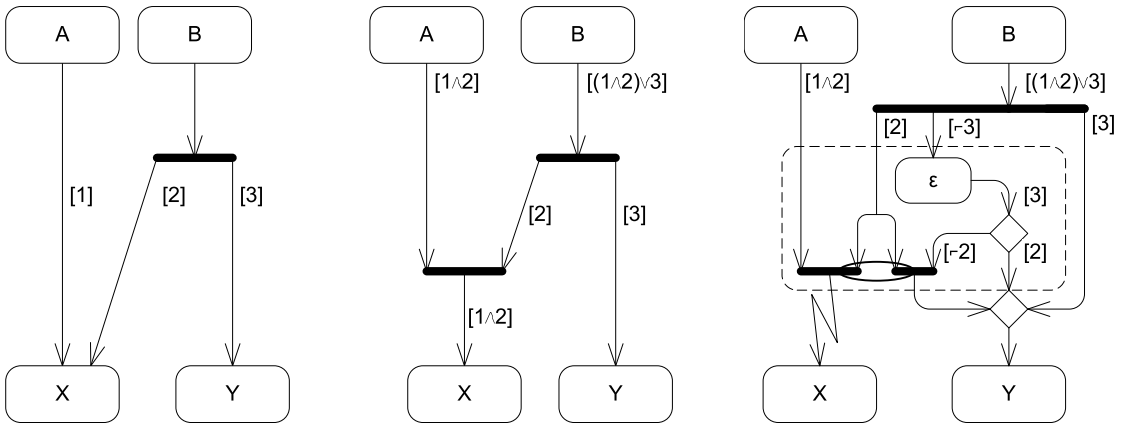
On closer inspection, we can see that there is another dependency between actions *Reverse* and *Brake* which is *Reverse* will never be executed after execution of *Brake* has started. Suchlike unobvious dependencies undermine a better perceptibility of relations between elements, which

is targeted by graphical notation. Furthermore, the example shows that highly sophisticated semantics of concepts are to be taken with a pinch of salt.

Nevertheless, we support token buffering at fork nodes as described in the following on the basis of the generic activity given in Fig. 4.23(a). A token may be buffered at the left outgoing edge of the fork node if a token is offered at $B$, condition 3 holds, and $X$ cannot accept a token due to a missing offer at $A$ or the failure of guard 1 while condition 2 holds. At the right outgoing edge of the fork node, a token is never buffered since it either may traverse the edge or the guard fails. A token offered at $B$ may also move downstream if a token is offered at $A$ and conditions 1 and 2 hold. Figure 4.23(b) shows the same situation after explication of the implicit join of incoming edges of $X$. Regarding the activation guard at the outgoing edge of $B$, guard 1 should be considered, however, to the edge connecting the fork and the join nodes, only condition 2 must be taken into account. Otherwise, token buffering would depend on a more special guard than it should. Consequently, to a join node representing an implicit join, propagation rules do not fully comply.

For a statical analysis, a flow of a token from $B$ to $X$ due to offers at $B$ and $A$ and guards 1 and 2 holding while guard 3 fails, is difficult to detect. The reason is that information about the existence of an offer at $A$ is not available to the thread executing the sequence ending with $B$.

The problem of token buffering is related to volatile offers. A token offered at $B$ is offered to the outgoing edges of the fork. Whether this offer is valid can only be determined at that point in time when tokens actually flow. For action $Y$, the offer is not stable since it is only valid if condition 3 holds while tokens flow. However, this situation cannot be modeled based on the implementation of volatile offers since the offer to the join node is volatile, too.




(a) Initial situation requiring token buffering.

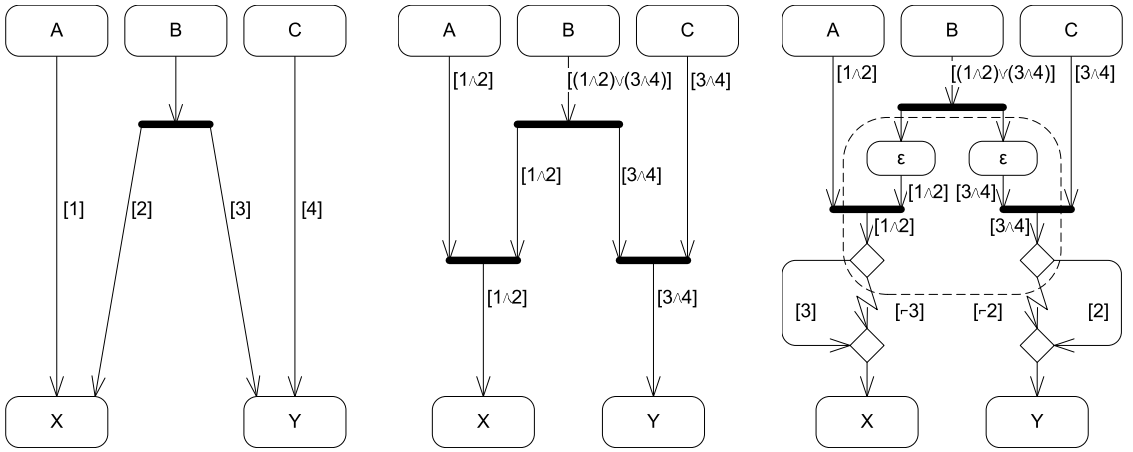
(b) Result of guard propagation and explication of implicit join.

(c) Explicit modeling of token buffering.

Figure 4.23: Explicit modeling of fork node buffering.

Fig. 4.23(c) shows a modification to the initial model making token buffering explicit. If condition 3 holds and a token is offered at $B$, this token is directly passed to $Y$. If to that time, also condition 2 holds, a token is offered to the join, if not, no token is offered. If condition 3 does not hold, the token activates action $\epsilon$, which has no effects but offering the token to $Y$ via a merge node. The token rests at the outgoing edge of $\epsilon$ until condition 3 holds. Note that if condition 3 does not hold, condition 2 must do so in order to satisfy the activation guard. If both conditions fail, the fork node will not offer any token to any of its outgoing edges. If now a token is offered at $A$ and conditions 1 and 2 hold, the token offered to the join node at the outgoing edge of the fork node is accepted causing the region to be interrupted.

An implementation of a fork node having two outgoing edges which possibly both have to buffer tokens is given in Fig. 4.24(c). It bases on the same idea as the implementation given in Fig. 4.23(c). Note that the join nodes intersecting the outline of the interruptible activity region are intended to be not owned by the region.




(a) Initial situation requiring token buffering at all outgoing edges.

(b) Result of guard propagation and explication of implicit join.

(c) Explicit modeling of token buffering at two outgoing edges of a fork node.

Figure 4.24: Explicit modeling of multiple fork node buffering.

### 4.2.7 Further Considerations on Control Nodes

For a proper consideration of control nodes in general, in particular when flows contain more than one control node, additional research is required including an appropriate formalization of control nodes' semantics. A formalization of control nodes would facilitate the definition of rules for transformations as well as to prove correctness of such rules.

Another approach — which is also supposable in combination with or based on a formalization of control nodes — is to establish a catalogue of patterns for which implementations are provided. Code snippets for pattern implementation could then be combined according to the matching of patterns in a complex activity graph.

An open question is how complex useful control flows may become. For sure, beyond some degree of complexity, models will also be beyond human comprehension. Consequently, it certainly is satisfactory to be able to handle flows of a complexity that is still understandable for developers. Criteria for how to determine whether a flow in this respect is worth being supported at code generation do not exist. Therefore, we assume that by the techniques presented above, a good portion of control flows which actually may ease modeling can be processed.

The initial limitation of our approach to support at most one control node between the source and the target of a flow has been pushed to support an unlimited number of control nodes of the same type. Furthermore, we consider a number of patterns which we described above. The fact that flows might occur in models that are not yet covered does not lapse our approach. Suchlike flows still can be implemented by hand as long as some kind of integration with the generated code is provided.

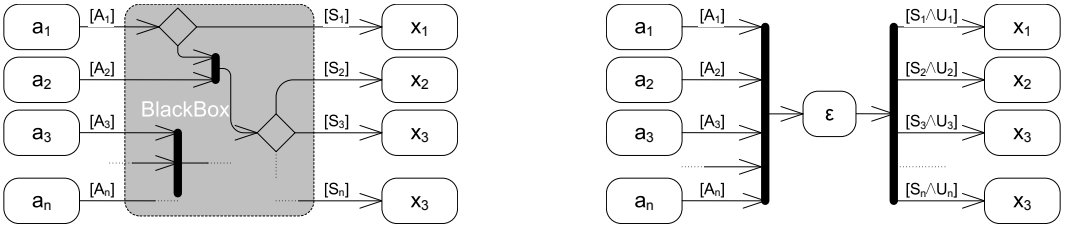
A low level approach to implement a complex flow as in Fig. 4.25(a) is to use a combination of a join node and a downstream fork node, as shown in Fig. 4.25(b). An implementation of any black box is possible by manually implementing the method `canJoin` (see Fig. 4.4) of the join node. This could be achieved by a specialization of the class implementing the join node and method overriding. The implementation of `canJoin` by the user must provide the following:

- check token offers and guards at incoming edges of the join node
- decide which actions $X_i$ must be started according to the black box semantics
- for each $X_i$ to be started, set $U_i$ to true
- evaluate guards of outgoing edges of the fork node
- if any guard evaluates to true, offer a token to $\epsilon$.

The action $\epsilon$ has no effects, it just serves as a segmentation of the flows. If a token is offered to the join node, the method `canJoin` is automatically called in order to determine whether $\epsilon$ can be

started. If it can be started, it has no effects, but the downstream fork node will offer tokens to actions $X_i$. The complete control flow net thus is implemented and integrated into the generated code.

However, an interface for such an implementation should be provided. It should comprise two elements: a modeling element to indicate a complex control flow which is implemented instead of being modeled and a hook in code where to insert the complex decision of action activation.




(a) N–N Black Box

(b) representing a N–N Black Box by a combination of *JoinNode* and *ForkNode*

Figure 4.25: Abstracting from black boxes.

In the example given by Fig. 4.25(a), `canJoin` must return true if a token is offered at $a_1$ while $A_1$ and $S_1$ both evaluate to true. In addition, the token offered at $a_1$ must be consumed and $U_1$ must be set to true. Another situation where the black box can emit a token is another token additionally offered at $a_2$ and $A_2$ holds while $S_1$ does not hold and at least one of $S_2$ or $S_3$ holds. Then, `canJoin` must set either $U_2$ or $U_3$ to true and consume the input offered at $a_1$ and $a_2$ and offer a token to $\epsilon$.

### 4.2.8 Implementing Object Flows

In Fig. 4.5, two options for passing a single object token are given:

```
int x = c(); d(x);        and        d(c());
```

The first alternative is more general since the second alternative requires an unguarded flow directly from action $c$ to action $d$. Since pins are a specialization of *MultiplicityElement*, they may hold multiple values at the same time according to their lower and upper bounds. Regarding this, a more general implementation of object flow is given in Listing 4.5.

```
1 List<int> x = ...
2 x.addAll(c());
3 d(x);
```

Listing 4.5: Implementation of an output pin

This implementation considers the flow of multiple tokens, but still is not compliant to the UML specification regarding multiplicity bounds of pins: *"An action may not put more values in an output pin in a single execution than the upper multiplicity of the pin"* [69, §11.3.27]. In order to implement output pins correctly, they must be represented by a list. Relating to the implementation of Listing 4.5, the length of the list of objects returned by `c()` must not be greater than the upper bound of $x$. Additionally, it must be assured that if an action is executed repeatedly, tokens of previous executions must flow downstream before tokens created by later executions cause the pin's upper bound to be violated. The semantics of activities does not define how to handle such an output pin overflow.

For guarded object flows, the execution must be paused after adding values to the list representing the source output pin and must not be resumed before the guard holds.

Actions with an input pin having a lower bound greater than 1 cannot be executed before the number of offered input tokens reaches the lower bound of the pin. How to consider multiplicity bounds of pins is shown in Listing 4.6.

```
1  List<int> x = ... // implementation of the output pin
2  List<int> i = ... // implementation of the input pin
3  int iupper = ...  // upper bound of input i
4  int ilower = ...  // lower bound of input i
5  ...
6  case <id_c>:
7    x.addAll(c());
8    int upper = x.size() > iupper ? iupper : x.size();
9    while (upper > ilower){
10     i.addAll(x.subList(0, upper);
11     x.sublist(0, upper).clear();
12     new ActivityThread(<id_d>).start();
13     upper = x.size() > iupper ? iupper : x.size();
14   }
15   break;
16 case <id_d>:
17   d(i);
18   ...
```

Listing 4.6: Implementation of an object flow considering input and output multiplicities.

If an output pin contains more elements than the upper bound of the downstream target pin, the target action must be executed repeatedly until the number of remaining token offers does no longer satisfy the lower bound of the target. Therefore, it is not sufficient to start a downstream action. Rather, it is necessary to start downstream actions in own threads in order to be able to start additional, concurrently executing actions (Listing 4.6, line 12).

An action cannot execute if the upstream output pin does not offer enough tokens to satisfy the lower bound of the action's input pin. In that case, the current thread ends. During the activity execution, the upstream output pin may obtain additional objects if the upstream action is executed again (referred to Listing 4.6, action $c$ may be executed again). The evaluation of bounds then is performed again and downstream actions might execute.

### Object Flows with Control Nodes

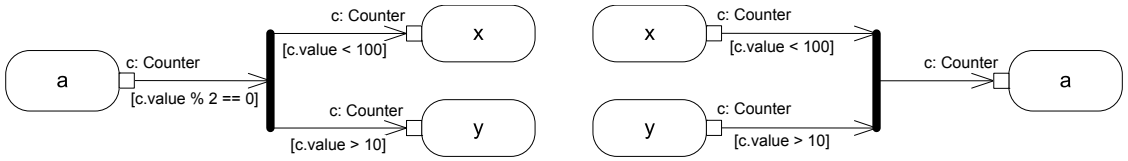
The implementation of control nodes needs only little adaptations to support object flow, if only one node is considered between a source and a target pin. If we assume that only one object flows at a time, this object must be available for guards. However, even though only one token flows at a time, multiple tokens may be offered by the source pin. Since the implementation of sequences of actions is not mapped to different thread classes but code for all sequences is contained in one single class divided into different case-blocks of a switch statement, it is not possible to refer to that individual object which has been created in the context of the current sequence execution. In consequence, if object tokens cannot flow because guards fail, for each token waiting at an output pin, a thread is waiting as well. Each of these threads will check guards for each waiting token. With respect to CPU time, optimizations should be applied, but conceptually, this approach will not fail as long as guard evaluation of different threads is synchronized in order to prevent race conditions. The number of waiting threads will decrease with each token flow.

If multiple objects must flow at a time because of lower bounds of input pins greater than 1, guards must be repeatedly evaluated for a set of object tokens until enough valid objects are found or guards have been evaluated for all objects. This requires more complex code and entails the problem that the number of waiting threads does not decrease synchronously with the number of available tokens. We leave this problem which is very likely solvable as well as the more complex code of guard evaluation for future work.

### Concurrent Object Flows

Figure 4.26 gives an example of concurrent object flows. It includes splitting a flow up into parallel flows (Fig. 4.26(a)) and joining two flows (Fig. 4.26(b)).

Listing 4.7 gives an implementation of a fork of an object flow, Listing 4.8 gives an implementation for joining object flows.




(a) Modeling concurrent object flows.

(b) Joining concurrent object flows.

Figure 4.26: Concurrent object flows.

The main difference compared to a fork node implementation in combination with control flows is that if objects traverse outgoing edges, these objects are copies of the offered object token (Listing 4.7, lines 20 and 26) and the source object token is removed from the offering output pin (line 30).

With regard to join node implementation, the most important change is that the join class no longer has lists representing incoming edges. It rather directly accesses output pins, from which object tokens are offered. Another striking change is that guard evaluation is done within the `canJoin()` method. The reason for this is that guard evaluation requires access to objects at output pins located at the end of other sequences. In order to avoid code duplication, accessing objects is implemented once as part of the join node implementation.

For both control node implementations, another important difference is that all offered object tokens must be considered. Therefore, guard evaluation is nested in a foreach-loop which is aborted as soon as an object is found, for which guards evaluate to true.

```
1  // Declaration of pins:
2  List<Counter> a_c = ...;
3  List<Counter> x_c = ...;
4  List<Counter> y_c = ...;
5
6  ...
7  while (!a_c.isEmpty()){
8    Iterator<Counter> it =
9      a_c.iterator();
10   Counter c;
11
12   while (it.hasNext()){
13     c = it.next();
14
15     if (c.value%2 == 0 &&
16         (c.value<100 ||
17          c.value>10)){
18
19       if (c.value<100){
20         x_c.add(c.clone());
21         new ActThread(<id_x>).
22           start();
23       }
24
25       if (c.value>10){
26         y_c.add(c.clone());
27         new ActThread(<id_y>).
28           start();
29       }
30       it.remove(c);
31     }
32   }
33   wait();
34 }
```

Listing 4.7: General implementation of an object flow containing a fork node.

```
1  // Declaration of pins:
2  List<Counter> a_c = ...;
3  List<Counter> x_c = ...;
4  List<Counter> y_c = ...;
5
6  class Join1 {
7    public boolean canJoin() {
8      Counter x_cToken;
9      Counter y_cToken;
10
11     for (Counter c: x_c){
12       if (c.value<100){
13         x_cToken = c;
14         break;
15       }
16     }
17     for (Counter c: y_c){
18       if (c.value>10){
19         y_cToken = c;
20         break;
21       }
22     }
23
24     if (x_cToken != null
25     && y_cToken != null){
26       x_c.remove(x_cToken);
27       y_c.remove(y_cToken);
28       a_c.add(x_cToken);
29       a_c.add(y_cToken);
30       return true;
31     }
32     else return false;
33   }
34 }
```

Listing 4.8: General implementation of a join of two object flows.

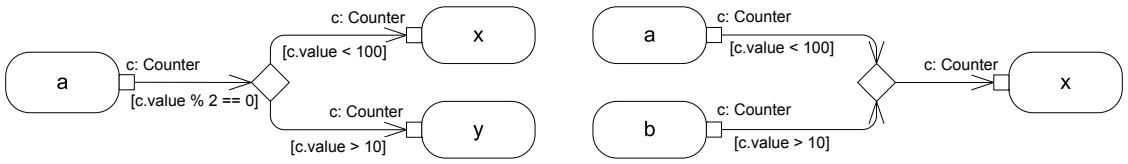
How to join concurrent flows is presented in Listing 4.9 which refers to Figure 4.26(b). An instance of join node (line 1) as well as a thread waiting for the two flows to be joined (line 2) are created upon starting the activity execution. In the case block representing the sequence which starts with action a, a loop implements the waiting until `canJoin()` returns true. After this loop, i. e. when the join actually has happened, a new thread waiting on other object tokens to be joined is created and started (line 16). Afterward, action $a$ is executed by calling method `a(a_c)` where $a\_c$ is the list representing the input pin of $a$ thus providing required input to the action execution.

```
1  Join1 j1 = new Join1();
2  new ActivityThread(<id_j1>).start();
3  ...
4  case <id_x>:
5    ...
6    x_c.addAll(x());
7    id = -1;
8    break;
9  case <id_y>:
10     ...
11   y_c.addAll(y());
12   id = -1;
13   break;
14 case <id_j1>:
15   while(! j1.canJoin()) wait();
16   new ActivityThread(<id_j1>).start();
17   id = <id_a>;
18   break:
19 case <id_a>:
20   a(a_c);
21   ...
```

Listing 4.9: Joining object flows.

### Alternative Object Flows

Figure 4.27 shows alternative object flows. It contains splitting a flow up into two alternative flows (Fig 4.27(a)) and merging independent flows (Fig. 4.27(b)). Listing 4.10 gives an implementation of an alternative branching of an object flow, Listing 4.11 gives an implementation for merging object flows.




(a) Modeling alternative object flows.

(b) Merging alternative object flows.

Figure 4.27: Alternative object flows.

Compared to the implementation of decision node with control flows, when using object flows, the offered object tokens must be placed on the input pin of the target action (Listing 4.10, lines 18 and 24) and removed from the offering output pin (line 30). Since all offered object tokens must be considered just as it is when implementing fork node, guard evaluation, token flow to target pins and resetting the thread id must be nested in a loop which is aborted if the id of the current thread has been changed (checked in line 29). If the id has not been changed, no decision has been taken and the thread must wait for additional tokens or a change of the system state (line 33).

Implementing a merge node, too, becomes more complex when applying object flows. It is no longer just setting the same id at the end of different sequences, i. e. as the last statement of different case blocks. As shown in Listing 4.11, the implementation given in the context of control

flows must be extended to consume tokens from output pins (lines 13 and 35) and place them on downstream input pins (lines 14 and 36) and guard evaluation must consider all object tokens available at the concerning output pin.

Since we consider only one token flowing at a time, a new thread executing the downstream sequence is created for each token to which guard evaluation succeeded (lines 19 and 41). For the last token, the current thread is reused by resetting its id (lines 16 and 38).

```
1  // Declaration of pins
2  List<Counter> a_c = new ...;
3  List<Counter> x_c = new ...;
4  List<Counter> y_c = new ...;
5  ...
6  case <id_a>:
7    ...
8    a_c.addAll(a());
9    while (!a_c.isEmpty()){
10     Iterator<Counter> it =
11       a_c.iterator();
12       Counter c;
13       while (it.hasNext()){
14         c = it.next();
15         if (c.value%2 == 0){
16           if (c.value<100){
17             id = <id_x>;
18             x_c.add(c);
19             break;
20           }
21           else if (c.value>10)
22           {
23             id = <id_y>;
24             y_c.add(c);
25             break;
26           }
27         }
28       }
29       if (id != <id_a>){
30         a_c.remove(c);
31         break;
32       }
33       wait();
34     }
35   }
36   break;
37 case ...
38
39
40
41
42
43
44
45
46
47
48
49
50
```

Listing 4.10: Implementation of an object flow containing a decision node.

```
1  // Declaration of pins
2  List<Counter> a_c = new ...;
3  List<Counter> b_c = new ...;
4  List<Counter> x_c = new ...;
5  ...
6  case <id_a>: ...
7    a_c.addAll(a());
8    id = <id_acont>;
9  case: <id_acont>:
10   while(true){
11     for(Counter c: a_c){
12       if (c.value<100){
13         a_c.remove(c);
14         x_c.add(c);
15         if (a_c.isEmpty())
16           id = <id_x>;
17         else{
18           new ActivityThread(<id_x>).
19             start();
20         }
21         break;
22       }
23     }
24     if (id == <id_acont>) wait();
25     else break;
26   }
27   break;
28 case <id_b>: ...
29   b_c.addAll(b());
30   id = <id_bcont>;
31 case <id_bcont>:
32   while(true){
33     for(Counter c: b_c){
34       if (c.value>10){
35         b_c.remove(c);
36         x_c.add(c);
37         if (b_c.isEmpty())
38           id = <id_x>;
39         else{
40           new ActivityThread(<id_x>).
41             start();
42         }
43         break;
44       }
45     }
46     if (id == <id_bcont>) wait();
47     else break;
48   }
49   break;
50 case ...
```

Listing 4.11: Implementation of a merged object flow.

#### Limitations of Object Flow Implementation

The implementation of object flows as presented here is not fully compliant to the UML specification. First of all, more than one token may flow at a time if the weights of concerned edges do not prevent from multiple objects flowing at the same time.

Furthermore, the presented approach does not consider multiplicity bounds of pins. To include this, guard evaluation must be done for all objects available for a flow and those objects which might flow must be marked, but not yet moved to the target input pin. If they are moved to the target pin before the lower bound of the receiving pin is reached, the moved object is no longer offered to other pins which might be able to consume it.

A merge of object and control flow is possible at join node. Control tokens must be offered together with object tokens, but only the latter ones are offered at outgoing edges.

A problem related to guard evaluation is that after applying propagation rules, referring to the object tokens in guards may result in name clashes. Therefore, naming of output pins must be thoroughly done. Alternatively, keywords could be used to access object tokens, e. g. *object* or *data*, or special characters could denote the object actually flowing, e. g. $.

### 4.2.9 Executing Activities

Listing 4.12 shows the skeleton of an activity implementation.

```
1  class Activity extends Thread{
2    private int id;
3
4    public static void activate(boolean concurrent){
5      if (concurrent) new Activity(0).start();
6      else new Activity(0).run();
7    }
8
9    public Activity(int id){
10     this.id = id;
11   }
12
13   public void run(){
14     while (id >= 0){
15       switch (id){
16         case 0:
17           new Activity(1).start();
18           new Activity(2).start();
19           new Activity(3).start();
20         case 1:
21           // Implement AcceptEventAction without incoming flows.
22           // After receiving a signal, reactivate AcceptEventAction
23           new Activity(1).start();
24           // Execute the downstream actions
25           ...
26         case 2:
27           // Implement AcceptEventAction without incoming flows.
28           // After receiving a signal, reactivate AcceptEventAction
29           new Activity(2).start();
30           // Execute the downstream actions
31           ...
32         case 3:
33           // Implement sequences of actions
34           ...
35       }
36     }
37   }
38 }
```

Listing 4.12: Implementation of an *Activity*.

The activity execution is started by calling the static method `activate(concurrent)`. The parameter `concurrent` determines if the activity is executed concurrently. An execution is created and the initiation sequence implemented in the `case 0` branch is run. In this branch, all sequences starting with an initial node as well as sequences starting with *AcceptEventAction*s (without incoming flows) are started. In the given example, the branches `case 1` and `case 2` are considered to start with such kind of automatically activated *AcceptEventAction*s. The reactivation after a signal receipt is implemented by creating a new thread running the same sequence again.

Since a flow reaching an *ActivityFinalNode* causes the execution to end and consequently, each executing action to be aborted, an activity itself can be seen as an *InterruptibleActivityRegion*. Therefore, in the next section, besides interrupting *InterruptibleActivityRegion*, ending an activity execution is presented.

### 4.2.10 Interruptible Activity Region

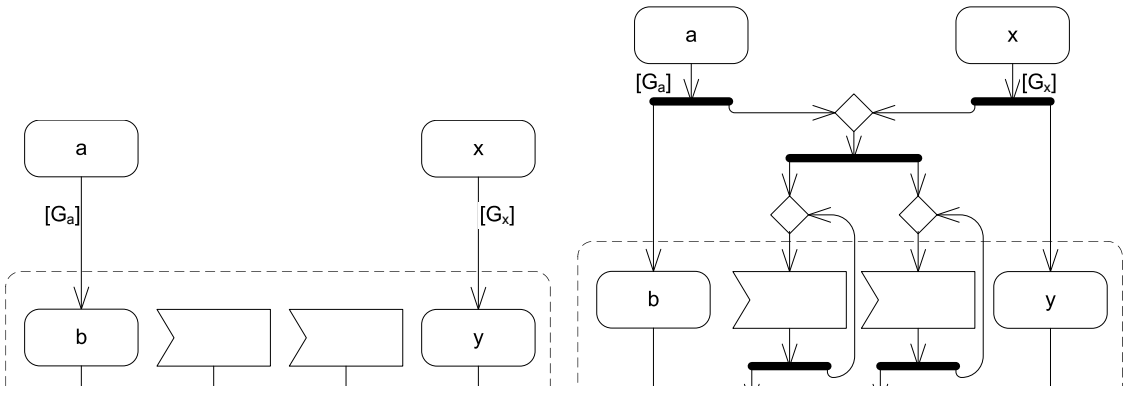
As its name suggests, *InterruptibleActivityRegion* groups flows and actions that are interruptible, i. e. which can be aborted. But yet another important feature is its impact on starting action executions:

> *AcceptEventAction*s in the region that do not have incoming edges are enabled only when a token enters the region, even if the token is not directed at the accept event action. [69, §12.3.33]

This is similar to the situation of *AcceptEventAction*s outside an *InterruptibleActivityRegion* which are activated when the containing activity starts executing.

Figure 4.28 shows an attempt to make the activation of *AcceptEventAction*s inside an *InterruptibleActivityRegion* explicit by a model transformation. Each flow entering the region is forked in order to enable the flow target as well as the *AcceptEventAction*s. Forked flows of all flows entering the regions are merged. This merged flow then is forked in order to enable all *AcceptEventAction*s.

This model transformation may result in multiple action executions for *AcceptEventAction*s inside the region if more than one token enters the region. However, the transformation is still semantically correct since action executions without incoming edges are executed again after a signal receipt. Thus, there is no difference between multiple concurrently executing *AcceptEventAction*s and a single execution: each signal arriving after a token entered the region and before this region is aborted is received.




(a) *InterruptibleActivityRegion* with implicit activation of *AcceptEventAction*s.

(b) *InterruptibleActivityRegion* with explicit activation of *AcceptEventAction*s.

Figure 4.28: Control flow semantics for *AcceptEventAction* without incoming flows inside *InterruptibleActivityRegion*

Listing 4.13 shows that the problem of *AcceptEventAction*s without incoming flows located in an *InterruptibleActivityRegion* can easily be faced at the level of generated code. The activation of such *AcceptEventAction*s is moved from the initiation sequence of the activity into an initialization method of the *InterruptibleActivityRegion*. By means of the boolean flag `init`, it is assured that activation of *AcceptEventAction*s only happens once. When actions inside the region are aborted, `init` must be set to `false` in order to cause the next entering token to activate *AcceptEventAction*s again.

```
1  class IR{
2
3    private boolean init = false;
4
5    public synchronized void init(){
6      if (! init){
7        init = true;
8        new <ActivityName>(1).start();
9        new <ActivityName>(2).start();
10     }
11   }
12   ...
13 }
```

Listing 4.13: Initialisation of *InterruptibleActivityRegion*.

Supporting abortion of executions related to actions inside a region necessitates to recognize entering and exiting flows as well as to distinguish *normal* exiting flows from interrupting ones. Figure 4.29 shows an example of an *InterruptibleActivityRegion* containing two sequences of actions. Since both sequences are mapped to an executing thread, the basic idea of implementation is:

- register threads executing actions contained in an *InterruptibleActivityRegion*
- unregister registered threats when leaving the *InterruptibleActivityRegion*
- if leaving the region via an interrupting edge, interrupt all registered threads.

Listing 4.14 shows an implementation for registering and unregistering threads (lines 3–8) as well as for interrupting threads if the region is interrupted (lines 9–14).

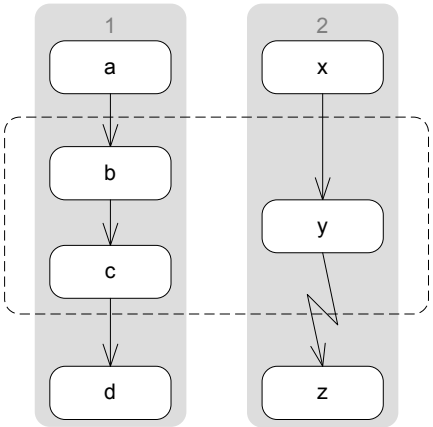



Figure 4.29: *InterruptibleActivityRegion* with an interrupted and an interrupting flow.

```
1  class IR{
2    List<Thread> threads = new ...;
3    public void add(Thread t){
4      threads.add(t);
5    }
6    public void remove(Thread t){
7      threads.remove(t);
8    }
9    public void terminate(){
10     for (Thread t: threads)
11       t.interrupt();
12     for (Thread t: threads)
13       t.join();
14   }
15 }
```

Listing 4.14: Implementation pattern for an *InterruptibleActivityRegion*.

An implementation of the left sequence 1 $(a, b, c, d)$ is given by Listing 4.15. When entering the region, i. e. after executing $a$ and before executing $b$, the current thread adds itself to a list of active threads inside the region (line 2). When leaving the region via a *normal* (i. e. non-interrupting) edge, the thread removes itself from the list as it is no longer affected by abortion of the region (line 13) [39]. Before and after executing an action, it must be tested if the region has been interrupted (lines 4 and 9). For long running actions, suchlike tests should be performed during action execution in order to be able to cooperatively end the action execution if needed.

The implementation of the right sequence 2 $(x, y, z)$ leaving the region via an interrupting edge is given by Listing 4.16. It also contains code for registering and unregistering the thread (lines 2 and 4). For actually aborting all actions inside the region, `ir1.terminate()` is called (line 5). The implementation of that method might either kill all threads, sent a message requesting the threads to terminate, or set a flag which is checked before new actions are executed. One of the two latter options probably is preferable as killing threads — although closer to the specification — is quite risky [39]. The implementation given in Listing 4.14, lines 9–14 notifies all affected threads and waits for their termination before the method call returns.

```
1  case 1: a();
2    ir1.add(this);
3    b();
4    if (isInterrupted){
5      id = -1;
6      break;
7    }
8    c();
9    if (isInterrupted){
10     id = -1;
11     break;
12   }
13   ir1.remove(this)
14   d();
15   ...
```

Listing 4.15: Implementation pattern for an interruptible action sequence.

```
1  case 2: x();
2    ir1.add(this);
3    y();
4    ir1.remove(this);
5    ir1.terminate;
6    z();
7    ...
8
9
10
11
12
13
14
15
```

Listing 4.16: Implementation pattern for an interrupting flow.

A proper implementation of interruptible regions requires, when interrupted, to clear those lists of join nodes which represent edges having their source located inside the region and fork nodes within the region to discard buffered tokens [39].

The implementation of *ActivityFinalNode* can be based on *InterruptibleActivityRegion*. The same mechanism to register active parts of an *InterruptibleActivityRegion* is used to manage active parts of an activity. The implementation of ending an activity then is implemented by considering the whole activity as an *InterruptibleActivityRegion* and each flow to an *ActivtyFinalNode* as an interrupting edge.

### 4.2.11 Preparing Activities by Model Transformations

A model transformation can keep our code generation straightforward by applying three steps:

1. make implicit forks and joins explicit
2. move sequences of actions to separate behaviors and replace them by *CallBehaviorAction*
3. if possible, replace multiple control nodes by a single one [39].

An activity and the result of the transformation is shown in Fig. 4.30. Note that the two object flows between `a1` and `a2` are not handled as an implicit fork and subsequent join since both flows can be implemented as in Fig. 4.5 [39].

The transformation $A^t$ of activity $A$ only contains actions of type *CallBehaviorAction* which implement simple sequences of actions. If $A$ contained a *CallBehaviorAction* (say action `a2`), it was moved to activity $X$. Consequently, each *CallBehaviorAction* of $A^t$ is mapped to an activity containing only a sequence of actions, each *CallBehaviorAction* occurring in a sequence represents a call of another behavior. All control nodes sequencing the sequences of actions remain in $A^t$. Thus, finding sequences is part of the transformation whereas sequencing of sequences is part of the code generation [39].

Since all actions still persist – even though at a different location – mapping actions of $A^t$ to actions of $A$ can easily be done if necessary e. g. for the development of a debugger. In our approach, debugging is done by using our interpreter, but adding debug information to the generated code in principle is possible [39].

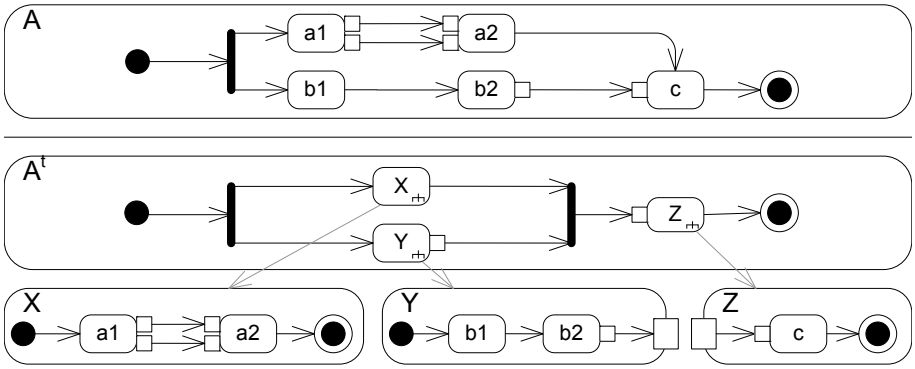



Figure 4.30: Transformation of an activity [39].


**What happens to *InterruptibleActivityRegion*?**

The proposed transformation leads to a model that in one concern is no longer in line with the UML semantics: Actions contained in the same region or activity partition may be moved into different activities, but still the information of being contained in a common container must persist. As can be seen in Fig. 4.29, such a situation occurs if an *InterruptibleActivityRegion* contains distinct sequences of actions. The implementation of all those sequences must refer to the same region (Listing 4.15 and Listing 4.16 both refer to $ir1$).

At the level of the transformed model, it is not possible to include actions that are owned by different activities in the same region — or in other words: one region can not be spread across multiple activities. This problem is fixed by including interruptible activity regions of the same name in each activity containing an action sequence which is affected by an abortion of the region or may cause it. Since all sequences are implemented in the same code class, they refer to the same region by name.

However, interruptible activity regions without any interrupting edge look quite strange, but we do not mind because the transformed model is an intermediate result at a level the user should not work on.

### 4.2.12 Semantical Resolution of Behavioral Concepts at a Glance

In the previous sections of this chapter we have outlined that activities can be divided into a set of sequences of actions. The semantics of such sequences is trivial, since it is the consecutive execution of the contained actions. These sequences of actions are mapped to threads. Thus, concurrency is supported. Guards at edges between actions or output and input pins are considered by suspending threads. *InterruptibleActivityRegion* is supported by registration and if necessary termination of threads affected by interruption of a region. Object flows are covered by the implementation of pins as typed sets.

Apart from a general pattern which partitions an activity into sequences of actions, semantics of control nodes is discussed and implementations are given which are needed to combine threads representing sequences of actions. Guard propagation, reordering of control nodes as well as implementation of complex flows by the user are considered for coping with flows containing multiple control nodes.

An important semantic issue we do not consider is the weight of edges. By specifying a weight, the number of tokens flowing together at the same time can be specified. In our approach, it is assumed that the weight of all edges is 1.

By intelligently combining the different concepts, we successfully implement activities in Java. The result is a kind of framework, in which actions can be executed. Therefore, the next step towards generating code for behavior is to examine the semantics of UML actions and derive implementations for actions. The following section is about this issue.

## 4.3 Actions

*Action* is the fundamental unit of executable functionality. UML's specification contains 37 concrete actions for various purposes, e. g.:

- specializations of *StructuralFeatureAction* support reading and writing of attribute values
- specializations of *LinkAction* provide creation and destruction of links of associations, as well as reading navigable association ends
- *CallOperationAction* causes a behavior which implements the called operation to execute
- *CallBehaviorAction* directly invokes another behavior.

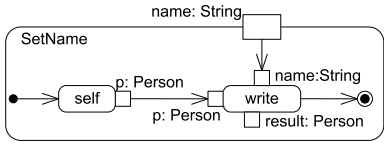
The sequencing of action executions is defined by control and object flows. Each action is executed in its own context, which — according to the specification — is the owner of the behavior containing the action [69]. If an action accesses features of a classifier, those features must be visible to the context object of the action.

We assume that the context of an action should be the context of the behavior containing the action rather than the owner of such a behavior itself, because the owner of a behavior might be another behavior. But the context of an activity is the owning *BehavioredClassifier*, if any, or if an activity is not directly owned by a behaviored classifier, its context is the first behaviored classifier reached by following the chain of ownership; if it is ultimately not owned by a classifier, then the context is the activity instance itself [69, §13.3.2]. By defining the context of an action as the owner of the behavior containing it, the context of an activity and an action contained by this activity might be different objects.

Besides graphical modeling of activities, UML encourages the use of a surface action language which encompasses both primitive actions and the control mechanisms provided by behaviors. A surface action language could introduce higher-level constructs such as e. g. a creation operation with initialization as a single unit as a shorthand for the create action to create an object and further actions to initialize attribute values and create objects for mandatory associations [69, §11.1].

Foundational UML (fUML) [72], a subset of UML, and the Action Language for Foundational UML (Alf) [71] both have been specified by the OMG. Although only very few new constructs have been introduced, using Alf may have advantages compared to traditional graphical modelling.

Fig. 4.31(a) shows an activity for setting the value of an attribute of the context object, Fig. 4.31(b) shows the same activity in Alf. Here, the graphical notation is more complex than a textual, code-like representation. The Quicksort example of the Alf specification [71, pp. 333, 335] contrasts an Alf implementation of 10 lines with a graphical representation consuming a whole A4 page. Another advantage of Alf is that it can be seen which feature is updated. In the graphical notation of UML, this detail is not presented [39].




(a) Graphical UML notation.

```
1 activity SetName(in _name:  String)
2 {
3    this.name = _name;
4 }
```

(b) Implementation in Alf

Figure 4.31: Graphical vs. textual representation of an activity [39].


But regardless of advantages and drawbacks of both, graphical representation and Alf, two problems remain unsolved:

- behavioral modeling in UML is based on very fine grained actions which are not suitable to reach a higher level of abstraction or bringing out a big picture
- the poor tool support of activities is not addressed by introducing another representation of activities [39].

As our approach comprises code generation from behavioral models, implementations of actions are required. Those implementations complete the code generated to implement activities as described in Sect. 4.2. Implementations of activities and actions together build a strong foundation for code generation from behavioral models.

### 4.3.1 Semantics of Actions and Implementations

In this section, we explain the semantics of those actions considered in our approach and give suitable implementations. For each action, we will refer to its purpose, its inputs and outputs, further features if applicable and, if relevant for our approach, its constraints.

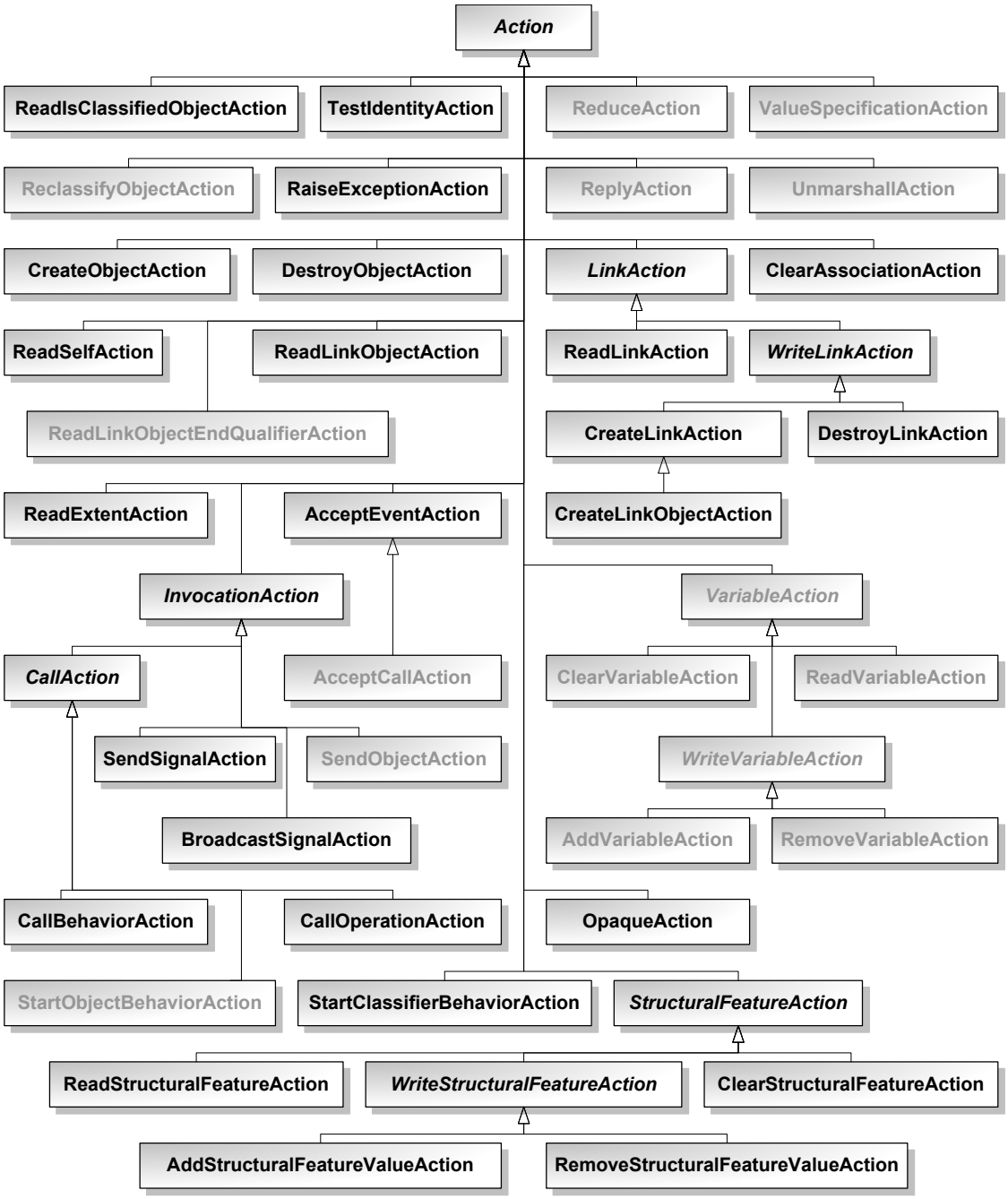



Figure 4.32: UML meta-model of Action.[4]

[4]Actions with names written gray are not considered, according to UML notation, abstract classes are labeled in italics.

Figure 4.32 shows all actions defined in UML and their generalization relationships. The black printed actions are considered in our approach in detail and explained in the following. These actions are related to static structure and therefore are of particular interest for our work. The gray printed actions are included in the figure for giving a survey of all actions available in UML.

At first, we will have a look at actions related to object creation and destruction. Afterward we address specializations of *StructuralFeatureAction*, followed by a discussion of specializations of *LinkAction*. Finally, we will survey actions for communication by signal processing and some other general purpose actions.

When referring to features of an action, say the type of its result, this is indicated by using gray font in angle brackets, e. g. `<result.type>`. If model elements are preprocessed by a macro, the name of the macro is written in green and a reference to the appendix follows the macro call (e. g. $\nearrow^{X.n}$).

### Visibility of Action

*Action* is a specialization of *NamedElement* and inherits the attribute *visibility*, but the semantics of visibility of an action is undefined. Actions are parts of activities and executed when control and object flow prerequisites are satisfied. Thus, the execution of an action is an effect of the execution of an activity which controls the executions of its own actions. Since actions are always owned by the activity in which they are contained, this owning activity can access them regardless of the specified visibility. Another activity cannot trigger the execution of an action, thus another visibility than protected seems useless to us[5]. We consider visibility of actions to be another case where UML designers have been fallen into the overgeneralization trap [91].

### CreateObjectAction

*CreateObjectAction* instantiates a classifier which may neither be abstract, nor an *AssociationClass*. It is associated with the classifier to be instantiated. The created instance is placed on an output pin. The type of the output pin must be the same as that of the created classifier and its multiplicity is 1..1. The action has no other effect like e. g. initialization of structural features. Therefore, it conforms to calling a parameterless constructor in Java. Such a constructor must be provided by all classifiers. An implementation is given by Listing 4.17.

```
<result.type> <result.name> = new <classifier>();
```

Listing 4.17: Implementation of *CreateObjectAction*.

### DestroyObjectAction

*DestroyObjectAction* action supports destruction of objects which are provided to an untyped input pin. Whether links in which the object participates are destroyed is specified by the attribute *isDestroyLinks*. Its default value is false, i. e. links are not deleted.

The semantics of an object destruction without destruction of links referencing this object is undefined. In Java, destroying the links is necessary to make the object accessible to the garbage collector. Other languages may support links referencing destroyed objects, but the impact on multiplicities or return values of actions reading link values is not considered by the specification.

The attribute *isDestroyOwnedObjects* specifies whether objects owned by a composite aggregation are destroyed together with the object. This is probably intended to be a shortcut for removing owned objects before destruction. However, whether to remove all owned objects is valid with respect to specified lower bounds is not considered.

How to properly implement this action depends on the target language. As we refer to Java, we give a strategy for how to let the garbage collector remove this object from the heap in Listing 4.18.

```
<target>.destroy(<isDestroyOwnedObjects>);
```

Listing 4.18: Implementation of *DestroyObjectAction*.

[5]Regarding specialization applied to activities, besides private visibility, protected visibility is feasible.

This implementation just delegates the destruction to the object to be destroyed (Listing 4.19). If owned objects are to be destroyed, too, then `destroy` is invoked for each owned object.

The actual destruction of an object is indirectly achieved by making an object accessible for the garbage collector. This requires that no reference to an object exists. Consequently, efficient access to all references is needed. An easy way to ensure efficient access to all references to objects is to replace attributes to non-primitive types by associations. Then it is possible to find all associations in which the classifier participates of which an instance is to be deleted, to identify links of which the instance to be destroyed is a value, and to remove such links:

$$\forall a \in Association : a.memberEnd.contains(instance.classifier) :$$
$$\forall l \in a, l.contains(instance) :$$
$$a.remove(l)$$

For a convenient implementation in Java, a set of all associations must be provided and links must provide the method `contains(instance)` which returns true, if `instance` is equal to any value of the link. The generation and initialization of such data structures as well as the implementation of such methods should be addressed as part of the code generation. Here, we assume that a mapping

$$Associations : Classifier \rightarrow \mathcal{P}(Association)$$

exists which maps a classifier to all associations in which it participates. The destruction of an object is given in Listing 4.19.

```
public void destroy(boolean destroyOwnedObjects){
  if (destroyOwnedObjects){
    for(Object o: ownedObjects){
      o.destroy(true);
    }
  }
  for(Association a: associations.get(this.class){
    for(Link l: a.getLinks()){
      if (l.contains(this)
        a.remove(l);
    }
  }
}
```

Listing 4.19: Implementation of object destruction by dereferencing.

### ReadSelfAction

By *ReadSelfAction*, the host object of an action is retrieved. The result is provided to the output pin *result*. The method `getContext()` used in the implementation of Listing 4.20 returns the instance which builds the context for the activity execution. The context of an activity is the owning *BehavioredClassifier*, if any. If an activity is not owned by a behaviored classifier, its context is the first behaviored classifier reached by following the chain of ownership. If it is ultimately not owned by a classifier, then the context is the activity instance itself [69, §13.3.2].

In order to be independent from the model at execution time, the context should be passed as a parameter to methods which start activities. Thus, the context is referenced by each activity execution and can easily be retrieved when a *ReadSelfAction* executes.

```
<result.type> <result.name> = this.getContext();
```

Listing 4.20: Implementation of *ReadSelfAction*.

### CallOperationAction

*CallOperationAction* is associated with an *operation*. It transmits an operation call request to the instance placed at its input pin *target*. Parameters of the called operation must be supplied by placing values on input pins. Therefore, number and type of parameters of the action must

conform to those of the called operation. Accordingly, the number and type of output pins of the action must be consistent with parameters of type *return*, *in-out*, and *out* of the operation.

The call may either be synchronous or asynchronous. If synchronous, the caller is blocked and continues execution when the behavior implementing the invoked operation completes and results are placed on the output pins. An implementation is given in Listing 4.21.

```
                                                        if (outputs.isEmpty())
<target>.<operation>(<createParameterValues(inputs)>↗A.2);
                                                                          fi
                                                       if (outputs.size()==1)
<result.type> <result.name> =
  <target>.<operation>(<createParameterValues(inputs)>↗A.2);
                                                                       else
                                                    for i=0..outputs.size()
<outputs[i].type> <outputs[i].name> = new <OutputPinImplementation>();
                                                                        end
<target>.<operation>(<createParameterValues(inputs, outputs)>↗A.3);
                                                                         fi
```

Listing 4.21: Implementation of *CallOperationAction* (*isSynchronous* = true).

The call `createParameterValues(inputs, outputs)` is intended to return a string representation of arguments of the operation call. It is used here in order to omit details of parameter implementations like using a collection type for multi-valued pins. Details are provided in Appendix A.

If the call is asynchronous, the caller continues execution immediately after the call request has been transmitted without waiting for the invoked behavior to complete and without receiving any return values like implemented in Listing 4.22. Nevertheless, the action must have output pins like it is when being marked synchronous even though these pins will never contain any values other than null. This obviously is an error in the constraints defined for this action where existence of output pins and conformance to according parameters of the called operation is not claimed for synchronous calls only.

```
new Thread(){
  public void run(){
    <target>.<operation>(<createParameterValues(inputs)>↗A.2);
  }
}.start();
```

Listing 4.22: Implementation of *CallOperationAction* (*isSynchronous* = false).

### ClearStructuralFeatureAction

*ClearStructuralFeatureAction* is a specialization of *StructuralFeatureAction*. *StructuralFeatureAction* is associated to the structural feature which is to be accessed and provides an input pin from which the object whose feature is to be accessed is obtained.

Additionally, *ClearStructuralFeatureAction* may have an output pin on which the modified object is placed. An implementation for this action is given in Listing 4.23. If the feature's upper bound is greater than 1, its implementation is a list which is to be cleared, otherwise, it is a single-valued attribute whose value is to be set to `null`.

```
                                          if <structuralFeature.upper != 1>
  <object>.<structuralFeature>.clear();
                                                                       else
  <object>.<structuralFeature> = null;
                                                                         fi
                                                        if (result != null)
  <result.type> <result.name> = <object>;
                                                                         fi
```

Listing 4.23: Implementation of *ClearStructuralFeatureAction*.

### ReadStrucutralFeatureAction

In addition to the characteristics inherited from *StructuralFeatureAction*, *ReadStructuralFeatureAction* must provide an output pin on which the result is put. Since the multiplicity of the output pin must be compatible with the multiplicity of the feature, the output pin may support multiple values even when the structural feature is single-valued. Therefore, the implementation given in Listing 4.24 determines whether a single value or an array of values is to be returned by checking the output pin's upper bound.

```
                    if (<result.upper != 1> && <structuralFeature.upper != 1>)
List<<result.type>> <result.name> = new LinkedList<<result.type>>();
<result.name>.addAll(<object.structuralFeature>);
                else if (<result.upper != 1> && <structuralFeature.upper == 1>)
List<<result.type>> <result.name> = new LinkedList<<result.type>>();
<result.name>.add(<object.structuralFeature>);
                                                                        else
<result.type> <result.name> = <object>.<structuralFeature>;
                                                                          fi
```

Listing 4.24: Implementation of *ReadStructuralFeatureAction*.

### AddStructuralFeatureValueAction

*AddStructuralFeatureValueAction* is a specialization of *WriteStructuralFeatureAction* which itself is a *StructuralFeatureAction*. *WriteStructuralFeatureAction* may have an input pin *value* and an output pin *result*. If present, the object obtained from the inherited input pin *object*, as modified, is placed on the output pin.

*AddStructuralFeatureValueAction* requires the optional input pin inherited by *WriteStructuralFeatureAction* and additionally has an input named *insertAt* if (and only if) the structural feature is ordered, which gives the position at which to insert the new value. If the structural feature is ordered and unique and the value to add is already contained in the set of its values, the value is moved to the specified position.

The attribute *isReplaceAll*, which by default is false, specifies whether or not the structural feature should be cleared before inserting a new value. If the value to add already exists, then the structural feature is not cleared. An implementation considering the characteristics of the structural feature as well as *isReplaceAll* is shown in Listing 4.25.

```
                                                          if (<isReplaceAll>)
if(!<object>.<structuralFeature>.contains(<value>))
  <object>.<structuralFeature>.clear();
                                                                            fi
           if (<structuralFeature.isUnique> && !<structuralFeature.isOrdered>)
if(!<object>.<structuralFeature>.contains(<value>))
  <object>.<structuralFeature>.add(<value>);
       else if (<structuralFeature.isUnique> && <structuralFeature.isOrdered>)
if(<object>.<structuralFeature>.conatins(<value>))
  <object>.<structuralFeature>.remove(<value>);
<object>.<structuralFeature>.add(<insertAt>, <value>);
      else if (<!structuralFeature.isUnique> && <structuralFeature.isrOdered>)
<object>.<structuralFeature>.add(<insertAt>, <value>);
                                                                        else
<object>.<structuralFeature>.add(<value>);
                                                                          fi
<output> = this;
```

Listing 4.25: Implementation of *AddStructuralFeatureValueAction*.

### RemoveStructuralFeatureValueAction

*RemoveStructuralFeatureValueAction*, too, is a *WriteStructuralFeatureAction*.

The boolean attribute *isRemoveDuplicates* specifies whether to remove duplicates of the value in non-unique structural features. If a non-unique structural feature is ordered and *isRemoveDuplicates* is false, then the action must have the input pin *removeAt* in order to allow for specifying the position of the value to remove. Otherwise, the value to be removed is obtained by the inherited input pin *value* and the action must not have a *removeAt* input pin. Listing 4.26 implements this action.

```
                                                    if (<isRemoveDuplicates>)
while(<object>.<structuralFeature>.contains(<value>))
  <object>.<structuralFeature>.remove(<value>);
      else if (!<structuralFeature.isUnique> && <structuralFeature.isOrdered>)
<object>.<structuralFeature>.remove(<removeAt>);
                                                                        else
<object>.<structuralFeature>.remove(<value>);
                                                                          fi
```

Listing 4.26: Implementation of *RemoveStructuralFeatureValueAction*.

### LinkAction

*"LinkAction is an abstract class for all kind of link actions that identify their links by the objects at the ends of the links"* [69, §11.3.21]. Figure 4.33 shows an excerpt of the UML abstract syntax from which the mechanism of identifying and accessing links can be seen.

A link action owns two or more instances of *LinkEndData* called *endData*. Each of these *endData* is associated with a property representing an association end and an input pin. That way, input pins are mapped to association ends. The input of a link action consists of the input pins associated to its *endData*. For identifying links, objects which are placed on the input pins of the link action are used. If all *endData* are associated with an input pin, all ends are supplied with a specific value and a single link can be identified. If one *endData* element has no input pin, the end to which this element is associated is called the *open end*.

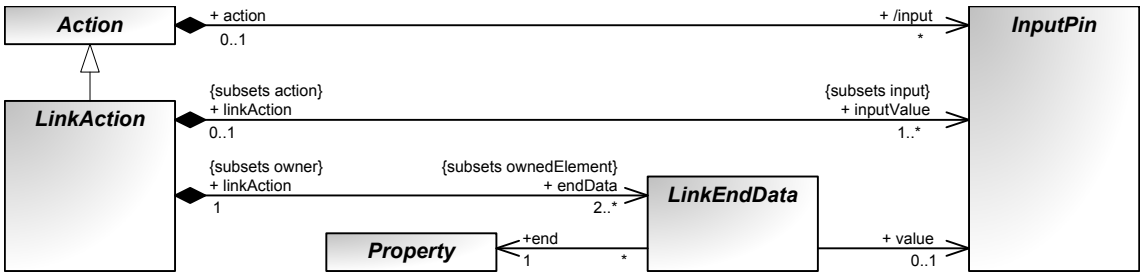



Figure 4.33: Abstract syntax of *LinkAction*.

### ReadLinkAction

*ReadLinkAction* reads the values of a navigable end of an association. It is a specialization of *LinkAction*. The end to be read is the open end, i. e. the *endData* element associated to this end has no input pin. Thus, no value for this end can be provided to the action.

The input of *ReadLinkAction* identifies a subset of all existing links of an association, such that the values for the specified ends of the link are the objects provided as input. The values of the open end of these identified links are the output of the action.

Listing 4.27 gives an implementation. The association concerned by the action is identified by the first *endData* element. Since the end of each *endData* element must belong to the same association, any *endData* element can arbitrarily be chosen for this purpose. The item *openEnd* denotes the *endData* element that is not associated with an input pin. The arguments passed to the `get` method of the association are provided by the macro `createParameterValues` (see Appendix A) operating on the set of *endData* except for the *openEnd*.

In Chapter 3, Sect. 3.6, an implementation of association is provided. This implementation offers methods to retrieve associated instances of a given context. *ReadLinkAction* can directly use these methods, if the method parameters and input pins are in the same order, e. g. alphabetically sorted by their names.

```
<result.type> <result.name> =
    <endData[0].end.association>.get<openEnd.name>(
      <createParameterValues(endData \openEnd)>↗A.2);
```

Listing 4.27: Implementation of *ReadLinkAction*.

### CreateLinkAction

This action is a *WriteLinkAction* which itself is a *LinkAction*. *EndData* of a *WriteLinkAction* must not have an open end, i. e. all *LinkEndData* elements must be associated to an input pin.

*EndData* is redefined in *CreeateLinkAction* so that its type is *LinkEndCreationData*, which is a specialization of *LinkEndData*. *LinkEndCreationData* adds the boolean attribute *isReplaceAll*. If the association end to which a link end creation data refers to is ordered, *LinkEndCreationData* provides an additional input pin for specifying the position where to insert the new link. An implementation for *CreateLinkAction* basing on the association implementation of Chapter 3, Sect. 3.6 is given in Listing 4.28. For unordered association ends, the parameter specifying the position for insertion of the new link is obsolete.

A correct consideration of the attribute *isReplaceAll* requires that the association provides additional methods for each end. These methods must delete all links in which a given instance participates, regardless of values at other ends. An implementation is given in Listing 4.29. Before creating a new link, these methods are called for association ends according to the specification of *isReplaceAll* of associated *endData*. The macro `createParameterValues` must provide values for link ends and positions where to insert values for ordered association ends.

```
                                                    for i=0..<endData.size>
                                                  if (<endData[i].isReplaceAll>)
<endData[i].end.association>.destroyAll(<endData[i].value>);
                                                                        fi end
<endData[0].end.association>.createLink(<createParameterValues(endData)>↗A.2);
```

Listing 4.28: Implementation of *CreateLinkAction*.

```
1 /* list of links of the association */
2 private List<Link> tuples = new LinkedList<Link>();
3
4 public void destroyAll(A a){
5   for(Link l : tuples){
6     if(l.a == a)
7       tuples.remove(l);
8   }
9 }
```

Listing 4.29: Implementation of *isReplaceAll*.

### CreateLinkObjectAction

A specialization of *CreateLinkAction* operating on association classes is *CreateLinkObjectAction*. The additional semantics provided by this action is that the link object is placed on the output pin *result*. Accordingly, the implementation given in Listing 4.30 is similar to that of create link (Listing 4.28). Considerations about *isReplaceAll* and ordered association ends are applicable in the same way as they are for *CreateLinkAction*. If a link already exists and no duplicates are allowed, than the action returns the existing link as result, if duplicates are allowed, lines 3 to 7 of Listing 4.31 are to be removed.

```
<output.type> <output.name> =
  <endData[0].end.association>.createLinkObject(
    <createParameterValues(endData)>↗A.2);
```

Listing 4.30: Implementation of *CreateLinkObjectAction*.

```
1  public Link createLinkObject(A a, B b, ...){
2    if (a != null && b != null && ...){
3      for (Link link : tuples){
4        if (link.a == a && link.b == b && ...){
5          return link;
6        }
7      }
8      Link l = new Link(a, b, ...);
9      tuples.add(l);
10     return l;
11   }
12   return null;
13 }
```

Listing 4.31: Adapted implementation of link creation for link object creation.

### ReadLinkObjectEndAction

*ReadLinkObjectEndAction* is another action operating on *AssociationClass*, which is both kind of an association as well as kind of a class. The features of an association class inherited by *Class* can be accessed as if the link object was a *normal* instance of a class. *ReadLinkObjectEndAction* is an action that provides access to objects via the ends of a link object. Thus, it is possible to access the objects which are related by a specific link object, however, end objects cannot access the link object but rather objects of opposite ends.

Inputs of *ReadLinkObjectEndAction* are the link object and the end to be read, the output is the end object. Listing 4.32 gives an implementation for this kind of action.

```
<result.type> <result.name> = <object>.get<end.name>();
```

Listing 4.32: Implementation of *ReadLinkObjectEndAction*.

### DestroyLinkAction

*DestroyLinkAction* is a *WriteLinkAction* that destroys a specified link. For the identification of the link to be destroyed, instances of *LinkEndDestructionData*, which is a specialization of *LinkEndData*, are used to specify the *endData*. *LinkEndDestructionData* supports to define whether duplicates are to be destroyed by means of the boolean attribute *isDestroyDuplicates*. Furthermore, for ordered non-unique ends, an input pin named *destroyAt* is used to specify the link by its position rather than by its end value.

Listing 4.33 gives an implementation which makes use of the link management provided by association implementation. From the action inputs, parameters for calling the method `destroyLink(...)` are obtained.

The actual implementation for deleting links identified by their end values is given by Listing 4.34. The deletion of links for non-unique, ordered associations is given by Listing 4.35.

```
<endData[0]>.<end>.<association>.destroyLink(
  <createParametersValues(input), isDestroyDuplicates>)↗A.2;
```

Listing 4.33: Implementation of *DestroyLinkAction*.

```
public void destroyLink(A a, ..., N n, boolean destroyDuplicates)
{
  List<Link> toRemove = new LinkedList<Link>();
  for (Link link : tuples_A)
    if (link.a == a && ... && link.n == n){
      toRemove.add(link);
      if (!destroyDuplicates)
        break;
    }
  }
  for (Link link : toRemove)
    tuples_A.remove(link);
    ...
    tuples_N.remove(link);
  }
}
```

Listing 4.34: Implementation of `destroyLink` in an implementation class of an association.

```
public void destroyLink(int posA, ..., int posN)
{
    tuples_a.remove(posA);
    ...
    tuples_N.remove(posN);
}
```

Listing 4.35: Implementation of `destroyLink` in an implementation class of an association for an ordered, non-unique association.

### ClearAssociationAction

This action destroys all links of a statically-defined association in which the object placed on the input pin at runtime participates. Since the action has no parameter to specify the end at which the object participates in the link, it is not possible to clear a reflexive association in a way that only links are destroyed in which the input object participates in a special role.

The implementation of this action is given by Listing 4.36. It calls a method that should be implemented in the scope of association implementation, as shown in Listing 4.37.

```
<association>.clear(<object>);
```

Listing 4.36: Implementation of *ClearAssociationAction*.

```
public void clear(Object value){
  List<Link> toRemove = new LinkedList<Link>();
  for (Link link : tuples){
    if (link.a == value || ... || link.n == value){
      toRemove.add(link);
    }
  }
  for (Link link : toRemove)
    tuples.remove(link);
  }
}
```

Listing 4.37: Implementation of `Clear`.

### SendSignalAction

A *SendSignalAction* sends a signal to a target object, which is placed on an input pin. The type of the signal is statically defined. The action may have additional inputs. The signal to be send is created by the action from its inputs.

At the target object, a *SignalEvent* occurs. The occurrence of such an event may have an immediate effect or it may be saved and trigger an effect later, i. e. by being processed by an *AcceptEventAction*. In either case, it is suitable to first store the occurrence of an event in the input event pool of the target object. If it can be processed, it is removed from the input event pool and associated data, e. g. the signal instance, are made available to the behavior consuming the event.

The event pool is part of the target object and therefore, adding an event to the pool is in the responsibility of the target object as well.

The implementation of *SendSignalAction* is given by Listing 4.38. It consists of the creation of the signal instance and of invoking method `receiveSignal(signal)` of the target object.

```
<signal.type> <signal.name> = new <signal.type>(
  <createParameterValues(inputs)>↗A.2);
<target>.receiveSignal(<signal.name>);
```

Listing 4.38: Implementation of *SendSignalAction*.

The implementation of method `receiveSignal(Signal s)` is given by Listing 4.39. The signal instance is added to the event pool of the target object. This makes the signal available for *AcceptEventActions*.

```
EventPool eventPool = new EventPool();

public void receiveSignal(Signal s){
  eventPool.add(s);
}
```

Listing 4.39: Implementation of signal receipt.

### AcceptEventAction

An *AcceptEventAction* is associated to at least one *Trigger*, which is associated to an *Event*. We only consider triggers which are associated with a *SignalEvent*. This defines for each *AcceptEventAction* a set of signals whose occurrence the action waits for.

The action may have outputs which either hold the received signal instance or its attributes, if the boolean attribute *isUnmarshall* is true. We only support the action if *isUnmarshall* is false.

The semantics of *AcceptEventAction* is primarily covered by the implementation of the event pool which we designed to intelligently dispatch signal instances to the correct action executions. The implementation of *AcceptEventAction* as given by Listing 4.40 and comprises only a call of the event pool which returns a signal instance. The types of events of triggers associated with the accept event action are passed as parameters to the event pool.

```
<result.type> <result.name> = eventPool.dispatch(<triggers.event>);
```

Listing 4.40: Implementation of *AcceptEventAction*.

### Implementing an Input Event Pool

For the implementation of sending and receiving signals, an input event pool is used for which an implementation is given in Listing 4.41. Objects serving as the context for accept event actions need to have an event pool and must provide a method for receiving signals (Listing 4.41, lines 4–9) and storing signals within their event pool. Signals passed to the event pool for this purpose are internally stored until they are consumed by an *AcceptEventAction*.

For convenience, activities might have direct access to the event pool of their context object instead of calling a method which just passes the signal event through.

When an *AcceptEventAction* is executed, the method `dispatch(signals)` is called. Within this method, the complete event pool is searched for a signal conforming to a signal type contained in the list passed as a parameter (lines 15 and 26–35). If no such signal is found, the thread waits (line 18) until it is notified after a new signal has been received (line 7). If such a signal is found (line 28), it is removed from the event pool (line 29) and returned to the *AcceptEventAction* (lines 30 and 23).

```
1  public class EventPool {
2    private List<Signal> events = new LinkedList<Signal>();
3
4    public void receive(Signal s){
5      synchronized (events) {
6        events.add(s);
7        events.notifyAll();
8      }
9    }
10
11   public Signal dispatch(List<Class<? extends Signal>> signals){
12     Signal s = null;
13     while (s == null){
14       synchronized (events) {
15         s = find(signals);
16         if (s == null){
17           try{
18             events.wait();
19           }catch (InterruptedException e){}
20         }
21       }
22     }
23     return s;
24   }
25
26   private Signal find(List<Class<? extends Signal>> signals){
27     for (Signal s: events){
28       if (signals.contains(s.getClass())){
29         events.remove(s);
30         return s;
31       }
32     }
33     return null;
34   }
35 }
```

Listing 4.41: Implementation of *AcceptEventAction*.

### BroadcastSignalAction

*BroadcastSignalAction* can be implemented as an iterative execution of *SendSignalAction* for each target object. The set of all objects which are potential targets is the result of *ReadExtentAction*, if it is executed for each classifier of the model. An implementation is given in Listing 4.42.

```
                                    for (Classifier c:  <Model.classifiers>)
                                          for (instance i:  c.readExtent())
i.receiveSignal(<signal>);
                                                                    end end
```

Listing 4.42: Implementation of *BroadcastSignalAction*.

In the implementation of *BroadcastSignalAction*, let *Model* be a class representing model characteristics needed at runtime, e. g. a list of all classifiers specified in a model.

### ReadExtentAction

*ReadExtentAction* returns all instances of a classifier placed at the input pin. In order to support this action, each classifier must provide a method returning all instances. Listing 4.43 gives an implementation of the action, Listing 4.44 gives an implementation for the method `readExtent()`. Furthermore, it shows how instances are managed in a static list of weak references in order to prevent a memory leak (line 3). Upon creation, each instance is added to the extent (line 6). When the extent is read, references of objects that have been removed by the garbage collector are removed from the extent (line 14). Objects reachable through the weak references are put into the result set (line 16). As a precaution, the garbage collector could be called before reading the extent to assure that no outdated values are contained in the result.

```
<output> = <classifier.name>.readExtent();
```

Listing 4.43: Implementation of *ReadExtentAction*.

```
1  public class <classifier.name> {
2    private static List<WeakReference<<classifier.name>>> extent =
3      new LinkedList<WeakReference<<classifier.name>>>();
4    // Constructor
5    <classifier.name>(){
6      extent.add(new WeakReference(this));
7    }
8
9    public List<<classifier.name>> readExtent(){
10     List<<classifier.name>> result =
11       new LinkedList<<classifier.name>>();
12     for(WeakReference<<classifier.name>> ref: extent){
13       if (ref.get() == null)
14         extent.remove(ref);
15       else
16         result.add(ref.get());
17     }
18     return result;
19   }
20 }
```

Listing 4.44: Implementation of `readExtent`.

### OpaqueAction

*OpaqueAction* is an action the semantics of which is not predefined by UML. It may have input and output pins, a *body* consisting of non-unique ordered strings specifying the semantics in one or more languages. For each string contained in *body*, the used language is specified in the same order as the body strings. An implementation is given in Listing 4.45.

```
                                                    for i = 0..<output.length>
List<<output[i].type>> <output[i].name>=new LinkedList<<output[i].type>>();
                                                                        end
public void <name>(<createInputParameters(input)>↗A.1){
                                                      for i = 0..<body.length>
                                                  if (<language[i]>.equals("java"))
    <body[i]>
                                                                      fi end
}
```

Listing 4.45: Implementation of *OpaqueAction*.

### RaiseExceptionAction

*RaiseExceptionAction* accepts an object on the input pin *exception*. The execution of the action effects this input object to be thrown as an exception. Listing 4.46 contains the line of code needed for implementation.

```
throw <exception>;
```

Listing 4.46: Implementation of *RaiseExceptionAction*.

### StartClassifierBehaviorAction

This action supports the dedicated starting of a classifier behavior. A classifier behavior is a behavior specification of a *BehavioredClassifier* — which is a specialization of *Classifier* — that describes the behavior of the classifier itself. *StartClassifierBehaviorAction "is provided to permit the explicit initiation of classifier behaviors, such as state machines and code, in a detailed, low-level* **raw** *specification of behavior"* [69, §11.3.46].

Considering the facts, that *"when an instance of a behaviored classifier is created, its classifier behavior is invoked"* [69, §13.3.4] and that *"if the behavior has already been initiated, or the object has no classifier behavior this action*[6] *has no effect"* [69, §11.3.46], it is not possible to imagine a context where to use this action. Possibly UML misses to define any means of deferring the invocation of classifier behavior as an effect of creating instances of behaviored classifiers.

Despite the question in which contexts, if at all, *StartClassifierBehaviorAction* is applicable, Listing 4.47 gives an implementation of starting the classifier behavior of an object: the method `startBehavior()` is invoked on the object which is placed on the input pin *object* of *StartClassifierBehaviorAction*. Since the action does not accept additional inputs, the classifier behavior cannot be supplied with parameter values. The same applies to output values.

An implementation of `startBehavior()` must be provided by each classifier, even if it does not own a behavior. In this case, the method implementation is empty as the action has no effect. If the classifier has a behavior which has already been started, the action has no effect, too. A suitable implementation is given by Listing 4.48. By checking (line 5) and setting (line 6) the boolean flag `isStarted`, only the first execution of the action has the desired effect. In order to prevent race conditions between concurrent action executions, the method `startBehavior()` is synchronized. The actual creation of an activity execution is achieved by the activation of the behavior in line 7 which refers to the activity implementation as proposed in Listing 4.12.

```
<object>.startBehavior();
```

Listing 4.47: Implementation of *StartClassifierBehaviorAction*.

```
1 private boolean isStarted = false;
2 private Behavior classifierBehavior = new <classifierBehavior.name>();
3
4 <classifierBehavior.visibility> synchronized void startBehavior(){
5   if (!isStarted){
6     isStarted = true;
7     classifierBehavior.activate();
8   }
9 }
```

Listing 4.48: Implementation of starting an owned behavior.

[6] StartClassifierBehaviorAction

#### CallBehaviorAction

A call behavior action directly calls another behavior. Whereas behavior invoked via *CallOperationAction* can be assigned to the context object of the invoked behavioral feature, identifying the context of a directly invoked behavior is problematic: the association between behavior and classifier is at the level of classes rather on the level of instances. Therefore, the context might be defined to be the same object that serves as the context of the behavior owning the calling action. Apart from this option, the context can be defined by using an *ActivityPartition* which refers to an instance: Behaviors of invocations contained by such a partition are the responsibility of the particular instance of the classifier represented by the partition.

Listing 4.49 gives an implementation. It considers both of the above described options regarding the context. In either cases, a context object is passed as a parameter to the creation of an activity execution. This context object either is the classifier represented by an activity group (line 4) or the context object of the invoking execution (line 6).

Another feature of *CallBehaviorAction* inherited from *CallAction* is `isSynchronous`. If false, the action execution ends immediately after starting the associated behavior, return values are discarded. If true, the action execution does not end before the invoked behavior execution ends. Results of the called behavior are placed at output pins of the action. In Listing 4.49, method `getResult()` is considered to return the outputs of the behavior. These values must be placed on the according output pins of the action.

```
1  <behavior.return.type> result = null;
2                                  if <action.owner> instanceOf ActivityGroup
3  <behavior.type> execution = new <behavior.type>(
4        <action.owner.represents>);
5                                                                        else
6  <behavior.type> execution = new <behavior.type>(this.getContext());
7                                                                          fi
8  execution.start(<createParameterValues(inputs)>↗A.2);
9                                                          if(isSynchronous)
10 {
11   execution.join();
12   <behavior.return.name> = execution.getResult();
13 }
14                                                                         fi
```

Listing 4.49: Implementation of *CallBehaviorAction*.

### 4.3.2 Action Execution and Static Structure

The precondition for executing an action is the satisfaction of all control and object flow prerequisites, i. e. object tokens must be available at all input pins and control tokens must be offered to all incoming control flow edges.

Postconditions of an action execution are token offers at outgoing control flows, object tokens created at output pins and side effects caused by the semantics of the action. If these side effects violate the specification of the structure, action semantics is either not specified or in the case of violating lower bounds, action semantics is defined to cause an invalid system state.

Since we generate code for accessing structural features in the context of code generation for static structure, implementation of concerning actions is achieved by mapping these actions to `set`, `get`, `add`, and `remove` methods. These methods then are called when an *Action* associated with the corresponding structural feature is executed. By doing so, we are not compliant to UML since our access methods do not violate constraints given by structural modeling but throw exceptions. Besides omitting side effects in favor of a valid static structure, throwing exceptions contradicts the UML specification since exceptions cannot be raised by an action execution unless the executed action is a *RaiseExceptionAction*.

For our implementations to be valid, UML were to be changed. But before arguing for such changes in the specification of the modeling language, we motivate this claim by shortly discussing how our full featured code is obtainable by adequate modeling.

## 4.4 Considering Constraints Defined by Static Structures

Generally, there are two ways of maintaining a valid system state:

- either check the current state before attempting any changes or
- try changes and recover in the case of failure.

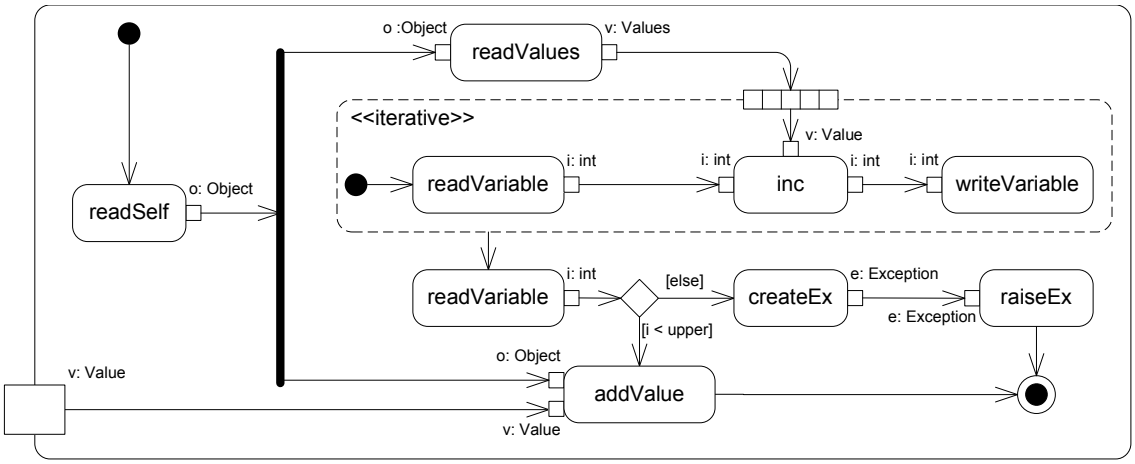
Since failure is not an option designated by UML semantics, both alternatives require to examine the system state and to figure out the permissibility of the desired side effects to the system. According to the set of supported actions, in our approach multiplicity bounds may be violated directly by actions or indirectly as a consequence of object destruction or by cascading destruction if composite aggregation is applied.

### 4.4.1 Modeling Multiplicity Bounds Checks

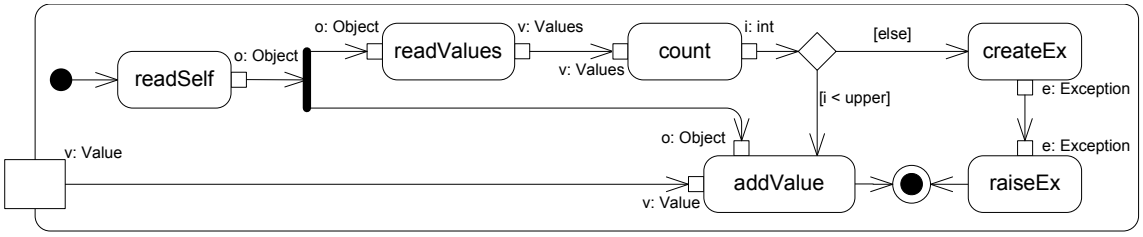
Checking bounds of multiplicities may be explicitly modeled, but resulting activities are not trivial as can be seen in Fig. 4.34(a). Instead of using structured nodes, *OpaqueAction* can be used to count the number of values of an attribute or association end in order to compare it to the specified bounds (Fig. 4.34(b)). The opaque action *count* returns the size of the input set provided at pin $v$.

Both examples of Fig. 4.34 raise an exception if bounds are violated. With little changes, these examples can be adapted to test for multiplicity bounds before attempting to add values and return a boolean result, thus supporting a previous test instead of causing an error requiring the retroactive recovery of a valid system state.

Although checking bounds can be included in behavioral models, we believe that the specification of actions should completely ensure the integrity of the underlying structural model. Multiplicity bounds are part of the structural specification of a system and there is no reason to discard this information. Modeling multiplicity bound checks adds significant modeling overhead and does not seem feasible since such checks must be modeled separately for each kind of access and for each feature.




(a) Counting values of a structural feature by means of *StructuredActivityNode*.




(b) Counting values of a structural feature by applying *OpaqueAction*.

Figure 4.34: Modeling checks of multiplicity bounds.

Generation of activities like those of Fig. 4.34 certainly is an issue of tooling. We discuss the inclusion of the generated parts of behavioral models based on information obtained from structural specifications in Chapter 7.

### 4.4.2 Avoiding Structural Feature Action and Link Action

Another, even more rigorous way to consider multiplicity bounds when accessing structural features is to provide an own implementation based on *OpaqueBehavior*, *OpaqueAction*, or *Operation* together with *CallOperationAction*. Since in our approach, methods for accessing features are generated as a part of the implementation of structures, calling these methods when *StructuralFeatureActions* are to be executed seems likely.

Irrespective of the way by which multiplicities are considered, the impact on activities is that each occurrence of *StructuralFeatureAction* has to be replaced by the provided action or behavior. Avoiding the use of *StructuralFeatureAction* on the one hand gives rise to better aligning semantics of behavior to the underlying static structure, on the other hand, it entails fundamental changes since now exceptions may occur where this has not been possible before.

In order to hide effects of replacing predefined UML actions by special, user defined or generated accession behaviors, exception handling as well as other kinds of handling invalid system states must be covered in the scope of the replacing modeling elements. For a more detailed discussion, we again refer to Chapter 7.

## 4.5 A More Convenient Semantics for UML Actions

In spite of the presented options for considering multiplicity bounds in behavioral models, a more convenient specification of the semantics of actions is justifiable. Certainly we do not expect too much of UML if we demand a higher level of abstraction from it. However, what we actually get is lower level of abstraction than typically provided by programming languages, many of which check correctness of accessing user defined structures such as arrays.

UML leaves the semantics of violations of constraints specified by structural modeling undefined. Consequently, complying with the constraints of structural models is in the responsibility of developers. Modeling, in particular consistently maintaining structural and behavioral models becomes laborious. But actually, using UML should make lives of software engineers easier. For this reason, semantics should be based on the needs of developers, i. e. behavior should not override structural constraints. Hence, we propose to introduce some additional features for actions.

### 4.5.1 Reporting about Success by Outputs

An optional boolean output would qualify *StructuralFeatureAction* to provide information about success or failure of accessing a structural feature. How to advance in case of failure remains in the responsibility of the modeler, but detecting necessity of recovering to a valid system state is supported by the system, thus reducing modelers' workloads.

Another way basically targeting the same idea of conveniently identifying invalid system states would be to introduce a new *Action*, say *TestConsistencyAction*, which supports checking consistency of actual values with respect to the specification of the concerned structural feature.

### 4.5.2 Reporting about Failures by Exceptions

We would prefer to change the specification of exceptions in UML. Only one action can raise an exception and the exception instance is to be passed to this action as an input. By that, the modeler can decide to indicate a failure during the execution of a behavior, but not a failure of an action. In other words, actions may report about success of their execution by output values, but making those output available to callers requires an explicit raising as an exception of it. Since modeling should make software specification more abstract, it would be of advantage to have some exception types predefined in UML with actions having meta-attributes to determine which kind of failures to report. In addition to exceptions reporting violations of upper or lower multiplicity bounds, it would be valuable to report an attempt to add existing values to features that have been defined as *unique*, or to remove values that are not included in the set of values of a feature.

Instead of using predefined exception types, these types could be assigned by the modeler by associating an action to the type of which an instance shall be raised as an exception.

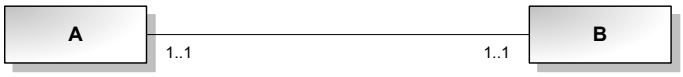
In either cases, failure during action execution could be processed by exception handlers or the exception propagation mechanism of UML would direct exceptions to callers of behaviors or operations.

### 4.5.3 Facing Temporarily Invalid System States

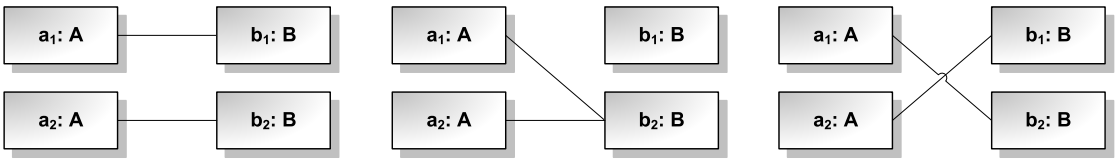
By the proposed means of coping with invalid system states, a particular case has not yet been considered, namely the intended temporary violation of multiplicity bounds. Violations of multiplicities sometimes are inevitable, e. g. when creating instances which must be linked with other instances.

*CreateObjectAction* creates an instance, but initializing it with required attribute values or links for associations is to be done by subsequent actions. Thus, immediately after the execution of a *CreateObjectAction* terminates, the system state might be invalid.

Another situation requiring to accept a temporarily invalid system state has already been mentioned in terms of discussing implementation of multiplicity bounds in Sect. 3.5.2. We once again refer to the example described by Génova et al. [30] and illustrated in Fig. 4.35.




(a) Class diagram with a 1-to-1 association.




(b) Initial system state.

(c) Intermediate wrong system state.

(d) Desired valid final system state.

Figure 4.35: Consequence of the equality of lower and upper bounds of multiplicities, based on [30]

Since a temporarily violation of the static structure sometimes is unavoidable, as can be seen in Fig. 4.35, UML should provide any means to cope with this problem.

Regions could be applied to introduce *transactions*. Figure 4.36 gives an example how the situation shown in Fig. 4.35 could be modeled with transactions. Inside the region, violations of

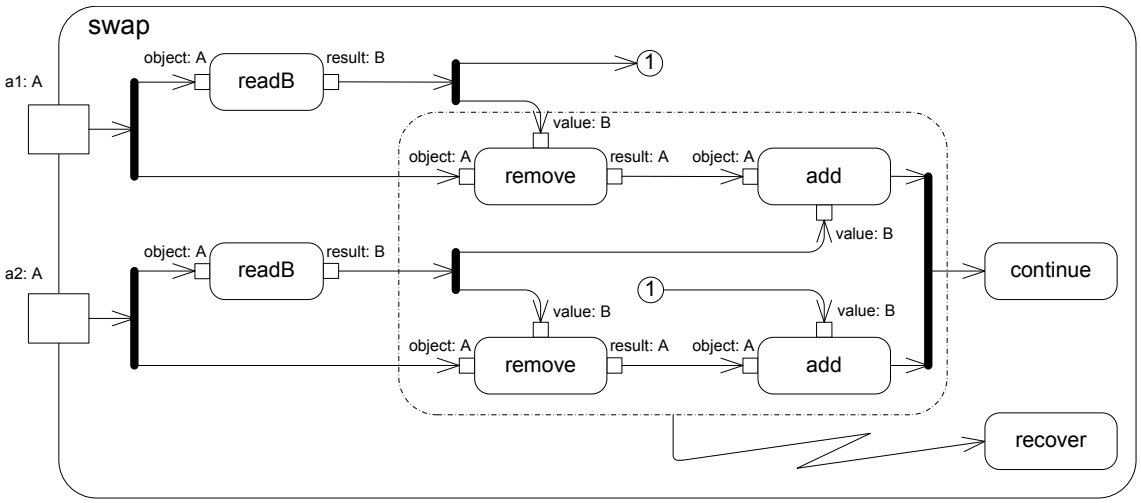



Figure 4.36: Proposal for transactions in UML activities.

structural constraints are permitted. When leaving the region, all constraints are checked. If the check is successful, the activity execution continues at action *continue*, if it fails, the region offers a token to the arrow edge. Thus, execution continues at action *recover*.

## 4.6 Summary

In this chapter, we introduced the key concepts of modeling dynamic behavior in UML, which is based on Action. The execution of actions is specified by modeling behaviors, in particular activities.

In the first part of this chapter, an approach how to translate activities into code is given with a detailed examination of semantics of flows and control nodes. The basic idea of translating activities into code is to identify sequences of actions which are mapped to sequences of method calls. Sequences of actions are sets of actions, which are connected by unbranched control or object flows.

Consideration of control nodes requires to evaluate guards in case of alternative branches and to implement concurrently executing threads in case of parallel branches. If guards are applied to outgoing edges of fork nodes, the special semantics of fork node buffering must be considered. Guards applied to flows which are not alternative branches require to pause execution of the concerned sequence of action while such guards do not hold.

For control nodes, implementations are given which implement the decision which downstream sequences to start next. If multiple control nodes occur between actions, flows and control nodes may be reordered or replaced in order to simplify the model while preserving its execution semantics. For this purpose, we introduce guard propagation and provide implementations for a couple of control flow patterns.

Basic considerations of control flow implementations are advanced to implement object flows as well. Furthermore, termination of action executions and flows by leaving interruptible activity regions via an interrupting edge are included in our approach.

In the second part of this chapter, the semantics of actions is examined and implementations in Java are sketched. The proposed implementations can complete the implementation generated when translating an activity into code.

Actions accessing the structural features are not designed to consider constraints of static structure, such as the size of sets of values for attributes or association ends. In order to align semantics of structural and behavioral models, techniques for modeling additional functionality like multiplicity bounds checking are presented and inclusion into tool support is considered. Furthermore, adaptations to the UML semantics specification are proposed.

In the next chapter, the modifications to input models discussed in Chapter 3 and Chapter 4 are formally specified.

# Chapter 5

# Model Transformation

In the previous sections we described the semantics of structural and behavioral modeling elements of UML, presented implementations based on the Java programming language, and discussed model transformations which reduce the level of abstraction by resolving abstraction patterns.

In this chapter, these model transformations are formalized by providing executable transformations in QVT[1]. In Sect. 5.1, we introduce the formalism actually used as well as some alternatives. By comparing them, we motivate our choice for QVT. A more detailed introduction of QVT is given in Sect. 5.2. We state the limitations of our executable transformations and general transformation issues in Sect. 5.3, afterwards we present and discuss transformation of static structure in Sect. 5.4 and of dynamic behavior in Sect. 5.5.

This chapter repeatedly refers to the diploma thesis of Schlecht [87], which was supervised by me. His work is concerned with the selection of a well suited formalism to be used for the formalization of the transformation from input models into implementation models from which to generate code. Further, he gives a set of transformations for the creation of most of the implementation patterns presented in Chapter 3 and Chapter 4 form input models.

## 5.1 Formalisms

For implementing model transformations, various formalisms and tools supporting them may be considered. On the basis of our experiences from coding transformations directly in Java, the primary objective of applying another formalism than source code is to separate the idea of what is done from the implementation details of how it is done. All considered formalisms therefore are declarative. Other criteria worth being considered at decision making are

- adequacy of a formalism
- preciseness of the formalism's specification
- clarity of formalizations
- maintainability of formalizations
- continuative tool support, executability of formalizations
- support of proofs of correctness.

Since our ambition is not only to formalize transformations rather than to use formalizations as implementations in ActiveCharts, not only availability of a tool is a must-have, but also some features of such a tool are decisive factors. These are

- conformance to the formalism
- interoperability
- ability of integration
- suitable documentation

[1] MOF 2.0 Query, View, and Transformation specification, Relations [70]

- usability
- free availability
- continuing development.

A detailed examination with respect to these criteria is carried out on *Extensible Stylesheet Language Transformations* (XSLT) [102], *Constraint Handling Rules* (CHR) [26, 27], *Triple Graph Grammars* (TGG) [88, 46], *Atl Transformation Language* (ATL) [24], and *MOF 2.0 Query/View/-Transformation* (QVT) [70] by Schlecht [87]. An overview presenting the results of this examination is provided by Table 5.1 and Table 5.2 for which the key is given with Table 5.3.

| | **XSLT** | **CHR** | **TGG** | **ATL** | **QVTO** | **QVTR** |
|---|---|---|---|---|---|---|
| Adequacy | - | - | ++ | ++ | ++ | ++ |
| Expressiveness | ++ | ++ | + | ++ | ++ | ++ |
| Precision of Specification | ++ | ++ | ++ | + | ++ | ++ |
| Clarity / Readability | - | - | ++ | - | -- | ++ |
| Maintainability | - | - | + | + | - | ++ |
| Continuity of Tool Support | ++ | ++ | ++ | ++ | ++ | ++ |
| Support of Proofs of Correctness | + | ++ | ++ | - | -- | ++ |

Table 5.1: Summary of results obtained from the analysis of different formalisms (by Schlecht [87]).

| | **XSLT** | **CHR** | **TGG** | **ATL** | **QVTO** | **QVTR** |
|---|---|---|---|---|---|---|
| Conformance | ++ | + | + | ++ | ++ | + |
| Interoperability | + | + | ++ | ++ | ++ | ++ |
| Integrability | ++ | - | ++ | ++ | ++ | ++ |
| Suitability of Documentation | ++ | - | -- | ++ | ++ | - |
| Usability | – | – | ++ | ++ | ++ | + |
| Free Availability | ++ | ++ | ++ | ++ | ++ | ++ |
| Continuation of Tool Support | ++ | - | ++ | ++ | ++ | - |

Table 5.2: Summary of tool support for transformation languages of Table 5.1 (by Schlecht [87]).

| Rating | Meaning |
|---|---|
| ++ | fully applies |
| + | somewhat applies |
| - | hardly applies |
| -- | applies not at all |

Table 5.3: Key to Table 5.1 and Table 5.2.

As can be seen from the presented analysis results, XSLT and CHR are not well suited formalisms since they both lack a proper interface for being applied to models. XSLT is designed to operate on XML files. Although UML models can be exported to XMI on which XSLT can directly operate, working on this level of abstraction is a way back from models to code, in this case the actual file format for persisting models. In CHR, additionally code is required for embedding CHR constraints in a host language.

TGG, ATL, and QVT (explicitly divided into its parts QVTO and QVTR which are explained in detail later) are well suited, with respect to QVT, QVTR is the best suited formalism but QVTO is backed up by better tool support. In the end, the excellent rating of QVTR regarding the suitability of the formalism, the fact of QVTR being an OMG standard and the availability of tool support led to QVTR to be used in ActiveCharts.

TGG and ATL both are formalisms that could be applied either. The special case of our transformation in which the source and the target models share the same meta-model is not well supported by TGG. Whereas in QVT, it is possible to copy whole structures, i. e. parts of the source model consisting of multiple elements and relationships between them, this cannot be done using TGG. When using ATL, some parts of our transformations cannot be expressed declaratively but must be specified operationally.

QVT primarily comprises two parts one of which is operational, the other one is declarative. While both parts can be used alone for operationally or declaratively specifying transformations, both approaches can be combined such that in an overall declarative transformation, some details are operationally specified. Although there are tools supporting either one or the other part of QVT, tool support for a combined application is missing. Hence a combination of declarative and operational proceeding is not further considered.

A transformation implemented in QVT *Operational Mappings* (QVTO) consists of mappings each of which describes relations between elements of the source model and corresponding elements of the target model. A complete transformation consists of a chain of mappings where, starting from some kind of general mapping, more and more detailed mappings are invoked.

Although benefiting from powerful expressiveness, a precise specification, excellent tool support and even though being an adequate formalism for model transformations as ours, advantages with regard to clarity and readability are — compared to the prototypical implementations in Java — marginal, if at all.

In contrast, QVT *Relations* (QVTR) , which is the declarative approach of QVT, stands to benefit from the same advantages like QVTO, but together with great readability and still fair tool support. A drawback is that there are very few tools supporting QVTR, however, at least one of them, namely *medini QVT*, is quite satisfactory for our purpose.

## 5.2 QVT

The concrete syntax of QVTR, tailored to those language elements we actually make use of, is given in an EBNF notation in Listing 5.1 and Listing 5.2. Terminal symbols are written **blue**, nonterminals are written *gray*, nonterminals for which no production rules are given are written in <green> color and put in angle brackets.

A transformation starts with the keyword **transformation** followed by the transformation name and a declaration of a set of models, typically a source and a target model by defining a name and the meta-model of each. Optionally, a definition of keys follows, the effect of which we describe in the course of the description of our transformation.

The main part of a transformation implemented in QVTR consists of relations such as shown in Listing 5.3. A relation defines patterns for different domains, typically one for the source model (lines 9–12) and one for the target model (lines 13–17).

*"A domain has a pattern, which can be viewed as a graph of object nodes, their properties and association links originating from an instance of the domain's type. Alternatively a pattern can be viewed as a set of variables, and a set of constraints that model elements bound to those variables must satisfy to qualify as a valid binding of the pattern. A domain pattern can be considered a template for objects and their properties that must be located, modified, or created in a candidate model to satisfy the relation"* [70, §7.2].

Additionally, a relation may include a **when** and a **where** clause. Such clauses may contain conditions or other relations. A relation must only hold for those bindings of variables for which conditions and relations contained in its **when** clause hold. If a relation holds for a set of bindings for variables in its domain patterns, then all relations contained in the **where** clause must hold under this set of bindings, too, and a valid binding for remaining unbound variables must exist.

Whether or not a relation holds may be determined with two different semantics in mind: *checking* semantics and *enforcement* semantics. Checking semantics is applied if the target domain $k$ is marked **checkonly**. Under this condition, a relation holds, if for each binding of variables

```
transformation ::= transformation <identifier>
                     ( modelDecl (, modelDecl)* )
                     {
                         keyDecl*
                         ( relation | query )*
                     }

modelDecl      ::= modelId : metaModelId
modelId        ::= <identifier>
metaModelId    ::= <identifier>

keyDecl        ::= key classId { keyProperty (, keyProperty)* } ;
classId        ::= <pathNameCS>
keyProperty    ::= <identifier>


relation       ::= [top] relation <identifier>
                     {
                      varDeclaration*
                      (domain | primTypeDomain)+
                      [when]
                      [where]
                     }

varDeclaration ::= <identifier> (, <identifier>)* : <TypeCS> ;

domain         ::= [checkonly | enforce] domain modelId template ;

primTypeDomain ::= primitive domain <identifier> : <TypeCS> ;

template       ::= objectTemplate [{ OclExpressionCS }]

objectTemplate ::= [<identifier>] : <pathNameCS> {
                      [propTemplate (, propTemplate)*]
                     }

propTemplate   ::= <identifier> = OclExpressionCS

when           ::= when { (OclExpressionCS ;)* }

where          ::= where { (OclExpressionCS ;)* }

query          ::= query <identifier> ( [paramDeclList] ) : <TypeCS>
                     (; | { OclExpressionCS })

paramDeclList  ::= paramDecl (, paramDecl)*

paramDecl      ::= <identifier> : <TypeCS>
```

Listing 5.1: Excerpt of the concrete syntax of QVTR [70].

```
OclExpressionCS::= <PropertyCallExpCS>
                 | <VariableExpCS>
                 | <LiteralExpCS>
                 | <LetExpCS>
                 |  <IfExpCS>
                 | ( OclExpressionCS )
                 | template
```

Listing 5.2: Concrete syntax of OCL Expressions [70].

of the **when** clause and variables of domains other than k, a valid binding of the remaining unbound variables of domain k exists that satisfies domain k's patterns and relations and conditions contained in the **where** clause [70, §7.10.1].

If a target domain *k* is marked **enforce**, the relation holds under the *enforcement* semantics if it either holds under the *checking* semantics or, if it does not, creating, modifying, or removing existing objects and assigning properties as specified in domain k's pattern results in a target model for which the relation now holds under the *checking* semantics.

Whereas the idea of the **checkonly** semantics is to assure a postulated relation between source and target model, the **enforce** semantics modifies the target model to satisfy the postulated relation.

A relation is of one of two kinds: either it is a *top-level* or a *non-top-level* relation. Top-level relations must always hold, whereas non-top-level relations must do so only if referenced in the **where** clause of another relation.

A generic example is given by Listing 5.3. The relation ***relationName*** matches each instance of the specified type (**typeName**, line 9) in the source model which holds a specific value in a specified attribute (`metaAttribute = value`, line 11) and enforces the existence of an instance of another type (**anotherTypeName**, line 13) in the target model. Furthermore, two attributes are supplied with values, one of which is identical to the value of the same attribute in the source instance (lines 15–16). This relation only must hold for those matches where the value of *srcClass*`.something` is not `0` (line 18). If the relation must hold, the relation ***anotherRelationName*** referred to in the **where** clause (line 19) must hold, too.

```
1  transformation transformationName
2  ( sourceModelId : sourceMetaModelId, targetModelId : targetMetaModelId )
3  {
4    key classId { propertyId },
5    key classId { propertyId2, propertyId3 };
6
7    top relation relationName
8    {
9      checkonly domain sourceModelId srcClass : typeName
10     {
11       metaAttribute = value
12     };
13     enforce domain tragetModelId trgClass : anotherTypeName
14     {
15       metaAttribute = value,
16       anotherMetaAttribute = srcClass.anotherMetaAttribute + newValue
17     };
18     when { srcClass.something <> 0 }
19     where { anotherRelationName(srcClass, trgClass); }
20   }
21
22   relation anotherRelationName
23   {
24     checkonly domain ...
25     enforce domain ...
26   }
27 }
```

Listing 5.3: Pseudo transformation.

Queries are functions without side effects, which may occur in a transformation like relations do and are referred to within **when** and **where** clauses. They can be used to perform checks or to compute values, which are to complex to be in-lined at the place where needed or if reuse is desired.

#### Notational Convention: Referencing Listings

As a shorthand pointer to queries or to relations which occur in `when` or `where` clauses, we include an arrow and the number of the concerned listing (e. g. $\nearrow^{X.n}$). Such listings are either contained in the appendix (e. g. $\nearrow^{B.12}$) or in this chapter (e. g. $\nearrow^{5.3}$).

By directly indicating within each listing where implementations of referenced relations and queries may be found, we will not include pointers to those listings in the textual description of the listings containing such hints.

## 5.3 Transformations by QVT Relations

Before we present the transformation which formalizes our implemented prototype, we shortly outline some limitations either induced by deficiencies of *medini QVT* or by pragmatical considerations, in particular in order to decrease the complexity of the transformation. Some additional issues are described by Schlecht [87], but omitted here because they are less relevant for us.

### 5.3.1 Limitations Caused by *medini QVT*

*medini QVT* does not support all features of QVTR as specified by OMG. Some are missing, others are implemented with a slightly differing semantics. Furthermore, some bugs in the implementation of *medini QVT* necessitate adaptations of transformations the necessity of which is not comprehensible unless knowing about their reasons. In the following, we shortly address those deficiencies of *medini QVT* which are or might be of relevance for our approach.

#### Collection templates

The QVTR specification defines *collection template*, which, in contrast to object template, specifies a pattern that matches a collection of elements. Collection template is not supported by *medini QVT*. Since collection template rather is syntactic sugar than a real necessity, each transformation can be implemented by using object templates — however, such a transformation will be less comfortable to read and write. Hence, this limitation leads to more complex relations, but it does not cause any severe or even unsolvable problems.

#### Black-Box Implementations

Black box implementations, which support using imperative concepts within a declarative formalism, are not supported by *medini QVT*as well. Again, this limitation has no serious effect with respect to our formalization since that can completely be covered by declarative concepts.

#### Enforcement Semantics — Creation of Objects

The enforcement semantics implemented in *medini QVT* differs from the specified enforce semantics. Due to the specification, an element must not be created if an element with the specified characteristics already exists. Primarily for performance reasons, checking for the existence of elements before creating them is omitted in *medini QVT*. Under special conditions, elements may be created multiple times [87]. However, the correct enforcement semantics can be enforced by using keys for those elements which by mistake are possibly repeatedly created.

#### Indices

*medini QVT* does not guarantee the ordering of elements even in ordered sets like `Sequence` or `OrderedSet`. Hence, the identification of elements must not be based on the position within the containing set.

### 5.3.2 Arbitrary Limitations for Decreasing Complexity

Even though we aim to support a wide range of modeling concepts, we accept to define some limitations in order to dispense with excessive handling of very complex or even exceptional cases such as unnamed model elements. In the following, we shortly introduce the general limitations of our formalization, which are not induced by limitations of our approach.

### Well Formed Models

Usually it is dispensable to supply all features of a modeling element with values, e. g. association ends may be unnamed. An association labeled *employment* between a class named *Employee* and another class named *Company* probably represents any kind of employment that is not better understandable if role names are given to association ends.

At some point in the processing from models to code, unnamed association ends must be named in order to distinctly name methods managing associations. Typically, unnamed association ends are named after the type to which they are connected.

Any kind of creating values for features of modeling elements, such as deriving a name from the context of an unnamed element, is not originally rooted in the transformation itself. Rather, it is a supplementation of a model that should be done before. We therefore assume that all features of modeling elements which are affected by our transformations are supplied with reasonable values.

### Multiple Inheritance

One of the key concepts in the object paradigm is generalization. In object-oriented languages, usually types can specialize other types — which in term are their general types — and implementations of types may inherit from the implementations of their general types.

In Java, a type is defined by the declaration of an `Interface` and implemented within a `Class`. For convenience, type specification and implementation can be combined by just writing a class serving as both, type specification and implementation. If types and implementations are strictly separated, implementations of types can be replaced with no need for any changes to other parts of the software.

Java supports multiple inheritance only on the level of types, not on their implementations, i. e. a class may implement multiple interfaces but can extend not more than one general class. Although there are some patterns which try to map multiple inheritance to languages which do not natively support it, we do not consider multiple inheritance in our transformation.

The relation between interfaces and classes in Java corresponds to the relation between *Interface*, *Classifier*, and *InterfaceRealization*. Although it is quite straightforward to map the UML meta-classes to Java language elements, we do not explicitly support this in our approach but consider all types to be defined only by classes.

### Nested Elements

A means for grouping elements in UML as well as in Java is *Package*. A package typically contains type definitions (interfaces), and type implementations (classes). A package may contain other packages. *Package* is a concept that supports organizing classes which have some characteristics in common, e. g. belonging to the same category or offering similar functionality.

Unless permitting the same name to different classifiers if both are contained in different packages and unless having impact on the visibility of features of a classifier to other classifiers, using packages and nesting packages has no deeper semantics. Since the structure of packages is not affected by our transformation, we consider all classes to be owned by the same package and omit nesting packages. We assume that a model containing only one single package can be created by moving all classes into this package and renaming them by building some kind of qualified name, e. g. by prepending all package names to the class name. After the transformation, the package structure can be restored by building all packages and moving classes into those packages according to their qualified name.

Regarding nested classes, the situation is somewhat more complex and flattening the hierarchy definitely is not possible since nested classes have access to private features of the containing classifier. By making nested classifiers instances of their own right, accessibility of private features of the formerly containing classifier is lost. However, to consider nested classifiers probably is straightforward by processing each nested classifier of a classifier while processing the containing classifier itself in a recursive way, but in order to keep our prototype simple, we do not yet consider nested classifiers.

### Application of Subsetting and Inheritance to Associations

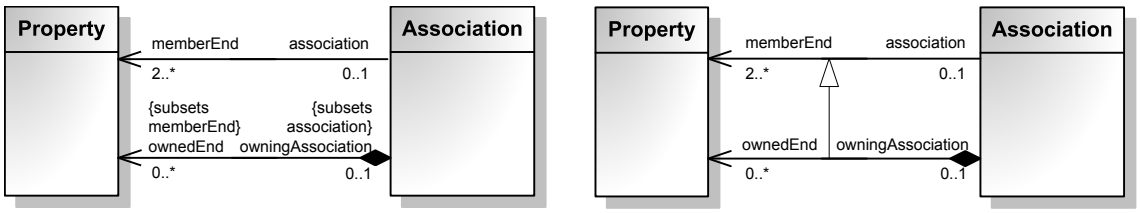
Subsetting and generalization are two different, but somehow similar concepts. Whereas subsetting is intensively used within the UML specification, generalization is not applied to associations in the UML specification at all.

The semantics of subsetting is that the set of values of an association end may be defined to be a subset of the set of values of another association end. Of course, some constraints must hold, e. g. the number and types of association ends of both associations must conform to each other, in particular, the subsetting end must be of the same or of a subtype of the subsetted end. Subsetting does not affect links, i. e. the links of an association of which one end subsets an end of another association are not related to the links of that other association.

But if an association is a specialization of another association, all links of the special association are also links of the general association, like instances of classes related by generalization are related to each other, as well. Generalization is applicable to associations since it is specified for classifiers, a subclass of which is association.

To give an example, we refer to the specification of *Association* in UML as shown in Fig. 5.1(a): there are two associations between *Association* and *Property*. One has the ends *association* and *memberEnd*, the other has the ends *owningAssociation*, which subsets *association*, and *ownedEnd*, which subsets *memberEnd*. The implication of subsetting here is, that a link between an association and a property for specifying an owned end requires the existence of a link between the same instances specifying the property to be a member end. Subsetting assures, that the set of owned ends is a subset of member ends. But the additional semantics of being an owned end is represented by an additional link in its own right. If a link between *association* and *memberEnd* is removed, a link associating the same instances by *owningAssociation* and *ownedEnd* must be removed if existing, too, in order to hold the subset relation consistent.

In contrast, the association for specifying owned ends could be modeled as a specialization of the association specifying member ends (as modeled in Fig. 5.1(b)). Then the creation of a link for defining an owned end would implicitly create a link defining a member end. This is the result of the special link in itself also being an instance of the general link. The deletion of the link would be possible in both associations (since the link is an instance of both) and intrinsically would affect the other association as well. However, changing an owned end to be a non-owned member end requires to delete the existing link, which is an instance of both associations, and to replace it by a newly created link which is an instance of the general association only. Making an existing end an owned end requires to delete the link of the general association and create a link of the special association.




(a) Subsetting of association ends.

(b) Generalization applied to associations.

Figure 5.1: Subsetting and Generalization applied to associations.

The implementation of associations by the Relationship Object Pattern allows for supporting both, subsetting and generalization. For subsetting, checks for ensuring consistency between sets and subsets of links must be inserted into manager classes. For generalization of associations, a straightforward implementation is to relate classes representing links by generalization as well.

A generalization relationship between links does not require a generalization relationship between association manager classes. But instances of links of an association must be included in the set of links of associations which are a general classifier. Therefore, operations for creating or deleting links must trigger operations in other associations related via generalization.

We investigated the realisability of both, subsetting and generalization applied to associations, by extending the Relationship Object Pattern on a level detailed enough to see that both concepts are technically feasible. However, an exhaustive consideration and discussion of implementation patterns for subsetting or generalization applied to associations is beyond the scope of this thesis.

#### No Support of Data-Types

*DataType* is a specialization of *Classifier* and is very similar to *Class*. In contrast to *Class*, it is not able to contain nested elements and its instances do not have their own identity. Since these differences are a limitation of data type compared to class, supporting data types should be possible on the basis of the support for classes, with little adaptations. However, in the scope of this work, we do not explicitly include data types.

#### Confinement to Most Common Features

In our transformation, source and target models share the same meta-model. In this regard, it is a special case whereas the general case is to transform models from a source meta-model into a target meta-model. If not sharing the same meta-model, all elements of the source model must explicitly be assigned to elements in the target model, whereas in our case, elements and subelements could also be cloned since elements are of the same types (referred to the meta-model) and have the same values. Cloning or copying elements — and in the case of numerous subelements, whole parts of a model — is not interesting and its detailed inclusion into our transformation does not contribute to a better understanding of our approach. Hence, we only include the copying of elements which are needed in subsequent steps of our model processing. Furthermore, we encapsulate copying in separate relations and exclude them from a detailed description.

### 5.3.3 General Aspects

The entire formalization of the transformation as implemented in our prototype ActiveCharts ranges from high level concepts to technical details of a quite low level of abstraction. Therefore, we focus on those parts which are relevant for pointing out our basic approach, that is, we concentrate on the tip of the iceberg and include less interesting technical details in the appendix and refer to them, if necessary, by giving a rough idea of omitted issues.

All relations for the model transformation are contained in a single QVTR transformation, i. e. no intermediate results are explicitly stored. The declaration of the transformation with its source and target models is given in Listing 5.4.

```
1 transformation mainTransformation(sourceModel : uml, targetModel : uml)
2 {
3   <key declarations>
4   <relations>
5 }
```

Listing 5.4: QVTR transformation: model declarations.

Before we go into a detailed discussion of the relations, we give some general background information about the transformation. This includes techniques we use to face problems arising from the limitations of *medini QVT* as well as some design decisions in the organization of relations which apply to both, transforming structures as well as transforming behavior.

#### Enforcing the Correct Enforcement Semantics by Using Keys

The semantics of **enforce** is that model elements are only created if they do not already exist, i. e. **enforce** guarantees the presence of an element, not its creation. Since *medini QVT* by mistake possibly repeatedly creates elements instead of matching previously created instances, the **enforce**-semantics must be fixed by using keys. A key is defined for each modeling element, i. e. for each meta-class, for which duplicate creation must be avoided. Meta-attributes used for identifying duplicates are included in the key declaration. In Listing 5.4, the real contents of line 3 are the key declarations contained in Listing 5.5.

Note that according to Schlecht [87], the use of keys will not prevent elements to be repeatedly created if more than one relation of those, which enforce the existence of such elements, are contained in the **where** clause of a single relation. This situation can be avoided by referring only to one of the concerning relations, say $R_1$ in the **where** clause where it actually appears and moving references to all other relations $R_i$ in the **where** clause of $R_1$. If $R_i$ denotes more than one relation,

```
1  -- Key Declarations
2  key PrimitiveType {name};
3  key Package {name};
4  key Class {name};
5  key Generalization {specific, general};
6  key Association {name};
```

Listing 5.5: QVTR transformation: Key declarations.

the same rule must be applied to the **where** clause of $R_1$. References to relations which enforce the existence of other elements are not affected.

### Copying Complex Model Elements

Transformation languages are designed to transform a source model into a target model, whereby the underlying meta-models usually are different. Our transformation aims to resolve complex UML structures by lower level elements with the target model still being a UML model. Therefore, an in-place-transformation where the target model initially is a copy of the source model, would be a suitable alternative, but unfortunately, since *medini QVT* does not support pattern matching in the target model, in-place-transformations are not well supported [87].

Alternatively, those parts of the source model which are not changed by the transformation can either be explicitly created — by enforcing the elements to exist in the target model with values of meta-attributes and references to other model elements specified exactly as in the target model — or be moved from the source model to the target model. In *medini QVT*, the second alternative can only be used if no relation attempts to match a pattern in the source model which has previously been moved to the target model, although the source model should not be changed during the transformation.

Since moving patterns from the source to the target model is problematic, in some cases model elements must be copied by hand, i. e. a relation must enforce an element and all its sub-elements to exist in the target model and set values of meta-attributes to the values specified in the source model. The relations for such a kind of copying model elements are not interesting, therefore, these relations are not discussed in the following but are included in the appendix. Furthermore, only those sub-elements are copied which are necessarily needed in the target model, i. e. elements which are supported in subsequent processing steps like code generation.

### Setting Up the Target Model

Before transforming the static structure and dynamic behavior specifications, one relation is needed for setting up the target model. This relation as shown in Listing 5.6 creates the only package named PSM which exists in the target model (line 6) and copies all element imports defined in the source model into the target model (line 9). The relation `Copy_ElementImport` is contained in the appendix, Listing B.13.

```
1  top relation ElementImport{
2    checkonly domain source srcPackage : Package {
3      elementImport = srcElementImport : ElementImport {}
4    };
5    enforce domain target trgPackage : Package {
6      name = 'PSM',
7      elementImport = trgElementImport : ElementImport {}
8    };
9    where { Copy_ElementImport(srcElementImport, trgElementImport);↗B.13}
10 }
```

Listing 5.6: Relation for creating a package and element imports.

#### Additional Information of Model Elements

During the transformation, some elements are created which have a very special meaning which must be considered when generating code from the target model. *Stereotype* is a concept that can be used to carry such additional information.

But using stereotypes requires their specification in a model profile, which is applied in the target model. Such a profile causes additional overhead without contributing to a better understandability of the transformation. Therefore, we decided to tag additional information to elements of the target model by using comments. This technique is easily implementable by covering the tagging of additional information in relations which are included in `where` clauses if tagging information is required.

Applying stereotypes admittedly is a more proper approach, but in favor of avoiding overhead while still obtaining executable transformations, comments are a good choice. The relations for adding comments and the contents of these comments are detailed in Appendix B.1.4.

## 5.4 Transforming Models of Static Structure

The transformation of static structure, i. e. of class diagrams, divides into three main parts:

1. insertion of implementation classes for associations
2. insertion of operations for accessing structural features
3. separation of generated code from user written code.

A detailed motivation and considerations concerning semantics of input and output models, adequacy of transformations etc. are all given in Chapter 3. Therefore, in this section, we will work with the patterns introduced in Chapter 3 without reflecting their semantics again.

We start with the transformation of classes with attributes, operations, and generalization relationships to other classes. Afterwards, we present the transformation of associations and association classes. Finally, additional modifications for implementing compositions are considered.

### 5.4.1 Transformation of Classes

A keynote of our approach is to hide generated code from the user, not only because generated code may cause confusion, but also because it should be protected from inadvertent as well as from intentional changes. For this reason, generated code should not be contained in the same source file that contains user written code [34].

In C#, the desired separation of generated and user written code both implementing the same class, can be achieved by applying partial classes [57].

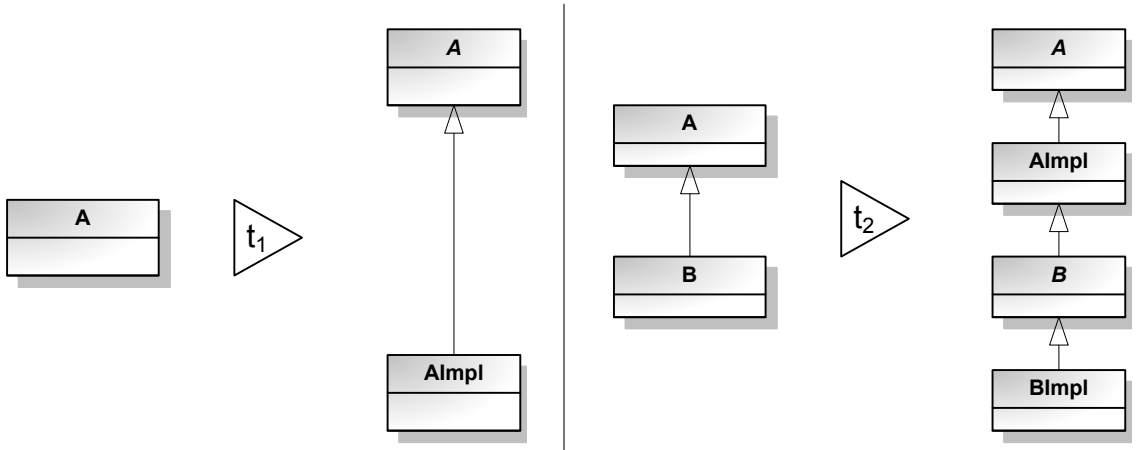
Regarding Java, generated and user written code must be contained in two distinct classes. This separation of generated and user written code is prepared by the transformation $t_1$ as shown in Fig. 5.2(a). Class $A$ from the source model (left) is transformed into a pair of two classes of the target model (right). In the target model, class $A$ is intended to contain generated code, like implementations of associations, attributes and behaviors, whereas class $AImpl$ is intended to hold user written code.

In order to clearly distinct the two kinds of created target classes, we term the general class $A$ a *conceptual class* and its specialization, class $AImpl$, we term *implementation class*.

If in a source model, two classes are related by generalization like in Fig. 5.2(b), transformation $t_2$ must assure that in the target model, class $B$ is no longer a specialization of class $A$ but of the newly inserted class $AImpl$.

Transformation $t_1$ is specified by relation `SingleClass` of Listing 5.7. It matches each source class (line 4) and creates an abstract class with the same name and visibility as the source class (lines 8–11). Additionally, a second class is created having the same name suffixed with `Impl` and the same visibility (lines 12–17) and being a specialization of the previously created class.

If a source class is abstract, then both, the conceptual class and implementation class are abstract, if the source class is not abstract, the conceptual class is abstract whereas the implementation class is not. A conceptual class is always abstract since it should never be instantiated alone. It builds a logical unit together with an implementation class and therefore must not exist on its own. This is insured by declaring the conceptual class to be abstract.




(a) Transformation of classes.

(b) Transformation of generalization.

Figure 5.2: Transformation of classes.

Figure 5.2(a) only gives a rough idea of the transformation of classes. Features of *A* in the source model must be distributed among the conceptual class and the implementation class, i. e. among *A* and *AImpl* of the target model. These details are covered by relations contained in the **where** clause of `SingleClass`.

In line 20 of `SingleClass` (Listing 5.7), a relation is called which tags the information of being an implementation class to the target implementation class *AImpl*. This information is needed for code generation in order to avoid confusion when source class names end with *Impl*, resulting in target classes named e. g. *AImpl* and *AImplImpl*.

```
1  top relation SingleClass
2  {
3    checkonly domain source srcPackage : Package {
4      packagedElement = srcClass : Class {}};
5
6    enforce domain target trgPackage : Package {
7      name = 'PSM',
8      packagedElement = trgClass : Class {
9        name = srcClass.name,
10       isAbstract = true,
11       visibility = srcClass.visibility },
12     packagedElement = trgImplClass : Class {
13       name = srcClass.name + 'Impl',
14       isAbstract = srcClass.isAbstract,
15       visibility = srcClass.visibility,
16       generalization = trgGeneralization : Generalization {
17         general = trgClass }}};
18
19   where {
20     SC_Comment(trgImplClass)↗B.32;
21     SC_PublicProtectedPackageProperty_Multiple(srcClass, trgClass)↗5.9;
22     SC_PublicProtectedPackageProperty(srcClass, trgClass)↗B.1;
23     SC_PublicProtectedPackageProperty_Derived_Multiple(
24         trgClass, trgImplClass, srcClass)↗5.10;
25     SC_PublicProtectedPackageProperty_Derived(
26         trgClass, trgImplClass, srcClass)↗B.2;
27 }
```

Listing 5.7: Transformation of classes.

Before discussing the other relations contained in the **where** clause of `SingleClass`, which handle the transformation of properties, we introduce the specification of transformation $t_2$ of Fig. 5.2(b). Since the information about a generalization relationship of classes is not considered by relation `SingleClass`, relation `SingleInheritance` of Listing 5.8 specifying transformation $t_2$ is needed.

This relation matches each possible pair of classes (lines 4–5) of the source model. In the **when** clause, it is checked whether the two classes are related by generalization (line 18). If they are, the relation must hold. In this case, the existence of an implementation class for the superclass is enforced (lines 10–11). Furthermore, the existence of the conceptual class for the subclass is enforced (lines 12–13). Finally, the generalization from the conceptual class of the subclass to the implementation class of the superclass is enforced (lines 14–15).

```
1  top relation SingleInheritance
2  {
3    checkonly domain source srcPackage : Package {
4      packagedElement = srcSuperClass : Class {},
5      packagedElement = srcSubClass : Class {}
6    };
7
8    enforce domain target trgPackage : Package {
9      name = 'PSM',
10     packagedElement = trgSuperImplClass : Class {
11       name = srcSuperClass.name + 'Impl' },
12     packagedElement = trgSubClass : Class {
13       name = srcSubClass.name,
14       generalization = trgGeneralization : Generalization {
15         general = trgSuperImplClass }}};
16
17   when {
18     srcSubClass.general->includes(srcSuperClass.oclAsType(Classifier)); }
19 }
```

Listing 5.8: Transformation of generalization relationships.

### Transformation of Owned Attributes

The transformation of properties, which are owned attributes of a classifier, is specified by a set of relations. Each of these relations handles only a subset of all owned properties. The subsets, into which the set of owned properties is divided, group properties regarding their characteristics of being derived or not and being single-valued or multi-valued, as shown by Table 5.4:

| | **isDerived** | | **upper bound** | |
|---|---|---|---|---|
| relation name | yes | no | $= 1$ | $> 1$ |
| SC_PublicProtectedPackageProperty_Multiple | | x | | x |
| SC_PublicProtectedPackageProperty | | x | x | |
| SC_PublicProtectedPackageProperty_Derived_Multiple | x | | | x |
| SC_PublicProtectedPackageProperty_Derived | x | | x | |

Table 5.4: Mapping from properties to processing relations.

The relations for processing owned attributes are all very similar to each other why it seems appropriate to discuss not all of them in detail but to include only some of them here and to refer to the appendix for the remaining relations.

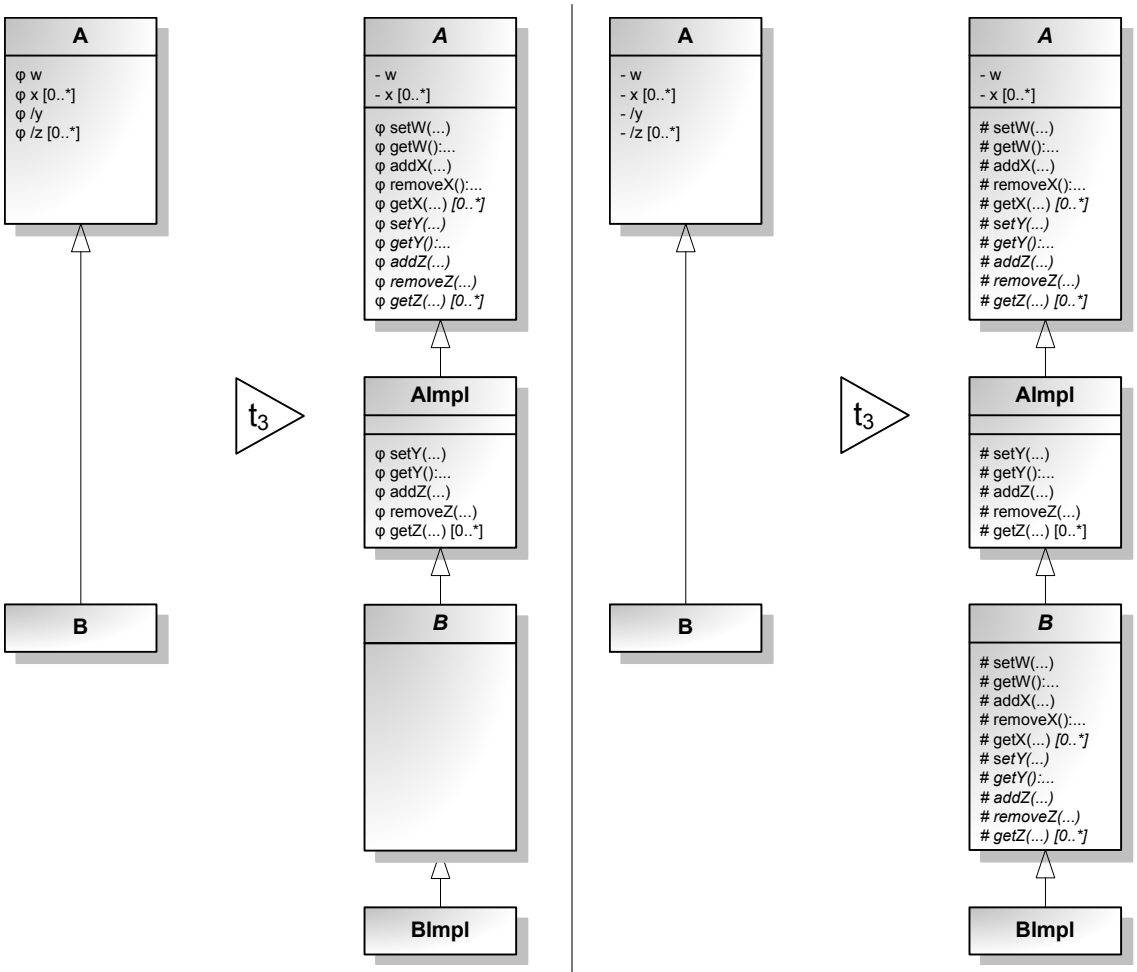
Figure 5.3 shows the transformation of attributes. The transformation of non-private attributes, i. e. of attributes with public, package, or protected visibility is shown in Fig. 5.3(a). The transformation of private attributes is shown in Fig. 5.3(b). In either case, the owned attributes are encapsulated as private attributes in the conceptual class, for accessing the attributes, operations are provided in the conceptual class as well.

For non-private attributes, operations for accessing owned attributes have the same visibility as the original attribute (we refer to this visibility by the letter $\varphi$). The names for these operations are formed by prepending a prefix before the name of the attribute. The prefixes are *set* and *get*, if the attribute is single-valued, or *add*, *remove*, and *get*, if it is multi-valued.

For private attributes, operations for accessing the attribute must be protected in order to restrict access to the attribute to the scope of the implementation class. In order to prevent from accessing the attribute in the scope of further specializations, operations for accessing private attributes must be redefined in specializations of the implementation class in the target model. The redefined operations must not access the attribute in order to hide it from specializations [34].

The general approach of transforming attributes comprises three steps:

- create private attributes in the conceptual class of the target model
- provide operations for accessing the private attributes with appropriate visibilities
- redefine operations for accessing private attributes in specializations of the concerning implementation classes.




(a) Transformation of non-private attributes.

(b) Transformation of private attributes.

Figure 5.3: Transformation of attributes.

Listing 5.9 shows the transformation of a non-derived, multi-valued attribute. In the `when` clause, the upper bound of the attribute is determined (line 12). Additionally, in order to prevent from processing owned ends in the same way as owned attributes, it is assured that the property is not an association end (line 13).

For each attribute conforming to the constraints defined in the `when` clause, a private attribute is enforced in the conceptual class of the target model (lines 8–9). In the `where` clause, a relation copying other characteristics of the property and a relation enforcing operations for accessing the property are included.

```
1  relation SC_PublicProtectedPackageProperty_Multiple
2  {
3    checkonly domain source srcClass : Class {
4      ownedAttribute = srcProperty : Property {
5        isDerived = false }};
6
7    enforce domain target trgClass : Class {
8      ownedAttribute = trgProperty : Property {
9        visibility = VisibilityKind::private }};
10
11   when {
12     srcProperty.upper > 1 or srcProperty.upper = -1;
13     srcProperty.association.oclIsUndefined();}
14
15   where {
16     Copy_Property(srcProperty, trgProperty)↗B.14;
17     SC_PropertyAccessOperation_Multiple(
18         false, srcProperty, trgClass)↗5.11; }
19 }
```

Listing 5.9: Transformation of non-private, multi-valued attributes.

Derived properties do not hold values rather values are computed at runtime. Accordingly, derived attributes are transformed into abstract operations in the conceptual class, which have the same visibility as the source attribute (or protected, if the attribute is private). These operations must be implemented by hand in the implementation class providing algorithms for actually deriving the values. An example is given by Listing 5.10. The relation for generating operations for accessing the attribute is included twice. In line 15, the relation is called for creating the abstract operation in the conceptual class, in line 16, it is called to create the concrete operation in the implementation class.

```
1  relation SC_PublicProtectedPackageProperty_Derived_Multiple
2  {
3    primitive domain trgClass : Class;
4    primitive domain trgImplClass : Class;
5
6    checkonly domain source srcClass : Class {
7      ownedAttribute = srcProperty : Property {
8        isDerived = true }};
9
10   when {
11     srcProperty.upper > 1 or srcProperty.upper = -1;
12     srcProperty.association.oclIsUndefined(); }
13
14   where {
15     SC_PropertyAccessOperation_Multiple(true, srcProperty, trgClass)↗5.11;
16     SC_PropertyAccessOperation_Multiple(
17         false, srcProperty, trgImplClass)↗5.11; }
18 }
```

Listing 5.10: Transformation of non-private, multi-valued attributes that are derived.

A relation enforcing operations for accessing a multi-valued attribute is presented in Listing 5.11. Within this relation, three operations are enforced for reading attribute values, for adding a new value and for removing a value. By a parameter, it is determined whether or not the operations should be abstract. If the relation is called for creating operations in a conceptual class (as in line 15, Listing 5.10), abstract operations are created, if called for creation of operations in a implementation class (as in line 16, Listing 5.10), all operations will be non-abstract.

In the **where** clause, the operations are marked with a comment in order to be able to identify these operations as a part of the implementation pattern for owned attributes when generating code from the target model. Some more detailed characteristics like visibility and parameters of the operations are handled in relations included in the **where** clause. For multi-valued attributes, relations `SC_GetterCharacteristics` (Listing 5.12) and `SC_AddRemoveOperationCharacteristics` (Listing 5.13) are included, for processing single-valued attributes, `SC_SetterCharacteristics` (included in Appendix B.1.1, Listing B.4) is used instead of relation `SC_AddRemoveOperationCharacteristics`.

A relation similar to `SC_PropertyAccessOperation_Multiple` of Listing 5.11 but for single-valued attributes is included in Appendix B.1.1, Listing B.3.

```
relation SC_PropertyAccessOperation_Multiple
{
  primitive domain isAbstractValue : Boolean;
  primitive domain srcProperty : Property;

  enforce domain target trgClass : Class {
    ownedOperation = trgClassGetter : Operation {
      isAbstract = isAbstractValue },
    ownedOperation = trgClassAddOperation : Operation {
      isAbstract = isAbstractValue },
    ownedOperation = trgClassRemoveOperation : Operation {
      isAbstract = isAbstractValue }};

  where {
    SC_PropertyAccessOperation_Comment(
        'Getter', srcProperty.name, trgClassGetter)↗B.33;
    SC_PropertyAccessOperation_Comment(
        'AddOperation', srcProperty.name, trgClassAddOperation)↗B.33;
    SC_PropertyAccessOperation_Comment(
        'RemoveOperation', srcProperty.name, trgClassRemoveOperation)↗B.33;
    SC_GetterCharacteristics(
        srcProperty, trgClassGetter)↗5.12;
    SC_AddRemoveOperationCharacteristics(
        'add', srcProperty, trgClassAddOperation)↗5.13;
    SC_AddRemoveOperationCharacteristics(
        'remove', srcProperty, trgClassRemoveOperation)↗5.13; }
}
```

Listing 5.11: Creation of accessor operations for multi-valued attributes.

```
relation SC_GetterCharacteristics
{
primitive domain srcProperty : Property;

enforce domain target trgOperation : Operation {
  name = 'get' + srcProperty.name.firstToUpper(),
  ownedParameter = trgParameter : Parameter {
    direction = ParameterDirectionKind::return,
    lower = srcProperty.lower,
    upper = srcProperty.upper },
  visibility = ownedElementVisibility(srcProperty.visibility)↗B.45 };

where {
  Copy_Parameter_ClassType(srcProperty.type, trgParameter)↗B.22;
  Copy_Parameter_PrimitiveType(srcProperty.type, trgParameter)↗B.23; }
}
```

Listing 5.12: Completion of the operation for reading access to attributes.

```
relation SC_AddRemoveOperationCharacteristics
{
  primitive domain namePrefix : String;
  primitive domain srcProperty : Property;

  enforce domain target trgOperation : Operation {
    name = namePrefix + srcProperty.name.firstToUpper(),
    ownedParameter = trgParameter : Parameter {
      name = srcProperty.name.firstToLower() },
    visibility = ownedElementVisibility(srcProperty.visibility)↗B.45 };

  where {
    Copy_Parameter_ClassType(srcProperty.type, trgParameter)↗B.22;
    Copy_Parameter_PrimitiveType(srcProperty.type, trgParameter)↗B.23; }
}
```

Listing 5.13: Completion of operations for writing access to multi-valued attributes.

The transformation of private attributes as depicted in Fig. 5.3(b) requires to adapt the relation of inheritance in order to hide protected methods for accessing attributes from specialized classifiers. For each privately owned attribute of classifier $A$, the access operations must be redefined in conceptual classes of specializations of $A$. Relations implementing this are included in B.5.

### Transformation of Owned Operations

With respect to their transformation, we consider two kinds of *Operation*:

- if behavioral aspects are modeled as well, an operation may have a behavior associated with it such that the code generated for this behavior is the implementation of the operation
- if behavior is not modeled or an operation is not associated with a behavior, then the implementation of such an operation is to be done manually after code generation.

In either case, the problem is that a behavior as well as code implementing an operation must have access to the features of the classifier owning the operation. Therefore, private operations must be transformed analogously to private attributes with protected visibility in the conceptual class in order to be accessible in the implementation class.

Listing 5.14 presents the first relation initiating the transformation of operations. It matches each source class (line 4) and the concerning conceptual class (lines 8–9) and implementation class (lines 10–11) of the target model. Enforcement of operations is covered by relations included in the **where** clause.

```
1  top relation OwnedOperation
2  {
3    checkonly domain source srcPackage : Package {
4      packagedElement = srcClass : Class {}};
5
6    checkonly domain target trgPackage : Package {
7      name = 'PSM',
8      packagedElement = trgClass : Class {
9        name = srcClass.name },
10     packagedElement = trgImplClass : Class {
11       name = srcClass.name + 'Impl' }};
12
13   where {
14     O_PublicProtectedPackageOperation(srcClass, trgClass, trgImplClass)↗5.15;
15     O_PublicProtectedPackageOperation_Abstract(srcClass, trgClass)↗5.16;
16   }
17 }
```

Listing 5.14: Transformation of operations.

Relation `O_PublicProtectedPackageOperation` of Listing 5.15 transforms operations which are not abstract (line 5) into a pair of operations, one of which being abstract (line 9) and located in the conceptual class, the other one being not abstract (line 13) and located in the implementation class. The **where** clause contains relations which supply both operations with some details which are copied from the source operation.

```
1  relation O_PublicProtectedPackageOperation
2  {
3    checkonly domain source srcClass : Class {
4      ownedOperation = srcOperation : Operation {
5        isAbstract = false }};
6
7    enforce domain target trgClass : Class {
8      ownedOperation = trgClassOperation : Operation {
9        isAbstract = true }};
10
11   enforce domain target trgImplClass : Class {
12     ownedOperation = trgImplClassOperation : Operation {
13       isAbstract = false }};
14
15   where {
16     Copy_Operation(srcOperation, trgClassOperation)↗B.17;
17     Copy_Operation(srcOperation, trgImplClassOperation)↗B.17; }
18 }
```

Listing 5.15: Enforcing target operations for non-private source operations.

Relation `O_PublicProtectedPackageOperation_Abstract` given in Listing 5.16 handles abstract operations. It enforces only one abstract operation in the conceptual class (line 8). Since being marked abstract, it is obviously not intended to provide an implementation.

In a correct model, a specialization of the class containing the abstract operation must provide an implementation, i. e. the special class must contain an operation with conforming name and parameters which is not abstract. This operation will be transformed by relation `O_PublicProtectedPackageOperation` which will enforce another abstract operation in the conceptual class of the special class. Although this duplicate occurrence of an abstract operation is useless, we do not mind about it since it causes no problems.

In the **where** clause a relation is contained which copies all operation details from the source model into the target model.

```
1  relation O_PublicProtectedPackageOperation_Abstract
2  {
3    checkonly domain source srcClass : Class {
4      ownedOperation = srcOperation : Operation {
5        isAbstract = true }};
6    enforce domain target trgClass : Class {
7      ownedOperation = trgClassOperation : Operation {
8        isAbstract = true }};
9
10   where {
11     Copy_Operation(srcOperation, trgClassOperation)↗B.17; }
12 }
```

Listing 5.16: Enforcing target operations for non-private, abstract source operations.

The transformation of private operations follows the semantics presented for the transformation of private attributes. The operation is included in the conceptual class with protected visibility and it must be redefined in the conceptual class of each specialization, if any, in order to avoid accessibility in the down-chain of generalization relationships.

The necessary adaptation of relation `SingleInheritance` is included in the Appendix (Listing B.5).

### 5.4.2 Transformation of Associations

In the discussion of the semantics of *Association* in Chapter 3, Sect. 3.2, we motivated that a semantically sound implementation is achievable, however, it is not simple. We argued to use the *Relationship Object Pattern* which is a natural fit to the specification of associations at runtime described as a set of links which consist of values each value being a reference to an object participating in that link.

In this section, we explain how to transform a source model in order to have the Relationship Object Pattern explicitly modeled for each association in the target model.

Conceptually, this transformation can be divided into two domains:

- replace an association by an implementation class and add a link class
- adapt classes participating in the association to the new structure e. g. by adding references to the association implementation class.

We term that class the instances of which represent links of an association a *link class*, the class creating, querying, and destroying links we term a *manager class*.

Figure 5.4 shows the general transformation of a ternary association applying the Relationship Object Pattern. The association is replaced by the manager class *ABC* and the link class *Link*. The classes participating in the association are no longer connected to each other rather they all reference the manager class and are referenced by the link class.

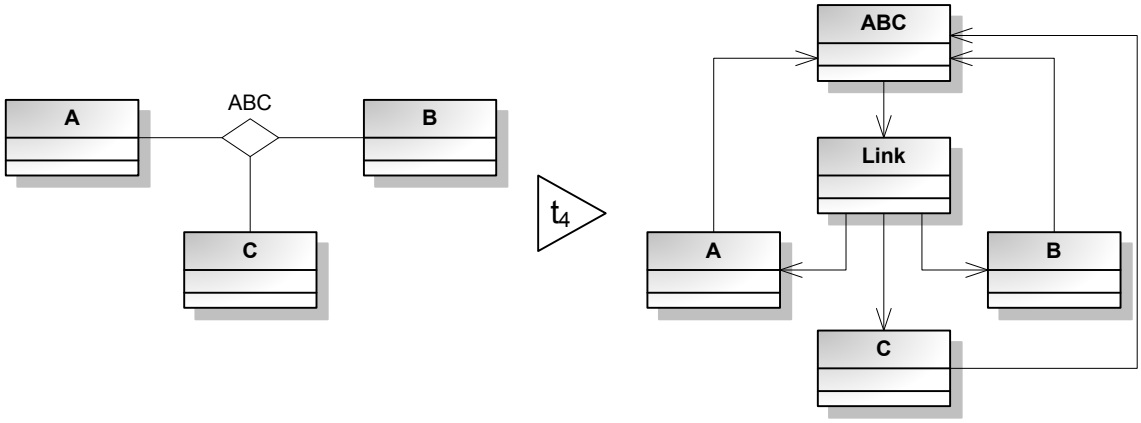



Figure 5.4: Transformation of a ternary association applying the Relationship Object Pattern.

All associations which occur in the Relationship Object Pattern in Fig. 5.4 are navigable in only one direction. Therefore, they can be implemented as properties in the class which has access to the class associated to the navigable end. In the following transformations, instead of one way navigable associations, we use properties for accessing the opposite ends and omit the creation of associations.

#### Creating Implementation Classes

The top-level relation `Associations` of Listing 5.17 mixes two separate relations for the transformation of associations and for the transformation of association classes, which are separately included in Listing B.9 and Listing B.10 of Appendix B.1.2. For brevity, both transformations are merged into a common representation covering transformation of associations and association classes. The parts only applicable for either associations or association classes are separated in sections enclosed by green bars. The green bars are labeled to indicate whether the enclosed content refers to association or association class transformation.

```
top relation Associations
{
                                                                    if isAssociation
  checkonly domain source srcAssociation : Association {};
                                                              else if isAssociationClass
  checkonly domain source srcAssociation : AssociationClass {};
                                                                                    fi

  enforce domain target trgPackage : Package {
    name = 'PSM',
    packagedElement = trgLinkClass : Class {
      name = srcAssociation.name,
      ownedOperation = trgLinkClassConstructor : Operation {
        name = srcAssociation.name,
        visibility = VisibilityKind::public }},
                                                                  if isAssociationClass
    packagedElement = trgLinkImplClass : Class {
      name = srcAssociation.name + 'Impl',
      ownedOperation = trgLinkImplClassConstructor : Operation {
        name = srcAssociation.name + 'Impl',
        visibility = VisibilityKind::public }},
                                                                                    fi
    packagedElement = trgManagerClass : Class {
      name = 'LinkManager_' + srcAssociation.name,
      ownedOperation = trgManagerClassConstructor : Operation {
        name = 'LinkManager_' + srcAssociation.name,
        visibility = VisibilityKind::private },
      ownedOperation = trgCreateLinkOperation : Operation {
        name = 'createLink',
                                                                  if isAssociationClass
        type = trgLinkClass,
                                                                                    fi
        visibility = VisibilityKind::public },
        ownedOperation = trgRemoveLinkOperation : Operation {
        name = 'removeLink',
        visibility = VisibilityKind::public },
      ownedOperation = trgGetLinkOperation : Operation {
        name = 'getLinks',
        visibility = VisibilityKind::public }}};

  where {
    A_LinkClass_Comment('Association(Class)', trgLinkClass)↗B.24;
    A_ManagerClass_Comment('Association(Class)', trgManagerClass)↗B.25;
    A_MemberClass_Comment(
        'Association(Class)', srcAssociation, trgPackage)↗B.26;
    A_MemberClass_Property_LinkManager(
        trgManagerClass, srcAssociation, trgPackage, trgPackage)↗5.18;
    A_LinkClass(srcAssociation, trgLinkClass)↗5.19;
    A_LinkClass_Constructor(srcAssociation, trgLinkClassConstructor)↗5.20;
                                                                  if isAssociationClass
    A_LinkClass_Constructor(srcAssociation,
                trgLinkImplClassConstructor)↗5.20;
                                                                                    fi
    A_ManagerClass(trgLinkClass, srcAssociation, trgManagerClass,
        trgCreateLinkOperation, trgRemoveLinkOperation,
        trgGetLinkOperation)↗5.21;
  }
}
```

Listing 5.17: General transformation of associations and association classes.

Relation `Associations` performs the following transformation steps:

- It matches each association (line 4) or association class (line 6) in the source model.
- For each source association, two classes are created in the target package, one class representing links (lines 11–15) and another class managing the link instances (lines 23–39). Both classes are named and are provided with a constructor, which — in the style of Java — is an operation named like the containing class.
- If the matched association is an association class, an implementation class must be created for the link class (lines 17–21). The constructor of the link class is public in order to allow the manager class to create links whereas the constructor of the manager class itself is private, since the manager class is designed to be a singleton [29].
- Operations for working with links are enforced in lines 25–39. If transforming an association class, the operation `createLink` must return the created link instance, therefore, a return type is specified for this operation in line 31.

In the **`where`** clause, relations are called which additionally add comments to the created classes and to the member end classes of the association (lines 42–45). When transforming association classes, the string parameter must read `'AssociationClass'`, otherwise, it must read `'Association'`. These comments are necessary in order to be able to identify the classes as parts of the implementation pattern of associations and association classes when generating code from the target model.

The remaining relations, which are described in detail later, adapt the classes participating in the association to the new structure and complete the link class as well as the manager class:

- Relation `A_MemberClass_Property_LinkManager` (line 47) adapts member end classes to the new structure: it creates a property which may hold a reference to the manager class.
- The link class is completed by two relations:
  - Relation `A_LinkClass` (line 48) adds a property to the link class for each association end. By that, instances of the link class are able to represent a link which holds a value for each associated instance.
  - Relation `A_LinkClass_Constructor` (line 49) adds a parameter for each association end to the constructor of a link class in order to allow for the creation of a link which is immediately supplied with the instances to be linked.
  - If an association class is transformed, the constructor of the link implementation class (which is not needed for plain associations) must be supplied with parameters for each association end, too. Therefore, relation `A_LinkClass_Constructor` is included twice but for the second time, with the link implementation class as the second parameter (line 52).
- Finally, the operations created in lines 25–39 must be completed by adding parameters, return types and by including information about upper and lower bounds. For this purpose, relation `A_Manager_Class` (line 56) is included.

### Providing Member End Classes with Access to Link Managers

Listing 5.18 contains relation `A_MemberClass_Property_LinkManager` which adds a reference to the manager class into each member end class:

- the manager class is referred to by the primitive domain *trgManagerClass* (line 3)
- each association end is matched in line 6
- each member end class is matched in lines 9–10
- in each matched member end class, a privately owned, appropriately named and typed attribute is created (lines 15–21).

The attribute for accessing the manager class is added to member end classes even if all opposite ends are not navigable, since non-navigability does not preclude the creation of links. Therefore, regardless of the navigability of opposite ends, in order to be able to create new links, each member must be able to access a manager class of an association.

The reference to the manager classes is static (line 19), since the manager class is implemented as a singleton. Simply because only one instance of a singleton class exists, all instances of a class accessing it may share this single instance.

```
1  relation A_MemberClass_Property_LinkManager
2  {
3    primitive domain trgManagerClass : Class;
4
5    checkonly domain source srcAssociation : Association {
6      memberEnd = srcMemberEnd : Property {}};
7
8    checkonly domain source srcPackage : Package {
9      packagedElement = srcMemberClass : Class {
10       name = srcMemberEnd.type.name }};
11
12   enforce domain target trgPackage : Package {
13     packagedElement = trgMemberClass : Class {
14       name = srcMemberEnd.type.name,
15       ownedAttribute = trgManagerClassProperty : Property {
16         name = trgManagerClass.name.firstToLower(),
17         type = trgManagerClass,
18         visibility = VisibilityKind::private,
19         isStatic = true,
20         defaultValue = trgPropertyValue : LiteralString {
21           value = 'null' }}}};
22
23   when {
24     srcMemberClass.ownedAttribute.oclIsUndefined()
25     or not srcMemberClass.ownedAttribute->exists(
26         p: Property | p.type = trgManagerClass; }
27
28   where {
29     A_MemberClass_Property_LinkManager_Comment(trgManagerClassProperty)↗B.27;
30   }
31 }
```

Listing 5.18: Linking member end classes to a link manager class.

In the **when** clause, it is checked that either the member end class does not have any owned attributes at all (line 24) or that no property exists the type of which is the manager class (line 26). By that, a multiple creation of a property for referencing a manager class is avoided. Such a multiple property creation would occur in reflexive associations, i. e. if a class participates in an association more than once in different roles.

In the **where** clause, additional information is tagged to the created attribute in order to be able to recognize its special semantics when generating code from the target model.

### Adding Attributes to Link Implementations

Relation `A_LinkClass` contained in Listing 5.19 completes the class implementing an association link. It matches each end of an association (line 4) and adds a privately owned attribute for each matched association end to the link class (lines 7–11).

Only reading access is provided by creating an operation for each end returning the value of the end (lines 12–18).

In the **where** clause, a string is tagged to the access operation indicating that its purpose is granting reading access to an attribute.

```
1  relation A_LinkClass
2  {
3    checkonly domain source srcAssociation : Association {
4      memberEnd = srcMemberEnd : Property {}};
5
6    enforce domain target trgLinkClass : Class {
7      ownedAttribute = trgProperty : Property {
8        name = roleNameFirstLower(srcMemberEnd)↗B.34,
9        type = trgPropertyType : Class {
10         name = srcMemberEnd.type.name },
11       visibility = VisibilityKind::private },
12     ownedOperation = trgGetter : Operation {
13       name = 'get' + roleNameFirstUpper(srcMemberEnd)↗B.35,
14       ownedParameter = trgGetterParameter : Parameter {
15         direction = ParameterDirectionKind::return,
16         type = trgGetterParameterType : Class {
17           name = srcMemberEnd.type.name }},
18       visibility = VisibilityKind::public }};
19
20   where {
21     SingleClass_PropertyAccessOperation_Comment(
22         'Getter', roleNameFirstLower(srcMemberEnd)↗B.34, trgGetter)↗B.28;
23   }
24 }
```

Listing 5.19: Adapting a link class to the structure of an association.

### Supplying Link Constructors with Parameters

Links have attributes each holding the value of one association end. Since links cannot be changed, attributes can only be accessed for reading. Values must be provided upon the creation of a link. Therefore, relation `A_LinkClass_Constructor` given in Listing 5.20 adds parameters to the constructor of a link class.

It matches each association end (line 4) and enforces the existence of a parameter for each end (lines 7–10).

```
1  relation A_LinkClass_Constructor
2  {
3    checkonly domain source srcAssociation : Association {
4      memberEnd = srcMemberEnd : Property {}};
5
6    enforce domain target trgOperation : Operation {
7      ownedParameter = trgInParameter : Parameter {
8        name = roleNameFirstLower(srcMemberEnd)↗B.34,
9        type = trgParameterType : Class {
10         name = srcMemberEnd.type.name }}};
11 }
```

Listing 5.20: Supplying a link class constructor with parameters.

### Creating Operations for Managing Links

The last step in the transformations of associations is to complete the operations actually supporting to create, to remove, and to query links. These operations have been created within the relation `Associations` of Listing 5.17.

Relation `A_ManagerClass` of Listing 5.21 performs the following transformation steps:

- it matches each end of an association (line 6)
- it creates two attributes intended to hold the lower (line 9) and upper (line 10) bounds of the currently matched end
- it creates operations for
  - creating new links (line 12)
  - removing existing links (line 18)
  - reading the association, i. e. accessing link values (line 24)
- for each such created operation, this relation provides an parameter for each association end which is named and typed like the currently matched end (lines 13–16, lines 19–22, and lines 25–28). A parameter for each end is needed to specify the values for a new link, to identify a link to be destroyed, or to identify the context, if a navigable end is read.

```
1  relation A_ManagerClass
2  {
3    primitive domain trgLinkClass : Class;
4
5    checkonly domain source srcAssociation : Association {
6      memberEnd = srcMemberEnd : Property {}};
7
8    enforce domain target trgManagerClass : Class {
9      ownedAttribute = trgLowerProperty : Property {},
10     ownedAttribute = trgUpperProperty : Property {}};
11
12   enforce domain target trgCreateLinkOperation : Operation {
13     ownedParameter = trgCreateLinkInParameter : Parameter {
14       name = roleNameFirstLower(srcMemberEnd)↗B.34,
15       type = trgCreateLinkParameterType : Class {
16         name = srcMemberEnd.type.name }}};
17
18   enforce domain target trgRemoveLinkOperation : Operation {
19     ownedParameter = trgRemoveLinkInParameter : Parameter {
20       name = roleNameFirstLower(srcMemberEnd)↗B.34,
21       type = trgRemoveLinkParameterType : Class {
22         name = srcMemberEnd.type.name }}};
23
24   enforce domain target trgGetLinkOperation : Operation {
25     ownedParameter = trgGetLinkInParameter : Parameter {
26       name = roleNameFirstLower(srcMemberEnd)↗B.34,
27       type = trgGetLinkParameterType : Class {
28         name = srcMemberEnd.type.name }}};
29
30   where {
31     A_ManagerClass_LowerProperty(srcMemberEnd, trgLowerProperty)↗5.22;
32     A_ManagerClass_UpperProperty(srcMemberEnd, trgUpperProperty)↗5.23;
33     A_ManagerClass_GetLinkOperation(trgLinkClass, trgGetLinkOperation)↗5.24;
34   }
35 }
```

Listing 5.21: Adapting a link manager class to the structure of an association.

In the **where** clause, two relations are called which complete the specification of the attribute holding the value of the lower bound (or upper bound respectively) of the currently matched end (line 31, 32 respectively).

A third relation supplies the *getLinks* operation with a return parameter (line 33).

Relation `A_ManagerClass_LowerProperty` (Listing 5.22) supplies the owned attribute intended to represent the lower bound of an association end with further details:

- a name derived from the word *lower* and the role name of the concerning end.
- a type, which is the primitive type `integer`
- a default value which value specified as the end's lower bound.

The attribute furthermore is specified to be private, read-only, and static.

```
1  relation A_ManagerClass_LowerProperty
2  {
3    primitive domain srcMemberEnd : Property;
4
5    enforce domain target trgLowerProperty : Property {
6      name = 'lower' + roleNameFirstUpper(srcMemberEnd)↗B.35,
7      type = trgPropertyType : PrimitiveType  {
8        name = 'Integer' },
9      visibility = VisibilityKind::private,
10     isStatic = true,
11     isReadOnly = true,
12     defaultValue = trgPropertyValue : LiteralString {
13       value = propertyLowerValue(srcMemberEnd)↗B.41 }};
14 }
```

Listing 5.22: Supplying an attribute representing a lower bound with a concrete value.

Relation `A_ManagerClass_UpperProperty` (Listing 5.23) is almost identical to relation `A_ManagerClass_LowerProperty` (Listing 5.22). The specification of the name and the default value of the attribute is adapted to comply to the upper bound of the concerning association end.

```
1  relation A_ManagerClass_UpperProperty
2  {
3    primitive domain srcMemberEnd : Property;
4
5    enforce domain target trgUpperProperty : Property {
6      name = 'upper' + roleNameFirstUpper(srcMemberEnd)↗B.35,
7      type = trgPropertyType : PrimitiveType {
8        name = 'Integer' },
9      visibility = VisibilityKind::private,
10     isStatic = true,
11     isReadOnly = true,
12     defaultValue = trgPropertyValue : LiteralString {
13       value = propertyUpperValue(srcMemberEnd)↗B.42 }};
14 }
```

Listing 5.23: Supplying an attribute representing an upper bound with a concrete value..

Relation `A_ManagerClass_GetLinkOperation` (Listing 5.24) adds a return parameter to the operation for reading the association, i. e. for the operation retrieving links. The return type of this parameter is the link class and its multiplicity ranges from 0 (since a request might result in no links to be returned) to unlimited (specified by $-1$, since any number of links possibly is to be returned).

```
1  relation A_ManagerClass_GetLinkOperation
2  {
3    primitive domain trgLinkClass : Class;
4
5    enforce domain target trgGetLinkOperation : Operation {
6      ownedParameter = trgReturnParameter : Parameter {
7        direction = ParameterDirectionKind::return,
8        type = trgLinkClass,
9        lowerValue = trgParameterLowerValue : LiteralInteger {
10         value = 0 },
11       upperValue = trgParameterUpperValue : LiteralUnlimitedNatural {
12         value = -1 }}};
13 }
```

Listing 5.24: Specifying the return value of an operation for querying links.

### 5.4.3 Modifying Member End Classes

According to the specification, adding or removing links has an impact on the state of instances of the owning classifier. Hence if an end is owned by an association, adaptations to the member end class are not necessary. But if an end owned by an association is *navigable*, access is granted to member end classes at opposite ends.

Listing 5.25 contains relation `A_MemberClass` for preparing operations for write access in each member end class:

- association ends are matched (line 4)
- an operation for adding a link is enforced (line 10)
- another operation for removing a link (line 11) is created.

```
1  top relation A_MemberClass
2  {
3    checkonly domain source srcAssociation : Association {
4      memberEnd = srcMemberEnd : Property {}};
5
6    enforce domain target trgPackage : Package {
7      name = 'PSM',
8      packagedElement = trgMemberClass : Class {
9        name = srcMemberEnd.type.name,
10       ownedOperation = trgAddOperation : Operation {},
11       ownedOperation = trgRemoveOperation : Operation {}}};
12
13   where {
14     A_MemberClass_AddRemoveOperation(
15         'add', srcMemberEnd, srcAssociation, trgAddOperation)↗5.26;
16     AC_MemberClass_AddRemoveOperation(
17         'add', srcMemberEnd, srcAssociation, trgAddOperation)↗B.11;
18     A_MemberClass_AddRemoveOperation(
19         'remove', srcMemberEnd, srcAssociation, trgRemoveOperation)↗5.26;
20     A_MemberClass_Prepare_GetOperation(
21         srcMemberEnd, srcAssociation, trgMemberClass)↗5.27;
22   }
23 }
```

Listing 5.25: Preparing operations for write access in member end classes.

In the `where` clause of relation `A_MemberClass`, relations are called which create parameters of accessor operations within member end classes. Two of those relations add parameters to the *add* operation of a member end class. The first one (line 15) only matches associations, the second

one (line 17) only matches association classes. For association classes, the operation for adding a new link requires an additional result parameter returning the created link instance. The former relation is printed in Listing 5.26, the latter one in Listing B.11 of Appendix B.1.2.

For the specification of parameters of a *remove* operation, which is also covered by the relation of Listing 5.26, but called with a different operation name prefix, no distinction between associations and association classes is needed.

For *get* operations, which are prepared by relation `A_MemberClass_Prepare_GetOperation` (included in line 20), such a distinction is not needed either. Preparation of *get* operations by relation `A_MemberClass_Prepare_GetOperation` is contained in Listing 5.27.

```
1  relation A_MemberClass_AddRemoveOperation
2  {
3    primitive domain namePrefix : String;
4    primitive domain srcClassMemberEnd : Property;
5
6    checkonly domain source srcAssociation : Association {
7      memberEnd = srcMemberEnd : Property {}};
8
9    enforce domain target trgOperation : Operation {
10     name = namePrefix + operationName(
11         srcAssociation.memberEnd->excluding(srcClassMemberEnd))↗B.36,
12     ownedParameter = trgInParameter : Parameter {
13       name = roleNameFirstLower(srcMemberEnd)↗B.34,
14       type = trgParameterType : Class {
15         name = srcMemberEnd.type.name }},
16     visibility = minimalImplVisibility(
17         srcAssociation.memberEnd->excluding(srcClassMemberEnd))↗B.43 };
18
19   when {
20     namePrefix<>'add' or not srcAssociation.oclIsKindOf(AssociationClass);
21     srcMemberEnd<>srcClassMemberEnd;
22   }
23
24   where {
25     Association_MemberClass_AddRemoveOperation_Comment(
26       namePrefix, 'Association',
27       'LinkManager'+associationName(srcAssociation)↗B.38, trgOperation)↗B.29;
28   }
29 }
```

Listing 5.26: Specification of parameters for accessor operations in member end classes.

*Add* and *remove* operations in member end classes must contain a parameter for each association end except the end to which the member end class itself is connected. In line 7 of Listing 5.26, each association end is matched, in the **when** clause in line 21, it is assured that the relation does not process the end to which the current member end class is connected. In line 20, it is assured that the relation does not transform operations for adding new links for association classes.

The operation is named (line 10) by appending all association ends' names to a prefix already bound by the calling relation (line 3). The visibility of the operation is determined by the most restrictive visibility specification of any end (line 16). Similar to private attributes, if an end is specified with private visibility, accessor operations have protected visibility.

In order to avoid accessibility in all subclasses, accessor methods for private association ends must be overwritten in specializations, if any. However, this is not included in the presented transformation yet.

If accessing association ends via member end classes is not desired for ends owned by the association, a relation must be included in the **when** clause evaluating to false if no operations are to be created for a source member end. Remember that in this case, higher order associations cannot be accessed via member end classes because ends must be owned by the association.

Finally, in the **where** clause of relation `A_MemberClass_AddRemoveOperation`, a relation is included which tags a comment to each transformed operation in order to identify the intended semantics when generating code from the target model.

Listing 5.27 shows the relation for preparing a *get* operation for association ends in a member end class. A classifier must own such an operation if at least one navigable end connected to another classifier exists. This is achieved in the following steps:

- each association end is matched (line 6)
- it is checked that the currently matched end is navigable (lines 12–13),
- it is ensured that the currently matched navigable end is an opposite end to that end, to which the classifier which owns the add operation is connected (line 14),
- a duplicate creation of a *get* operation is avoided (lines 15–16).

```
1  relation A_MemberClass_Prepare_GetOperation
2  {
3    primitive domain srcClassMemberEnd : Property;
4
5    checkonly domain source srcAssociation : Association {
6      memberEnd = srcMemberEnd : Property {}};
7
8    enforce domain target trgMemberClass : Class {
9      ownedOperation = trgOperation : Operation {}};
10
11   when {
12     srcAssociation.navigableOwnedEnd->exists(
13       p : Property | p = srcMemberEnd);
14     srcMemberEnd <> srcClassMemberEnd;
15     not trgMemberClass.ownedOperation->exists(
16       o : Operation | o.name = 'get'+roleNameFirstUpper(srcMemberEnd))↗B.35;
17   }
18
19   where {
20     A_MemberClass_GetOperation(
21         srcMemberEnd, srcClassMemberEnd, srcAssociation, trgOperation)↗5.28;
22     A_MemberClass_GetOperation_Parameter(
23         srcMemberEnd, srcClassMemberEnd, srcAssociation, trgOperation)↗5.29;
24   }
25 }
```

Listing 5.27: Preparing an operation for reading ends of a higher order association.

Relation `A_MemberClass_Prepare_GetOperation` of Listing 5.27 only enforces an operation without specifying the values for any further characteristics. The specification of further details is covered by two relations contained in the **where** clause.

The first of both relations, relation `A_MemberClass_GetOperation` is shown in Listing 5.28. It specifies the following characteristics of the add operation:

- the name of the operation (lines 8–9)
- the return parameter (lines 10–17) with appropriate type (lines 12–13) and multiplicity bounds according to those of the end to be read (lines 14–17)
- the visibility according to the end (line 18).

In the **where** clause, relations are included which tag information to the operation needed to identify the operation as part of the implementation pattern of associations when generating code from the target model.

The second relation is given by Listing 5.29. Relation `A_MemberClass_GetOperation_Parameter` adds parameters to the operation. Additional parameters are necessary to specify the context,

```
1  relation A_MemberClass_GetOperation
2  {
3    primitive domain srcMemberEnd : Property;
4    primitive domain srcClassMemberEnd : Property;
5    primitive domain srcAssociation : Association;
6
7    enforce domain target trgOperation : Operation {
8      name = 'get' + roleNameFirstUpper(srcMemberEnd)↗B.35 +
9        operationSuffix(srcMemberEnd, srcClassMemberEnd, srcAssociation)↗B.40,
10     ownedParameter = trgReturnParameter : Parameter {
11       direction = ParameterDirectionKind::return,
12       type = trgParameterType : Class {
13         name = srcMemberEnd.type.name },
14       lowerValue = trgParameterLowerValue : LiteralInteger {
15         value = srcMemberEnd.lower },
16       upperValue = trgParameterUpperValue : LiteralUnlimitedNatural {
17         value = srcMemberEnd.upper }},
18     visibility = srcMemberEnd.visibility };
19
20   where {
21     A_MemberClass_GetOperation_Comment(srcAssociation, trgOperation)↗B.30;
22     AC_MemberClass_GetOperation_Comment(srcAssociation, trgOperation)↗B.30;
23   }
24 }
```

Listing 5.28: Specifying an operation for reading association ends.

for which an association end is to be accessed, i. e. , the values of all other ends. For binary associations, the context is implicitly defined by the instance on which a get operation is called. Therefore, the relation must not process binary associations. This is achieved by the first condition of the **when** clause (line 16).

The remaining two conditions in the **when** clause determine for which association ends, a parameter is added to the *get* operation currently processed: For higher order associations, when reading an association end, values must be provided for those ends building the context, i. e. all ends except the end to be read (line 17). The end, to which the classifier owning the operation is connected should not be included as a parameter either (line 18), otherwise, it would be necessary to pass the instance, on which the operation is invoked, as a parameter.

```
1  relation A_MemberClass_GetOperation_Parameter
2  {
3    primitive domain srcOperationMemberEnd : Property;
4    primitive domain srcClassMemberEnd : Property;
5
6    checkonly domain source srcAssociation : Association {
7      memberEnd = srcMemberEnd : Property {}};
8
9    enforce domain target trgOperation : Operation {
10     ownedParameter = trgInParameter : Parameter {
11       name = roleNameFirstLower(srcMemberEnd)↗B.34,
12       type = trgParameterType : Class {
13         name = srcMemberEnd.type.name }}};
14
15   when {
16     srcAssociation.memberEnd->size() > 2;
17     srcMemberEnd <> srcOperationMemberEnd;
18     srcMemberEnd <> srcClassMemberEnd; }
19 }
```

Listing 5.29: Supplying an operation for reading association ends with parameters.

For all other ends of the association, a parameter is added to the get operation. Each parameter is named like the role name of the concerning end. The type is identified by the name of the classifier connected to the concerning end.

### 5.4.4 Transformation of Compositions

The implementation of composition as proposed in Sect. 3.6.3 necessitates that a manager class implementing a composition can access manager classes of other associations in which parts of the composition participate. This access is made explicit by inserting an association between concerned manager classes. Listing 5.30 shows a relation for this purpose. This relation

- matches each composition, i. e. each association of the source model of which an end's aggregation kind is *composite* (lines 3–6)
- matches all associations without further restrictions (line 8)
- matches the manager class of the composition (lines 12–13)
- matches the manager class of the other association (lines 14–15)
- enforces an association between the manager class of the composition and the manager class of the association (lines 16–18).

In the `when` clause, it is ensured that the matched association is not identical to the matched composition (line 22) and that the matched association has an end connected to the classifier which is a part of the composition (lines 23–25).

```
1  top relation Composition
2  {
3    checkonly domain source srcCompositionAssociation : Association {
4      memberEnd = srcCompositeMemberEnd : Property {
5        aggregation = AggregationKind::composite },
6      memberEnd = srcOtherMemberEnd : Property {}};
7
8    checkonly domain source srcOtherAssociation : Association {};
9
10   enforce domain target trgPackage : Package {
11     name = 'PSM',
12     packagedElement = trgCompositionManagerClass : Class {
13       name = 'LinkManager'+associationName(srcCompositionAssociation)↗B.38 },
14     packagedElement = trgOtherManagerClass : Class {
15       name = 'LinkManager'+associationName(srcOtherAssociation)↗B.38 },
16     packagedElement = trgManagerAssociation : Association {
17       name = trgCompositionManagerClass.name + '__' +
18              trgOtherManagerClass.name }};
19
20   when {
21     srcCompositionAssociation.memberEnd->size() = 2;
22     srcCompositionAssociation <> srcOtherAssociation;
23     srcOtherAssociation.memberEnd->exists(
24         p : Property | p.type = srcCompositeMemberEnd.type
25                      or p.type = srcOtherMemberEnd.type); }
26
27   where {
28     Composition_ClassRelation_Multiplicity(
29         trgCompositionManagerClass,
30         trgOtherManagerClass,
31         trgManagerAssociation)↗5.31; }
32 }
```

Listing 5.30: Associating a manager class of a composition to manager classes of associations.

In the **where** clause, a relation is called which adds navigable owned ends to the association between the manager class of a composition and manager classes of associations in which the parts of the composition participate.

This relation is given in Listing 5.31. In this relation, the manager class of a composition and the manager class of an association in which a part of the composition participates are bound as primitive domains.

Then, for both manager classes, an association end with appropriate type is created. The upper and lower bound of both ends are 1, visibility of both ends is private.

```
relation Composition_ClassRelation_Multiplicity
{
  primitive domain trgMemberEndClass1 : Class;
  primitive domain trgMemberEndClass2 : Class;

  enforce domain target trgAssociation : Association {
    navigableOwnedEnd = trgOwnedEnd1 : Property {
      type = trgMemberEndClass1,
      lowerValue = trgOwnedEnd1LowerValue : LiteralInteger {
        value = 1 },
      upperValue = trgOwnedEnd1UpperValue : LiteralUnlimitedNatural {
        value = 1 },
      visibility = VisibilityKind::private },
    navigableOwnedEnd = trgOwnedEnd2 : Property {
      type = trgMemberEndClass2,
      lowerValue = trgOwnedEnd2LowerValue : LiteralInteger {
        value = 1 },
      upperValue = trgOwnedEnd2UpperValue : LiteralUnlimitedNatural {
        value = 1 },
      visibility = VisibilityKind::private }};
}
```

Listing 5.31: Supplying ends of associations between manager classes with values.

## 5.5 Transforming Models of Dynamic Behavior

The transformation of activities differs from that of static structure in the number of supported meta-classes, i. e. modeling elements. For static structure, the specified transformations support *Package*, *Class*, *Association*, *AssociationClass*, *Property*, *Operation*, *Parameter*, *ParameterDirectionKind*, and *VisibilityKind*. In contrast, the transformation of activities requires to support much more meta-classes, namely seven kinds of *ControlNodes*, *Activity* and *InterruptibleActivityRegion*, all supported kinds of *Action*[2], four kinds of *ObjectNode* as well as *ControlFlow* and *ObjectFlow*.

The large number of concerned meta-classes is problematic for the creation of the target model. In **checkonly** domains, which typically refer to the source model, it is possible to use abstract types for matching a set of concrete type instances, but in **enforce** domains which are used to create instances in the target model, abstract types cannot be used. Instead of dynamically choosing the type of enforced elements, a distinct relation must be provided for each considered type.

Alternatively, the source activity can be copied into the target model where it is changed in the run of the transformation.

In order to avoid references from the target model into the source model, all references from the copied activity to model elements that are not copied because they are not part of the activity must be updated to reference elements of the target model. This is a purely technical aspect for which relations are listed in Appendix B.2, Listing B.49 and Listing B.50.

The first step of the transformation of behavior is to include each activity of the source model into the target model. In UML, other kinds of behavior exist, such as *StateMachine*, however, we support the full transformation of a model only if it solely contains behaviors of type *Activity*.

[2]UML defines 34 different kinds of action.

Relation `Activity` of Listing 5.32

- matches each class and each of its owned behaviors (lines 3–4) in the source model
- matches to each source class the conceptual class in the target model (lines 6–9)
- moves the matched activity into the set of owned behaviors of the conceptual class in the target model (line 10).
- creates an operation containing the behavior as its implementation[3] (lines 11–13).

In the **where** clause, parameters of the activity are copied to this operation.

```
1  top relation Activity
2  {
3    checkonly domain source srcClass : Class {
4      ownedBehavior = srcActivity : Activity {}};
5
6    enforce domain target trgPackage : Package {
7      name = 'PSM',
8      packagedElement = trgClass : Class {
9        name = srcClass.name,
10       ownedBehavior = trgClass.ownedBehavior->prepend(srcActivity),
11       ownedOperation = trgOperation : Operation {
12         name = 'run' + srcActivity.name.firstToUpper(),
13         method = OrderedSet{trgClass.ownedBehavior->at(1)} }}};
14
15   where {
16     Copy_Activity_Parameter(srcActivity, trgOperation)↗B.51;
17     Activity_CallBehaviorAction(trgClass,
18         trgClass.ownedBehavior->at(1),
19         trgClass.ownedBehavior->at(1))↗5.34;  }
20 }
```

Listing 5.32: Moving activities from the source model into the target model.

Another relation called in the **where** clause extracts sequences of actions which are moved each into its own activity. Afterwards, sequences of actions are replaced by call behavior actions which call the activities containing the sequences.

### 5.5.1 Extraction of Action Sequences

We introduced *sequences of actions* in Sect. 4.2.3. Figure 5.5 (which is identical to Fig. 4.30 in Sect. 4.2.11) shows the transformation from activity $A$ into $A^t$, which is the extraction of sequences and their inclusion in the original behavior by means of *CallBehaviorAction*.

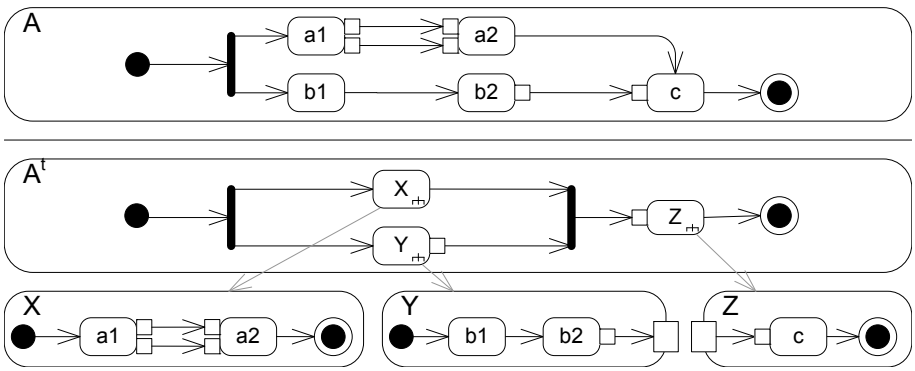



Figure 5.5: Extraction of sequences of actions from an activity [39].


[3]The meta-attribute *method* may contain a behavior that serves as the behavioral specification of an operation.

Here, we formally specify how to determine sequences of actions. The general idea is as follows:

- A sequence of actions is an ordered set of actions. The order represents the sequencing of executions. The first action of a sequence, we term the *head*.
- If all outgoing edges of a head action reach the same target action, and the target action has no other incoming flow, then this target is included in the sequence of actions, and we term such an action *flow body action*.
- If a flow body action is the head of another sequence, it becomes a flow body action of the concatenation of both sequences.

Listing 5.33 is a straightforward specification for checking an action being a flow body action. Note that this relation does not modify the target model. It is needed in another relation to determine whether an action is to be processed or not.

Given an action (line 6) and an edge (line 7) contained in the same activity (line 5 ), then a given action bound in the primitive domain (line 3) is a flow body action, if

- for any pair of the given and another action (line 10),
  for which an edge exists that starts at the other action (line 11)
  and ends at the given action (line 12)
- no other edge exists (line 13) which
  is an incoming edge of the given action not starting at the other action (line 14)
  or is an outgoing edge of the other action not ending at the given action (line 15).

In case of object flow, the criterion is that output pins are attached to the source of the flow, i. e. the *other* action and input pins are attached to the flow target, i. e. the *given* action.

```
1  top relation Activity_IsFlowBodyAction
2  {
3    checkonly domain source srcAction : Action {};
4
5    checkonly domain source srcActivity : Activity {
6      node = srcOtherAction : Action {},
7      edge = srcEdge : ActivityEdge {}};
8
9    when {
10     srcAction <> srcOtherAction;
11     srcEdge.source = srcOtherAction and
12     srcEdge.target = srcAction;
13     not srcActivity.edge->excluding(srcEdge)->exists( a : ActivityEdge |
14       (a.target = srcAction and a.source <> srcOtherAction)
15       or (a.target <> srcAction and a.source = srcOtherAction));  }
16 }
```

Listing 5.33: Checking actions for being part of a sequence.

In order to extract sequences of actions, a relation must identify these sequences and move them into subactivities. Such a relation is given in Listing 5.34. Relation *Activity_CallBehaviorAction* matches each node of type *Action* of an activity. By means of the relation of Listing 5.33, which is included in the `when` clause, it is ensured that the action is no flow body action. Since an action that is no flow body action can only be the head of a sequence of actions, the extraction of this sequence is prepared:

- a call behavior action is created (lines 9–11) as well as an activity owned by this call behavior action (lines 12–26)
- an initial node and an activity final node are created and added to the nodes owned by the new activity (lines 15–16 and lines 17–18)
- activity edges from the initial node to the head action and from the head action to the final node are inserted and labeled with names (lines 19–22 and lines 23–26).

```
1  relation Activity_CallBehaviorAction
2  {
3    primitive domain trgClass : Class;
4
5    checkonly domain source srcActivity : Activity {
6      node = srcAction : Action {}};
7
8    enforce domain target trgActivity : Activity {
9      node = trgCallBehaviorAction : CallBehaviorAction {
10       name = 'CallBehaviorAction_' + srcActivity.name +
11           '_Subactivity_StartsWith_' + srcAction.name,
12       behavior = trgSubactivity : Activity {
13         name = srcActivity.name +
14             '_Subactivity_StartsWith_' + srcAction.name,
15         node = trgInitialNode : InitialNode {
16           name = 'Init' },
17         node = trgFinalNode : ActivityFinalNode  {
18           name = 'End' },
19         edge = trgInitialEdge : ControlFlow {
20           name = 'Init_' + srcAction.name,
21           source = trgInitialNode,
22           target = srcAction },
23         edge = trgFinalEdge : ControlFlow {
24           name = srcAction.name + '_End',
25           source = srcAction,
26           target = trgFinalNode }}}};
27
28   when {
29     not Activity_IsFlowBodyAction(srcAction, srcActivity)↗5.33; }
30
31   where {
32     Activity_Subactivity(
33         srcAction, srcActivity, trgActivity, trgFinalEdge, trgClass)↗5.35;
34     Activity_CallBehaviorAction_Incoming(
35         trgCallBehaviorAction, srcAction)↗5.37;
36     Activity_CallBehaviorAction_Outgoing(
37         trgCallBehaviorAction, srcAction, srcActivity)↗5.39; }
38 }
```

Listing 5.34: Preparing subactivities for sequences of actions.

So far, a new activity has been created with an initial node and an activity final node. Flows from the initial node to the head action as well as from the head action to the activity final node have been inserted. In order to complete the extraction of sequences, three things must be done:

- all actions of the same sequence of actions must be included in the new activity as well as incoming edges of those actions
- each flow ending at the first action of a sequence must be updated in order to end at the call behavior action which replaces the sequence in the original activity
- each flow starting at the last action of a sequence must be updated in order to start at the call behavior action which replaces the sequence in the original activity.

These modifications are achieved by performing the relations in the **where** clause of *Activity_CallBehaviorAction*.

Relation *Activity_Subactivity* of Listing 5.35 is called in order to actually move all actions of a sequence into a newly created subactivity. The first action of the sequence, the source and target activity, the final edge from the new subactivity and the owning class are passed as arguments. The relation matches the given class (line 8) and the new subactivity by name (lines 9–10), adds the given action to the set of owned actions (line 11) and evaluates relation *Activity_FlowBodyAction* of Listing 5.36. All arguments are passed through to this relation.

```
1  relation Activity_Subactivity
2  {
3    primitive domain srcAction : Action;
4    primitive domain srcActivity : Activity;
5    primitive domain trgActivity : Activity;
6    primitive domain trgFinalEdge : ControlFlow;
7
8    enforce domain target trgClass : Class {
9      ownedBehavior = trgSubactivity : Activity {
10       name = srcActivity.name+'_Subactivity_StartsWith_'+srcAction.name,
11       node = trgSubactivity.node->including(srcAction) }};
12
13   where {
14     Activity_FlowBodyAction(
15         srcAction, srcActivity, trgSubactivity, trgFinalEdge)↗5.36; }
16 }
```

Listing 5.35: Moving flow body actions into subactivities.

In relation *Activity_FlowBodyAction* (Listing 5.36), downstream actions of a sequence are included into subactivities into which such sequences are isolated:

- the action is included into the set of nodes of the subactivity (line 9)
- all incoming edges of the downstream action are included in the set of owned edges of the subactivity (line 10).

After this modification, the action is part of the subactivity as well as all its incoming edges. But now, the origin of the incoming edge of the activity final node must be updated. This is done in line 14 and it is renamed in line 13. In the **when** clause it is checked that actions are only added if they are flow body actions and if they are part of that sequence which is to be included in this subactivity. The query for checking this last condition is given in the Appendix B.2 in Listing B.56.

```
1  relation Activity_FlowBodyAction
2  {
3    primitive domain srcAction : Action;
4
5    checkonly domain source srcActivity : Activity {
6      node = srcOtherAction : Action {}};
7
8    enforce domain target trgActivity : Activity {
9      node = trgActivity.node->including(srcOtherAction),
10     edge = trgActivity.edge->union(srcOtherAction.incoming) };
11
12   enforce domain target trgFinalEdge : ControlFlow {
13     name = srcOtherAction.name + '_End',
14     source = srcOtherAction };
15
16   when {
17     srcAction <> srcOtherAction;
18     Activity_IsFlowBodyAction(srcOtherAction, srcActivity)↗5.33;
19     isSubsequentSubactivityAction(
20         srcOtherAction, srcAction, srcActivity)↗B.56; }
21 }
```

Listing 5.36: Updating the structure of a subactivity.

While moving actions into new activities which are associated to call behavior actions, it is necessary to update flows starting or ending at such moved actions. Relations for updating incoming flows are given in Listing 5.37 and Listing 5.38. For incoming edges, relation *Activity_CallBehaviorAction_Incoming* updates the target of control flows by executing relation *Activity_CallBehaviorAction_Incoming_Change*, which actually replaces the target of the control flow.

If object flows are to be considered, another relation is needed which maps input pins of the first action of a sequence to input pins of the call behavior actions as well as to activity parameter nodes of the behavior associated to the call behavior action. A relation for changing incoming object flows is given in Listing B.54 in Appendix B.2.2. It must be included in the **where** clause which contains relation *Activity_CallBehaviorAction_Incoming*.

```
1  relation Activity_CallBehaviorAction_Incoming
2  {
3    primitive domain trgCallBehaviorAction: CallBehaviorAction;
4
5    checkonly domain source srcAction : Action {
6      incoming =  srcEdge : ControlFlow {}};
7
8    where {
9      Activity_CallBehaviorAction_Incoming_Change(
10         trgCallBehaviorAction, srcEdge)↗5.38; }
11 }
```

Listing 5.37: Identifying flows to head actions.

```
1  relation Activity_CallBehaviorAction_Incoming_Change
2  {
3    primitive domain trgCallBehaviorAction: CallBehaviorAction;
4
5    enforce domain target trgEdge : ControlFlow {
6        target = trgCallBehaviorAction };
7  }
```

Listing 5.38: Updating the target of incoming edges of head actions to call behavior actions associated with subactivities.

For outgoing edges, similar relations are needed. Listing 5.39 matches those edges which must be updated. But whether or not a flow source must be updated is more complex to determine than it is for flow targets.

For a given action *srcAction* and a call behavior action *trgCallBehaviorAction* which is associated to the behavior into which *srcAction* is moved, any edge of activity *srcActivity* which contains *srcAction* before it is moved possibly must be updated. The source of an edge must be updated, if its source is the last action of a sequence. This is the case if

- the source is an action (line 11)
- the target is either not an action (line 12) or, if it is an action, has more than one incoming flow, i. e. there is an implicit join before the next action (line 13) or there is an implicit fork before the next action (line 14)
- provided that the source is not a flow body action (line 15), it is the action to be moved, i. e. the sequence consists of only one action (line 17), **or** it is an action contained in the sequence (line 18).

The actual replacement of the flow source is performed within relation *Activity_CallBehaviorAction_Outgoing_Change* given in Listing 5.40.

If object flows are to be considered, another relation must be included in the **where** clause of relation *Activity_CallBehaviorAction_Outgoing*. A relation updating outgoing object flows is

```
relation Activity_CallBehaviorAction_Outgoing
{
  primitive domain trgCallBehaviorAction: CallBehaviorAction;
  primitive domain srcAction : Action;

  checkonly domain source srcActivity : Activity {
    node = srcNode : ActivityNode {
      incoming =  srcEdge : ControlFlow {}}};

  when {
    srcEdge.source.oclIsKindOf(Action);
    not srcNode.oclIsKindOf(Action)
      or srcNode.incoming->size() > 1
      or srcEdge.source.outgoing->size() > 1;
    not Activity_IsFlowBodyAction(
        srcEdge.source.oclAsType(Action), srcActivity)
      and srcEdge.source = srcAction
      or isSubsequentSubactivityAction(
        srcEdge.source.oclAsType(Action), srcAction, srcActivity); }

  where {
    Activity_CallBehaviorAction_Outgoing_Change(
        trgCallBehaviorAction, srcEdge)↗5.40; }
}
```

Listing 5.39: Identifying outgoing edges of the last flow body action of a sequence.

given in Listing B.55 in Appendix B.2.2. This relation maps the output pins of the last action of a sequence to output pins of the call behavior action which is associated to the activity containing the sequence of actions. In addition, activity parameter nodes are added to the activity to ensure that the inputs of the call behavior action and its associated activity match.

```
relation Activity_CallBehaviorAction_Outgoing_Change
{
  primitive domain trgCallBehaviorAction: CallBehaviorAction;

  enforce domain target trgEdge : ControlFlow {
      source = trgCallBehaviorAction };
}
```

Listing 5.40: Updating the source of an outgoing edge of the last flow body action of a sequence.

### 5.5.2 Limitations of Transforming Behaviors

In the previous section, we provide transformations of behavior which consists of identification of sequences of actions and extraction of such sequences. This is the basic step needed for generating code conforming to the patterns presented in Sect. 4.2.11.

The transformation of behaviors does not create a transformed behavior rather than it is an in-place transformation of a behavior that is moved from the input model into the target model. This is acceptable for a prototype, however, the activity should be copied and modifications should be applied to the copy of the behavior in the target model.

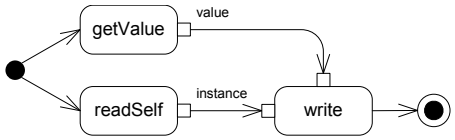
The presented relations cover the basic idea of separating flows which can be implemented as threads. Advanced considerations like propagation of guards or decompositions of complex control flow nets as discussed in Sect. 4.2.5 are not yet considered in our transformation.

Relations identifying sequences of actions and moving them into subactivities are based on control flow analysis. Analysing object flows for this purpose is not presented. The main idea of moving action sequences into own activities can be figured out without distinctly including object flows. Relations processing control flows are less complex since control flows connect actions

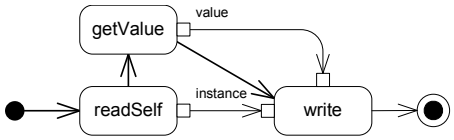
directly. Object flows connect output and input pins. Concerned actions are the owners of the pins. This makes relations processing object flows more complex.

By omitting object flows, relations remain less complex for a second reason: including object flows requires to differentiate between both kinds of flows since a control flow is represented by an edge of type *ControlFlow* whereas an object flow is represented by an edge of type *ObjectFlow*.

Another issue regarding flows is that data flows may induce unintended concurrency if an action requires inputs which is provided by different actions. An example is given by Fig. 5.6. Figure 5.6(a) shows an activity for adding a value to a structural feature by means of the *AddStructuralFeatureValueAction write*. Its inputs are obtained from two independent actions: the *ReadSelfAction readSelf* providing the object on which to access the structural feature and an *OpaqueAction getValue* providing the value to add. Both actions may — but not necessarily have to — be executed concurrently. An extraction of sequences of actions applied to the activity given by Fig. 5.6(a) would result in three sequences. A modification of the activity is given in Fig. 5.6(b). By the additionally inserted control flows (marked bold), the concurrency between action *readSelf* and *getValue* is resolved and a sequencing of action executions is given. Based on the control flow, a single sequence of actions can be determined.




(a) Unintended concurrency induced by independent input data.




(b) Serialization of sequences by additional control flows.

Figure 5.6: Avoiding unintended concurrency by additional control flows.

If claiming that control flows must always be present even if the execution semantics is determined by data flows, sequencing of actions can be based on the sole analysis of control flows. The existence of such additional control flows can be checked by a model validation: each action must have at least one outgoing control flow and if it has input pins, it also must have at least one incoming control flow. Furthermore, outgoing object flows must reach actions which are also reachable by following downstream control flows. Object flows then must be considered at the beginning and end of action sequences, within action sequences, they become a matter relevant for the transition from models to code.

## 5.6 Summary

In this chapter we give an overview of some formalisms for the transformation of models and justify our choice for QVTR.

The presented QVTR relations resolve complex modeling concepts of design models and the resulting models serve as implementation models at a lower level of abstraction.

Associations are resolved by the Relationship Object Pattern, i. e. by inserting classes for implementing associations.

For classes in the input model, a pair of classes related by a generalization relationship is created in the target model. The intention of this pattern is to hide generated code in the general class from the user, who may possibly be implementing algorithms that are not contained in the input model directly into the specialized class. Operations for accessing association ends and attributes are also included in the more general classes.

Behavior is accounted for by the reorganization of activities that moves sequences of actions into subactivities which are incorporated into the original activity using call behavior action.

From the models resulting from the application of the presented transformation relations, code can be generated in a straightforward approach as detailed in the following chapter.

# Chapter 6

# Code Generation

In this chapter, we describe the transition from models to code. The transformations specified in the previous chapter resolve modeling concepts, which cannot be directly implemented in Java, as well as most of them like e. g. association cannot be implemented in other object-oriented languages, too. After the resolution of these concepts, the transformed models serve as implementation models the elements of which can be mapped to target language concepts.

This chapter starts with a brief view over the role of code generation in MDD approaches in Sect. 6.1. After that, the criteria relevant for the decision which formalism to use for specifying code generation templates are motivated. Two general options of how to proceed from models to code are presented in Sect. 6.3. Section 6.4 introduces the selected formalism Mof2Text .

Finally, the actual generation of code is presented in two parts, starting with the generation of code for static structure. The generation of code for dynamic behavior follows in the last section.

## 6.1 The Role of Code Generation in MDD Approaches

Although models could be compiled to runnable software, they usually are transformed to source code. The generated code can be processed by existing compilers and is the specification of the software on the lowest level of abstraction reached on the way from models to executable code.

If we consider the step from source code to runnable machine code acceptably investigated and take techniques of compiling or interpreting for granted, a core item of model driven development probably is the step from models to code. In this step, the gap between models and runnable software actually is bridged.

The concrete way of how to descend from a modeling language to a programming language is not predetermined, particularly regarding technical options: a code generator may be completely implemented as a piece of software or its implementation might be based on frameworks or executable formalisms, either operational or declarative.

But even though the transition from models to code is a core issue of MDD, the harder part is the semantical mapping from modeling concepts to target language features. Therefore, this part is more intensively discussed in this thesis whereas code generation is included in order to provide a continuous, integrated approach covering all the way from highly abstract conceptual models to runnable software.

## 6.2 Formalisms

Implementing a code generator in any imperative or object-oriented programming language is an option, however, it not really is a well suited proceeding. We discard it for the same reasons why we discard this option with regard to implementing model transformations: An imperative or object-oriented programming language is a bad choice if a separation of transformation rules from technical implementation details is desired. Therefore, we look for a formalism which supports to specify a mapping from models to code on an abstract level hiding the bothersome necessities of real implementations.

According to the considerations concerning the choice of a formalism for specifying model transformations, a declarative approach is favored. Again, as proposed by Schlecht [87], additional criteria considered are

- adequacy of a formalism
- preciseness of the formalism's specification
- clarity of formalizations
- maintainability of formalizations
- continuous tool support, ability to execute formalizations
- support of proofs of correctness.

Since the applied formalism should be backed up by tool support in order to use the formalization as an implementation in ActiveCharts, not only availability of a tool is demanded, but also the following requirements must be satisfied by a tool [87]

- conformance to the formalism
- interoperability
- ability of integration
- suitable documentation
- usability
- free availability
- continuing development.

Whereas the choice which formalism to use for specifying model transformations is based on a detailed analysis by Schlecht [87], the decision what formalism to apply for code generation is not backed up by such an analysis. Rather, this decision is based on the expertise of other developers by means of an intensive investigation and evaluation of reports comprising experiences about possibly applicable formalisms.

Some formalisms that are considered for model transformations have proved to be inadequate, such as XSLT or CHR and are not taken into account for code generation either due to the fact that, for the same reasons why they do not qualify for model transformation, they disqualify for code generation, too. In particular, the criterion for exclusion is the cumbersome interfacing to the modeling domain.

An approach which is worth being favored is *MOF to Text Template Language* (Mof2Text) [68] as this formalism is a standard released by OMG. Therefore it is a natural complement to other OMG standards on which ActiveCharts is based, such as UML and QVT-R. An evaluation of the level of compliance to the criteria for decision making formulated above proved the adequacy of Mof2Text.

Tool support for Mof2Text is little with regard to the number of mature tools. However, *Acceleo*[1] is a tool supporting Mof2Text while excellently satisfying the demands. For a detailed presentation of the evaluation of the applicability of Mof2Text and *Acceleo* in ActiveCharts, we refer to Schlecht [87].

## 6.3 Transformation Strategies

In our approach, we transform models in order to resolve highly abstract modeling concepts into models which, concerning abstraction, level with object-oriented languages. The resulting platform specific model (PSM) is not the only possible source for code generation. A transformation into the domain of the target language, i. e. a language specific model (LSM) which is an instance of a metamodel of the target language, is an equivalent approach. But it requires a meta model for

[1] http://www.eclipse.org/acceleo

the target language. *MoDisco*[2] provides a metamodel for the Java programming language and a model to text transformation based on *Acceleo* templates. Accordingly, we identify two strategies of code generation:

- specifying a mapping from a PSM to the target language by means of *Acceleo* templates
- specifying a transformation from a PSM to a LSM and using an automatic code generation.

The first option is the obvious strategy which most often is applied. It focuses on the specification of model to text transformation patterns.

The second alternative moves the transition to a programming language into the modeling domain. Depending on the adequacy of the metamodel of the LSM, the mapping from a LSM to code is very simple and straightforward. Ideally, it is included in a tool, as it is e. g. in *MoDisco*.

A problem of the second approach is a trade-off between level of abstraction of the programming language metamodel on the one hand and the influence a developer has on the design of generated code on the other hand. If the metamodel of a target language offers no abstraction over the underlaying represented target language, the transformation of a LSM to code requires no semantical resolution at all. In this case, the design of resulting code is completely predetermined by the transformation of a PSM to a LSM.

If the level of abstraction of a LSM is little, the influence on the design of code generated from the LSM still is high. But a metamodel on a low level of abstraction requires more detailed transformations, which in term become more complex, less readable and less maintainable.

A metamodel on a higher level of abstraction makes the transformations less complex, however, the mapping from the LSM to code is more complex. Since this mapping is supposed to be covered by tool support, it no longer is in the responsibility of the developer. Thus, the developer has less influence on the design of the actual implementation of a model in code.

The main benefit of a second model transformation and code generation from a LSM is that only one formalism is applied. But for the time being, the available metamodels for Java are on a level of abstraction which makes the transformation into a LSM harder to understand than a transformation of our PSM into code.

Since we base on OMG standards, we discard the option of defining our own Java metamodel and applying a model transformation that creates a LSM in favor for directly generating code from the PSM applying Mof2Text.

## 6.4 Mof2Text

Mof2Text is a standard released by OMG in version 1.0 in 2008. It is designed for model to text transformations and aligned with other OMG standards such as UML 2.0, MOF 2.0, and OCL 2.0.

The rules for transforming model elements to text are specified in *templates*, which are grouped in *modules*. *"A template specifies a text template with placeholders for data to be extracted from models. These placeholders are expressions specified in terms of metamodel entities and are evaluated over instances of these metamodel entities"*[68, § 8.1.3]. The text producing expressions, i. e. the placeholders, of a template are grouped in *blocks*.

Besides templates, modules can contain *queries*. A query specifies a side-effect-free operation. Mof2Text modules, templates, blocks, and queries are easily readable. Therefore, we explain the syntax by means of few concrete examples.

The syntax of a template is given in Listing 6.1. The items of the template language are written in gray color, types of model elements are written **red**.

```
1 [template public classToJava(c : Class)]
2 class [c.name/] {
3   // Constructor
4   [c.name/]() {
5   }
6 }
7 [/template]
```

Listing 6.1: Syntax of a Mof2Text template.

[2] http://www.eclipse.org/MoDisco

In the first line of Listing 6.1, the template `classToJava` is declared. It is processed once for each class of the input model. The class element is accessible inside the template through the variable `c`. Access to elements of `c` must be marked as contents of the template by including it in square brackets. E. g. `[c.name/]` returns the name of the class bound to variable `c`. Every character not included in square brackets is added to the output. For a class named `Person`, an execution of template `classToJava` results in the code of the following Listing 6.2:

```
class Person {
  // Constructor
  Person() {
  }
}
```

Listing 6.2: Code generated by a simple template.

Class properties can be included in template processing in two ways. The property can be accessed as it is done for the class name in Listing 6.1 or another template may be called as shown in Listing 6.3.

```
[template public classWithAttributesToJava(c : Class)]
class [c.name/]{
  // Attributes
  [attributeToJava(c.attribute)/]
  // Constructor
  [c.name/](){
  }
}
[/template]

[template public attributeToJava(a : Attribute)]
[a.type.name/] [a.name/];
[/template]
```

Listing 6.3: Syntax of a Mof2Text template.

The advantage of the latter option is that for properties, the template processing engine iterates over all existing elements. If class `Person` has attributes `name` and `lastName`, both of type String, the code given in Listing 6.4 is generated when processing `classWithAttributesToJava`.

```
class Person {
  // Attributes
  String name;
  String lastName;
  // Constructor
  Person() {
  }
}
```

Listing 6.4: Code generated by a simple template.

An alternative way to process multi-valued properties effecting in the same output is using the *for* block. The example given in Listing 6.5 produces exactly the same output as the template given in Listing 6.3. The impact of either grouping template expressions in own templates or including them inline in another template is on the specification of transformations only, not on the produced output.

Including the processing of attributes directly in the processing of a class makes a template more complex and less readable. In contrast, encapsulation of attribute processing in its own template contributes to a more modular design of transformation patterns, particularly facilitating template reuse.

```
1  [template public classWithAttributesToJava(c : Class)]
2  class [c.name/]{
3    // Attributes
4    [for(a : Attribute | c.attribute)]
5    [a.type.name/] [a.name/];
6    [/for]
7    // Constructor
8    [c.name/](){
9    }
10 }
11 [/template]
```

Listing 6.5: Syntax of a Mof2Text template.

A drawback of this modular design is that an invocation of each template requires to pass all elements needed inside the invoked template as parameters. Invocations of templates become complex and inside the invoked template, the origin of values for actual parameters cannot be seen. Therefore, we prefer the option to include most transformation steps in-line.

In the following, we give a top down presentation of our formalization for the code generation process. We start at the generation of source files for classes and owned behaviors omitting details of how to implement owned members — such as owned attributes and operations of classes or nodes and edges of activities — by using *snippets*. A *snippet* is a sequence of lines within a code generation template that is cut out and referred to by `[include <name>/]`. All variables valid in the scope of the include statement are valid in the included snippets, too. Snippets are for presentation purposes only. Before executing the code generator, all `include` statements must be replaced by the referenced snippet. The first line `snippet::<name>:` is to be omitted. We will descend into the details of code generation for owned elements by following the chain of `include` statements and discussing the included snippets.

## 6.5 Generating Code for Static Structures

The generation of code for static structures comprises the creation of classes and populating created class files with content for owned elements such as attributes and operations. Since during model transformations, associations between classes have been made explicit by introducing association implementation classes, code generation for associations is covered as well by the template processing classes instead of requiring an own template matching associations.

Listing 6.6 shows the general template for class processing. It produces no output for activities (line 2) as well as it does not process classes which are not owned by the package named `PSM` (line 6). For all other classes, a file is created (line 7) and populated with contents by including snippet `CreateClassContent` (line 8).

```
1  [template public generateElement(c : Class)]
2   [if  (c.oclIsKindOf(Activity))]
3    [comment : nothing to do here /]
4   [else]
5    [comment : process PSM package only /]
6    [if (c.containingPackages()↗C.11->last().name.equalsIgnoreCase('PSM'))]
7     [file(classFileName(c)↗C.12, false, 'UTF-8')]
8      [include CreateClassContent /]
9     [/file]
10   [/if]
11  [/if]
12 [/template]
```

Listing 6.6: Generation of class files for UML classes.

### 6.5.1 Generating Code for Classes and Associations

Listing 6.7 contains the generation of a package declaration and an import statement for the classes contained in package `java.util`. By this, it is possible to refer to the Java type `List`, which we make use of for implementing multi-valued properties and operation parameters. Furthermore, the class declaration is contained considering the visibility of the class, the keyword `abstract` for abstract classes, the class name and the name of the general class if any. Within the class body, several snippets are included. These are needed for

- processing owned attributes
- processing owned operations
- generating code needed for the implementation of participation in associations
- implementing a link manager (i. e. a class implementing an association)
- generating code for behaviors included in general classes
- generating code for owned behaviors

```
1  snippet::createClassContent:
2  package [containingPackages(c).name->sep('.') /];
3  import java.util.*;
4
5  [visibilityString(c.visibility)↗C.14/][if(c.isAbstract)] abstract [/if]
6   class [c.name /]
7  [if(not c.generalization->isEmpty())] extends [superClass(c)↗C.15.name /][/if]
8  {
9   [comment : handle Properties/]
10  [include handleClassProperties /]
11  [comment : LinkMemberClass/]
12  [include handleLinkMember /]
13  [comment : LinkManager/]
14  [include handleLinkManager /]
15  [comment : Operation/]
16  [include handleOperation /]
17  [comment : handle general Behaviors/]
18  [include GeneralClassifierBehaviorImplementation /]
19  [comment : handle owned Behavior/]
20  [include ClassifierBehaviorImplementation /]
21 }
```

Listing 6.7: Generation of class contents.

Some snippets only contribute to the content of specific classes, e. g. code for participation in an association is only generated if the processed class participates in an association and code for implementing associations is only generated for manager classes. The order of snippets in the listing is designed to contribute to the readability of generated code. The order in which snippets are presented here is intended to group the snippets according to their purpose.

### 6.5.2 Generating Code for Owned Attributes

The code generation for owned attributes is covered by snippet `handleClassProperties`. It is given by Listing 6.8 and it creates a field declaration in the source file. It

- processes each owned attribute in a loop (lines 2, 8)
- inserts the specified visibility modifier (line 3) which is obtained from query *visibilityString*
- inserts the optional keywords `static` (line 4) and `final` (line 5) if needed

- determines the type of the field according to type and upper bound of the owned attribute (line 6); if the attribute is multi-valued, query `typeString` returns a typed list (`List<PropertyType>`)
- inserts the name of the attribute (line 6)
- assigns a default value, if a default value is modeled (line 7).

By the declaration of a separator in line 2, a line feed is inserted after each field declaration.

```
1 snippet::handleClassProperties:
2 [for(p : Property | c.ownedAttribute) separator('\n')]
3  [visibilityString(p.visibility) /]
4  [if(p.isStatic)] static [/if]
5  [if(p.isReadOnly)] final [/if]
6  [typeString(true, p.type, p.upper)↗C.16 /][p.name /]
7  [if(not p.default.oclIsUndefined())] = [defaultStringValue(p)↗C.20 /][/if];
8 [/for]
```

Listing 6.8: Generation of code for the implementation of owned attributes.

### 6.5.3 Generating Code for Owned Operations

Code generation for owned operations as shown in Listing 6.9 is very similar to that of owned attributes, since a method declaration like a member field declaration has a visibility modifier (line 3), optional keywords `static` (line 5) and `final` (line 6), a type, and a name (line 7). However, there are some differences:

- method signatures not necessarily have a return type: accordingly, the called query `returnType` either returns `void`, a type name, or a typed collection such as `List<typeName>`.
- operations may have parameters: the parameters for the method signature are obtained form query `inParameterString`, which creates a string consisting of pairs of type names and parameter names. For multi-valued parameters, a typed set is used (e. g. `List<typeName>`).
- method may be abstract: the keyword `abstract` must be considered by either including it in the method declaration (line 4) as well as by closing an abstract method declaration with a semicolon (line 11) or by generating a method body for which content is provided by `generateOperationBody` (lines 13–15).

```
1  snippet::handleOperation:
2  [for(o : Operation | c.ownedOperation) separator('\n')]
3   [visibilityString(o.visibility) /]
4   [if(o.isAbstract)]abstract [/if]
5   [if(o.isStatic)]static [/if]
6   [if(o.isLeaf)]final [/if]
7   [returnType(c, o)↗C.16 /][o.name /](
8   [for(p : Parameter | inParameters(o)↗C.17) separator(', ')]
9    [inParameterString(true, p)↗C.19 /]
10  [/for])
11  [if(o.isAbstract)];
12  [else]
13  {
14   [include generateOperationBody /]
15  }
16  [/if]
17 [/for]
```

Listing 6.9: Generation of method signatures implementing owned operations.

Before descending into the details of code generation for method bodies, the code needed for association implementation is presented.

### 6.5.4 Generating Code for Link Members

Classes participating in associations need access to the association. For this purpose, the *handle* pattern is introduced in Sect. 3.6.1. Listing 6.10 covers the generation of code for demanding and obtaining a handle from a link manager class:

- for each owned attribute that refers to a link manager, indicated by a special comment, an invocation of the `getHandle` method is included in a static initializer block (lines 3–10)
- for each such attribute a callback method for obtaining the actual handle for a link manager class is generated (lines 13–17)
- the provided handle is assigned to the owned attribute (lines 14–15); in order to prevent from changing the handle, it can only be updated once, i. e. if being `null` (line 14).

```
1  snippet::handleLinkMember:
2  [if(isLinkMemberClass(c)↗C.33)]
3   static
4   {
5    [for(p : Property | c.ownedAttribute)]
6     [if(isLinkMemberClassProperty(p)↗C.35)]
7      [p.type.name /].getHandle([c.name /].class);
8     [/if]
9    [/for]
10  }
11  [for(p : Property | c.ownedAttribute)]
12   [if(isLinkMemberClassProperty(p))]
13    public static void takeHandle([p.type.name/] linkManager){
14     if([c.name/].[p.name/] == null){
15      [c.name/].[p.name/] = linkManager;
16     }
17    }
18   [/if]
19  [/for]
20 [/if]
```

Listing 6.10: Generation of code implementing the participation in associations.

### 6.5.5 Generating Code for Link Managers

Link manager classes must provide access to the association to participating classes. The generation of code for this task as well as for internally storing links is presented in Listing 6.11:

- Query `isLinkManager` (line 2) is used to determine, whether or not a comment has been attached to the currently processed class during model transformation, indicating that this class represents a link manager class. Only link manager classes are processed.
- A line of code for implementing the singleton pattern is inserted (line 3).
- Method `getHandle` is generated in lines 4–11:
  Inside the body of this method, code is generated to supply each participating class with a handle. The participating classes are identified by looking up the parameters of the `createLink` method. In order to create an association link, a value must be supplied for each association end and therefore, the parameters of this method must contain all association ends.
- Finally, a private list for storing links is prepared (lines 12–17); the type of the link class is looked up from the return type of the `getLinks` method, which returns a link of the association.

```
1  snippet::handleLinkManager:
2  [if (isLinkManager(c)↗C.34)]
3   private static final [c.name /] INSTANCE = new [c.name /]();
4   public static void getHandle(Class<?> memberEnd){
5   [for(t : Type | inParameters(c.ownedOperation->any(
6    o : Operation | o.name = 'createLink')).type->asSet()) separator('\n')]
7    if ( memberEnd == [t.name /].class ){
8     [t.name /].takeHandle( INSTANCE );
9    }
10  [/for]
11  }
12  private List<[c.ownedOperation->any(
13   o : Operation | o.name = 'getLinks').ownedParameter->any(
14   p : Parameter | p.direction.toString() = 'return').type.name /]> links =
15   new LinkedList<[c.ownedOperation->any(
16   o : Operation | o.name = 'getLinks').ownedParameter->any(
17   p : Parameter | p.direction.toString() = 'return').type.name /]>();
18 [/if]
```

Listing 6.11: Generation of code for link manager classes.

### 6.5.6 Generating Code for Method Bodies

The generation of code for method bodies is covered by special snippets depending on the kind of operation for which code is generated. Three kinds of operations must be distinguished:

- owned operations of a *normal* class
- owned operations of a *normal* class to handle participation in an association
- owned operations of link manager classes.

The decision is implemented as shown in Listing 6.12:

- constructors are identified by name, since constructors have the same name as the owning class (line 3); the body implementation is provided by snippet `generateConstructorBody`
- methods which have been inserted by the model transformation process are marked by attached comments which now are used for deciding which kind of method body to create (lines 6–32)
- for operations which are neither a constructor nor have a comment for identification attached, a default return statement is created (line 34)
- if an operation is associated with a behavior implementing the effect of the operation, then the code of the method body is to activate the associated behavior (lines 37–43) either concurrently (line 39) or synchronously (line 41).

Specific implementations for operations are generated for constructors, for getter and setter methods accessing owned attributes as well as add and remove methods if the accessed owned attribute is multi-valued. For operations of classes participating in associations as well as for operations of link manager classes, snippets are included which make a further distinction between the purpose of operations, which is either to create, retrieve, or remove association links.

```
1  snippet::generateOperationBody:
2  [comment : Constructor /]
3  [if(o.name = c.name)]
4    [include generateConstructorBody /]
5  [comment : Getter /]
6  [elseif (o.ownedComment->exists(n : Comment | n._body.startsWith(
7    'Operation#Kind:Getter')) and c.ownedAttribute->exists(
8    p : Property | p.name = substringName(o, 'Operation#Kind:Getter')↗C.36))]
9    return [substringName(o, 'Operation#Kind:Getter') /];
10 [comment : Setter /]
11 [elseif(o.ownedComment->exists(n : Comment | n._body.startsWith(
12   'Operation#Kind:Setter')) and c.ownedAttribute->exists(
13   p : Property | p.name = o.ownedParameter->first().name))]
14   [include generateSetterBody /]
15 [comment : AddOperation /]
16 [elseif(o.ownedComment->exists(n : Comment | n._body.startsWith(
17   'Operation#Kind:AddOperation')) and c.ownedAttribute->exists(
18   p : Property | p.name = o.ownedParameter->first().name))]
19   [include generateAddBody /]
20 [comment : RemoveOperation /]
21 [elseif(o.ownedComment->exists(n : Comment | n._body.startsWith(
22   'Operation#Kind:RemoveOperation')) and c.ownedAttribute->exists(
23   p : Property | p.name = o.ownedParameter->first().name))]
24   [include generateRemoveBody /]
25 [comment : LinkManager /]
26 [elseif(c.ownedComment->exists(n : Comment | n._body.startsWith(
27   'Class#Kind:LinkManager')))]
28   [include generateLinkManagerBody /]
29 [comment : LinkMemberClass (begin) /]
30 [elseif(c.ownedComment->exists(n : Comment | n._body.startsWith(
31   'Class#Kind:LinkMemberClass')))]
32   [include generateLinkMemberBody /]
33 [else]
34   [defaultReturnValue(c, o, returnType(c, o))↗C.21 /]
35 [/if]
36
37 [for (method : Activity | o.method) separator (';\n')]
38   [if (o.concurrency.toString() = 'concurrent')]
39     [method.name/].activate(true);
40   [else]
41     [method.name/].activate(false);
42   [/if]
43 [/for]
```

Listing 6.12: Generation of method bodies for owned operations.

### Generating Code for Constructors

Listing 6.13 contains the template for generating a method body of a class constructor.

- If a generalization relationship to another class exists, the super constructor is called ( lines 3–5). The parameters for this invocation are obtained from the query `inSuperParameters`.
- After the call of `super`, multiplicity bounds are checked for multi-valued attributes (lines 7–15) having the same name as a constructor parameter:
  - if bounds are violated, an `IllegalArgumentException is thrown` (line 10)
  - if the value to assign is `null`, a `NullPointerException` is thrown (line 12), assignment of an empty list, however, is allowed.
- After checking bounds, code is generated to assign the values of parameters to owned attributes having the same name in order to initialize the newly created instance (lines 16–20).

```
1  snippet::generateConstructorBody:
2  [if(not c.generalization->isEmpty())]
3   super([for(p : Parameter | inSuperParameters(c, o)↗C.18) separator(', ')]
4    [p.name /]
5   [/for]);
6  [/if]
7  [for(p : Parameter | o.ownedParameter) separator('\n')]
8   [if(c.ownedAttribute->exists(a : Property | a.name = p.name)
9    and (p.upper > 1 or p.upper = -1))]
10   if([p.name /] == null) { throw new NullPointerException(); }
11   if([p.name /].size() < [p.lower /] || [p.name /].size() > [p.upper /]) {
12    throw new IllegalArgumentException();
13   }
14  [/if]
15 [/for]
16 [for(p : Parameter | o.ownedParameter) separator('\n')]
17  [if(c.ownedAttribute->exists(a : Property | a.name = p.name))]
18   this.[p.name /] = [p.name /];
19  [/if]
20 [/for]
```

Listing 6.13: Generation of a constructor with parameters.

### Generating Method Bodies for Accessing Owned Attributes

Listing 6.14 shows the generation of code for a method setting the value of an owned attribute.

- If the attribute is mandatory, i. e. its lower bound is greater than 0 or if it is multi-valued, the parameter value to be assigned must not be null (lines 2–8).
- Furthermore, if multi-valued, the number of elements to be set must conform to the lower and upper bounds of the attribute (lines 9–17).
- Finally the actual parameter is assigned to the owned attribute (line 18).

Note that for a multi-valued attribute with 0 as its lower bound, it is legal to assign an empty list, but it is not allowed to assign `null`.

```
1  snippet::generateSetterBody:
2  [if (o.ownedParameter->first().lower > 0
3   or (o.ownedParameter->first().upper > 1)
4   or (o.ownedParameter->first().upper = -1)]
5    if([o.ownedParameter->first().name /] == null) {
6      throw new NullPointerException();
7    }
8  [/if]
9  [if (o.ownedParameter->first().upper > 1)
10  or (o.ownedParameter->first().upper = -1)]
11   if ([o.ownedParameter->first().name /].size()
12     < [o.ownedParameter->first().lower /]
13   || [o.ownedParameter->first().name /].size()
14     > [o.ownedParameter->first().upper /]) {
15     throw new IllegalArgumentException();
16   }
17 [/if]
18 this.[o.ownedParameter->first().name/] = [o.ownedParameter->first().name/];
```

Listing 6.14: Generation of code for setting attribute values.

If an attribute is multi-valued, its value either can be set or a value can be added to the set of previously assigned values. Code for this is generated as shown in Listing 6.15. The presented template creates

- code to prevent from assigning `null` values to attributes (lines 2–4)
- code to prevent adding a value while violating the upper bound of an attribute (lines 5–8)
- code to actually add the value if both checks have been passed (lines 9–10).

```
1  snippet::generateAddBody:
2  if([o.ownedParameter->first().name /] == null) {
3   throw new NullPointerException();
4  }
5  if(this.[o.ownedParameter->first().name /].size() ≥ [c.ownedAttribute->any(
6    p : Property | p.name = o.ownedParameter->first().name).upper /]) {
7    throw new IllegalArgumentException();
8  }
9  this.[o.ownedParameter->first().name /].add(
10   [o.ownedParameter->first().name /]);
```

Listing 6.15: Generation of code for adding attribute values.

Similar to the content of Listing 6.15, the removal of attribute values is supported by generating code as given in Listing 6.16:

- code is generated to check that it is not tried to remove a `null` value (lines 2–4), since it is not possible to add the value `null` by the generated setter methods (as given by Listing 6.14)
- code is generated that tests for a violation of the lower bound of the attribute (lines 5–8)
- code is generated for actually implementing the removal of a given value (lines 9–10).

```
1  snippet::generateRemoveBody:
2  if([o.ownedParameter->first().name /] == null) {
3    throw new NullPointerException();
4  }
5  if(this.[o.ownedParameter->first().name /].size() ≤ [c.ownedAttribute->any(
6    p : Property | p.name = o.ownedParameter->first().name).lower /]) {
7    throw new IllegalArgumentException();
8  }
9  this.[o.ownedParameter->first().name /].remove(
10   [o.ownedParameter->first().name /]);
```

Listing 6.16: Generation of code for removing attribute values.

### Generating Method Bodies for Managing Association Links

If an operation is marked to be a link manager class method, it is either for

- creating a new link
- retrieving a link or a set of links
- destroying a link.

The decision as implemented in Listing 6.17 is made by evaluating the name of the operation.

An operation body for creating a new link is generated as shown in Listing 6.18. Firstly, a check is implemented preventing from creating links with `null` values (lines 3–7). If the value of any parameter is `null`, an exception is thrown (line 6).

```
snippet::generateLinkManagerBody:
[if(o.name = 'createLink')]
    [include CreateLinkBody /]
[elseif(o.name = 'getLinks')]
    [include GetLinkBody /]
[elseif(o.name = 'removeLink')]
    [include RemoveLinkBody /]
[/if]
```

Listing 6.17: Distinction of methods for accessing association ends from participating classes.

Afterwards, a check is generated to prevent from creating a link violating multiplicity bounds. Therefore, the upper bound must be tested for each association end. The outer `for` block (lines 8–14) causes a `getLink` invocation for each end, the inner `for` block (lines 9–11) constructs the parameter list for each such invocation: by replacing the parameter bound in the outer `for` block by `null`, the associated end becomes the open end of the `getLink` invocation. Thus, each end is queried and the size of the returned set of links is compared with the concerning upper bound.

Code for the actual creation of the link must distinguish between two cases:

- Creating a link for an association class (lines 15–21)
- Creating a *normal* link for an association (lines 21–27).

In the first case, a line for creating an instance of the association class is generated (lines 17–18), internally stored (line 19) and returned (line 20). The class name consists of the name of the type returned by the operation and the suffix *Impl*.

In the latter case, an instance of the association is created and internally stored. The name of the link class is looked up from the return type of the operation `getLinks`.

In both cases, the values associated to the link ends are passed to the constructor of the link.

```
snippet::CreateLinkBody:
[if(o.name = 'createLink')]
  if([for(p : Parameter | inParameters(o)) separator('||')]
    [p.name /] == null
  [/for]){
    throw new NullPointerException();
  }
  if([for(p : Parameter | inParameters(o)) separator('||')]
    getLinks( [for(a : Parameter | inParameters(o)) separator(',')]
                [if(p.name <> a.name)] [a.name /] [else] null [/if]
              [/for]).size() == upper[p.name.toUpperFirst() /]
  [/for]){
    throw new IllegalArgumentException();
  }
  [if(c.ownedComment->exists(n : Comment | n._body.startsWith(
      'Class#Kind:LinkManager#AssociationClass')))]
    [o.type.name /] link = new [o.type.name /]Impl(
    [for(p : Parameter | inParameters(o)) separator(',')][p.name/][/for]);
    links.add(link);
    return link;
  [else]
    links.add(new [c.ownedOperation->any(
      o : Operation | o.name = 'getLinks').type.name /](
      [for(p : Parameter | inParameters(o)) separator(',')]
        [p.name /]
      [/for]));
  [/if]
[/if]
```

Listing 6.18: Generation of code for creating association links.

Listing 6.19 contains the snippet for generating the code of an operation for retrieving links. Such an operation returns a list of links complying to the provided context.

First, the declaration of a list holding the result set is generated (line 3).

Then, a `for` loop is generated for iterating over all existing links (lines 4–10). Inside this loop, an `if` statement is created. The condition to be tested is that each parameter of the operation is either `null` or equals a value of the link (lines 5–7). Which parameter must equal which link value is determined by the name of the parameter and the link value. If the condition holds, the link is a candidate to be included into the result set of the operation (line 8).

```
1  snippet::GetLinkBody:
2  [if(o.name = 'getLinks')]
3    List<[o.type.name /]> result = new LinkedList<[o.type.name /]>();
4    for([o.type.name /] l : links){
5      if([for(p : Parameter | inParameters(o)) separator(' && ')]
6      ([p.name /] == null || l.get[p.name.toUpperFirst() /]() == [p.name /])
7      [/for]) {
8        result.add(l);
9      }
10   }
11   return result;
12 [/if]
```

Listing 6.19: Generation of code for retrieving association links.

The code needed for the implementation of an operation for removing links is very similar to that for creating links. Consequently, the snippet for creating such code presented in Listing 6.20 is very close to the snippet of Listing 6.18. The difference is that not the upper bound for association ends is tested but the lower bound (line 10) and that not links are created and added into the internal list of links but removed from that list (line 19).

```
1  snippet::RemoveLinkBody:
2  [if(o.name = 'removeLink')]
3    if([for(p : Parameter | inParameters(o)) separator(' || ')]
4      [p.name /] == null[/for]){
5      throw new NullPointerException();
6    }
7    if([for(p : Parameter | inParameters(o)) separator(' || ')]
8      getLinks([for(a : Parameter | inParameters(o)) separator(', ')]
9        [if(p.name <> a.name)] [a.name /] [else] null [/if]
10     [/for]).size() == lower[p.name.toUpperFirst() /]
11   [/for]){
12     throw new IllegalArgumentException();
13   }
14   for([c.ownedOperation->any(
15     o : Operation | o.name = 'getLinks').type.name /] l : links){
16     if([for(p : Parameter | inParameters(o)) separator(' && ')]
17       l.get[p.name.toUpperFirst() /]() == [p.name /]
18     [/for]){
19       links.remove(l);
20       break;
21     }
22   }
23 [/if]
```

Listing 6.20: Generation of code for destroying association links.

### Generating Method Bodies for Accessing Association Ends

The last kind of operations to consider are those offering access to association ends in participating classes. During the model transformation, such operations have been marked by comments starting with the prefix *Operation#Kind:LinkMemberClassOperation#*. The decision about the concrete kind of operation is made by evaluating the tagged comment like shown in Listing 6.21. In the included snippets, tagged strings are evaluated for obtaining additional information.

```
1  snippet::generateLinkMemberBody:
2  [if(o.ownedComment->exists(n : Comment | n._body.startsWith(
3    'Operation\#Kind:LinkMemberClassOperation\#add')))]
4      [include MemberAddLinkBody /]
5  [elseif(o.ownedComment->exists(n : Comment | n._body.startsWith(
6    'Operation\#Kind:LinkMemberClassOperation\#remove')))]
7      [include MemberRemoveLinkBody /]
8  [elseif(o.ownedComment->exists(n : Comment | n._body.startsWith(
9    'Operation\#Kind:LinkMemberClassOperation\#get')))]
10     [include MemberGetLinkBody /]
11 [/if]
```

Listing 6.21: Distinction between methods for accessing associations from member classes.

If an operation represents the method to add a new link to an association, the `createLink` method of the link manager class must be called. How the necessary code is generated is given by Listing 6.22.

At first, it must be determined whether the association is an association class or not. In either case, an invocation of the method to create a new link in the manager class (see Listing 6.18) is inserted (line 6 and line 16). Only if the created link is an instance of an association class, this link is returned by the manager class. Thus, the main difference between the if-block from line 3 to line 15 handling association classes and the else-block from line 15 to line 25 handling normal associations is the `return` statement generated in line 5.

```
1  snippet::MemberAddLinkBody:
2  [if(o.ownedComment->exists(n : Comment | n._body.startsWith('add')))]
3   [if(o.ownedComment->exists(n : Comment | n._body.startsWith(
4    'add#AssociationClass')))]
5    return [substringName(
6    o,'add#AssociationClass').toLowerFirst()/].createLink(
7    [for(p : Parameter | c.ownedAttribute->any(
8     a : Property | a.type.name = substringName( o,
9      'add#AssociationClass')).type.oclAsType(Class).ownedOperation->any(
10      o : Operation | o.name = 'getLinks').ownedParameter->select(
11       i : Parameter | i.direction.toString() = 'in')) separator(', ')]
12    [if(o.ownedParameter->exists(v : Parameter | v.name = p.name))]
13     [p.name /] [else] this [/if]
14   [/for]);
15  [else]
16   [substringName(o, 'add#Association').toLowerFirst() /].createLink(
17    [for(p : Parameter | c.ownedAttribute->any(
18     a : Property | a.type.name = substringName( o,
19      'add#Association')).type.oclAsType(Class).ownedOperation->any(
20       o : Operation | o.name = 'getLinks').ownedParameter->select(
21        i : Parameter | i.direction.toString() = 'in')) separator(', ')]
22    [if(o.ownedParameter->exists(v : Parameter | v.name = p.name))]
23     [p.name /] [else] this [/if]
24   [/for]);
25  [/if]
26 [/if]
```

Listing 6.22: Generation of code for linking objects.

In Listing 6.22, the name of the link manager to call is obtained from the comment tagged to the operation. For better readability, the prefix *Operation#Kind:LinkMemberClassOperation#* is omitted in lines 4, 6, 9, 16, and 19.

The parameter list for the call of method `createLink` is constructed by obtaining all in-parameters of the `getLinks` method of the link manager class. If a parameter of the same name exists for the currently processed operation, then the name of this parameter is included into the generated method invocation. Otherwise, the keyword `this` must be used to pass the member end class object as a link value to the `createLink` method of the link manager class.

Code for removing an existing link from an association is very similar to the code for creating a new link. Consequently, the generation of the concerned method bodies only slightly differ from each other. Listing 6.23 shows the snippet that supplies a `removeLink` method with its body implementation. The difference to the snippet for generating the implementation of an `addLink` method (as in Listing 6.22) is that instead of `createLink`, the method `removeLink` of the association manager class is invoked. All other characteristics of the generation of the method body are identical, in particular the processing of the parameter list for the call of the manager class methods. However, evaluation of tagged comments of course must be adapted to the processing of a remove link operation.

```
1  snippet::MemberRemoveLinkBody:
2  [if(o.ownedComment->exists(n : Comment | n._body.startsWith('remove')))]
3   [if(o.ownedComment->exists(n : Comment | n._body.startsWith(
4    'remove#AssociationClass')))]
5    [substringName(o, 'remove#AssociationClass').toLowerFirst()/].removeLink(
6    [for(p : Parameter | c.ownedAttribute->any(
7     a : Pr   operty | a.type.name = substringName(o,
8      'remove#AssociationClass')).type.oclAsType(Class).ownedOperation->any(
9       o : Operation | o.name = 'getLinks').ownedParameter->select(
10       i : Parameter | i.direction.toString() = 'in')) separator(', ')]
11    [if(o.ownedParameter->exists(v : Parameter | v.name = p.name))]
12     [p.name /] [else] this [/if]
13   [/for]);
14  [else]
15   [substringName(o,'remove#Association').toLowerFirst() /].removeLink(
16    [for(p : Parameter | c.ownedAttribute->any(
17     a : Property | a.type.name = substringName(o,
18      'remove#Association')).type.oclAsType(Class).ownedOperation->any(
19       o : Operation | o.name = 'getLinks').ownedParameter->select(
20        i : Parameter | i.direction.toString() = 'in')) separator(', ')]
21    [if(o.ownedParameter->exists(v : Parameter | v.name = p.name))]
22     [p.name /] [else] this [/if]
23   [/for]);
24  [/if]
25 [/if]
```

Listing 6.23: Generation of code for unlinking objects.

The implementation of a method that reads links from an association is similar to the two previously presented methods for adding or removing links, but it differs in two details:

- the method invoked on the link manager class requires the specification of an open end, i. e. exactly one parameter must not contain a value
- the result of the invoked method must be processed and a result set must be constructed which is returned.

Generating the parameters for the call of the `getLinks` method must identify the open end for which no value is provided:
In contrast to the methods for adding or removing links where all link ends are specified, when querying an association, this is done for one special end which is called the *open end* while providing the context, i. e. values for all other association ends. The open end is determined by comparing the names of the parameters of the `getLinks` method of the manager class with the name of

the operation currently processed. If after removing the prefix *get*, the name of the operation equals the name of the parameter, then the keyword `null` must be inserted to identify this link end as the open end. Otherwise, either the parameter is passed through or the keyword `this` must be inserted, like it is done for parameters of `addLink` or `removeLink` operations above. The creation of parameters is done in two alternative blocks, either for an association class or for a plain association.

The set of links returned by the link manager class must be processed by the member end class. A statement for creating the result list is included in line 3. In line 41, the value of the open end is inserted into the result set. The statement returning the result set is generated in line 48.

The complete template is given in Listing 6.24.

```
1  snippet::MemberGetLinkBody:
2  [if(o.ownedComment->exists(n : Comment | n._body.startsWith('get')))]
3   List<[o.type.name /]> result = new LinkedList<[o.type.name /]>();
4   [if(o.ownedComment->exists(
5    n : Comment | n._body.startsWith('get#AssociationClass')))]
6    for([substringName(o, 'get#AssociationClass#LinkManager') /]
7     l : [substringName(o, 'get#AssociationClass').toLowerFirst()/].getLinks(
8      [for(p : Parameter | c.ownedAttribute->any(
9       a : Property | a.type.name = substringName(
10      o, 'get#AssociationClass')).type.oclAsType(Class).ownedOperation->any(
11      o : Operation | o.name = 'getLinks').ownedParameter->select(
12       i : Parameter | i.direction.toString() = 'in')) separator(', ')]
13      [if(o.name.substring(4).toLowerFirst().matches(
14     p.name + '|' + p.name + 'By.*'))]
15       null
16      [elseif(o.ownedParameter->exists(v : Parameter | v.name = p.name))]
17       [p.name /]
18      [else]
19       this
20      [/if]
21     [/for]))
22   [else]
23    for(Link_[substringName(o, 'get#Association#LinkManager') /]
24     l : [substringName(o, 'get#Association').toLowerFirst() /].getLinks(
25      [for(p : Parameter | c.ownedAttribute->any(
26       a : Property | a.type.name = substringName(
27       o, 'get#Association')).type.oclAsType(Class).ownedOperation->any(
28       o : Operation | o.name = 'getLinks').ownedParameter->select(
29        i : Parameter | i.direction.toString() = 'in')) separator(', ')]
30       [if(o.name.substring(4).toLowerFirst().matches(
31        p.name + '|' + p.name + 'By.*'))]
32        null
33       [elseif(o.ownedParameter->exists(v : Parameter | v.name = p.name))]
34        [p.name /]
35       [else]
36        this
37       [/if]
38      [/for]))
39   [/if]
40   {
41    result.add(l.
42    [if(o.name.matches('.*By.*'))]
43     [o.name.substring(1, o.name.index('By') - 1) /]
44    [else]
45     [o.name /]
46    [/if]());
47   }
48   return result;
49  [/if]
```

Listing 6.24: Generation of code for retrieving linked objects.

### 6.5.7 Including Code for Owned Behavior

Depending on the source model and model transformations, behavior may be modeled by independent activities, activities which are owned behaviors of classifiers or activities which are classifier behaviors. In the examples given here, we only consider the behavior which is a classifier behavior. Other owned behavior can be considered as well, but we give no example for this case.

For linking static structure with dynamic behavior, we include references to classifier behavior as well as the implementation of *StartClassifierBehaviorAction* in the implementation of classes, as shown in Listing 6.25. For the classifier behavior, a private field is generated (line 3). A flag indicating whether or not the classifier behavior has been started is generated by line 4. From line 6 to 11, code for implementing a *StartClassifierBehaviorAction* is generated. The flag `isStarted` is checked, and if the behavior has not yet been started, the flag is set to true before the classifier behavior is executed.

```
1  snippet::ClassifierBehaviorImplementation:
2  [for (activity : Activity | c.classifierBehavior)]
3   private [activity.name/] classifierBehavior;
4   private boolean isStarted = false;
5
6   [visibilityString(activity.visibility)/] void synchronized startBehavior(){
7    if (! isStarted){
8     isStarted = true;
9     classifierBehavior = new [activity.name/](this);
10   }
11  }
12  [include CreateAbstractMethodsForOpaqueActions /]
13 [/for]
```

Listing 6.25: Generation of code for a classifier behavior.

The code for the activity itself is generated by a separate template which generates its own file. It is presented in the next section. If code for the activity should be located in the owning class, the template had to be included here.

In our approach, we consider that opaque actions are used for plugging in user code. During model transformations, sequences of actions have been identified and been moved into activities which are related to the activity formerly owning the concerned actions by means of call behavior actions. We term an activity only containing a sequence of actions *subactivity*, to an original activity, we refer to as a *main* activity.

The implementation of opaque actions is enforced by generating abstract method declarations in the class owning the behavior, as shown in Listing 6.26.

```
1  snippet::CreateAbstractMethodsForOpaqueActions:
2  [for (node:CallBehaviorAction | activity.node)]
3   [for (sub:Activity | node.behavior)]
4    [for (init:InitialNode | sub.node)]
5     [if (init.outgoing->asSequence()->first().target.oclIsKindOf(Action))]
6      [let actions:OrderedSet(Action) = getSequence(init)↗C.24->asOrderedSet()]
7       [for (a:OpaqueAction | actions)]
8         abstract [visibilityString(a.visibility)/]
9         [if (action.output->asSet()->isEmpty())] void
10        [else] [action.output.type.name/] [/if]
11        [action.name/]([paramString(action)↗C.22 /]);
12      [/for]
13     [/let]
14    [/if]
15   [/for]
16  [/for]
17 [/for]
```

Listing 6.26: Generation of abstract method declarations for opaque actions.

Since a *CallBehaviorAction* has exactly one behavior, the `for` loop starting in line 3 is executed once. Inside the loop, a nested loop iterates over all initial nodes. According to the transformation of models of dynamic behavior, a subactivity has only one initial node with a single outgoing flow. The target of that flow is an action (line 5). For this action, the sequence of itself and all downstream actions is obtained by calling the query `getSequence` in line 6. For each opaque action of this sequence (line 7), an abstract method is generated (lines 8–11). By that, the implementation at runtime is enforced and when generating code for activities, method calls for opaque actions can be included.

If the action has no outputs, the method return type is `void` (line 9), otherwise, the return type is the type of the action output (line 10).

Inside the `for` block (lines 7–12), specific code could also be generated for other kinds of actions according to the outlined implementations of Sect. 4.3.1. Handling other kinds of actions would be included in another `for` block inserted after the `for` block, which only handles opaque actions (lines 7–12).

During model transformation, for each class, a pair of classes is generated in the target model. One of these two classes is intended to contain the generated code for accessing owned attributes and association ends as well as implementation of dynamic behaviors, the other one is a specialization of the first class which includes the user code inside the body implementation of methods corresponding to opaque actions. The abstract methods for opaque actions have been created in the general class. In its specialization, these methods must be implemented.

A default implementation for methods corresponding to such opaque actions is generated by the snippet given in Listing 6.27. The generation process is similar to that for the abstract methods in the super class. But it is nested in a `for` block iterating over all superclasses. For each superclass, in another nested `for` block, all owned behaviors are processed. If an activity contains any opaque action, a comment is included in the generated code and similar to the generation of abstract methods, empty method implementations are created.

```
1  snippet::block::GeneralClassifierBehaviorImplementation:
2  [for (super : Classifier | c.general)]
3   [for(behavior : Activity | super.ownedElement)]
4    [if hasCallBehavior(behavior)↗C.28]
5     // Implement opaque actions of activity [behavior.name/]
6    [/if]
7    [for (cba : CallBehaviorAction | behavior.node) separator ('\n')]
8     [for (action : OpaqueAction | cba.behavior.oclAsType(Activity).node)]
9      [let s: String = paramString(action) ]
10      [visibilityString(action.visibility)/]
11      [if (action.output->asSet()->isEmpty())]
12       void
13      [else]
14       [action.output.type.name/]
15      [/if]
16      [action.name/]([s/])
17      {
18       // TODO Auto-generated method stub
19       System.out.println("Executing [action.name/]");
20      }
21     [/let]
22    [/for]
23   [/for]
24  [/for]
25 [/for]
```

Listing 6.27: Generation of default implementations for opaque actions.

After having generated code for static structures and having prepared a link between static structures and dynamic behavior, templates for generating implementations for activities are discussed in the following.

## 6.6 Generating Code for Dynamic Behavior

A code generator that processes activities must distinguish between the main activity, where action sequences have been replaced by call behavior actions, and subactivities that solely contain those sequences of actions. Main activities are the starting point for code generation for dynamic behavior. Therefore, we start with the code generation template for main activities. Afterwards, we move towards the code generation for control flows including control nodes. The complex semantics of *JoinNode* requires its own template for the generation of join node implementations. Finally, the code generation for the implementation of action sequences, i. e. subactivities, is presented.

In the previous chapter, object flows are not included in the model transformation of dynamic behavior. Consequently, we do not include object flows at code generation either. Considering object flows makes templates more complex and less readable, while not generally changing the nature of the presented templates.

### 6.6.1 Processing Classifier Behavior

Listing 6.28 shows the general structure of the template for creating a class implementing an activity. The template is executed for each activity, but for subactivities, it produces no code. The decision is made by evaluating the name (line 2): if it contains the string *StartsWith*, the activity is identified as a subactivity. If the activity shall be implemented as an inner class, this template must be included in the class generation template and the `file` pattern becomes obsolete.

The rest of this template covers some technical instructions such as a Java package declaration and import statements (line 7–10) as well as a class declaration (line 13–20) for the activity.

```
1  [template public generateActivity(a : Activity)]
2   [if (a.name.contains('StartsWith'))]
3    [comment : nothing to do here /]
4   [else]
5    [file(activityFileName(a)↗C.13, false, 'UTF-8')]
6
7    package [containingPackages(a).name->sep('.') /];
8
9    import java.util.*;
10   import java.lang.Runnable;
11
12   /** This class implements the UML Activity [a.name/]. */
13   [visibilityString(a.visibility)/]
14   [if(a.isAbstract)]
15    abstract
16   [/if]
17   class [a.name /]
18   [if(not a.generalization->isEmpty())]
19    extends
20   [superClass(a).name /][/if]
21 {
22   [include ActivityContents/]
23 }
24   [/file]
25  [/if]
26 [/template]
```

Listing 6.28: Generation of a sourcefile for each activity.

### 6.6.2 Generating Activity Class Content

The actual and more individual content for each class implementing an activity is generated by snippet `ActivityContents` (line 22) given in Listing 6.29. It contains a field declaration for the context object (line 4) as well as a constructor supplying the field with its value (line 9–19). This constructor creates an ActivityThread for each node that is reachable from any initial node and

```
1  snippet::ActivityContents:
2  [include JoinNodeClassImplementation/]
3
4  private [a._context.name/] context;          // The context object
5  private boolean terminated = false;          // Activity termination flag
6
7  /** Creates an instance of [a.name/] and starts the execution of all
8    * sequences directly reachable from any initial node. */
9  public [a.name/]([a._context.name/] context){
10   this.context = context;
11   [for (node: InitialNode | a.node)]
12     [for (edge: ControlFlow |node.outgoing)]
13       new ActivityThread([a.node->asSequence()->indexOf(edge.target)/]);
14     [/for]
15   [/for]
16   [for (node: JoinNode | a.node)]            // start job for join nodes
17     new ActivityThread([a.node->asSequence()->indexOf(node)/]);
18   [/for]
19 }
20 /** Generate an inner class for implementing the control and data flow. */
21 [include ActivityThreadImplementation/]
22 /** Generate subactivies. */
23 [include SequenceImplementation/]
```

Listing 6.29: Generation of class contents for activities.

for each join node, which waits for being able to join tokens offered to incoming edges. Snippet `JoinNodeClassImplementation` (line 2) generates the complex implementation of join nodes as inner classes.

An ActivityThread is the implementation of the control flow in a main activity. It is contained in an inner class which is generated by snippet `ActivityThreadImplementation` (line 21). Finally, snippet `SequenceImplementation` (line 23) covers the implementation of subactivities.

The field `terminated` (line 5) indicates that, if set to true, an activity final node has been reached during the execution of an activity thread and all other running activity threads of the same activity execution must be aborted. It is required to properly implement join nodes.

### 6.6.3 Generating Code for Control Flow

Listing 6.30 contains snippet `ActivityThreadImplementation` which generates the control flow of a main activity. It contains a class declaration, a field declaration for an id which is a selector for the subactivity to execute and a constructor (line 9–12) which takes the value for this id.

```
1  snippet::ActivityThreadImplementation:
2  private class ActivityThread implements Runnable{
3    /** The id for selecting code sequences to be executed */
4    private int fId = 0;
5
6    /**Constructor creating and starting the execution of a code sequence.
7     * @param id the code sequence to be executed.
8     * @param context the instance which is the context of this activity. */
9    private ActivityThread(int id){
10     this.fId = id;
11     new Thread(this).start();
12   }
13   [include ActivityExecutionImplementation/]
14 }
```

Listing 6.30: Generation of code for the activity execution.

```
1  snippet::ActivityExecutionImplementation:
2  /** This method executes the activity */
3  public void run(){
4    while (fId > 0){
5      switch(fId){
6      [let seqNr : Sequence(ActivityNode) = a.node->asSequence()]
7        [for (node : ActivityNode | a.node)]
8          [if (node.oclIsKindOf(CallBehaviorAction))]
9            case [ seqNr->indexOf(node) /]:
10             [camelCase('ex', node.name)↗C.23 /]();
11             [let nextNode : ActivityNode =
12                    node.outgoing->asSequence()->first().target]
13               [if (nextNode.oclIsKindOf(JoinNode))]
14                 // next node is a join: add token to incoming list!
15                 [let eId : Integer = nextNode.incoming->asSequence()
16                           ->indexOf(node.outgoing->asSequence()->first())]
17                   join[seqNr->indexOf(nextNode) /].add[eId/](new Token());
18                 [/let]
19                 fId = 0;
20                 synchronized(context){ context.notifyAll(); }
21               [elseif (nextNode.oclIsKindOf(ForkNode))
22               or (nextNode.oclIsKindOf(DecisionNode))
23               or (nextNode.oclIsKindOf(MergeNode))
24                or (nextNode.oclIsKindOf(FinalNode))]
25                 fId = [seqNr->indexOf(nextNode)/];
26               [/if]
27             [/let]
28             break;
29         [/if]
30         [if (node.oclIsKindOf(ControlNode)
31           and not node.oclIsKindOf(InitialNode))]
32           case [ seqNr->indexOf(node) /]:
33             [include ControlNodeImplementation/]
34             break;
35         [/if]
36       [/for]
37       default: throw new RuntimeException("No such sequence in [a.name/]");
38     [/let]
39     }
40   }
41 }
```

Listing 6.31: Generation of code for the selection of sequences.

The run method of an ActivityThread is generated by snippet `ActivityExecutionImplementation` (line 13) and given by Listing 6.31 (line 4–40). It consists of a loop that is executed while the value of `fId` is greater than 0. Inside the while-loop, a switch statement is used to determine, according to the value of `fId`, which part of the activity to execute. The switch statement is assembled as follows:

1 Each node of a main activity is visited (line 7).

2 If the node is a *CallBehaviorAction* (line 8), a case block is prepared containing a method call invoking the appropriate action sequence (line 10).

3 The next node after a *CallBehaviorAction* is identified (line 11–12). If it is a *JoinNode* (line 13), a token is added into the appropriate incoming edge (line 17) and `fId` is set to 0. By that, this activity thread ends. The activity thread which has been started for the join node will continue activity execution, if the join node has input at all incoming edges and guards hold. In order to force the evaluation of the join node, `notifyAll()` is invoked on the context object (line 20).

4 If the next node is a *ForkNode* (line 21), *DecisionNode* (line 22), or *MergeNode* (line 23), `fId` is set to the appropriate value.

5 If the node is a *ControlNode* (line 30), but not an *InitialNode* ( line 31), code for handling control nodes is included by snippet `ControlNodeImplementation` (line 33).

In order to handle an invalid value of `fId`, a default statement is added throwing a `RuntimeException` (line 37).

### 6.6.4 Generating Code for Control Nodes

Listing 6.32 distinguishes the control node for which code is to be created. *DecisionNode*, *ForkNode*, and *JoinNode* are covered by own snippets.

A merge node is implemented by assigning the index of the node connected to its outgoing edge to `fId` (line 11). We assume that a merge node has only one outgoing edge and thus, the surrounding `for` loop executes in a single iteration. By having updated the value of `fId`, the activity execution continues at the correct position.

A flow final node is implemented by ending the current activity thread by setting `fId` to 0 (line 14).

In case of an *ActivityFinalNode*, the field `terminated` is set to true (line 16) and waiting threads are notified (line 17). Then, the current activity thread is ended by assigning 0 to `fId` (line 18).

```
1  snippet::ControlNodeImplementation:
2  [if (node.oclIsKindOf(DecisionNode))]
3    [include DecisionNodeImplementation/]
4  [elseif (node.oclIsKindOf(ForkNode))]
5    [include ForkNodeImplementation/]
6  [elseif (node.oclIsKindOf(JoinNode))]
7    [include JoinNodeImplementation/]
8  [elseif (node.oclIsKindOf(MergeNode))]
9    // implementation of a merge node
10   [for (e : ActivityEdge | node.outgoing->asSequence())]
11     fId = [seqNr->indexOf(e.target)/];
12   [/for]
13 [elseif (node.oclIsKindOf(FlowFinalNode))]
14   fId = 0;
15 [elseif (node.oclIsKindOf(ActivityFinalNode))]
16   terminated = true;
17   synchronized(context){ context.notifyAll(); }
18   fId = 0;
19 [/if]
```

Listing 6.32: Generation of code for control nodes.

### 6.6.5 Generating Code for Join Nodes

The implementation of joining control flows requires the explicit implementation of join nodes as inner classes. The implementation of joining flows bases on the implementation of join nodes.

#### Generating Join Node Implementation

Snippet `JoinNodeImplementation`, which is included in the snippet creating the content of activities (Listing 6.29, line 2), is given by Listing 6.33. It consists of three parts:

- declaration of a inner class representing control tokens (line 3)
- implementation of join nodes (lines 7–39) comprising
  - an implementation of incoming edges (lines 9–17)
  - the generation of a method for offering tokens to the node (lines 13–15)
  - the implementation of actually joining offered tokens (lines 19–38)
- creation of an instance for each join node of an activity (lines 41–42).

```
1  snippet::JoinNodeClassImplementation:
2   /* Control Token implementation */
3   private class Token{}
4
5 [let seqNr : Sequence(ActivityNode) = a.node->asSequence()]
6   [for (node : JoinNode | a.node)]
7   /* Implementation of join node [ seqNr->indexOf(node) /] */
8   private class Join[seqNr->indexOf(node)/]{
9     [for (inEdge : ActivityEdge| node.incoming)]
10      [let joinId : Integer = node.incoming->asSequence()->indexOf(inEdge) ]
11    private List<Token> l[ joinId /] = new LinkedList<Token>();
12
13    public void add[ joinId /](Token token) {
14      l[ joinId /].add(token);
15    }
16      [/let]
17    [/for]
18
19    public boolean canJoin() {
20      if(!(
21    [for (inEdge : ActivityEdge| node.incoming) separator ('||')]
22        l[ node.incoming->asSequence()->indexOf(inEdge) /].isEmpty()
23    [/for]
24      )){
25        if(
26    [for (inEdge : ActivityEdge| node.incoming) separator ('&&')]
27      ([ inEdge.guard.stringValue() /])
28    [/for]
29        && ([ node.outgoing->asSequence()->first().guard.stringValue() /])
30        ){
31    [for (inEdge : ActivityEdge| node.incoming)]
32          l[ node.incoming->asSequence()->indexOf(inEdge) /].remove(0);
33    [/for]
34          return true;
35        }
36      }
37      return false;
38    }
39  }
40
41  private Join[seqNr->indexOf(node)/] join[seqNr->indexOf(node)/] =
42    new Join[seqNr->indexOf(node)/]();
43  [/for]
44 [/let]
```

Listing 6.33: Generation code for join nodes.

The type `Token`, which is implemented as an inner class (line 3), is required for the implementation of incoming edges of the join node.

Join nodes of an activity are implemented as inner classes (line 8), too. The class name is a concatenation of the prefix `Join` and the index of the node in the sequence of nodes (line 5).

For each incoming edge, its index in the sequence of edges is determined (line 10). A list containing offered tokens (line 11) as well as a method for adding offered tokens to that list (line 13–15) are created.

The method `canJoin()` (line 19–38) returns true, if all incoming edges have tokens (line 20–24) and if guards hold (line 25–30). Before returning true (line 34), tokens offered at incoming edges are consumed (line 31–33). If not all guards hold or if not all incoming edges are supplied with token offers, `canJoin` returns false (line 37).

Finally, code is inserted to create an instance for each join node (lines 41–42).

### Generating Code for Joining Control Flows

Code generation for handling a join node is given in Listing 6.34. For each join node, an ActivityThread is started when starting the execution of an activity. This thread waits (line 6) while it cannot join tokens offered to incoming edges (line 4).

```
1  snippet::JoinNodeImplementation:
2  // implementation of a join node
3  synchronized (context) {
4    while (! join[seqNr->indexOf(node) /].canJoin() && !terminated){
5      try {
6        context.wait();
7      } catch (InterruptedException e) {
8        e.printStackTrace()
9      }
10   }
11   fId = [seqNr->indexOf(node.outgoing->asSequence()->first().target)/];
12 }
13 if (terminated)      // start a new job waiting for a join?
14   fId = 0;
15 else
16   new ActivityThread([a.node->asSequence()->indexOf(node)/]);
```

Listing 6.34: Generation of code for *JoinNode*.

In line 11, code is generated to update `fId`. The current activity thread thus will continue activity execution. In order to be able to join other tokens at the join node, another thread must be created. Therefore, depending on whether the activity has been terminated (line 13), either a new thread is started (line 16) or it is ended (line 14).

## 6.6.6 Generating Code for Decision Nodes

Generation of code implementing a decision node is given by Listing 6.35. It consists of two parts:

- waiting until a decision can be made
- causing the activity execution to continue according to the result of the decision.

```
1  snippet::DecisionNodeImplementation:
2  // Implementation of a decision node
3  synchronized (context) {
4    while (!([node.incoming->asSequence()->first().guard.stringValue()/] && (
5    [for (e : ActivityEdge | node.outgoing->asSequence()) separator ('||')]
6    ([e.guard.stringValue()/])
7    [/for]
8    ))){
9      try {
10       context.wait();
11     } catch (InterruptedException e) {
12       e.printStackTrace();
13     }
14   }
15   [for (e : ActivityEdge | node.outgoing->asSequence())]
16     [ifLiteral(node.outgoing->asSequence()->indexOf(e))↗C.27 /]
17       ([e.guard.stringValue()/])
18     fId = [seqNr->indexOf(e.target)/];
19   [/for]
20 }
```

Listing 6.35: Generation of code for *DecisionNode*.

The first part is implemented as a while loop, which is executed repeatedly as long as the decision node cannot emit a token at any of its outgoing edges. This is the case if the guard of the incoming edge does not hold (line 4) or for all of the outgoing edges (line 5), guards do not hold, too (line 6). Inside the loop body, the thread waits (line 10).

When the loop is no longer executed, it is determined at which outgoing edge to continue. For this purpose, a complex if-then-else statement is created:
For each outgoing edge (line 15), a condition is appended. The query `ifLiteral` returns the string `if` for the first edge, `else if` for all other edges (line 16). After the keyword `if` or `else if`, the guard is included (line 17). In the body of each `if` or `else if` block, the value of `fId` is set to the index of the target node of the outgoing edge (line 18).

Since the code which determines where to continue the activity execution is only reached when the guard of at least one outgoing edge holds, `fId` definitely will be set in one of the `if`/`else if` blocks.

### 6.6.7 Generating Code for Fork Nodes

Generation of code implementing a fork node is given by Listing 6.36. It is almost identical to the template for generating the implementation of a decision node.

The only difference compared to the implementation of a decision node is that tokens may be emitted at multiple outgoing edges, i. e. the execution may continue with concurrently executed outgoing flows. Therefore, the evaluation of guards at outgoing edges is not implemented by a sequence of `if ... else if ... else` but by a set of `if` statements which are independent from each other (line 16). If the condition, which is the guard of an outgoing edge holds, a new ActivityThread is created (line 17). The index of the target of the outgoing edge is provided as a parameter. The `fId` of the current activity thread is set to 0 (line 19).

```
1  snippet::ForkNodeImplementation:
2  // implementation of a fork node
3  synchronized (context) {
4    while (!([node.incoming->asSequence()->first().guard.stringValue()/] && (
5    [for (e : ActivityEdge | node.outgoing->asSequence()) separator ('||')]
6      ([e.guard.stringValue()/])
7    [/for]
8    ))){
9      try {
10       context.wait();
11     } catch (InterruptedException e) {
12       e.printStackTrace();
13     }
14   }
15   [for (e : ActivityEdge | node.outgoing->asSequence())]
16     if ([e.guard.stringValue()/])
17       new ActivityThread([seqNr->indexOf(e.target)/]);)
18   [/for]
19   fId = 0;
20 }
```

Listing 6.36: Generation of code for *ForkNode*.

### 6.6.8 Generating Code for Sequences of Actions in Subactivities

The implementation of action sequences is given in Listing 6.37. Code is generated for each *CallBehaviorAction* (line 2). The signature of a method implementing a sequence is generated in lines 6–9. It consists of the visibility modifier `private`, the return type `void`, if the action has no outputs (line 7) or the return type of the action outputs (line 8), followed by the method name, built by the prefix `ex` and the node name. If the node has inputs, method parameters are included by calling the query `paramString` (line 9).

Since a *CallBehaviorAction* has exactly one behavior, the `for` loop of lines 10–26 is executed once. Inside the loop, a nested loop iterates over all initial nodes (lines 11–25). According to

the transformation of models of dynamic behavior, a subactivity has only one initial node with a single outgoing flow. The target of that flow is an action (line 13). For this action, the sequence of itself and all downstream actions is obtained by calling the query *getSequence* (line 14). For each action of this sequence (line 16), if it is an *OpaqueAction* (line 19), a method call is generated (line 20). The method calls on the context object are valid since code generation for static structures includes the generation of abstract methods for *OpaqueActions*. By that, the implementation at runtime is enforced.

Inside the `for` block (lines 16–22), specific code could also be generated for other kinds of actions according to the outlined implementations of Sect. 4.3.1. Handling other kinds of actions would be included in `else-if` blocks inserted after the `if` block, that only handles opaque actions (lines 19–21).

```
1  snippet::SequenceImplementation:
2    [for (node : CallBehaviorAction | a.node)]
3
4   /** Implement the action sequence of CallBehaviorAction [node.name/]
5    *  Generated by ActiveCharts. */
6    private
7    [if (action.output->asSet()->isEmpty())] void
8    [else] [action.output.type.name/] [/if]
9    [camelCase('ex', node.name)/]([paramString(action) /]){
10     [for (sub : Activity | node.behavior)]
11       [for (initial : InitialNode | sub.node)]
12         [if (initial.outgoing->asSequence()->first().target.
13                                                 oclIsKindOf(Action))]
14           [let actions : OrderedSet(Action) = getSequence(initial)
15                                                   ->asOrderedSet()]
16             [for (action : Action | actions)]
17               [include ConsiderGuards/]
18               [include ConsiderIRs/]
19               [if (action.oclIsKindOf(OpaqueAction))]
20                 context.[action.name/]([paramString(action) /]);
21               [/if]
22             [/for]
23           [/let]
24         [/if]
25       [/for]
26     [/for]
27   }
28   [/for]
```

Listing 6.37: Generation of code for sequences of actions.

### Considering Guards

Regardless of the kind of action to execute, guards must be tested before in order to only execute actions if all token flow prerequisits are satisfied (line 17 in Listing 6.37). The generation for the required code is given in Listing 6.38.

Since the result of the evaluation of a guard may depend on values in the context object, the code for testing guards must not be interrupted by the system scheduler in order to prevent changes to the context object during guard evaluation. In order to be thread safe, it is contained in a `synchronized` block (lines 3–16).

Between actions in a sequence of actions, exactly one control flow exists. The `for` block (lines 4–15) will only match this single control flow. If the flow is guarded (line 5), the guard is evaluated inside the loop condition of a `while` loop (line 6). If the guard does not hold, the thread is suspended (line 7) and may be resumed if being notified after a modification of the context object, or if the tread is to be aborted (lines 9–12). If it is to be aborted, the `fId` is set to 0.

```
snippet::ConsiderGuards:
// wait until all guards hold
synchronized(context){
[for (g_edge : ActivityEdge | action.incoming->asSequence())]
  [if not gedge.guard.oclIsUndefined()]
  while (![g_edge.guard.stringValue()/]) && ! terminated()){
    try {
      context.wait();
    } catch (InterruptedException e) {
      // This sequence is aborted...
      fId= 0;
    }
  }
  [/if]
[/for]
}
```

Listing 6.38: Generation of code for considering guarded flows.

The code lines after testing guards are only reached if the conditions of guards hold, i. e. if the tokens actually may traverse edges. In order to behave correctly if a sequence is aborted, i. e. if `fId` is set to 0, before executing any further lines of code, the value of `fId` must be checked.

A proper consideration of guards requires modifications to the code generated for merge nodes, as given in Listing C.1 in Appendix C.1

### Considering Interruptible Activity Regions

After having checked guards and the value of `fId`, it is necessary to consider the effects of interruptible activity regions before executing actions, because action executions might be affected by the abortion of an interruptible activity region, if actions are located inside such a region, or the token flow might trigger the abortion of a region, if the traversed edge is an interrupting edge.

Listing 6.39 shows how to generate code considering interruptble activity regions.

```
snippet::ConsiderIRs:
// manage entering, leaving, and aborting interruptible activity regions...
synchronized(context){
[for (enteredRegion : InterruptibleActivityRegion | action
  .inInterruptibleRegion)]
  [if (action.incoming->asSequence()->first().source
    .inInterruptibleRegion->excludes(enteredRegion))]
  register(ActivityThread.this, ir[enteredRegion.node.name/]);
  [/if]
[/for]
[for (leftRegion : InterruptibleActivityRegion | action
  .incoming->asSequence()->first().source.inInterruptibleRegion)]
  [if (action.inInterruptibleRegion->excludes(leftRegion))]
  unregister(ActivityThread.this, ir[leftRegion.node.name/]);
  [/if]
[/for]
[for (ir : InterruptibleActivityRegion | action
  .incoming->asSequence()->first().interrupts)]
  interruptRegion(ir[ir.node.name/]);
[/for]
}
```

Listing 6.39: Generation of code for entering, leaving and aborting interruptible activity regions.

Implementing the semantics of interruptible activity regions consists of three parts:

- registering threads entering a region (lines 4–10)
- unregistering threads leaving a region (lines 11–16)
- interrupting regions if tokens traverse interrupting edges (lines 17–20).

A thread executing a sequence of actions must be registered for an interruptible activity region, if the target of a flow is located inside an interruptible activity region which does not contain the source of the flow.

A thread must be unregistered if the source of a flow is contained in an interruptible activity region in which the target is not contained. Here, it makes no difference if the edge connecting two actions is a normal edge or an interrupting edge. The current thread must be unregistered in either case in order to prevent it from being aborted or from aborting itself.

If the incoming edge of the action to be executed next is an interrupting edge, the concerning interruptible activity region is to be aborted.

The generation of the methods `register`, `unregister`, and `interrupt` is contained in the Appendix C.3 in Listing C.8.

## 6.7 Summary

With the code generation templates presented in this chapter, the PSM generated by the model transformations discussed in Chapter 5 can be translated into a basic Java implementation.

Regarding static structure, code for implementing classes with owned attributes as well as classes implementing associations are considered. Operations for accessing owned attributes as well as association ends are included, too. Furthermore, the implementation of the *handle pattern* as introduced in Sect. 3.6.1 is part of the presented code generator.

Concerning dynamic behavior, abstract methods are generated for the implementation of opaque actions. Furthermore, the general pattern for activity execution and the implementation of sequences of actions is covered by the presented templates. The complete semantics of control nodes is considered, but limited to the constraint that control flows between actions may only contain one single control node. The effects of guarded flows are considered as well.

Although not all modeling elements which are considered in our discussion of UML semantics in Chapter 3 and Chapter 4 are included, these templates give a solid foundation to base on for further template developments to include additional modeling concepts such as object flows, interruptible activity regions or the implementation of other kinds of actions as sketched in Sect. 4.3.1.

The presented transformations are included in a prototypical implementation which represents our most advanced version of the ActiveCharts implementation. A survey from the beginning of our research in the development of a MDD approach and supportive tooling is presented in the following chapter.

# Chapter 7

# Tool Support

In order to contribute to Model-Driven Development, our transformations from models to executable code ultimately must be workable. We prove this by prototypical implementations. This chapter surveys the evolution of these prototypes, starting with the first approach of interpreting UML2 activities and ending with our most recent approach of completely translating models to runnable code.

Section 7.1 outlines the first step from a bundle of tools to an integrated IDE. Section 7.2 comprises the technological evolution of the translation from models to code implemented in Java. The step from a Java implementation to transformation languages is sketched in Sect. 7.3. In Sect. 7.4 we roughly glance over UML compliance of related tools, compare the different stages in our own history of tool support, point out limitations as well as proposals for extensions.

## 7.1 Evolution of ActiveCharts

The story of ActiveCharts is a story of several steps each of which contributed to the current state of our knowledge and tool support concerning activities as a means of modeling software behavior. Although applying different techniques to base activity execution on, the extent of tool support has always been targeted to cover aspects which are typical for activities, such as concurrency, interruptible activity region, or object flow.

Concerning static structure, class, generalization, attribute, and binary association are supported. Over time, the extent to which UML concepts are supported has slightly increased, e. g. attention has been given to higher order associations and association classes.

### 7.1.1 ActiveChartsIDE

The first prototype, ActiveChartsIDE, primarily serves as an implementation of the formal semantics of activities from Sarstedt [83]. Basic information about Sarstedt's implementation is contained in [84], a detailed tutorial on its usage is given in [82]. Its main purpose is the visualization of token flow in activity diagrams and the execution of call operation actions, which are implemented in methods of context classes.

Although ActiveChartsIDE succeeds in visualizing the token flow in activity diagrams and provides debugging facilities for interactive stepping through activity executions, the implementation suffers from deficiencies with regard to runtime behavior. The straightforward implementation of the formalization given by means of Abstract State Machines [42] results in inefficient code.

Trying to model a controller for elevators revealed the inability to stop elevator cabins on floor level due to badly effects on signal processing caused by inefficient algorithms. Moreover, using Microsoft VISIO [8, 98] as a UML editor and for visualizations, ActiveChartsIDE is at a disadvantage compared to tools entirely based on products licenced at no charge. These two points clearly indicated the promising nature of a reimplementation comprising redesign.

The complex architecture of ActiveChartsIDE is depicted by Fig. 7.1. The tool comprises three parts: VISIO for modeling and visualization, some kind of development environment for writing and compiling code and ActiveChartsIDE itself.

Starting from graphical model representations, ActiveChartsIDE imports activity diagrams and persists them in a textual, XML-based representation. Furthermore, it generates code for

the implementation of the static structure as modeled in class diagrams. Implementations of call behavior actions are added by the user in a separate code file. Generated code and user written code are compiled to an executable.

The interpreter uses the activity import module to load either activity diagrams or activities stored as xml files. It executes the loaded activities and calls methods contained in the executable. Visualization of an execution with debugger facilities is provided by the *Visualization* component of ActiveChartsIDE, which starts and controls the execution of the interpreter. The debugger supports stepping through activities, definition of break points, and inspection of context classes. It communicates with the visualization component which depicts the state of an activity execution by diagram augmentation with tokens and highlighting of edges traversed by tokens.

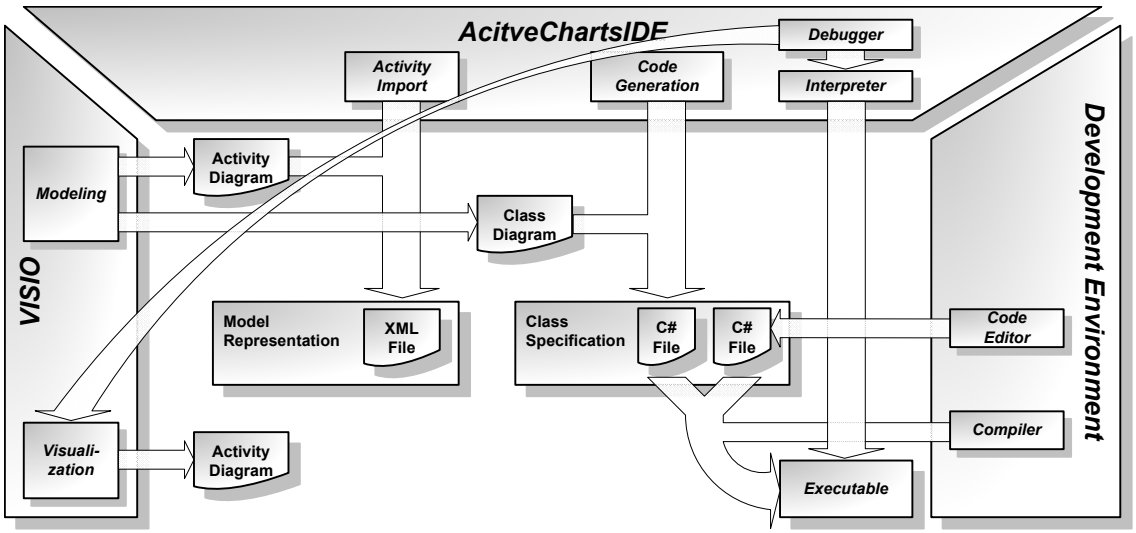



Figure 7.1: Architecture of ActiveChartsIDE.

### 7.1.2 Migration to Eclipse

The following reasons triggered a re-implementation of ActiveChartsIDE as a bundle of eclipse plugins:

- basing on eclipse offers to use and tightly integrate visual modeling editors for UML which are implemented as eclipse plugins as well
- an implementation of the UML2 meta-model is available as part of the eclipse Model Development Tools (MDT).

Thus, an eclipse based re-implementation offers better tool support for modeling, and the need to implement a custom meta-model becomes obsolete.

#### Improvements of the Execution Engine

Since the implementation language of our tool no longer is Microsoft C# but Java, the execution engine is implemented completely anew. The recalculation of many items of information at every execution step is now avoided by using lookup tables and results of earlier execution step computations [13, 36].

#### Java Based Code Generation

The transition from C# to Java as a target language for code generation from class models entails as a necessity to substitute the partial class concept by another mechanism available in Java as well. Furthermore, the very rudimentary support of association classes and the missing consideration of composition semantics is targeted by the mapping from UML class diagrams to Java code [33].

## 7.2 Approaching a Complete Model Translation

Although execution speed has significantly advanced with the re-design and re-implementation of the activity interpreter, aspiring the translation of UML activities to object-oriented code is due to a tighter integration of both, structural and behavioral models. As this thesis focuses on dependencies between structures and behaviors, the previous chapters attest that behavior execution must include aspects of structures. Such aspects, e. g. multiplicity bounds, must be made available to an execution engine unless these information is obtained from code of generated classes. To that effect, an activity execution engine must also process the model of static structure.

Another solution in favor of a once more faster activity execution is to translate activities into code in the way of our approach. By doing so, structure and behavior are represented in the same way (as code) and an execution environment for activities becomes obsolete.

### 7.2.1 ActiveCharts First Generation Prototype

Our first implementation of a code generator for static structures is purely programmatic. Figure 7.2 overviews the interplay between the modules of the eclipse based ActiveCharts. Third party UML editor plugins support graphical modeling of UML diagrams. An implementation of the UML metamodel based on the Eclipse Modeling Framework provides the foundation for building a UML model according to the graphical specification.

The model transformation directly operates on the model generated by the graphical UML editors. The resulting transformed model is the input for the code generator, which writes Java source code line by line. The generated code is translated by the Java compiler.

The activity interpreter operates on the model for executing the activity and on the class files for invoking operations and accessing attributes and association ends.

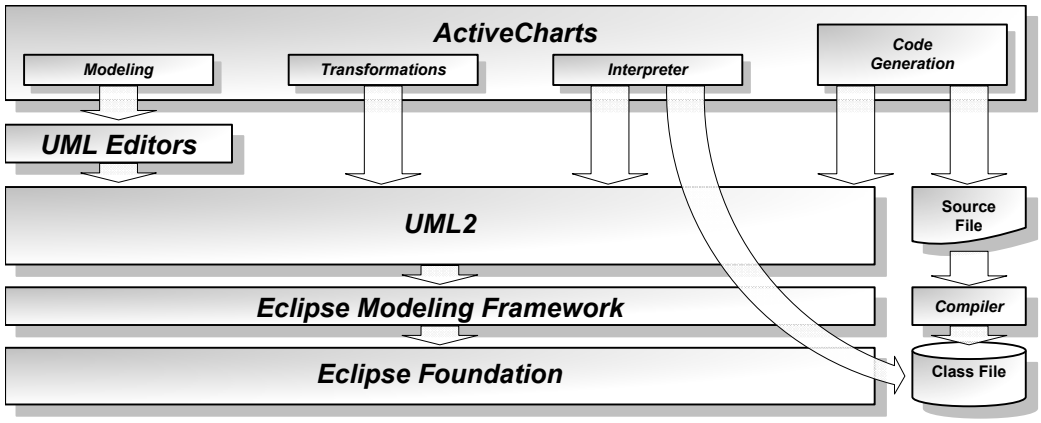



Figure 7.2: Architecture of ActiveCharts First Generation Prototype.

Implementations of model transformation and code generation on such a low level of abstraction tend to become very complex, badly readable and almost un-maintainable. However, due to the multitude of limitations revealed by an evaluation of tools either based on template languages or on transformation frameworks such as *openArchitectureWare*[1], directly accessing models and source code representations seems favorably. In fact, most limitations are found when using template languages, which most often are proprietary developed by tool vendors.

### 7.2.2 ActiveCharts Second Generation Prototype

A first attempt to base code generation on a higher level of abstraction is to create an *Abstract Syntax Tree (AST)*. In contrast to directly writing code, the construction of such a tree is flexible with regard to when a certain item of information is implemented: when writing code line by line, the order in which information is added to the generated code is inherent; when constructing a tree, child nodes can be added to each node at any time during the generation process.

[1] *OpenArchitectureWare* by now is part of Xpand, which itself is included in the eclipse project Model To Text (M2T) (https://projects.eclipse.org/projects/modeling.m2t).

As shown in Fig. 7.3, the code generator accesses the model to build an AST covering both, structural and behavioral aspects of the model. When the tree is complete, it is written to Java source files and compiled to Java classes.

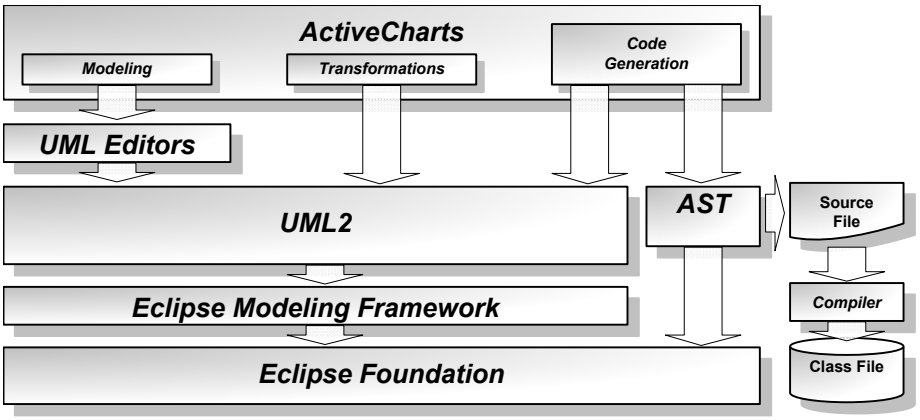



Figure 7.3: Architecture of ActiveCharts Second Generation Prototype.

## 7.3 ActiveCharts Third Generation Prototype

We often refer to the positive effects of working on higher level of abstractions. Unfortunately, the positive effects make no impact on our first generation prototype. But by applying transformation languages supporting a descriptive specification, the translation of models to code can be coped with on a much higher level of abstraction.

The third generation of our tool directly uses the formalized transformations. We consider the executable specifications of our model transformation as presented in Chapter 5 and the transition to code as stated in Chapter 6 to be well readable, maintainable, and therefore, a good basis for extensible tool implementation. From this point of view, the formalizations provided as part of this work is not only a contribution to theory but also an initial implementation of a core that could be advanced and refined to a release candidate. Figure 7.4 shows the third generation prototype. As can be seen on the left side, modeling still is based on the UML2 metamodel and graphical editors based on it. The model transformation is achieved by *medini QVT*, the source files are generated by *Acceleo*. A runtime module connecting the model and the generated code no longer exists, since our third generation tool completely translates the system specification into runnable code for which no interpreter other than the JVM[2] is needed.

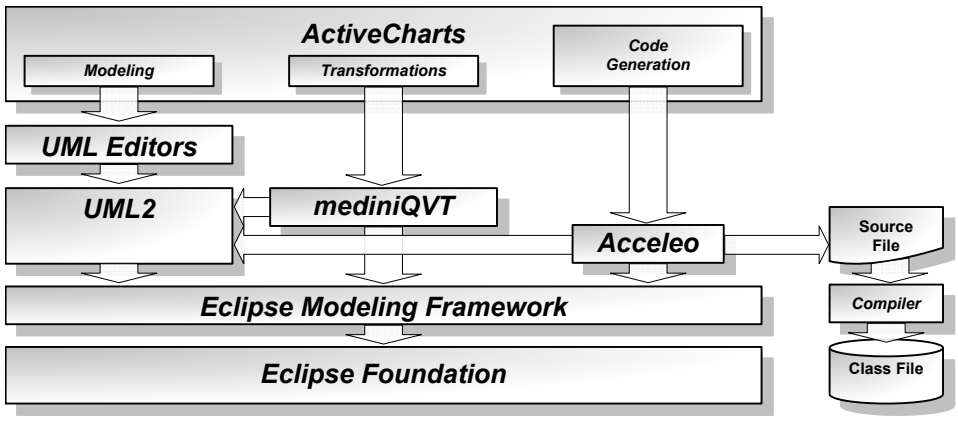



Figure 7.4: Architecture of ActiveCharts Third Generation Prototype.

[2]Java Virtual Machine

## 7.4 Discussion

In this section, we glance over UML compliance of tools that are comparable to our implementations and compare the characteristics of the different generations of our implementations to each other.

### 7.4.1 Level of Compliance Compared to Related Tools

Many tools claim to be UML compliant, but most of them are not really. For example, ownership of association ends is rarely supported. Tools either make associations or associated classifiers to end owners, without any possibility to change this and with no semantic effects resulting form the choice for one or the other option. As a matter of fact, tools on such a level of UML compliance cannot generate code with the same preciseness as pursued by us.

A detailed analysis of UML tools [25] in 2009 reveals, that modeling activities is well supported by very few tools, e. g. Papyrus [114]. The processing of activities, either execution by interpreters or code generation, is only rudimentarily supported, if at all. Modeling static structure is quite fairly supported by most tools, although advanced concepts like end ownership are often missing. Generally speaking, modeling and processing of models should be improved. The situation regarding tool support as examined in 2009 has not substantially changed until today [3, 6].

Code generation usually bases on special frameworks or on template languages. Most often, basic implementations or templates are supplied, which can be extended to cover a comprehensive code generation. However, both mechanisms are often proprietary and consequently, interoperability of code generators based on templates is not given or restricted, i. e. templates cannot be used in tools other than from the same vendor.

Another common problem is, that models are not directly accessible. If a tool with suchlike restrictions uses templates and a template is processed for each class, then code generation for associations becomes difficult since neither the model is accessible, nor can templates be defined to be processed for each association. Advanced concepts like implementation of compositions by cascading object destruction often cannot be addressed by those tools.

### 7.4.2 Architecture

Our eclipse based implementations rest on a freely available, UML compliant meta-model implementation. The design of our first generation prototype, however, offers too little abstraction by directly writing source code into Java files. The hardest problem to face under this approach is that code generation either must produce lines of code in the order they appear in the source file or points where to insert code into previously created code fragments must somehow be identified.

Our second generation prototype solves this problem by basing on the AST. But code generation is still cumbersome since the level of abstraction is low. For a single line of code, a couple of tree nodes must be created representing e. g. statements, assignments, operators, or arguments. The resulting code is no good basis for a code generator that should be maintainable and extensible.

In consideration of the fact that users might be interested in adjusting the output of a code generator to their individual needs, in particular because supporting additional modeling concepts might be demanded, an acceptable degree of readability, maintainability, and extensibility is inevitable. Such a high degree has not been reached before basing on standardized transformation languages for model to model as well as model to text transformations. Therefore, our third generation prototype combines powerful and comprehensive initial implementations with an architecture applying techniques and formalisms which accommodate extending or tailoring tools to individual preferences and requirements.

### 7.4.3 Comparison of Runtime Characteristics

Regarding the execution of behavior, we identified some runtime characteristics and compared the results obtained by evaluating the ActiveCharts interpreter, the eclipse based interpreter and the output of our third generation prototype. The considered characteristics are

- duration of executions
- code size
- memory allocation.

A detailed presentation is given in our publication *UML Activities at Runtime* [36]. In the following, we roughly summarize the main findings.

As expected, using lookup tables for token flow computation results in faster executions of activities. Our eclipse based interpreter outruns the ActiveCharts interpreter, except if many activities are executed which are not very complex. In such a scenario, creating lookup tables may consume more time than can be saved during the entire activity execution. The fastest activity execution is achieved by code generation for activities, as provided by the third generation prototype.

Regarding code size, our eclipse based interpreter is smaller than the ActiveCharts interpreter. Whether or not the difference in code size results from the less complex flow algorithms due to the use of lookup tables cannot be answered for sure, since it probably also is influenced by the application of different target languages[3]. But in contrast to the code generation approach for activities, code size of interpreters is constant. With growing complexity and size of input models, the size of model representations[4] needed by interpreters turned out to grow slower than the generated code. However, small activities result in code smaller than the interpreter code. But since generated code is dependent from many factors it is not feasible to try to give a model marking a break-even point. For very large models, the interpreter approach scales better in terms of code size, but worse in terms of execution duration.

Allocation of memory scales differently for interpreter and generator approaches. Our interpreters allocate memory for storing the token configuration and furthermore for holding an instance of the entire model. Therefore, beside the memory needed to load the model, much memory is allocated if many tokens are present, i. e. many tokens wait for guards or for tokens to join with.

How much memory is allocated when running generated code depends on the structure of a model, in particular on the number of join nodes, since for them, instances are created regardless of whether tokens are offered to incoming edges or not. Generally, tokens processed by interpreters correspond to threads executing generated code. Depending on the stack frame of a thread, many waiting threads can cause more or less memory allocation, even if they represent control tokens, for which interpreters always demand the same memory allocation.

### 7.4.4 Current Limitations

In this section, we do not discuss limitations of our approach, but we list limitations of our tool support.

ActiveChartsIDE uses VISIO for modeling. With our first eclipse implementations, we also developed class diagram and activity diagram editors offering support for basic modeling concepts. Whereas with VISIO, a fairly usable editor is employed, our own editors are less usable, but tailored to our demands. Since in the last years, third party UML editors have increased in usability and amount of supported modeling concepts, development of our own editor has been discontinued.

ActiveChartsIDE also provides a visualization of token flow [32]. Although aspired for our eclipse based interpreter as well, visualization is only included on a very basic and rudimentary level. With regard to our code generation approach, code for visualizing the execution of an activity could be inserted as a kind of debug info, however, it is not addressed yet.
Furthermore, visualization is limited to our own editors. Including visualization in a third party editor is possible if source code is available. PapyrusUML [114] is an open source tool thus qualifying itself to be a basis for a modeling and visualization tool. An extension of PapyrusUML is currently considered, but has not yet been started.

Still a problem is the fact that there is no editor basing on eclipse UML available which supports all modeling elements our approach uses. With regard to modeling structures, support of higher order associations often is not supported, with regard to modeling behavior, support of actions most often is unsatisfactory.
Actually, our test models have not been developed by drawing diagrams but by using the model editor comprised in eclipse UML. Although it is extremely uncomfortable to design models in a tree view representation, it actually is the only way to get models containing those modeling concepts which to deal with constitutes the strength of our approach.

[3] ActiveChartsIDE generates C# code, the eclipse based prototypes create Java code.
[4] Models are typically persisted as xml files.

### 7.4.5 Possible Extensions

A valuable extension were the development of a modeling editor and visualization component overcoming the limitations described above. Furthermore, debugging facilities like those offered by ActiveChartsIDE are not yet available in our eclipse based implementations.

With basic considerations about distributed activities [77], deployment diagrams are to be included in our approach. If doing so, visualization options for deployments could be elaborated and included in a next generation prototype.

Since UML offers further formalisms for behavioral modeling, namely state machine and collaboration, diagramming could be extended to support both in order to be prepared for a possible extension of our approach based on the semantic description of Kohlmeyer [51, 50].

#### Model Generation

In Chapter 3, we argued for a proper consideration of constraints specified in structural models, such as multiplicity bounds and aggregation or composition semantics. In particular, we proposed to check bounds of association ends and to reject modifications causing a violation of them. In Sect. 4.4.1, we presented an activity for checking multiplicity bounds when accessing structural features. Thus, we provide means for considering structural integrity on the level of modeling as well as on the level of implementation.

Regarding the option to include checking multiplicity bounds in behavioral models, good tool support should generate the needed activities since those activities are not trivial and must be created and maintained in case of changing the structural model. Generating such activities is possible due to structural models containing all required information.

The option of addressing the structural integrity of a system in code by implementing actions with additional semantics not backed up by the specification of UML is problematic because the implementation is not aligned to the modeling language semantics. Accordingly, it is preferable to reflect the entire semantics of the intended implementation into the model. But this turns the idea of MDD upside down: instead of deriving implementations from models, a model is derived from its semantically stronger implementation. Such a model explicitly shows that structural integrity is checked and exceptions are raised when a change results in an invalid system state. This explicit representation of accounting for structural integrity is a better strategy to deal with the missing alignment between the semantics of actions on the one hand and of structural features on the other hand. An example is given in Fig. 7.5, which shows an activity implementing an *AddStructuralFeatureValueAction* with additional semantics, which is throwing an exception in case of a violation of multiplicity bounds.

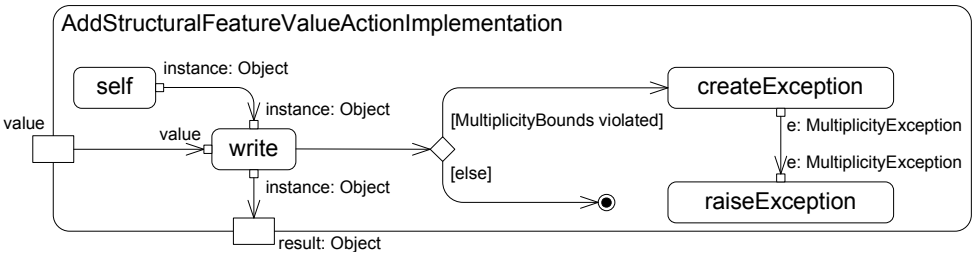



Figure 7.5: Activity adding semantics to *AddStructuralFeatureValueAction*.

Each *AddStructuralFeatureValueAction* of a model can easily be augmented with the additional semantics of *AddStructuralFeatureValueActionImplementation* of Fig. 7.5:

- for each action $a$ to be supplied with the additional semantics, an activity *AddStructuralFeatureValueActionImplementation* must be created
- action $a$ must be replaced by a *CallBehaviorAction* associated to the created activity
- action *write* in the created activity must be replaced by action $a$ (from the first bullet point).

Since several possible ways exist to cope with multiplicity bound violations, tooling designs should consider to offer adaptability to developers. In the following, we outline a design incorporating this demand. It is based on a three tier architecture as shown in Fig. 7.6.

- Level $L_0$ contains an abstract activity serving as the root for accessing structural features. *AddValue* is an *OpaqueAction* which implements the activity shown in Fig. 7.5.
- Level $L_1$ contains activities which are specializations of the abstract activity of $L_0$. Such an activity exists for each structural feature that is accessed from within dynamic behavior specifications. It specifies the types of object nodes owned by the abstract activity from $L_0$ to the types required in the context of each structural feature as well as it redefines *AddValue* according to the concrete context.
- Level $L_2$ denotes the set of activities in which structural features actually are accessed. Instead of using *StructuralFeatureAction*, *CallBehaviorAction* associated to an activity of $L_1$ is applied.

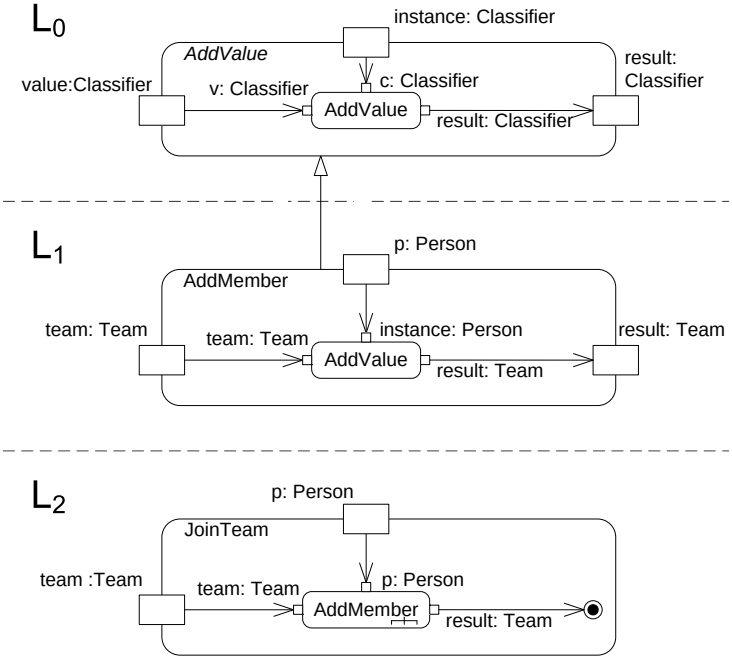



Figure 7.6: Three tier architecture for considering structural integrity.

The basic consideration behind this three layer architecture is that instead of using *StructuralFeatureAction*s of UML, activities like that of Fig. 7.5 should be used. But since these activities serve as a documentation of the actual implementation of associations rather than code is generated for them, they must not be changed. Therefore, in order to prevent any changes and to achieve a higher level of abstraction, the semantics of implementations for accessing structural features is encapsulated by opaque actions.

This actions throw exceptions if multiplicity bounds are violated. The reaction to an exception may be modeled on $L_2$ level, i. e. individually for each occurring access. If handling exceptions on $L_1$ level, each access on the same structural feature is uniformly specified. If specifying exception handling on $L_0$ level, violations of the structural integrity are uniformly handled for all structural features.

An example is given in Fig. 7.7. Instead of throwing an exception — what actually happens in case of a multiplicity bounds violation — a boolean result indicates whether or not a value could be added. The same adaptations could be applied on the $L_2$ level, thus affecting only the attempt of adding a person to a team in the activity *JoinTeam*. If applied to the activity *AddValue* on level $L_0$, any attempt to add a value to any structural feature would no longer throw an exception, but rather a boolean indicator of success would be provided.

This example also shows that generalization applied to activities can beneficially be used in modeling behavior and motivates future work on its semantics as well as its inclusion in tooling. It allows adapting the responses to invalid system state modifications locally, globally, or globally

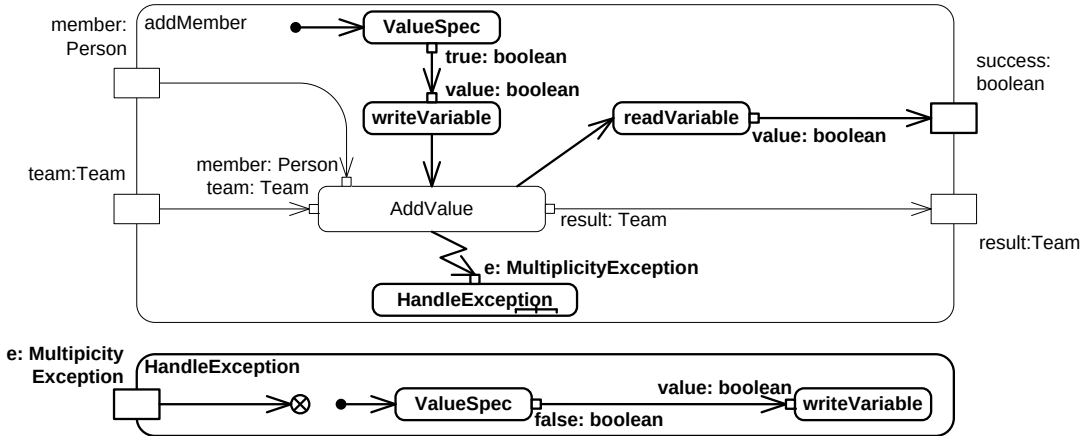



Figure 7.7: Adapting reaction to multiplicity bounds violations.

but restricted to an individual structural feature. Furthermore, it introduces a straightforward option to align models with our implementations.

### 7.4.6 Concluding Remarks

With our tools, we provide a basis for processing activities based on two different, alternative techniques: interpretation and code generation. Both have their strengths and therefore, including both is a good choice for comprehensive tool support. The applied architecture and formalisms particularly with regard to the implementation of code generators contribute to an excellent degree of extensibility and maintainability, which is not reached by our implementations of interpreters. Using standards such as QVT or Mof2Text to implement our own prototypes demonstrates the benefits of working at higher levels of abstraction, not only in model-driven development itself, but also in the development of MDD tools.

Employment of our tool in practical courses has revealed that generated code for static structure can easily be used and accessing features is convenient since provided methods have meaningful names derived from input models. Likewise, the implementation of behavior by providing code for actions but without being responsible for control and data flows has turned out to be well manageable. Modular implementation of more complex algorithms, based on interacting actions and each representing relatively small, self-contained tasks each of which represents only a small portion of the overall behavior, inherently results in clearly structured code. However, inexpedient input models may markedly diminish this effect.

# Chapter 8

# Discussion

In this chapter, we discuss our approach in terms of its characteristics, in particular compared to related work, its validity, and ways of advancing our approach itself as well as the UML specification. Moreover, we outline some basic concepts of promoting our approach towards a workflow execution engine considering two aspects, failure recovery and persistence.

## 8.1 Characteristics of Our Approach

In this thesis, we outline some severe shortcomings of the UML specification. Concerning *Association* and *Property*, these are the flawed assumption of replaceability of both concepts as well as dependencies between concepts concerning *Association* that are actually designed to be orthogonal, e. g. end ownership and navigability. We address such issues and provide solutions by proposing a transformation from models to object-oriented code covering the general case of associations, i. e. we are not limited to binary associations, cover association classes and properly consider end multiplicities, navigability, and visibility.

On the other hand, we also address semantics of behaviors and actions. Instead of detachedly regarding structural and behavioral modeling, we integrate both dimensions of modeling by aligning semantics of structures and behaviors, e. g. by considering multiplicity bounds when executing actions. A focus on these relations between structural and behavioral modeling has not yet been targeted by other approaches with the same emphasis.

Although research results concerned with the above mentioned issues have been published in recent years, these works differ from ours in their extent on the one hand, i. e. in the set of considered details, as well as in their objective target on the other hand.

Concerning classes, we do not in detail focus on generalization relationships, in particular multiple inheritance or the like, and we exclude interfaces and *generics* completely, although both concepts are widely-used. The reason for this is that the kind of relationships that has great impact on behavior is association rather than generalization. Furthermore, interfaces and associations are concepts which are difficult to combine, as we will explain later. And finally, generic classes (also often called template classes) as well as generic methods are a feature which rather make high demands on the compiler than on a code generator, which can map generic parameters of model elements directly to generic parameters in code, if the target language supports generics like e. g. Java and C# do.

Regarding activities, other research work most often is concerned with one of three hot spots:

- mapping UML activities to some other formalisms for specifying workflows, such as e. g. BPMN, or general formalisms like Petri Nets
- executing activities by interpreters
- generating code from activities.

Our estimation is, that the order of the above list represents the amount of research carried out in the three considered fields. Therefore, we are contributing to a less deeply explored issue, which in our opinion is of great relevance for practitioners in software engineering. We assume that the relatively little effort put about code generation from activities results from the fact that in contrast to UML class models, UML activities are absolutely no natural fit to object-oriented languages.

Furthermore, semantics of UML activities has been fundamentally changed with the release of UML 2.0 in 2005. Compared to *State Machine*, *Activity* is a much younger formalisms for which tool support not yet is on a par with tool support for other behavioral formalisms.

To close — or at least, to narrow — the gap between activities and code, the scope of supported UML elements usually is restricted. In order to define a common restriction which assuredly contains all basic UML elements of activities, new specifications such as fUML [72] have been released. Our approach supports activities to a much wider extent.

Before we will provide references to related work, oppose it to our approach, outline commonalities as well as differences, and compare the extent of supported elements, we discuss the reason why we do not consider interfaces in our approach in detail.

### 8.1.1 Association and Interfaces

In object-oriented design, it is desired to keep type declarations and implementations apart from each other. An interface is the declaration of a type, a class is its implementation. If two classes are to be associated, each class should reference a type rather than an implementation. By doing so, types can be re-implemented and multiple implementations for the same type may exist. Which implementation is actually used is not visible to classes referencing a type. In particular, it is possible to exchange implementations without any need of adaptations to the classes referencing the implemented types.

The specification of UML states that *"an association between an interface and any other classifier implies that a conforming association must exist between any implementation of that interface and that other classifier. In particular, an association between interfaces implies that a conforming association must exist between implementations of the interfaces."* [69, §7.3.24]

The claim obviously fails to provide a valid description for the semantics of an association between interfaces such as shown in Fig. 8.1. Figure 8.2(a) shows the implication of the first sentence of the above quotation. The result is that not classes are referenced, but interfaces, i. e. a reference is not committed to a particular implementation, but to a type, complying with good object-oriented design as presented above. However, the semantics of a single association is lost. Since there are three properties implicated, it is not possible to represent the relationship between implementations of interfaces by a binary association, if at least one interface is implemented by more than one class.

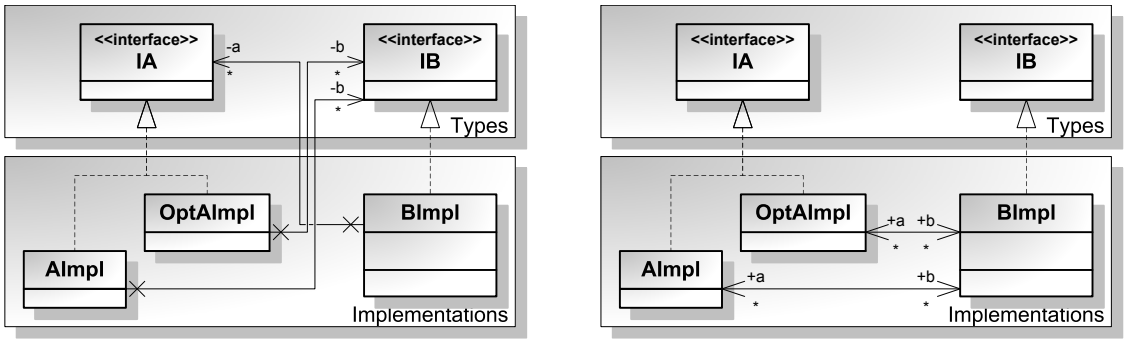
The implications of the second sentence can be seen when looking at Fig. 8.2(b): whereas the association specified on the type level intends to associate an instance implementing interface *IA* with an instance implementing *IB*, the semantics of an association between classes on the level of implementations as demanded by the specification is not the same. First, the role name *a* appears twice in *BImpl* each representing an end of an association to a subtype of *IA*, and second, the multiplicity bounds of the *a* end of the type level association cannot be represented correctly by bounds of the *a* ends on implementation level[1].


Figure 8.1: Association between interfaces.

Our approach is able to generate code preserving the association semantics correctly, as shown in Fig. 8.3. By generating a manager class and using the interface types as values of the link class, links can represent any combination of instances which are subtypes of associated interfaces. However, it is necessary to generate code for referencing the manager class as well as for obtaining a handle, if needed, into each class directly implementing an interface participating in the association. Methods for accessing the association can be inserted into the interfaces and must be implemented in all implementations of a type as well.

In favour of clearly focusing on associations between classes without any ambiguities, we exclude interfaces as specified in the UML from our approach.

[1] This problem does not occurs if the lower bound is zero and the upper bound is unlimited.




(a) Associations between implementations of associations and classifiers.

(b) Association between all implementations of interfaces.

Figure 8.2: Associations between implementations.

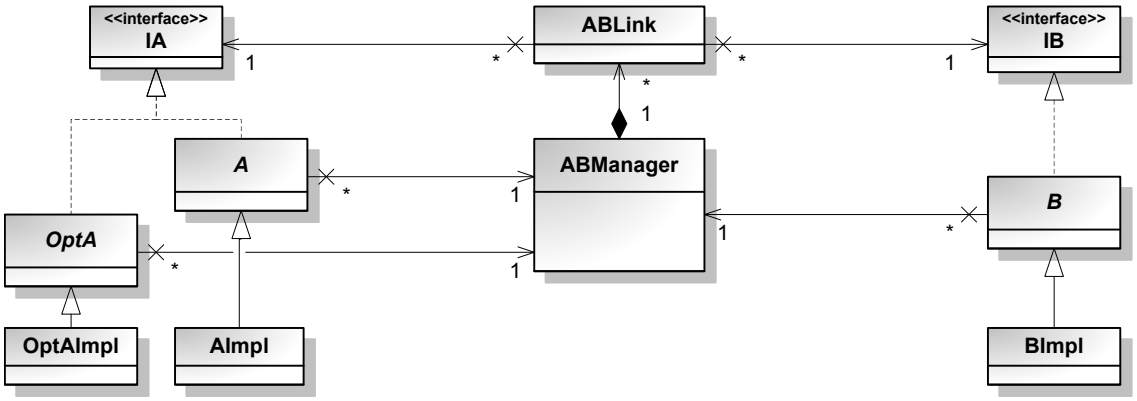



Figure 8.3: Implementation of associations between interfaces in ActiveCharts.

## 8.2 Related Work

We divide this section in two parts. First, we discuss work related to our approach on the basis of targeting the same elements of structural models. Second, we turn towards work that is like our approach concerned with behaviors and actions.

### 8.2.1 Related Research on Structural Modeling

Basic elements of structural modeling are entities and their relationships, as covered by ER diagrams [16]. In UML, entities are represented by *Class*. The predominant kind of relationships between classes are represented by *Generalization* and *Association*.

Translating classes and generalization into code is not difficult at all. Including class attributes is more difficult, in particular if multiplicity bounds are considered. The translation of associations is most difficult, unless accepting some limitations.

A straightforward and often applied strategy of implementing associations is to use references to the opposite class in each class participating the association. Akehurst et al. [1] and Génova et al. [30] discuss this technique of association implementation. Since Akehurst et al. do not consider visibility, the problem of mutually updating private attributes in member end classes is not addressed. Furthermore, it is stated that navigable ends must be owned by the classifier whereas non-navigable ends must be owned by the association. Our work differs from Akehurst et al. by considering visibility and by considering that a navigable association end can be owned either by the association or by a member end class. However, Akehurst et al. discuss other characteristics like e. g. qualified associations which are omitted by us.

Génova et al. [30] consider visibility and state that associations with both ends defined private cannot be managed because synchronizing references is not possible. Using an implementation class for an association is discussed and discarded, because it is assumed that it does not solve the problem either, since it involves auxiliary classes that cannot provide access to the methods for managing the association to some classes excluding all other classes. Unlike Génova et al. , we show how to provide access to links to only some classes. For this purpose, a highly sophisticated code pattern has been elaborated. Besides its ability to facilitate selective access which is restricted to a set of classes, our handle pattern captivates by its brevity, in particular if global handles are applicable which are completely based on static features of concerned classes resulting in the absence of extra CPU time costs.

Some tools, e. g. Fujaba [111] provide access to attributes and even to associations by generating code according to the Fujaba Association Specification [14, 28, 64], which is similar to the approaches of Akehurst et al. [1] or Génova et al. [30]. Although in Fujaba, visibility is included, if both ends of an association are private, generated code cannot be compiled because of an attempt to access private methods of another class. Albeit visibility is correctly taken into account, Fujaba fails to resolve the conflict of absent accessibility induced by limited visibility.

Although providing a more comprehensive formalization of associations, Diskin and Dingel [22] focus on the set theory and mappings to represent associations omitting visibility. Therefore, their work serves as a basis for understanding and implementing associations, but consideration of all features of this modeling concept demands for extensions to their work.

Furthermore, there a numerous attempts to formalize UML, based on different languages or formalisms like *Z* [93, 60] or a *System Model* based on pure mathematics, as applied by the UML Semantics Project [12]. The problem of this kind of research is that although *"the imprecisions and ambiguities of natural language make it difficult to detect and correct subtle errors, incompleteness, and inconsistencies"* [12], a formalization itself certainly will not expose all consistency flaws and subtle errors. In fact, formalizations build a basis on which correctness and consistency could be determined, but this determination is additional research work that must be done.

Another problem is that formalization usually is done for some special purpose. If ownership, navigability, or visibility are not relevant, these concepts consequently are often not covered by formalizations such like [93, 60]. Since we are not aware of a formalization of sufficient maturity covering all the aspects relevant for our work, and which is generally accepted as a correct UML formalization, we directly base on the mostly textually given specification of UML.

### 8.2.2 Related Tools for Structural Modeling

Numerous tools claim to generate code from UML class diagrams, however, documentation about how details of a source model are considered is not provided. An evaluation of freely available — or at least freely evaluable — tools listed in the *UML Vendors Directory* reveals that tools rather provide an infrastructure supporting to implement code generators.

Provided implementations can at best serve as examples because details like multiplicities most often are not at all or only rudimentarily considered. Whether or not such tools can be extended to provide comprehensive code generation primarily depends on

- internal model representation
- code generation frameworks or template mechanisms
- embedding of templates in the code generation process.

If the internal model representation is based on a simplified UML model, the simplifications may lead to less expressive models, e. g. by not supporting dedicated end ownership for associations. For instance, the recent version of Enterprise Architect [110] handles navigability detachedly from end ownership. Consequently, it is possible to define a non-navigable association end owned by an end classifier, albeit an end that is also an owned attribute always represents a navigable association end.

A limitation often seen at template-based approaches is that the content produced by processing a template is contained in a separate source file. Consequently, the policy of how to apply templates determines which model elements can produce output and the output of which elements is isolated in separate files. Suchlike limitations are problematic because they prevent the generation of a pair of classes like proposed in our approach in order to separate generated code from code which is to be completed by the user.

If a template is processed for each instance of a specific UML type, say for each class, code generation for an association is not possible unless the specification of a template and a model element for which to apply it is supported. A model transformation prior to the code generation process may solve this problem, if transformations are supported and the mechanism is powerful enough.

The template mechanism of Enterprise Architect [110] is an example suffering the above described limitations. Classes are processed by templates, but not the entire model is accessible for template designers but only a representation of the currently processed class. Navigating along associations does not give access to the opposite classes but only to some information of the opposite ends such like the name of its type. But accessing the type and its properties is not possible.

### 8.2.3 Alternative Approaches of Implementing Structural Models

Apart from implementing association by applying any kind of association implementation pattern in a traditional object-oriented language, a fundamentally different approach is to make association a member of provided language concepts. The two primary approaches to achieve this are the development of appropriate libraries and language extensions.

#### Libraries

Designing libraries providing association implementations and a convenient interface for accessing this implementations can be seen as a lightweight approach towards making association a language concept as demanded by Rumbaugh [79] already in 1987.

Although this might be a way of introducing relationships to object-oriented languages, the benefit of employing such libraries is possibly small. An implementation of a library mainly hides complex code that is needed for a proper relationship implementation from the user and provides its reuse. When generating code, the complex implementation of relationships can be hidden in superclasses as well. Furthermore, implementations can be tailored to the actual needs in the concrete context of a model [34]. In addition, it is possible to transform models in a way that makes implementation classes for associations explicit. Such a decorated model is a good documentation for the implementation of the original conceptual model.

Instead of generating code for associations, a tool also could use libraries implementing associations to keep the code generation process less complex. Libraries of a kind generally suited for

such an approach have been introduced by Nelson et al. [62] and Østerbye [75]. But since these approaches are not concerned with UML associations but with relations between classes in general, they do neither support ownership nor visibility of association ends.

The aspect-oriented approach of Pearce and Noble [76, 20] achieves a very clear distinction of code needed for the class implementation and the code of association implementation, but again, this approach is not explicitly designed to implement UML associations. Furthermore, benchmark results regarding execution time provided by the authors show that a direct implementation of association may be remarkably faster than the implementation of libraries. We assume that for other libraries, the drawback very likely exists, too.

**Language Extensions**

For language extensions, the same general consideration as for libraries is valid: it moves the point where associations are to be resolved to the language compiler which must resolve additional programming language constructs. Having a look at the whole process of compiling a model to executable code, the programming language is just one level of abstraction that is passed on the way from the high level — which is the model — down to the very low level — which is the machine's code. From a strict MDD's point of view, the level of abstraction of the programming language — and consequently the question whether associations are decomposed when translating a model to code or when compiling the code to a lower level representation — is irrelevant. Although we do not reject the idea of a first class construct for associations in programming languages, we consider it not to be necessary for the implementation of UML models [34].

*RelJ* by Biermann and Wren[9] is a language comprising a subset of Java and relationships as a language feature. According to its grammar, it neither supports higher order associations, nor navigability or visibility of association ends. Furthermore, association classes are limited to associations with attributes, but operations are not supported.

*Rumer* by Balzer [5] is another language supporting binary relationships. Being limited to binary relationships, Rumer cannot implement UML associations in general. Furthermore, navigability is not supported as well as visibility is not.

Nelson et al. [63] present the language *Affinity* with relation as a built in feature. But semantics of relation in *Affinity* differs from the semantics of UML association. A relation is coupled with equality of instances. Therefore, beside equality as is in Java , another kind of equality based on *egal* [4] is introduced. Thus, *Affinity* is equipped with concepts which are not part of the object paradigm. Since we focus to generate code for mainstream object-oriented languages, approaches like *RelJ*, *Rumer*, or *Affinity* disqualify themselves.

Another approach for implementing associations has been based on concern-oriented reuse [6]. The approach is powerful and therefore in principle well suitable for dealing with the problem of the lack of associations as a stand-alone concept in programming languages. However, it is limited to binary associations and the objective is to provide a basis for the implementation of binary associations on a general basis. Aspects such as memory requirements and runtime behavior are taken into account in such a way that the developer can configure the right one from a set of predefined association implementations. The reusability and adaptability of a general association concept is therefore in the foreground.

An attempt to introduce associations as a programming language concept is Umple [95]. It is a set of extensions to object oriented languages that provides a concrete textual syntax to UML abstractions such as association [3]. Badreddin et al. [3] use Umple for enforcing multiplicity constraints and referential integrity. An analysis of association usage in open source projects clearly shows that various combinations of multiplicity bounds occur in real systems and a proper consideration in code is necessary. Umple honors multiplicities by not performing creation or destruction of links that would result in an invalid system state. The actual execution of an operation is indicated by a boolean return value. However, Umple is also limited to binary associations and also assumes that the opposite association ends are always accessible and thus does not allow to freely specify the visibility of association ends.

### 8.2.4 Related Research on Behavioral Modeling

Over the last years, *Activity* has been addressed by a growing number of researches and tools. A survey giving a detailed analysis of UML modeling tools [25] reveals, that support of activities regarding modeling is well in only very few tools, e. g. Papyrus [114]. Further processing of activities, in particular execution by interpreters or code generation is only rudimentarily supported, if at all.

"Activities are often associated with business processes. Our approach cannot fully compete with state-of-the-art workflow execution engines, to which such processes might be deployed, since such engines typically provide additional features, e. g. failure recovery. But activities can also be used to describe short-lived workflows that do not require recovery capabilities with process state preservation etc. E. g. the workflow of purchasing a ticket required to use a public transportation system: the use of a workflow engine for the process of selecting a departure and a destination station, a departure or arrival time and selecting a connection that meets the given constraints is oversized. For such purposes, we consider generating code in line with our approach a good option.

Some advantages of generating code are that we can profit from compiler technique benefits like code optimizations which would have to be re-implemented in an interpreter if runtime characteristics of compiled code are desired. Not building an interpreter but transforming activities to code and thus making the Java virtual machine to the model interpreter is our goal.

Our approach supports implementations for accessing structural features and maps it to corresponding actions, thus achieving the coupling of structures and behaviors. By additionally accurately considering the subtleties of the UML token flow semantics, our approach supports behavioral modeling according to the specification. However, as composing complex flows by combining multiple control nodes between actions makes code generation very difficult – if not impossible – our approach is limited to a single control node between two actions. This limitation may be dropped when implementing control nodes as dedicated classes with methods for token propagation and token consumption.

Although other approaches dealing with code generation for activities exist, these approaches most often focus on some special application in which code generation for activities is only a minor part. UML based Web Engineering (UWE) [48, 49] is concerned with this issue, but even claiming to be based on standards, UWE only uses UML notation, but as can be seen in examples, the semantics of implicit joins is not considered [49, pp. 171,176][56].

Another approach for comprehensive MDD is implemented in UJECTOR. Since the provided examples only consist of simple sequences of actions [96, pp. 30,34], we can not assess the value of code generation for activities of this tool.

Sulistyo and Prinz propose *recursive modeling* to obtain complete code [94], but the introduced code patterns neither support concurrency nor a delay of token flow due to not holding guards.

Bhattacharjee and Shyamasundar present an approach for *validated code generation for activity diagrams* [7]. This approach overcomes limitations of the so far mentioned works by mapping activities to Esterel, a language supporting concurrency. However, even here some problems are not addressed, e. g. the fact that it can not be statically decided which threads will be joined when reaching a join node. A delay of a token flow due to not satisfied guard conditions as well as the token buffering semantics of fork nodes are not considered.

Executable UML [59] gives detailed advice how to build executable UML models, but without addressing details of compiling models to code. BridgePoint[109] is a tool based on executable UML, but modeling UML2 activities — in contrast to state machines — is not well supported. Instead, Object Action Language (OAL) is used, which is less expressive than Alf as concurrency is not supported on the same level. Deferring an activity execution due to non satisfied guards is not possible in OAL, too. Thus, central concepts which we address in this thesis are not applicable there.

Summing up we can find the token flow semantics of activities being not sufficiently implemented in current approaches or tools using more restrictive formalisms such as subsets of the UML specification of activities or action languages. As explained, a decision node cannot be implemented only by using an `if-then-else` statement. If the guard of each outgoing edge is not satisfied, the decision is delayed until one guard holds. Suchlike effects of the token flow semantics are very different to what programmers are used to and possibly therefore often not considered – but dealing with them is the essence of this contribution." [39] [2]

## 8.3 Feasibility and Validity of Our Approach

Models containing behavior and static structure are a necessary and sufficient foundation for generating the runnable code of an application [44]. Feasibility of our attempt to take account for

[2] Reprinted by permission from Springer Nature Customer Service Centre GmbH: [39] ©2011, (doi: 10.1007/978-3-642-21470-7_15.)

both aspects particularly considering constraints imposed to behavior by the systems structure specification, such as multiplicities, is self-evident.

With ActiveCharts, an integration of behavior and static structures has been proposed, which is applied in research, e. g. for modeling and simulation in the field of telecommunication systems[58] as well as in real projects. In 2012, the ActiveChartsIDE has been adapted to the special demands of a German premium automotive manufacturer when specifying the *automatic start/stop*. Automatic start/stop switches off the engine, whenever its power is not needed, neither for propulsion, nor for driving the alternator while the vehicle decelerates to stop, and immediately restarts the engine immediately when power is needed again. Based on the success of this first application of ActiveChartsIDE, further projects followed to model new vehicle features.

Apart from such activities with industrial partners, our approach has been applied in practical courses of our institute at Ulm University in order to estimate the degree of feasibility and gain information about how to improve our work. In particular, we compared the benefit between using our prototypical tools and using other modeling tools and code generators. We provide details about our expertise on designing experiments in order to compare different approaches in our publication [38].

Also, basing on standards is feasible, especially if models and formalisms for their transformations are released by the same body, as it is the case in our approach by transforming UML models using QVT and MOFM2T. Although other formalisms could be employed, as we pointed out, our choice is well motivated.

Preferring a mainstream language in favor of newly developed languages or libraries such as described in the previous section seems beneficial to us since how to extend languages to support relations is still discussed in the community. Furthermore, such languages often are not expressive enough, e. g. if not being multi-threaded, generating code for activities cannot be done as proposed by us.

The separation of reducing semantic complexity in a first step and changing the presentation of a model by transcending to a textual representation in another step is feasible for two reasons:

- it allows for choosing a well suited formalism for each of both steps instead of being committed to one formalism,
- it supports a language independent decomposition of semantically complex modeling concepts to object-oriented concepts.

By this separation of handling semantics in a first step and handling syntax in a second step, we succeed in supporting highly abstract modeling concepts, like composition: at model transformation, associations between manager classes, which represent associations on which a composition has impact, i. e. which are concerned by cascading deletes, can be inserted. At code generation, the collaboration between associated manager classes can be implemented.
With regard to behavior, the separation of sequences of actions which can be implemented in a single thread and complex flows, which may demand for a highly sophisticated coordination between ending threads and threads to be started is a good example for the benefit offered by detaching the semantical resolution from the translation into another representation.

The validity of our approach has not been formally proved in the scope of this thesis. The main reason for this is, that UML in itself is not a formal basis on which a proof could be founded. Rather, UML is a specification primarily formulated in natural language.

A proof of our concept could base on any formalization of UML, but to our best knowledge, such a formalization does not yet exist. A formalization of UML as a part of our approach also has not been aspired. Unfortunately, an attempt of formalizing UML, the UML semantics project [12], has never been completed.

In terms of validity, we nevertheless assume, that our approach withstands due to

- an intensive investigation of the ULM specification,
- a detailed description of the probably intended semantics and
- an attempt to elaborate patterns matching the assumed semantics or a modified semantics wherever modifications seem appropriate.

Regarding static structure, the correctness of our approach can be validated by comparing class diagrams to the generated code. Since classes and properties of UML can directly be mapped to

object-oriented language constructs, the comparison of input models and output code is a sufficient validation method.

The validation of association semantics can also be done through code inspection, because the ability to link instances to each other depends on the existence of appropriate mappings and methods for managing them. The implementation of multiplicity bounds checks is to be realized within these access methods and can also be proved by code inspection.

The semantics of behaviors, however, should be validated by dynamic tests. The implementation pattern for activities as in our approach is designed to closely fit to a modified input model. The modification to the input model is considered to have no impact on execution semantics.

We assure that the execution of generated code complies to the specified original input model by a set of test activities. These activities are processed by the presented model transformation and code generation and the generated code is executed. In the case of several possible, valid execution orders, it must be ensured that any valid order can occur, but no invalid orders occur. An efficient way to check activity executions for compliance with semantics of input models is achieved by specifying the effects of opaque actions in the input model as follows:

- print the action name beginning with an upper case letter when the execution starts
- wait a randomly specified amount of time
- print the action name beginning with a lower case letter when the execution ends
- print and exclamation mark (!) followed by the action name beginning with a lower case letter when an execution is aborted.

Different orders of action executions can be forced by either using guards or by varying the time that action executions consume. In our test activities, guards of flows refer to a property of the context class which is set before the activity execution starts. Additionally to the above mentioned semantics of actions of test activities, the property to which guards refer may be changed by some actions. In this way, not only different action execution orders are forced, but it is also possible to check whether guards are properly considered. An execution path in an activity must continue exactly at the time when a previously unfulfilled condition is satisfied as a result of the change in the context object.

After an activity has been executed multiple times, it can be checked by searching in the printed output, whether all executions are valid. For example, if an interruptible activity region is contained in the activity, only those actions may be aborted which are located inside the region and if a token passed the interrupting edge. The fact that a token traversed an interrupting edge can be seen by the output produced by the execution of the action which is the target of the interrupting edge.

For a large number of activity executions and a wide variety of valid action sequences, a convenient albeit time-consuming method of validation is to check the first output and to remove it (together with all duplicates) from the list of activity execution outputs. This will eventually either find an invalid activity execution output or empty the list.

Another strategy is to search for invalid execution outputs by regular expressions, e. g. if action $A$ is followed by action $B$, the upper case letter B must never appear before the lower case letter a.

It is also possible to analytically determine all possible orders and then force each order by either appropriately defining guards or by appropriately setting the duration of action executions through wait statements. However, this strategy is only applicable if the number of valid executions is manageable.

For a proper validation, the set of test activities must meet the following two coverage criteria:

- all elements supported by our approach are to be included
- for each such element, its semantics must be completely covered.

We claim that our approach can handle guards, all kinds of control nodes, concurrency and interruptible activity regions correctly. Furthermore, our approach can also handle data flows correctly, although this is not yet automated in all details by our model transformation and code generation. Some extensions are needed here. For instance, discarding data objects within interruptible activity regions if the region is left over an interrupting edge, or passing data over control nodes, requires minor adaptations to the generated code.

To meet the first coverage criterion, the test set contains the following elements:

- guarded flows
- flow final node and activity final node
- basic object flows, where output of actions are provided as input for other actions
- decision node with guarded outgoing edges
- fork node with guarded outgoing edges
- merge node
- join node
- interruptible activity region and interrupting edges.

The validation of the complete semantical extent of decision and fork nodes, as required by the second coverage criterion, is assured by:

- no guard of outgoing edges holds
- exactly one guard of outgoing edges holds
- more than one guard of outgoing edges hold
- all guards of outgoing edges hold.

For interruptible activity regions, we consider:

- leaving the region by a non-interrupting edge
- aborting the region due to a token traversing an interrupting edge
- having several, nested interruptible activity regions
- having several, overlapping interruptible activity regions.

The test activities and the produced output is contained in Appendix D.

Apart from the test described above, the applicability of our approach has been tested in real projects, embedded in practical courses of the university. Besides many small test projects with no real system in mind, TravelPlaner — to which we refer in a technical report [37] — is a project designed to estimate the impact of our approach. It comprises a detailed and complex static structure employing composition, higher oder association, generalization applied to association classes as well as to *normal* classes and a behavior containing concurrent execution paths, control and object flows as well as control nodes. The aim of TravelPlaner was to get an impression of the share of the generated code in the whole system and to show that the generated code and user code are easy to integrate and that the generated code actually supports the implementation of a system.

## 8.4 Advancing Towards Workflow Execution

Due to the fact that UML activities are often used to model workflows — even though activities are not really designed for this purpose — adaptations of activities and of our approach in this direction certainly are valuable.

A workflow modeled in UML typically consists of multiple actions which are executed according to the token flow semantics defined by UML, but actions do not necessarily start as soon as all required tokens are available. Workflows are often suspended for some time. When which workflow is continued may depend on various conditions, e. g. the progress of other workflows or the arbitrary choice of the person responsible for the next step. Consequently, running workflow instances may remain in a system for quite a long time. By that, workflow instances are prone to be terminated during execution due to a system failure. In order to avoid complete re-execution, recovery at the point of termination or somewhere — typically very short — before this point is desirable.

Up to now, we considered UML activities to be used for short running workflows, like the example given in our TravelPlaner project: purchasing a ticket for public transportation. In the following, we give a short use case:

**TravelPlaner Use Case:**
*A user provides input data, which are the departure and destination stop and departure and arrival time. Based on these pieces of information, the system searches connections between both stops departing and arriving within the given time constraint. The user chooses one of the offered connections which is passed back to the system for evaluating the possibility of seat reservations. Only if seats are available for reservation, input is requested from the user. The number of requested seats, or 0 if no seats are available, is passed to the system and reservations are created. The connection, reservations, and the calculated ticket price are displayed. If confirmed and payed according to the selected method, a ticket is printed, if refused, no ticket is printed and seat reservations are destroyed, if any.*

We consider a workflow as the presented use case to be *short running*, since its execution typically will successfully terminate within few minutes. Furthermore, an activity implementing the given use case could simply be restarted, if its execution is aborted in any abnormal way. Such workflows are not comparable to long running workflows which, if aborted cannot be started from the very beginning because of dependencies to other systems or the real world, e. g. the process of building a car is to be continued if a workflow specifying automated production steps has been aborted; otherwise, the partially built car could not be completed.

At a first glance, activities in our sample projects seem to be related to *micro processes* as defined by Künzle and Reichert [53] describing the internal behavior of an object instance. But UML actions not only cover the micro level, e. g. the access to attributes, but also the macro level which is the interaction of objects, achieved by *CallbehaviorAction* and *CallOperationAction* as well as communication by means of sending and accepting signals. By that, modeling processes on the *micro* and *macro* level is possible, however, UML is lacking a clear distinction of both aspects which is aspired by PHILharmonicFlows [17].

Künzle identifies the missing separation of different aspects of workflow modeling and execution as a crucial issue of current approaches [52]. PHILharmonicFlows is designed to explicitly offer a data centric as well as a process centric view, user integration as well as a clear separation of internal object behavior on the one hand and object interaction on the other hand. Although having a similar execution semantics which possibly might be mapped to the UML token flow semantics for activity execution, UML Activity is a more general and more abstract concept, not primarily designed for workflow specification. Activity is intended to also fit to the specification of embedded systems without user interaction, thus being an instance of a very different, more general domain than dedicated workflow modelling approaches.

The strength of UML activities to be applicable to many domains of behavioral modeling — since not being designed to fit primarily to workflow modeling — entails that workflow modeling is possible on a sketchy level only. Whether or not it is possible to implement the core issues of PHILharmonicFlows in UML cannot be answered in the scope of this thesis. To a certain degree, a mapping to UML by means of profiles and stereotypes very likely is achievable. But the question whether both approaches are equivalent is not constructive: a very specialized language such as PHILharmonicFlows definitely will outrun a general purpose language such as UML when both approaches are applied in the domain for which the more special one has been designed for.

However, activities in UML may be used for workflow modeling on both levels, internal object behavior as well as interactions between objects and activity executions. Although not providing a real data centric view, data flow supports the specification of data related issues in activities as well. As a result of possibly complex specification of inter-object communication by means of *CallBehaviorAction*, *CallOperationAction* as well as the application of signal and time events as triggers within activities, activity executions may turn into long running workflows. Therefore, we shortly outline two aspects for advancing the applicability of UML Activity in the workflow domain: recovery and persistency.

### 8.4.1 Implementing Recovery in ActiveCharts

In our approach, concurrency of parallel branches in an activity is considered by starting a dedicated thread for each branch. Thus, an activity execution may consist of a single thread or comprise several threads which are concurrently running. Therefore, if we accept that

- object flows are explicitly modeled, i. e. no data are transmitted by writing values into variables and reading from them in downstream actions

- no side effects like any change to the underlying static structure occurs, i. e. no values are added or removed to attributes or association ends

then activities as implemented in our approach can be recovered by

- creating the threads which were active when the activity execution failed
- starting the execution of each thread at the correct position.

The ability to recover can be implemented by writing a logfile. For each thread that is started, an entry is appended including the thread's id and the consumed input data, if appropriate. For each thread that ends, an entry is appended including the thread's id and provided output data, if any. If a thread is reused by changing its id, an entry for the termination of the thread with the old id and an entry for the starting of a new thread with the current id is appended, including input and output data as appropriate.

If somewhere in a sequence of actions, a point where to recover is demanded, this sequence either must be split into two sequences each mapped to its own thread (e. g. by including a merge node with a single incoming edge), or a second id reporting the progress of execution within a sequence of actions must be introduced. With such a second id, it is possible to skip statements implementing actions which, in terms of the token flow, are located upstream to the position at which to recover.

In Fig. 8.4, we give an example: it shows an activity for some kind of online ordering. The action *login* provides user account data, the action *order* provides the items and quantities the customer wants to order. With these data, an invoice can be created which the user must confirm. If he does so, the order is saved, otherwise, the order is skipped, i. e. the activity ends without saving the order. In case of an unusual termination of the activity, it should be recovered without asking the user for input he provided before termination.

The gray boxes group those actions which are executed in the same thread. The id for such threads is given in the lower right corner. According to the idea to start threads for each sequence (or to reuse a thread by changing its id), an execution can be recovered at the end of a thread or the start of the next thread respectively.

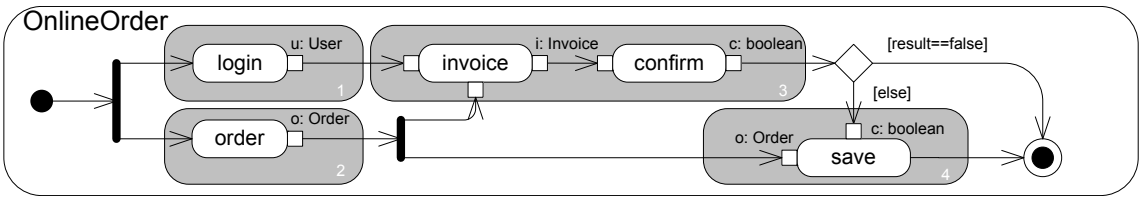



Figure 8.4: Activity modeling an online order workflow.

A logfile which is the result of a possible execution is given in Fig. 8.5. In green color, entries reporting about the starting of a thread are listed, entries reporting the termination of threads are printed in red color. In blue color, entries are listed which report about provided or consumed data.

Recovery is possible at any point $t_i$ in the logfile. To recover at $t_0$ or $t_1$, the activity must be started normally. Recovering at $t_2$ requires to start thread 2, but not thread 1 since it has been ended. Instead, its output must be restored and thread 3 must be started. For recovering at $t_3$, the outputs of thread 1 and thread 2 must be restored and thread 3 and thread 4 must be started. If recovering at $t_4$, the outputs of thread 1 and thread 2 must be restored but the objects consumed by thread 3 must be removed since thread 3 will not be started again. Instead, its output is restored before thread 4 is started.

Each point in time $t_i$ represents the transition from one gray box of Fig. 8.4 to another. If recovering between the actions of box 3 is demanded, the activity implementation must be adapted to contain a progress id as in Listing 8.1. In line 16, an entry is added to the logfile reporting that an object token traversed the edge between *invoice* and *confirm*. The logfile also must include the object token $i$ itself. Then, it is possible to recover after action *invoice* by providing the object token $i$ and starting a thread with id 32.

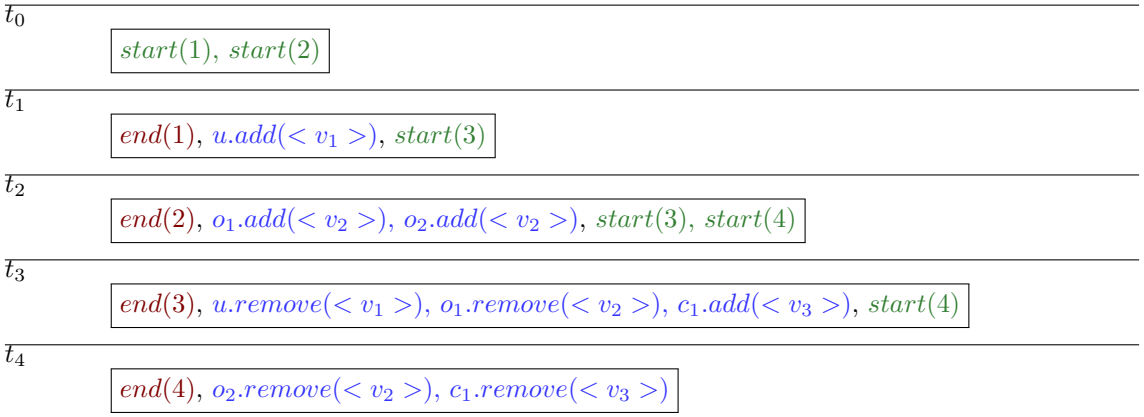



Figure 8.5: Support for activity recovery.

```
User user;
Order order;
Invoice i;
(...)
switch (id){
  case  0: (...)
           break;
  case  1: (...)
           break;
  case  2: (...)
           break;
  // begin sequence 3
  case  3: user = u.remove(0);
           order = o1.remove(0);
           i = invoice(user, oder);
           log(id(32));
  case 32: boolean accept = confirm(i);
           c.add(accept);
           if (accept){
            log(t3);
            id = 4;
           } else {
            log(...);
            id = -1;
           }
           break;
  // end sequence 3
  case  4: (...)
           break;
  }
}
```

Listing 8.1: Implementation of a recovery point between actions.

This example suggests how to include ability for recovery in our concept of activity execution by threads. We provide no further detail since we just wanted to figure out a general idea of advancing towards failure recovery.

### 8.4.2 Implementing Persistence in ActiveCharts

Persistence is more difficult to automate, since it must be modeled which values to store and which technology to use. Van Tricht [97] developed a framework for ActiveCharts which supports persisting instances of classes modeled in class diagrams as well as activities. The data are stored in XML representations.

If a large number of objects — or data not organized in objects — is to be stored, XML files are no suitable technology. Instead, powerful database systems might be applied.

In the current state of ActiveCharts, accessing a database for retrieving or storing values is in the responsibility of the user and must be included in opaque actions or behaviors. However, a more convenient mechanism is desirable and achievable. But enhancements for storing objects into databases or creating objects from database query results requires some kind of configuration or additional information tagged to model elements, e. g. by introducing stereotypes. By that, information about tables and columns to which attribute values refer to can be specified.

A very simple approach is to create a database with a table for each class. Each attribute of a class refers to a column in the according table. It is state of the art to create suchlike mappings automatically, however, when to create objects and when to persist them must be annotated in the model. Assuming that a mapping between objects and tables exists, actions for accessing the database could be specified. For writing into the database, instances placed on an input pin of an action could be mapped and inserted into the according tables. Referential integrity could be considered by navigating across associations, provided that the static structure and the database structure are compatible in this respect. For creating objects, some key values could be provided as an input for an action that returns instances matching the given keys.

For supporting all – or at least the most common — features of today's database systems to the full extent, either extensions to the UML are required or a properly designed profile must be developed. Other approaches probably are more a workaround than a reasonable and reliable lightweight approach. In the scope of our work, we did not make a point of elaborating a mechanism for persistence. However, as stated above, there are frameworks that could be applied and such an application should be backed up by a UML profile or UML modifications in order to provide a clear semantics, syntax, and notation.

## 8.5 UML Simplified

This thesis refers to Unified Modeling Language Super Structure Specification 2.4.1, which has been the current UML version at the time our latest contributions have been published. Meanwhile, UML Simplified has been introduced, which has been released as UML 2.5 Beta 2 by OMG. Version 2.5 is formally a minor revision to the UML 2.4.1 specification, having been substantially re-written as solicited by the UML Specification Simplification RFP[3]. The main difference to the version before are editorial issues: whereas formerly, modeling elements where described one after the other, each with an own section covering abstract syntax, semantics and notation, currently elements are grouped and share common sections including abstract syntax, semantics and notation of each member of such a group.

UML 2.4.1 is organized in chapters dealing with different views on a system, such as static structure, collaboration, deployment, or behavior. Each element of the meta model is introduced in the chapter covering the concerning modeling purpose. There, all characteristics of each modeling element are specified: its purpose, properties and relations to other modeling elements, its semantics, rational, and notation, if applicable.

UML 2.5 Beta 2 is organized differently by focusing on different aspects of the specified elements. It therefore introduces the syntax as well as properties of and relationships between modeling elements in one part of the document whereas semantics is covered by another part. Whereas in UML 2.4.1, semantics is scattered all over the specification, in UML 2.5, specification of each element is split into parts located at different positions in the document.

Changes to the contents of the language, such as its extent, syntax, or semantics are little. Uml 2.5 was declared as a Beta 2 version until December 2017, before it was finally released as UML 2.5.1 [73]. Since to this time, our research work as well as the preparation of this thesis was almost completed, we refer to the previous UML version, to which also the publications related to this thesis refer to.

[3] request for proposals

## 8.6 Current Trends in Model-Driven Sofware Engineering

Below is a brief overview of how research and tool support in model-driven engineering has evolved over the last few years, outlining the main topics currently being discussed in the community

The two most relevant conferences focusing on Model-Driven Engineering are *International Conference on Model-Driven Engineering Languages and Systems (MODELS)* and *European Conference on Modelling Foundations and Applications (ECMFA)*. The first seven editions of MODELS were dedicated to UML, the conference itself being entitled *Intl. Conf. on the Unified Modeling Language, Modeling Languages and Applications*. From 2005 on, it is no longer directly associated with UML, although still intensively concerned with it, and is constantly emerging to a more language independent format. Accordingly, it increasingly covers more language independent topics such like modelling in general, metamodeling, or development and specification of modeling languages whereas questions such as semantics of a concrete language or even only subsets of a language, for instance UML behaviors, are less often addressed.

A similar trend can be seen in MODELS European counterpart ECMFA. The focus generally is more on the evolution of languages including domain-specific and aspect-oriented modelling on the one hand and the process of Model-Driven Engineering in itself, as for instance model management, on the other hand.

However, even UML Activities, particularly their semantics and code generation from them, has not been addressed by the community of the above mentioned conferences in recent years, several publications prove that both topics still are highly relevant. An empirical evaluation of current UML modeling tools [81] of 2015 rates the capabilities of current tools, but the evaluation only surveys class diagrams, state machines and sequence diagrams but not activities for modeling behavior. Another survey [55] of 2014 states that practitioners of the embedded systems domain do not consider code generation issues to be a major problem of MDD approaches. But again, UML activities are not primarily considered since this formalism plays a minor role in this domain. These two contributions show that still today, the option to use other formalisms than UML Activities for behavioral modeling, is very popular, at least in some domains. Consequently, often other formalisms than UML2 Activities are addressed by practitioners as well as researchers, when it is aimed to generate code from behavioral models. In this regard, our work still is unique with regard to the examined formalism and its accuracy.

Further, the importance of our work is confirmed by Mussbacher et al. trying to assess the relevance of Model-Driven Engineering thirty years from now on [61]. Lacking tool support for keeping models of different levels of abstraction in sync is identified as a current problem. The main cause why models of different levels of abstraction diverge is that often specification is done on a high level of abstraction whereas corrections of design faults or bug fixing is done on lower levels of abstraction, often directly in generated code. Changes of generated code are not transfered back into models. This problem can either be faced by updating models according to changes in code or by providing a process following the idea to exclusively work on the higher level of abstraction, i. e. the models, and generate the artifacts of the lower levels of abstraction [40]. Alternatively, the redundancy which is induced by having artifacts of at least two different levels of abstractions describing the same system, i. e. the models and the generated code, can be avoided by directly executing the models in order to make generated code obsolete [21]. With ActiveCharts, we follow the idea of exclusively working on the modeling level for specifying structure and control or data flows of an application. The problem of time consuming model transformation and code generation, particularly in case of large models, is a drawback of model centric approaches, which might be adequately faced by incremental approaches [74].

The choice to support UML Activities for modeling systems in general is a good one, since currently, Activity diagrams are well known and often applied [78]. The fact that activity diagrams are considered to be widely used especially for software process specification in the way activities are seen by us, too, motivated an approach for software process verification [54], which would also fit fine into ActiveCharts. Another approach to testing Java implementations against its UML class model [15] possibly could be used to verify our code generation results for static structure.

As this brief literature review shows, code generation from class models as well as behaviors has rarely been addressed in the recent past. However, numerous contributions are somehow related to code generation issues, which suggests that code generation is still a key topic in model-driven engineering domain.

# Chapter 9

# Conclusion

UML notation is a de-facto standard which is well known and widely used. But generally speaking, its semantics is much less well understood. Many developers who are using UML for documentation purposes or in model based development processes focus on the notation while semantics is not properly considered.

According to OMG's MDA, we are interested in the question to what extent UML can be beneficially used in model-driven development. Many tool vendors claim full UML compliance of their tools, but realistically judged, capabilities of most tools are not more than the creation of class stubs from UML class diagrams or generation of code from very simple activities. The value of such kind of tool support is little, because it covers only that part of implementing models that could easily be done manually. Such a level of tool support may result in some time saving, but the same effect is achievable if using IDEs with good code completion and generation of boilerplate code. Where tool support is really needed, it is absent leading to a situation in which application of modeling concepts does not depend on the problem to solve but on their straightforwardness to be implemented: *"Higher-order (e.g., ternary) associations are difficult to implement and are generally avoided"* [93].

The aim of model driven development is, that concepts are applied which are closer to the problem domain than to the target domain which is the actual implementation. Whether or not higher order associations are used should depend on the problem to be solved: if it can be described well with the help of higher order associations, these should be applied and their implementation should be managed by tools. This claim not only counts for associations but for all modeling concepts offering a higher level of abstraction. On the way to provide tools which comply to this claim, still a lot of work is to be done. This thesis contributes some basic foundations to a better tool support of model-driven development using UML.

## 9.1 Contributions

Our contributions apply to different aspects of evolving a Model-Driven Development approach. Coping with Model-Driven Development in terms of elaborating a development approach or tool support requires

- to understand the semantics of the used modelling language,
- to map this semantics to constructs offered by the target language which typically are on a lower level of abstraction, and
- to somehow define or implement these mappings.

All the knowledge gathered in the process of understanding the source and target formalisms and implementing a mapping in between may effectuate ideas for evolving applied formalisms in order to simplify mappings. The evolution of applied formalisms thus is a fourth issue to be added to the three issues stated above.

Modifications to applied formalisms can be proposed, but often their implementation is not in the responsibility of the proposer. Hence, findings concerning applied formalisms not necessarily trigger an improvement of the initial situation.

This thesis contributes to all four topics outlined above by its

- detailed examination of the semantics of UML
- resolution of modeling concepts of a high level of abstraction to concepts of a lower level of abstraction
- formalization of such resolutions and a prototypical implementation for validating the approach
- proposals of enhancements to the UML semantics.

In the following, we shortly survey our contributions.

**Semantical Resolution of Static Structures**

With *Association*, UML introduces a complex modeling concept that cannot be directly mapped to underlying target language constructs. Associations are difficult to implement in particular if

- they have more than two ends
- composition semantics is applied
- associations are related by generalization.

We present an implementation pattern which considers ownership, navigability, and visibility of association ends. It is straightforward with regard to the implementation of higher order associations and *AssociationClass*. In addition, it covers generalization between associations as well as between association classes.

**Semantical Resolution of Behaviors**

We design a basic pattern for activity implementation. This patten maps the token flow semantics of activities to threads of the Java programming language. Into this pattern, implementations of actions can be embedded.

The effects of control nodes and guards are considered by starting and pausing threads implementing guarded flows. But the pattern is only applicable to activities if flows contain at most one control node.

By the definition of guard propagation rules, control and object flows which pass multiple control nodes possibly can be transformed to flows which pass only one single control node on their way from a source to the target. By such a transformation, activities with flows containing more than one control node become applicable to our implementation pattern.

For some kinds of more complex flows, to which guard propagation rules are not applicable, we introduce reordering of nodes in order to obtain patterns which are translatable, too. For flows which are not covered by the proposed reordering patterns, we outlined the specification of interfaces facilitating the inclusion of manually implemented complex flow algorithms.

Concurrency and signal processing are also included in our approach. *InterruptibleActivityRegion*, which is rarely supported by currently available tools, is implemented by us, too.

**Formalization of Semantical Resolutions**

With a set of QVTR relations, we do not only formalize the transformation from complex modeling concepts to model patterns which are more easily translatable to code. Since the transformation is executable in *medini QVT*, it also serves as a referential implementation of our prototype.

Beyond that, the declarative description of a semantical resolution turned out to be much more convenient to develop, to extend, and to maintain thus being a promising technique on which to base tool support.

**Formalization of Code Generation**

The gap from the transformed model to the textual representation in the target language, which in our approach is Java, is bridged by another standardized formalism: MOFM2T. Like QVTR, MOFM2T is executable by tools and can be used for real tool implementation.

### Tool Support

Formalizations — and by using executable formalisms also implementations — of model transformations and code generation as proposed in our approach are completely based on OMG standards for which eclipse[1] based tools are available.

Models are represented using the EMF-based implementation of the UML2 metamodel of the eclipse project *UML2*[2].

Apart from the implementation of model transformations and code generation, we consider to cover the elimination of UML deficiencies by intelligent tooling, e. g. by generating parts of models in order to replace actions with poorly specified semantics. In the long run, revisions of UML should clear out these deficiencies.

In the context of our tooling, we better align semantics of static structure and dynamic behavior. Basic consideration to inheritance applied to activities are presented in order to make tool support both powerful and convenient.

### Discussion of UML Semantics and Proposals for Enhancements

To all parts of our work, a previously carried out detailed examination of the UML semantics is of vital importance. As a result of this examination, we discuss some consolidated findings:

**Misalignment Concerning the Semantics of Structures and Behaviors.**
On the one hand, multiplicities are an integral part of associations, since association ends cannot be defined without multiplicity bounds. On the other hand, violations of multiplicity bounds do not affect the execution of actions and behaviors. Whereas specification of multiplicity bounds is a necessity for structural modeling, consideration is not even optionally possible but must be completely modeled. This is not practical because it requires much effort, results in complex activities and is error prone.

**Integrating Exceptions to Actions Semantics.**
Exceptions in UML are raised by *RaiseExceptionAction*. An exception cannot occur as a result of the execution of another action. Consequently, the semantics of actions cannot imply an exceptional termination of an action, but only of an activity (or a structured node). We propose to change this and to make exceptions a mechanism that might be used to indicate a clash of semantics, e. g. structural specifications like multiplicity bounds and behavioral specifications like adding values to a structural feature causing the violation of an upper bound.

**Specification of Stereotype *«Create»*.**
Lower bounds are problematic with respect to instance creation and initialization. The stereotype *«Create»* is defined by UML, but its specification is not broad enough to effectively face this problem.

**Specifications of Transactions.**
Transactions as known from databases are a solution which is applicable to UML as well. This concept could be used to define actions which may cause constraints specified upon the static structure to be temporarily not satisfied. By that, the modeler could clearly define when structural integrity may be discarded for a moment and when it is properly considered again.

**Recovery of UML Activities.**
Since the workflow like notation and semantics of activities comforts the specification of workflows, activities should provide means for including recovery aspects. We outlined the implementation of recovery for activities, but it is questionable whether it is feasible to log activity execution on a level which facilitates recovery at any point.

[1] eclipse
[2] http://www.eclipse.org/modeling/mdt/?project=uml2

## 9.2 Outlook

Model-Driven Development is a promising approach, although tool support is still rudimentary. In this work, we introduce the entire tool development process as part of our ActiveCharts project. We start with a discussion of semantics. Understanding how complex modeling concepts can be implemented is supported by the presentation of implementation patterns for structural and behavioral concepts such as association and activity. Based on this, we formalize the mapping of abstract modeling concepts into concrete constructs of the target language via an executable formalism in order to directly obtain prototypical tool support. On this basis, tools could provide high-level model transformations and code generators.

Model-Driven Development is more than just applying models as a documentation or blueprint. In terms of UML, it is also a step towards a higher level of abstraction, as the modeling language provides abstractions over concepts of the programming language. To establish this level of abstraction in the long term as the level at which problems are addressed and solutions developed, tomorrow's developers must be instructed to think and work at a higher level of abstraction than today's programming languages.

In addition, modeling is also more than visual programming, and without general agreement to this claim, it will not be a resounding success. If one considers a model merely as an abstraction of the code to be developed, i. e. if having the implementation already in mind while modeling, modeling concepts that are difficult to implement continue not to be used in models. However, if the level of abstraction the developer is actually working on is to be increased, the mapping of abstract modeling concepts to concrete programming language elements must be covered by tools just as today's development environments allocate programming language constructs to the code instructions executed by the real hardware or virtual machines.

Demands for higher abstractions in programming languages, especially relations, have often been formulated, sometimes even up to a few decades ago [79, 66, 103]. Although Model-Driven Development as an idea to address such claims is not entirely new, it still can not provide satisfactory technology to cope with the complexity of today's system and software engineering. Making the MDD tools of tomorrow as advanced as today's programming IDEs are, requires a lot of work. Our approach can give orientation to the direction in which modeling languages such as the current main stream language UML can develop, it can provide impetus for further tool support and possibly inspire future work.

# Zusammenfassung

Modellierung ist ein Konzept, das in vielen Bereichen des Ingenieurwesens Anwendung findet. Dementsprechend gibt es verschiedene Arten von Modellen, die sich aber immer auf ein und denselben Grundgedanken beziehen: Ein Modell ist ein Stellvertreter für ein tatsächlich existierendes oder gedachtes Gebilde. Modelle unterscheiden sich im Grad der Abstraktion sowie in ihrem inneren Aufbau und ihrem Zweck.

Beispielsweise ist ein Bauplan zum einen eine grafische Spezifikation der Form und Abmessungen von Bauteilen, zum anderen eine Darstellung, wie mehrere Einzelteile zusammen Baugruppen bilden, die schließlich gemeinsam das Gesamtgebilde darstellen. Sie stellen eine zweidimensionale Abstraktion eines dreidimensionalen Objektes dar, wie z. B. die in Abbildung 10.1 gezeigte Konstruktionszeichnung des Wright Flyers von 1903.

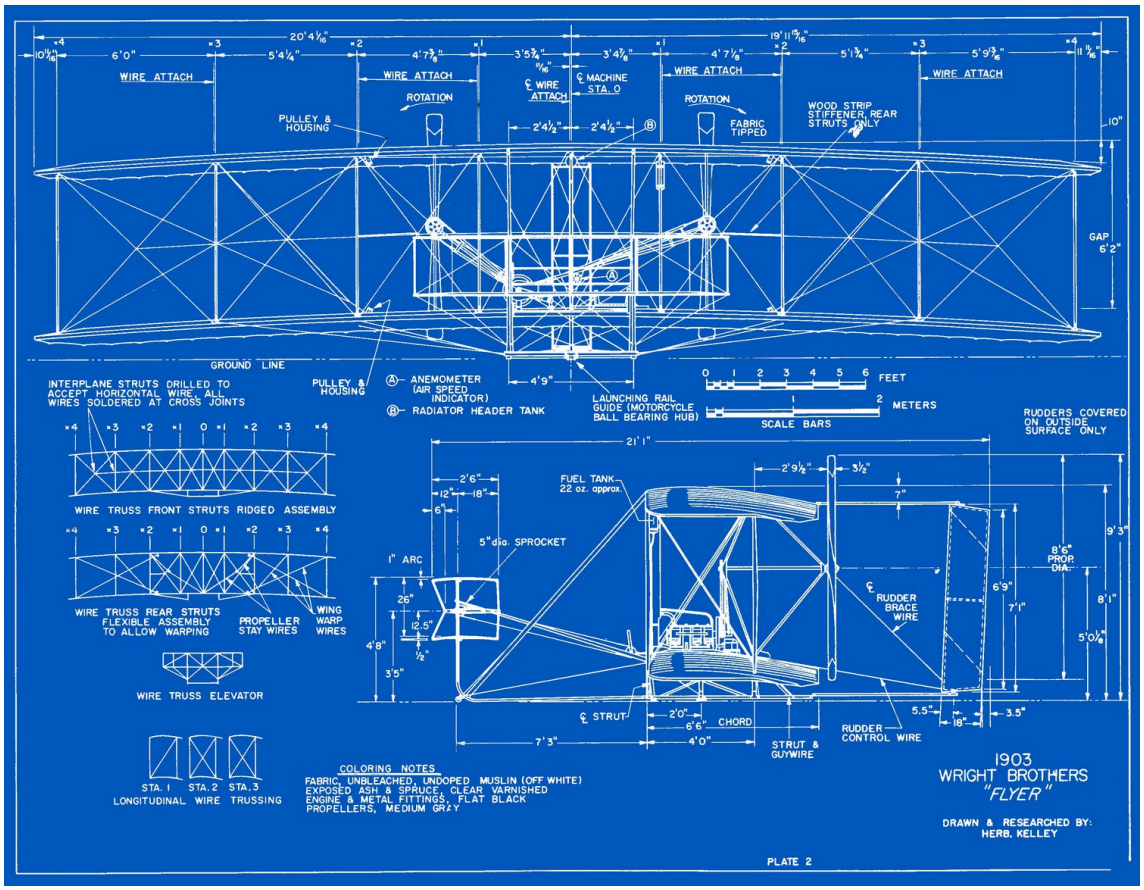



Abbildung 10.1: Blaupause des Wright Flyer von 1903[1].

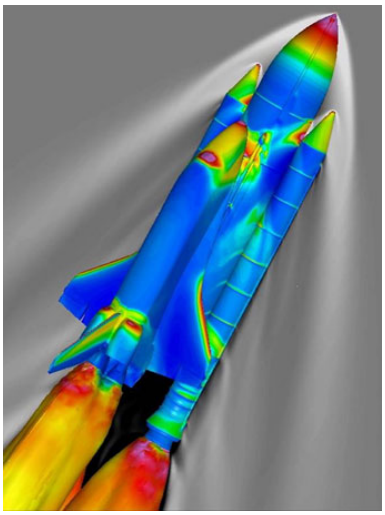
Im Hinblick auf die Form eines Körpers ist ein dreidimensionales Modell als maßstabgetreuer Nachbau weniger abstrakt als ein Bauplan, da es den modellierten Gegenstand nicht auf eine zweidimensionale Abbildung reduziert. Solch ein Modell kann das reale Objekt repräsentieren, bevor dieses existiert. So können die aerodynamischen Eigenschaften eines Autos oder eines Flugzeugs

[1]Basierend auf den Plänen von Christman, http://wright-brothers.org [104].

anhand maßstabsgetreuer Modelle im Windkanal überprüft werden. Abbildung 10.2(a) zeigt das Windkanalmodell des Panavia PA200 Tornado[2]. Der Vorteil solcher Modelle ist, dass Konstruktionsfehler — hier im Bereich der Aerodynamik — erkannt werden können, bevor kostenintensive Prototypen gebaut werden und die Risiken von Fahr- oder Flugtests reduziert werden.

Mit zunehmender Rechenleistung verlagert sich die Modellierung kontinuierlich von physischen Modellen weg hin zu Computer-Simulationen. Der Vorteil von Computermodellen besteht darin, dass derartige Modelle in der Lage sind, Entwurfsfehler aufzuzeigen, die in Umgebungen auftreten, die experimentell nicht nachgestellt werden können, wie z. B. Schwerelosigkeit oder Vakuum, beides Bedingungen unter denen Raumfahrzeuge und Satelliten betrieben werden. Abbildung 10.2(b)[3] zeigt die Druck- und Strömungsverhältnisse um das Space Shuttle mit Feststoffboostern und Tank im Startaufstieg bei Mach 2,46 in einer Höhe von ca. 20.000 Metern [101].

(a) Windkanalmodell des Panavia PA200[2].



(b) Aerodynamisches Modell des Space Shuttle mit externem Tank und Feststoffraketen[3].

Abbildung 10.2: Reales und virtuelles aerodynamisches Modell.

In der Softwareentwicklung können Modelle in der gleichen Weise gewinnbringend eingesetzt werden: ein Modell kann ein komplexes Softwaresystem vor dessen Implementierung hinsichtlich vieler Eigenschaften vertreten. So können beispielsweise einzelne Komponenten eines eingebetteten Systems implementiert, getestet und integriert werden bevor andere Komponenten verfügbar sind, sofern die fehlenden Bestandteile durch Prototypen oder ausführbare Modelle simulierbar sind.

Dennoch unterscheiden sich Softwaremodelle von den übrigen oben erwähnten Modellen: falls ein Modell in maschinenlesbarer Form vorliegt, dann stammt das Modell als auch der modellierte Gegenstand, nämlich die Software, aus der gleichen Umgebung, d. h. beides sind Gebilde die von der gleichen Art von Maschinen verarbeitet werden können. Daher ist es prinzipiell möglich, ein Entwurfsmodell in eine Implementierung zu übersetzen[4]. Modellgetriebene Softwareentwicklung ist ein Ansatz, der sich genau darauf konzentriert.

Der Eigentliche Vorteil wird durch den höheren Grad der Abstraktion erreicht, denn Modellierungssprachen sind für gewöhnlich besser auf den Fachbereich des Problems zugeschnitten. Die Übersetzung in lauffähigen Maschinencode kann durch entsprechende Werkzeuge geschehen. Ein bekannter Ansatz dazu ist MDA[5], ein offener Standard, der einen Satz von Transformationen definiert, die plattformunabhängige Modelle, also Modelle die vollkommen frei von Einschränkungen der auf dem Zielsystem verwendeten Systemsoftware sind, in konkrete Implementierungen,

[2]Eigenes Werk des Autors, aufgenommen im Deutschen Museum, mit freundlicher Genehmigung des Deutschen Museums, München.

[3]Diese Abbildung wurde ursprünglich von der NASA veröffentlicht und ist gemeinfrei [101].

[4]Das Herstellen eines Objektes nach dem Vorbild eines Modells ist nicht auf Software begrenzt, wie beispielsweise die Herstellung von Werkstücken mittels CNC-Fräsen zeigt. Jedoch werden dazu schon sehr aufwendige Maschinen benötigt, die bedient und gewartet werden müssen und die vollständige Herstellung komplexerer Gebilde wie das eines Fahr- oder Flugzeuges sind voll automatisiert derzeit nicht denkbar.

[5]Model-Driven Architecture

die von der Technologie der Zielplattform abhängig sein können und typischerweise auch sind, überführt.

Diese Arbeit zielt auf die Übersetzung einer sehr abstrakten Modellierungssprache in eine allgemeine objektorientierte Programmiersprache ab. Besonderes Augenmerk liegt dabei auf zwei wesentlichen Eigenschaften eines Systems:

- der Systemarchitektur, d. h. der statischen Struktur
- der Funktionalität, d. h. dem dynamischen Verhalten.

Die Modellierungssprache, in welcher die Modelle spezifiziert werden, ist die Unified Modeling Language (UML), ein Standard der von der Object Management Group (OMG) herausgegeben wurde. Die Zielsprache in der die Systeme implementiert werden ist Java. Diese Wahl ist willkürlich und wir hätten sowohl die Eingabemodelle wie für die daraus generierten Implementierungen auch andere Sprachen auswählen können. Jedoch sind beide Sprachen in der industriellen wie auch der akademischen Welt sehr populär und ihre Anwendung ist bei Forschern wie auch bei Anwendern gleichermaßen weit verbreitet. Insbesondere stellt die UML heute einen de-facto Standard in der Modellierung von Softwaresystemen dar.

Eine Eigenheit der UML besteht darin, dass die Semantik nicht formal beschrieben ist. Sie ist mehrdeutig und folglich existieren verschiedene Interpretationen. Darüber hinaus widerspricht sich die Spezifikation hinsichtlich der Modellierung von statischen Strukturen auf der einen und der von dynamischen Verhalten auf der anderen Seite teilweise. Ansätze, die sich nur auf einen der beiden Aspekte entweder auf die statische Struktur oder auf das dynamische Verhalten konzentrieren, können die Widersprüche zwischen den beiden Bereichen der Modellierung nicht bewältigen. Im Gegensatz dazu zielt diese Arbeit darauf ab, die Schwächen und Fehler der UML im Zusammenspiel zwischen Struktur- und Verhaltensmodell zu identifizieren:

- In einem ersten Schritt wird die Semantik der Modellierung statischer Strukturen im Detail untersucht, mit dem Hauptaugenmerk auf Klassen mit ihren Attributen sowie Beziehungen zwischen Klassen untereinander. Ein zentraler Punkt dabei ist die Frage, wie die Semantik von Assoziationen, insbesondere Symmetrie, Navigierbarkeit, Sichtbarkeit von Assoziationsenden und der Besitz von Assoziationsenden in Konzepte der Zielprogrammiersprache umgesetzt werden können.
- In einem zweiten Schritt wird die Semantik von Aktivitäten untersucht und eine Abbildung der Token-Flow-Semantik auf Threads der Zielsprache erarbeitet. Die größte Herausforderung besteht hier darin, alle Arten von Kontrollknoten zu unterstützen und gleichzeitig die Auswirkungen von bedingten Flüssen angemessen zu berücksichtigen.
- Aktivitäten bilden die Ausführungsumgebung für Aktionen. In einem dritten Schritt wird die Semantik einer Teilmenge von Aktionen diskutiert. Diese Menge umfasst hauptsächlich diejenigen Aktionen, die semantisch mit statischen Strukturen zusammenhängen, z. B. durch Lesen oder Aktualisieren von Klassenattributen oder durch Erzeugen oder Lesen von Klassenverknüpfungen in Assoziationen.
- Basierend auf diesen drei Schritten geben wir eine formale und ausführbare Beschreibung, wie Modelle unter Berücksichtigung der semantischen Details von Eingabemodellen in Code übersetzt werden können.
- Wir diskutieren Modifikationen der UML, um die Semantik von statischen Strukturen mit der Semantik des dynamischen Verhaltens in Einklang zu bringen und umgekehrt.

Im Zuge der Auseinandersetzung mit den oben aufgeführten Punkten trägt diese Arbeit zum Verständnis und zur weiteren Entwicklung von modellgetriebener Softwareentwicklung bei durch

- ein besseres und tieferes Verständnis der Semantik und einer semantikerhaltenden Implementierung von Assoziationen, insbesondere auch mehrstelliger Assoziationen und Assoziationsklassen
- eine allgemeine Abbildung der Token-Fluss-Semantik von Aktivitäten auf Java-Threads, einschließlich der Semantik von Kontrollknoten und der Kombination von Kontrollknoten mit bedingten Flüssen

- Vorschläge, wie die UML-Semantik verbessert werden kann, um die Semantik struktureller Merkmale (z. B. Klassenattribute) in Einklang mit der Semantik von Aktionen zu bringen, die auf strukturelle Merkmale zugreifen (z. B. durch lesen oder verändern von Attributwerten)
- eine Transformation von sehr abstrakten Modellierungskonzepten hin zu weniger abstrakten Programmiersprachenkonzepten sowie ein Übergang von Modellen zu Code, die beide formal durch ausführbare deklarative Formalismen angegeben werden, die als Standard der Object Management Group veröffentlicht sind.

Basierend auf diesen Beiträgen könnte die Effizienz von modellgetriebener Softwareentwicklung gesteigert werden, indem die Fähigkeit der Werkzeuge verbessert wird, Modelle zu verarbeiten, die die volle Vielfalt von UML-Modellierungskonzepten verwenden. Entwicklungswerkzeuge müssen in die Lage versetzt werden, den vollständigen semantischen Inhalt der Eingabemodelle, wie in dieser Arbeit beschrieben, in eine moderne Zielsprache zu übersetzen, damit der Vorteil der Modellierungssprachen, nämlich das höhere Abstraktionsniveau, im Entwicklungsprozess gewinnbringend ausgeschöpft werden kann.

# Appendix A

# Macros for Action Implementation

This part of the appendix contains macros referenced from listings contained in Chapter 4.

```
String createInputParameter( <InputPin[] input> ){
  String result = "";
  if( <input>.length > 0 ){
    result = <input[0].type>+" "+<input[0].name>;
  }
  for (int i = 1; i < <input>.length; ++i){
    result += ", "+<input[i].type>+" "+<input[i].name>;
  }
  return result
}
```

Listing A.1: Macro building a string representation from input pins.

```
String createParameterValues( <InputPin[] input> ){
  String result = "";
  if( <input>.length > 0){
    result = <input[0].value>;
  }
  for (int i = 1; i < <input>.length; ++i){
    result += ", " + <input[i].value>;
  }
  return result;
}
```

Listing A.2: Macro building a string representation of input values.

```
String createParameterValues( <InputPin[] input> , <OutPin[] output> ){
  String result = "";
  if( <input>.length > 0){
    result = <input[0]>;
  }
  for (int i = 1; i < <input>.length; ++i){
    result += ", "+<input[i]>;
  }
  for (int i = 0; i < <output>.length; ++i){
    result += ", " + <output[i]>;
  }
  return result;
}
```

Listing A.3: Macro building a string representation of input and output values.

# Appendix B

# QVT Relations and Queries

This part of the appendix contains parts of the model transformation discussed in Chapter 5 which are subject to low level details. We provide those parts here since they are nevertheless needed to execute transformations.

## B.1 Structural Transformation

Here we present those parts of the transformation related to structural aspects of the model.

### B.1.1 Transforming Attributes

```
1  relation SC_PublicProtectedPackageProperty
2  {
3    checkonly domain source srcClass : Class {
4      ownedAttribute = srcProperty : Property {
5        isDerived = false,
6        upper = 1 }};
7
8    enforce domain target trgClass : Class {
9      ownedAttribute = trgProperty : Property {
10       visibility = VisibilityKind::private }};
11
12   when {
13     srcProperty.association.oclIsUndefined(); }
14
15   where {
16     Copy_Property(srcProperty, trgProperty)↗B.14;
17     SC_PropertyAccessOperation(false, srcProperty, trgClass)↗B.3; }
18 }
```

Listing B.1: Transforming non-private, non-derived, single-valued owned attributes.

```
relation SC_PublicProtectedPackageProperty_Derived
{
  primitive domain trgClass : Class;
  primitive domain trgImplClass : Class;

  checkonly domain source srcClass : Class {
    ownedAttribute = srcProperty : Property {
      isDerived = true,
      upper = 1 }};

  when {
    srcProperty.visibility <> VisibilityKind::private;
    srcProperty.association.oclIsUndefined(); }

  where {
    SC_PropertyAccessOperation(true, srcProperty, trgClass)↗B.3;
    SC_PropertyAccessOperation(false, srcProperty, trgImplClass)↗B.3;  }
}
```

Listing B.2: Transforming non-private, derived, single-valued owned attributes.

```
relation SC_PropertyAccessOperation
{
  primitive domain isAbstractValue : Boolean;
  primitive domain srcProperty : Property;

  enforce domain target trgClass : Class {
    ownedOperation = trgClassGetter : Operation {
      isAbstract = isAbstractValue },
    ownedOperation = trgClassSetter : Operation {
      isAbstract = isAbstractValue }};

  where {
    SC_PropertyAccessOperation_Comment(
        'Getter', srcProperty.name, trgClassGetter)↗B.33;
    SC_PropertyAccessOperation_Comment(
        'Setter', srcProperty.name, trgClassSetter)↗B.33;
    SC_GetterCharacteristics(srcProperty, trgClassGetter)↗5.12;
    SC_SetterCharacteristics(srcProperty, trgClassSetter)↗B.4; }
}
```

Listing B.3: Adding access operations for owned attributes.

```
relation SC_SetterCharacteristics
{
  primitive domain srcProperty : Property;

  enforce domain target trgOperation : Operation {
    name = 'set' + srcProperty.name.firstToUpper(),
    ownedParameter = trgParameter : Parameter {
      name = srcProperty.name.firstToLower(),
      lower = srcProperty.lower,
      upper = srcProperty.upper },
    visibility = ownedElementVisibility(srcProperty.visibility)↗B.45 };

  where {
    Copy_Parameter_ClassType(srcProperty.type, trgParameter)↗B.22;
    Copy_Parameter_PrimitiveType(srcProperty.type, trgParameter)↗B.23; }
}
```

Listing B.4: Adding getter operations for accessing owned attributes.

### Adaptations for the Transformation of Privately Owned Attributes and Operations

The following listings include relations for the transformation of privately owned attributes and operations.

```
top relation SingleInheritance
{
  checkonly domain source srcPackage : Package {
    packagedElement = srcSuperClass : Class {},
    packagedElement = srcSubClass : Class {}
  };

  enforce domain target trgPackage : Package {
    name = 'PSM',
    packagedElement = trgSuperImplClass : Class {
      name = srcSuperClass.name + 'Impl' },
    packagedElement = trgSubClass : Class {
      name = srcSubClass.name,
      generalization = trgGeneralization : Generalization {
        general = trgSuperImplClass }}};

  when {
    srcSubClass.general->includes(srcSuperClass.oclAsType(Classifier)); }

  where {
      hidePrivateOperation(srcSuperClass, trgSubClass)↗B.6;
      hidePrivateMultivaluedAttributes(srcSuperClass, trgSubClass)↗B.7;
      hidePrivateSinglevaluedAttributes(srcSuperClass, trgSubClass)↗B.8;
  }
}
```

Listing B.5: Handling private attributes and private operations.

```
relation hidePrivateOperation
{
  checkonly domain source srcSuperClass: Class{
      ownedOperation = privOp : Operation {
      visibility = VisibilityKind::private
      }
  };

  enforce domain target trgSubClass : Class {
    ownedOperation = hideOp : uml::Operation {
      name = privOp.name,
      ownedComment = commentOp : uml::Comment {
        _body = 'hideOperation'+privOp.name.firstToUpper()
      },
      visibility = uml::VisibilityKind::protected
    }
  };

  where {
    Copy_Operation_Parameter(privOp, hideOp); }
}
```

Listing B.6: Hiding private operations in subclasses.

```
relation hidePrivateMultivaluedAttributes
{
  checkonly domain source srcSuperClass: Class{
      ownedAttribute = privAttr : Property {
        visibility = VisibilityKind::private
      }
  };

  enforce domain target trgSubClass : Class {
    ownedOperation = hideAdd : uml::Operation {
      ownedComment = commentAdd : uml::Comment {
        _body = 'hideAttribute'+privAttr.name.firstToUpper()
      },
      ownedParameter = trgAddParameter : Parameter {
        direction = ParameterDirectionKind::_in,
        name = privAttr.name,
        type = privAttr.type,
        lower = 1,
        upper = 1
      },
      name = 'add' + privAttr.name.firstToUpper(),
      visibility = uml::VisibilityKind::protected
    },
    ownedOperation = hideRemove : uml::Operation {
      ownedComment = commentRemove : uml::Comment {
        _body = 'hideAttribute'+privAttr.name.firstToUpper()
      },
      ownedParameter = trgRemoveParameter : Parameter {
        direction = ParameterDirectionKind::_in,
        name = privAttr.name,
        type = privAttr.type,
        lower = 1,
        upper = 1
      },
      name = 'remove' + privAttr.name.firstToUpper(),
      visibility = uml::VisibilityKind::protected
    },
    ownedOperation = hideGet : uml::Operation {
      ownedComment = commentGet : uml::Comment {
        _body = 'hideAttribute'+privAttr.name.firstToUpper()
      },
      ownedParameter = trgGetParameter : Parameter {
        direction = ParameterDirectionKind::return,
        name = privAttr.name,
        type = privAttr.type,
        lower = privAttr.lower,
        upper = privAttr.upper
      },
      name = 'get' + privAttr.name.firstToUpper(),
      visibility = uml::VisibilityKind::protected
    }

  };

  when {
    privAttr.upper <> 1 ; }
}
```

Listing B.7: Inserting protected attributes for hiding private, multi-valued attributes in subclasses.

```
relation hidePrivateSinglevaluedAttributes
{
  checkonly domain source srcSuperClass: Class{
      ownedAttribute = privAttr : Property {
        visibility = VisibilityKind::private
      }
  };

  enforce domain target trgSubClass : Class {
    ownedOperation = hideRemove : uml::Operation {
      ownedComment = commentRemove : uml::Comment {
        _body = 'hideAttribute'+privAttr.name.firstToUpper()
      },
      ownedParameter = trgSetParameter : Parameter {
        direction = ParameterDirectionKind::_in,
        name = privAttr.name,
        type = privAttr.type,
        lower = 1,
        upper = 1
      },
      name = 'set' + privAttr.name.firstToUpper(),
      visibility = uml::VisibilityKind::protected
    },
    ownedOperation = hideGet : uml::Operation {
      ownedComment = commentGet : uml::Comment {
        _body = 'hideAttribute'+privAttr.name.firstToUpper()
      },
      ownedParameter = trgGetParameter : Parameter {
        direction = ParameterDirectionKind::return,
        name = privAttr.name,
        type = privAttr.type,
        lower = privAttr.lower,
        upper = privAttr.upper
      },
      name = 'get' + privAttr.name.firstToUpper(),
      visibility = uml::VisibilityKind::protected
    }
  };

  when {
    privAttr.upper = 1; }
}
```

Listing B.8: Inserting protected attributes for hiding private, single-valued attributes in subclasses.

### B.1.2 Transformations of Associations and Association Classes

```
top relation Association
{
  checkonly domain source srcAssociation : Association {};

  enforce domain target trgPackage : Package {
    name = 'PSM',
    packagedElement = trgLinkClass : Class {
      name = 'Link' + associationName(srcAssociation)↗B.38,
      ownedOperation = trgLinkClassConstructor : Operation {
        name = 'Link' + associationName(srcAssociation)↗B.38,
        visibility = VisibilityKind::public }},
    packagedElement = trgManagerClass : Class {
      name = 'LinkManager' + associationName(srcAssociation)↗B.38,
      ownedOperation = trgManagerClassConstructor : Operation {
        name = 'LinkManager' + associationName(srcAssociation)↗B.38,
        visibility = VisibilityKind::private },
      ownedOperation = trgCreateLinkOperation : Operation {
        name = 'createLink',
        visibility = VisibilityKind::public },
      ownedOperation = trgRemoveLinkOperation : Operation {
        name = 'removeLink',
        visibility = VisibilityKind::public },
      ownedOperation = trgGetLinkOperation : Operation {
        name = 'getLinks',
        visibility = VisibilityKind::public
      }
    }
  };

  when {
    not srcAssociation.oclIsTypeOf(AssociationClass);
    srcAssociation.memberEnd->size() > 1; }

  where {
    A_LinkClass_Comment('Association', trgLinkClass)↗B.24;
    A_ManagerClass_Comment('Association', trgManagerClass)↗B.25;
    A_MemberClass_Comment(
          'Association', srcAssociation, trgPackage)↗B.26;
    A_MemberClass_Property_LinkManager(
        trgManagerClass, srcAssociation, trgPackage, trgPackage)↗5.18;
    A_LinkClass(srcAssociation, trgLinkClass)↗5.19;
    A_LinkClass_Constructor(srcAssociation,
        trgLinkClassConstructor)↗5.20;
    A_ManagerClass(trgLinkClass, srcAssociation, trgManagerClass,
        trgCreateLinkOperation, trgRemoveLinkOperation,
        trgGetLinkOperation)↗5.21;
  }
}
```

Listing B.9: Transformation of association excluding association classes.

```
top relation AssociationClass
{
  checkonly domain source srcAssociationClass : AssociationClass {};

  enforce domain target trgPackage : Package {
    name = 'PSM',
    packagedElement = trgLinkClass : Class {
      name = srcAssociationClass.name,
      ownedOperation = trgLinkClassConstructor : Operation {
        name = srcAssociationClass.name,
        visibility = VisibilityKind::public }},
    packagedElement = trgLinkImplClass : Class {
      name = srcAssociationClass.name + 'Impl',
      ownedOperation = trgLinkImplClassConstructor : Operation {
        name = srcAssociationClass.name + 'Impl',
        visibility = VisibilityKind::public }},
    packagedElement = trgManagerClass : Class {
      name = 'LinkManager_' + srcAssociationClass.name,
      ownedOperation = trgManagerClassConstructor : Operation {
        name = 'LinkManager_' + srcAssociationClass.name,
        visibility = VisibilityKind::private },
      ownedOperation = trgCreateLinkOperation : Operation {
        name = 'createLink',
        visibility = VisibilityKind::public,
        type = trgLinkClass },
      ownedOperation = trgRemoveLinkOperation : Operation {
        name = 'removeLink',
        visibility = VisibilityKind::public },
      ownedOperation = trgGetLinkOperation : Operation {
        name = 'getLinks',
        visibility = VisibilityKind::public
      }
    }
  };

  when {
    srcAssociationClass.memberEnd->size() > 1; }

  where {
    A_LinkClass_Comment('Association(Class)', trgLinkClass)↗B.24;
    A_ManagerClass_Comment('Association(Class)', trgManagerClass)↗B.25;
    A_MemberClass_Comment(
        'Association(Class)', srcAssociationClass, trgPackage)↗B.26;
    A_MemberClass_Property_LinkManager(
        trgManagerClass, srcAssociationClass, trgPackage, trgPackage)↗5.18;
    A_LinkClass(srcAssociationClass, trgLinkClass)↗5.19;
    A_LinkClass_Constructor(srcAssociationClass,
        trgLinkClassConstructor)↗5.20;
    A_LinkClass_Constructor(srcAssociationClass,
                trgLinkImplClassConstructor)↗5.20;
    A_ManagerClass(trgLinkClass, srcAssociationClass, trgManagerClass,
        trgCreateLinkOperation, trgRemoveLinkOperation,
        trgGetLinkOperation)↗5.21;
  }
}
```

Listing B.10: Transformation of association classes.

```
relation AC_MemberClass_AddRemoveOperation
{
  primitive domain namePrefix : String;
  primitive domain srcClassMemberEnd : Property;

  checkonly domain source srcAssociationClass : AssociationClass {
    memberEnd = srcMemberEnd : Property {}};

  enforce domain target trgOperation : Operation {
    name = namePrefix + operationName(
        srcAssociationClass.memberEnd->excluding(srcClassMemberEnd))↗B.36,
    ownedParameter = trgInParameter : Parameter {
      name = roleNameFirstLower(srcMemberEnd),
      type = trgParameterType : Class {
        name = srcMemberEnd.type.name }},
    visibility = minimalImplVisibility(
        srcAssociationClass.memberEnd->excluding(srcClassMemberEnd))↗B.43,
    type = trgType : Class {
      name = srcAssociationClass.name }};

  when { srcMemberEnd <> srcClassMemberEnd; }

  where {
    A_MemberClass_AddRemoveOperation_Comment(namePrefix, 'AssociationClass',
        'LinkManager_' + srcAssociationClass.name, trgOperation)↗B.29;
  }
}
```

Listing B.11: Specification of operations for write access on an association class.

```
relation A_MemberClass_GetOperation_Binary
{
  primitive domain srcClassMemberEnd : Property;

  checkonly domain source srcAssociation : Association {
    memberEnd = srcMemberEnd : Property {}};

  enforce domain target trgMemberClass : Class {
    ownedOperation = trgOperation : Operation {}};

  when {
    srcAssociation.memberEnd->size() = 2;
    srcAssociation.navigableOwnedEnd->exists(
        p : Property | p = srcMemberEnd);
    srcMemberEnd <> srcClassMemberEnd;
    not trgMemberClass.ownedOperation->exists(
        o : Operation | o.name = 'get' + roleNameFirstUpper(srcMemberEnd));
  }

  where {
    A_MemberClass_GetOperation(
        srcMemberEnd, srcClassMemberEnd, srcAssociation, trgOperation)↗5.28;
  }
}
```

Listing B.12: Preparing an operation for reading ends of a binary association.

### B.1.3 Copying Model Elements

In the following, we give the code needed to copy elements from the source model into the target model without any changes to the copied structures.

```
relation Copy_ElementImport
{
  primitive domain srcElementImport : ElementImport;

  enforce domain target trgElementImport : ElementImport {
    alias = srcElementImport.alias,
    importedElement = srcElementImport.importedElement,
    visibility = srcElementImport.visibility };
}
```

Listing B.13: Copying *ElementImport*s.

```
relation Copy_Property
{
  primitive domain srcProperty : Property;

  enforce domain target trgProperty : Property {
    aggregation = srcProperty.aggregation,
    default = srcProperty.default,
    isDerived = srcProperty.isDerived,
    isDerivedUnion = srcProperty.isDerivedUnion,
    isLeaf = srcProperty.isLeaf,
    isOrdered = srcProperty.isOrdered,
    isReadOnly = srcProperty.isReadOnly,
    isStatic = srcProperty.isStatic,
    isUnique = srcProperty.isUnique,
    lower = srcProperty.lower,
    name = srcProperty.name,
    upper = srcProperty.upper };

  where {
    Copy_Property_ClassType(srcProperty.type, trgProperty)↗B.15;
    Copy_Property_PrimitiveType(srcProperty.type, trgProperty)↗B.16;
  }
}
```

Listing B.14: Copying attributes of *Property*.

```
relation Copy_Property_ClassType
{
  primitive domain srcType : Class;

  enforce domain target trgProperty : Property {
    type = trgPropertyType : Class {
      name = srcType.name }};
}
```

Listing B.15: Enforcing existence of types of owned attributes.

```
relation Copy_Property_PrimitiveType
{
  primitive domain srcType : PrimitiveType;

  enforce domain target trgProperty : Property {
    type = srcType };
}
```

Listing B.16: Copying primitive tapes of owned attributes.

```
relation Copy_Operation
{
  primitive domain srcOperation : Operation;

  enforce domain target trgOperation : Operation {
    concurrency = srcOperation.concurrency,
    isAbstract = srcOperation.isAbstract,
    isLeaf = srcOperation.isLeaf,
    isOrdered = srcOperation.isOrdered,
    isQuery = srcOperation.isQuery,
    isStatic = srcOperation.isStatic,
    isUnique = srcOperation.isUnique,
    lower = srcOperation.lower,
    name = srcOperation.name,
    upper = srcOperation.upper };

  where {
    Copy_Operation_ClassType(srcOperation.type, trgOperation)↗B.18;
    Copy_Operation_PrimitiveType(srcOperation.type, trgOperation)↗B.19;
    Copy_Operation_Parameter(srcOperation, trgOperation)↗B.20; }
}
```

Listing B.17: Copying attributes of *Operation*.

```
relation Copy_Operation_ClassType
{
  primitive domain srcType : Class;

  enforce domain target trgOperation : Operation {
    type = trgOperationType : Class {
      name = srcType.name }};
}
```

Listing B.18: Enforcing the existence of return types of operations.

```
relation Copy_Operation_PrimitiveType
{
  primitive domain srcType : PrimitiveType;

  enforce domain target trgOperation : Operation {
    type = srcType };
}
```

Listing B.19: Copying primitive return types of operations.

```
relation Copy_Operation_Parameter
{
  checkonly domain source srcOperation : Operation {
    ownedParameter = srcParameter : Parameter {}};

  enforce domain target trgOperation : Operation {
    ownedParameter = trgParameter : Parameter {
      default = srcParameter.default,
      direction = srcParameter.direction,
      effect = srcParameter.effect,
      isException = srcParameter.isException,
      isOrdered = srcParameter.isOrdered,
      isStream = srcParameter.isStream,
      isUnique = srcParameter.isUnique,
      lower = srcParameter.lower,
      name = srcParameter.name,
      upper = srcParameter.upper,
      visibility = srcParameter.visibility }};

  where {
    Copy_Parameter_ClassType(srcParameter.type, trgParameter)↗B.22;
    Copy_Parameter_PrimitiveType(srcParameter.type, trgParameter)↗B.23;
  }
}
```

Listing B.20: Copying attributes of operation parameters.

```
relation Copy_Activity_Parameter
{
  checkonly domain source srcActivity : Activity {
    ownedParameter = srcParameter : Parameter {}};

  enforce domain target trgOperation : Operation {
    ownedParameter = trgParameter : Parameter {
      default = srcParameter.default,
      direction = srcParameter.direction,
      effect = srcParameter.effect,
      isException = srcParameter.isException,
      isOrdered = srcParameter.isOrdered,
      isStream = srcParameter.isStream,
      isUnique = srcParameter.isUnique,
      lower = srcParameter.lower,
      name = srcParameter.name,
      upper = srcParameter.upper,
      visibility = srcParameter.visibility }};

  where {
    Copy_Parameter_ClassType(srcParameter.type, trgParameter)↗B.22;
    Copy_Parameter_PrimitiveType(srcParameter.type, trgParameter)↗B.23;
  }
}
```

Listing B.21: Enforcing the existence of parameter types.

```
relation Copy_Parameter_ClassType
{
  primitive domain srcType : Class;

  enforce domain target trgParameter : Parameter {
    type = trgParameterType : Class {
      name = srcType.name }};
}
```

Listing B.22: Enforcing the existence of parameter types.

```
relation Copy_Parameter_PrimitiveType
{
  primitive domain srcType : PrimitiveType;

  enforce domain target trgParameter : Parameter {
    type = srcType };
}
```

Listing B.23: Copying primitive parameter types.

### B.1.4 Marking Model Elements by Comments

```
relation A_LinkClass_Comment
{
  primitive domain srcAssociationKind : String;

  enforce domain target trgClass : Class {
    ownedComment = trgComment : Comment {
      _body = 'Class#Kind:LinkClass#' + srcAssociationKind }};
}
```

Listing B.24: Marking a link implementation class.

```
relation A_ManagerClass_Comment
{
  primitive domain srcAssociationKind : String;

  enforce domain target trgMemberClass : Class {
    ownedComment = trgComment : Comment {
      _body = 'Class#Kind:LinkManager#' + srcAssociationKind }};
}
```

Listing B.25: Marking a link manager class.

```
relation A_MemberClass_Comment
{
  primitive domain srcAssociationKind : String;

  checkonly domain source srcAssociation : Association {
    memberEnd = srcMemberEnd : Property {}};

  enforce domain target trgPackage : Package {
    packagedElement = trgClass : Class {
      name = srcMemberEnd.type.name,
      ownedComment = trgComment : Comment {
        _body = 'Class#Kind:LinkMemberClass#' + srcAssociationKind }}};
}
```

Listing B.26: Marking a class connected to an association end.

```
relation A_MemberClass_Property_LinkManager_Comment
{
  enforce domain target trgProperty : Property {
    ownedComment = trgComment : Comment {
      _body = 'Property#Kind:LinkMemberClassProperty#LinkManager' }};
}
```

Listing B.27: Marking the owned attribute of a member end class which references the link manager class.

```
relation SingleClass_PropertyAccessOperation_Comment
{
  primitive domain srcKindName : String;
  primitive domain srcPropertyName : String;

  enforce domain target trgOperation : Operation {
    ownedComment = trgComment : Comment {
      _body = 'Operation#Kind:' + srcKindName + '#' + srcPropertyName }};
}
```

Listing B.28: Marking operations for accessing association ends within link class.

```
relation A_MemberClass_AddRemoveOperation_Comment
{
  primitive domain srcOperationKind : String;
  primitive domain srcAssociationKind : String;
  primitive domain srcManagerClassName : String;

  enforce domain target trgOperation : Operation {
    ownedComment = trgComment : Comment {
      _body = 'Operation#Kind:LinkMemberClassOperation#' +
              srcOperationKind + '#' +
              srcAssociationKind + '#' + srcManagerClassName }
  };
}
```

Listing B.29: Marking operations for accessing association ends within a member end class.

```
relation A_MemberClass_GetOperation_Comment
{
  checkonly domain source srcAssociation : Association {};

  enforce domain target trgOperation : Operation {
    ownedComment = trgComment : Comment {
      _body = 'Operation#Kind:LinkMemberClassOperation#get#Association#' +
              'LinkManager' + associationName(srcAssociation) }};

  when { not srcAssociation.oclIsKindOf(AssociationClass); }
}
```

Listing B.30: Marking operations reading association ends in member end classes.

```
relation AC_MemberClass_GetOperation_Comment
{
  checkonly domain source srcAssociationClass : AssociationClass {};

  enforce domain target trgOperation : Operation {
    ownedComment = trgComment : Comment {
      _body = 'Operation#Kind:LinkMemberClassOperation#' +
              'get#AssociationClass#LinkManager_' +
              srcAssociationClass.name }};
  }
```

Listing B.31: Marking operations reading association ends of an association class in member end classes.

```
relation SC_Comment
{
  enforce domain target trgClass : Class {
    ownedComment = trgComment : Comment {
      _body = 'Class#Kind:ImplClass' }};
}
```

Listing B.32: Marking a class being an implementation class.

```
relation SC_PropertyAccessOperation_Comment
{
  primitive domain srcKindName : String;
  primitive domain srcPropertyName : String;

  enforce domain target trgOperation : Operation {
    ownedComment = trgComment : Comment {
      _body = 'Operation#Kind:' + srcKindName + '#' + srcPropertyName }};
}
```

Listing B.33: Marking operations accessing owned attributes.

### B.1.5 Queries

```
query roleNameFirstLower(m : Property) : String
{
  if(not m.name.oclIsUndefined() and m.name  <>  '')
    then m.name.firstToLower()
  else
    m.type.name.firstToLower()
  endif
}
```

Listing B.34: Query for obtaining a role name with lower first letter.

```
query roleNameFirstUpper(m : Property) : String
{
  if(not m.name.oclIsUndefined() and m.name <> '')
    then m.name.firstToUpper()
  else
    m.type.name.firstToUpper()
  endif
}
```

Listing B.35: Query for obtaining a role name with upper first letter.

```
query operationName(m : OrderedSet(Property)) : String
{
  operationName(m, '')↗B.37
}
```

Listing B.36: Query for obtaining an operation name.

```
query operationName(m : OrderedSet(Property), n : String) : String
{
  if(m->size() = 0)
    then n
  else operationName(
    m->excluding(m.first()), n.concat(roleNameFirstUpper(m.first())↗B.35))
  endif
}
```

Listing B.37: Query for obtaining a concatenation of property names.

```
query associationName(a : Association) : String
{
  if(not a.name.oclIsUndefined() and a.name <> '')
    then '_'.concat(a.name)
  else
    associationName(a.memberEnd, '')↗B.39
  endif
}
```

Listing B.38: Query for obtaining the name of an association of deriving a name from its ends.

```
query associationName(m : OrderedSet(Property), n : String) : String
{
  if(m->size() = 0)
    then n
  else
    associationName(m->excluding(
      m.first()), n.concat('_').concat(roleNameFirstLower(m.first())↗B.34))
  endif
}
```

Listing B.39: Query for obtaining a concatenation of property names.

```
query operationSuffix(m1: Property, m2: Property, a: Association): String
{
  if(not a.memberEnd->excluding(m2)->exists(
      p : Property | p.type = m2.type)) then
    ''
  else
    if(m1.type = m2.type) then
      ''
    else
      'By'.concat(roleNameFirstUpper(m2))
    endif
  endif
}
```

Listing B.40: Query for making operation names distinct if multiple ends are of the same type.

```
query propertyLowerValue(p : Property) : String
{
  if(p.lower.oclIsUndefined()) then
    '1'
  else
    if(p.lower = -1) then
      '*'
    else
      toString(p.lower)
    endif
  endif
}
```

Listing B.41: Query returning a properties lower multiplicity bound.

```
query propertyUpperValue(p : Property) : String
{
  if(p.upper.oclIsUndefined()) then
    '1'
  else
    if(p.upper = -1) then
      '*'
    else
      toString(p.upper)
    endif
  endif
}
```

Listing B.42: Query returning a properties upper multiplicity bound.

```
query minimalImplVisibility(m : OrderedSet(Property)) : VisibilityKind
{
  if(m->exists(
      p : Property | p.visibility = VisibilityKind::private
                  or p.visibility = VisibilityKind::_package
                  or p.visibility = VisibilityKind::protected))
  then
    VisibilityKind::protected
  else
    VisibilityKind::public
  endif
}
```

Listing B.43: Query for deriving a visibility from visibilities of a set of properties.

```
query minimalVisibility(m : OrderedSet(Property)) : VisibilityKind
{
  if(m->exists(
      p : Property | p.visibility = VisibilityKind::private))
  then
    VisibilityKind::private
  else if(m->exists(
      p : Property | p.visibility = VisibilityKind::_package))
  then
    VisibilityKind::_package
  else if(m->exists(
      p : Property | p.visibility = VisibilityKind::protected))
  then
    VisibilityKind::protected
  else
    VisibilityKind::public
  endif endif endif
}
```

Listing B.44: Query returning the most restrictive visibility of a set of properties.

```
  query ownedElementVisibility(vis : VisibilityKind) : VisibilityKind
  {
    if(vis = uml::VisibilityKind::private)
      then uml::VisibilityKind::protected
      else vis
    endif
  }
```

Listing B.45: Query returning the visibility for the implementation of an owned element.

```
query modulo(a : Integer, b : Integer) : Integer
{
  a - ((a / b).floor() * b)
}
```

Listing B.46: Query implementing a *modulo* operator.

```
query toString(i : Integer) : String
{
  toString(OrderedSet{'0','1','2','3','4','5','6','7','8','9'},
           OrderedSet{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, i, '')
}
```

Listing B.47: Query implementing a mapping from Integer to String.

```
query toString(ns : OrderedSet(String),
               ni : OrderedSet(Integer), i : Integer, s : String) : String
{
  if(ni->exists(n : Integer | i = n)) then
    s.concat(ns->at(i + 1))
  else
    toString((i / 10).floor()).concat(toString(ns, ni, modulo(i, 10), s))
  endif
}
```

Listing B.48: Query translating an Integer into its string representation.

## B.2 Behavioral Transformation

### B.2.1 Copying Model Elements

```
top relation Activity_TypedElement
{
  checkonly domain source srcTypedElement : TypedElement {
    type = srcType : Class {}};

  where {
    Activity_TypedElement_Change(srcType.name, srcTypedElement)↗B.50; }
}
```

Listing B.49: Realtion enforcing the existence of all typed elements in the target model.

```
relation Activity_TypedElement_Change
{
  primitive domain trgClassName : String;

  enforce domain target trgTypedElement : TypedElement {
    type = trgType : Class {
      name = trgClassName }};
}
```

Listing B.50: Relation enforcing the existence of a type.

```
relation Copy_Activity_Parameter
{
  checkonly domain source srcActivity : Activity {
    ownedParameter = srcParameter : Parameter {}};

  enforce domain target trgOperation : Operation {
    ownedParameter = trgParameter : Parameter {
      default = srcParameter.default,
      direction = srcParameter.direction,
      effect = srcParameter.effect,
      isException = srcParameter.isException,
      isOrdered = srcParameter.isOrdered,
      isStream = srcParameter.isStream,
      isUnique = srcParameter.isUnique,
      lower = srcParameter.lower,
      name = srcParameter.name,
      upper = srcParameter.upper,
      visibility = srcParameter.visibility }};

  where {
    Copy_Parameter_ClassType(srcParameter.type, trgParameter)↗B.52;
    Copy_Parameter_PrimitiveType(srcParameter.type, trgParameter)↗B.53;
  }
}
```

Listing B.51: Relation for copying activity parameters.

```
relation Copy_Parameter_ClassType
{
  primitive domain srcType : Class;

  enforce domain target trgParameter : Parameter {
    type = trgParameterType : Class {
      name = srcType.name }};
}
```

Listing B.52: Relation enforcing the existence of the types of activity parameters.

```
relation Copy_Parameter_PrimitiveType
{
  primitive domain srcType : PrimitiveType;

  enforce domain target trgParameter : Parameter {
    type = srcType };
}
```

Listing B.53: Relation enforcing the existence of primitive types of activity parameters.

### B.2.2 Considering Object Flows

```
relation Activity_CallBehaviorAction_Incoming_Data
{
  checkonly domain source srcAction : Action{
    input = srcParam : InputPin {}
  };

  enforce domain target trgCallBehaviorAction : CallBehaviorAction{
    argument = trgParam : InputPin{
      name = '_' + srcParam.name,
      type = srcParam.type,
      lower = srcParam.lower,
      upper = srcParam.upper,
      incoming = srcParam.incoming
    }
  };

  enforce domain target trgSubactivity: Activity{
    ownedParameter = param : Parameter {
      name = srcParam.name,
      direction = uml::ParameterDirectionKind::_in,
      type = srcParam.type
    },
    node = actParam : ActivityParameterNode{
      name = srcParam.name,
      type = srcParam.type,
      parameter = param
    },
    edge = flow : ObjectFlow{
      source = actParam,
      target = srcParam,
      name = source.name + '->' + target.name
    }
  };
}
```

Listing B.54: Updating the target of incoming object flows of head actions to call behavior actions associated with subactivities.

```
relation Activity_CallBehaviorAction_Outgoing_Data
{
  checkonly domain source srcAction : Action{
    output = srcParam : OutputPin {}
  };

  enforce domain target trgCallBehaviorAction : CallBehaviorAction{
    result = trgParam : OutputPin{
      name = '_' + srcParam.name,
      type = srcParam.type,
      lower = srcParam.lower,
      upper = srcParam.upper,
      outgoing = srcParam.outgoing
    }
  };

  enforce domain target trgSubactivity: Activity{
    ownedParameter = param : Parameter {
      name = srcParam.name + 'Parameter',
      direction = ParameterDirectionKind::return,
      type = srcParam.type
    },
    node = actParam : ActivityParameterNode{
      name = '_AP_' + srcParam.name,
      type = srcParam.type,
      parameter = param
    },
    edge = flow : ObjectFlow{
      source = srcParam,
      target = actParam,
      name = source.name + '->' + target.name
    }
  };
}
```

Listing B.55: Updating the source of an outgoing object flow of the last flow body action of a sequence.

### B.2.3 Queries

```
query isSubsequentSubactivityAction(
  a1 : Action, a2 : Action, a : Activity) : Boolean
{
  if(a2.outgoing.first().target.oclIsUndefined()) then
    false
  else if(a2.outgoing.first().target.oclAsType(Action) = a1) then
    true
  else if(Activity_IsFlowBodyAction(
    a2.outgoing.first().target.oclAsType(Action), a)) then
    isSubsequentSubactivityAction(
      a1, a2.outgoing.first().target.oclAsType(Action), a)
  else
    false
  endif
  endif
  endif
}
```

Listing B.56: Query checking whether an action occurs after another action in the same sequence of actions.

# Appendix C

# Code Generation

This appendix contains some modifications to listings provided for generating code from transformed models and introduces additional listings in order to consider effects of guards at merge nodes, to implement a foundation for supporting object flows, and to generate code for supporting interruptible activity regions. Note that these listings require input models which have been generated by the relations as modified for supporting object flows, contained in Appendix B.2.2.

Furthermore, listings are provided showing the implementation of queries called from code generation templates.

After query implementations, a survey over the complete composition of the template snippets presented in Chapter 6 is given by Fig. C.1 and Fig. C.2.

## C.1 Considering Guarded Flows

Listing 6.32 contains a simplified processing of merge node which only produces correct code in case that incoming and outgoing flows of the merge node are not guarded. Listing C.1 shows how to consider guarded flows together with merge node.

The outgoing flow of the current node, which is the incoming flow of the merge node is tested in line 4, the outgoing edges of the target of the outgoing flows, which are the outgoing edges of the merge node, are tested in lines 5–6.

If the current thread is interrupted while waiting for the guards to evaluate to true, fId is set to 0 to terminate the current thread. Otherwise, if guards evaluate to true, fId is set to the appropriate value for continuing the activity execution.

```
1  [if (node.outgoing->asSequence()->first().target.oclIsKindOf(MergeNode))]
2  // next node is a merge
3  synchronized (context) {
4    while (! ([node.outgoing->asSequence()->first().guard.stringValue()/]
5      && [node.outgoing->asSequence()->first().target.outgoing
6                ->asSequence()->first().guard.stringValue()/])){
7      try { context.wait(); }
8      catch (InterruptedException e) { fID = 0; }
9    }
10 }
11 fId = [seqNr->indexOf(nextNode)/];
12 [/if]
```

Listing C.1: Consideration of guards at merge nodes.

## C.2 Considering Object Flows

The following Listing C.2 presents how to implement a wrapper class for multiple action outputs. For multiple activity outputs, a very similar template can be used matching each activity (line 1) of the transformed input model.

```
1  [template public generateCombinedClasses(c : OpaqueAction)]
2    [comment @main /]
3
4    [comment create classes for multiple output pins /]
5    [if (c.output->asSequence()->size() > 1)]
6      [file(outputFileName(c), false, 'UTF-8')]
7  /**
8   * Generated file: Wrapper class for flows between opaque actions!
9   */
10 package [containingPackages(c.activity).name->sep('.') /];
11       [visibilityString(c.visibility)/] class [c.output.type.name/]
12 {
13   /**
14    * Specification of values
15    */
16     [for (pin : OutputPin | c.output)]
17   private [pin.type.name/] [pin.name/];
18     [/for]
19
20   /**
21    * Specification of setter and getter methods
22    */
23       [for (pin : OutputPin | c.output)]
24   public void ['set'.camelCase(pin.name)/]([pin.type.name/] value){
25     this.[pin.name/] = value;
26   }
27
28   public [pin.type.name/] ['get'.camelCase(pin.name)/](){
29     return this.[pin.name/];
30   }
31       [/for]
32 }
33     [/file]
34   [/if]
35 [/template]
```

Listing C.2: Generation of a wrapper for multiple action outputs.

The following Listing C.3 shows, how to declare local variables for inputs and outputs of call behavior actions of the transformed models. This variables are needed in order to provide outputs of a sequence of actions as inputs for another sequence of actions. Therefore, the variables must be declared outside the `run` method of class `ActivityThread`, e. g. in Listing 6.30 before including `ActivityExecutionImplementation`.

```
[let seqNr : Sequence(ActivityNode) = a.node->asSequence()]
// Prepare input and return parameters for action sequences
  [for (node : ActivityNode | a.node)]
    [if (node.oclIsKindOf(CallBehaviorAction))]
      [for (input : InputPin | inputActionValues(
                                node.oclAsType(Action))->asOrderedSet())]
        [if (input.upper>1)]
List<[input.type.name/]> [node.name/][input.name/] =
                                new LinkedList<[input.type.name/]>();
        [else]
[input.type.name/] [node.name/][input.name/] = null;
        [/if]
      [/for]
      [for (output : OutputPin | node.oclAsType(CallBehaviorAction).output)]
        [if (output.upper>1)]
List<[output.type.name/]> [node.name/][output.name/] =
                                new LinkedList<[output.type.name/]>();
        [else]
[output.type.name/] [node.name/][output.name/] = null;
        [/if]
      [/for]
    [/if]
  [/for]
[/let]
```

Listing C.3: Preparing local variables representing inputs and outputs of sequences of actions.

The following Listing C.4 provides a template generating methods to set the inputs of an action sequence. The generated methods are to be called after creating a new activity thread that requires inputs before that thread is started. The variables holding the values have been declared by the code generated by the template given in Listing C.3. The content of Listing C.4 should be included in Listing 6.30 after the content of Listing C.3.

```
[let seqNr : Sequence(ActivityNode) = a.node->asSequence()]
// Prepare methods for setting sequence inputs
  [for (node : ActivityNode | a.node)]
    [if (node.oclIsKindOf(CallBehaviorAction))]
      [for (input : InputPin | inputActionValues(
                                node.oclAsType(Action))->asOrderedSet())]
        [if (input.upper>1)]
public void set[input.name/]([input.type.name/] paramValue){
  [node.name/][input.name/].add(paramValue);
}
        [else]
public void set[input.name/]([input.type.name/] paramValue){
  [node.name/][input.name/] = paramValue;
}
        [/if]
      [/for]
    [/if]
  [/for]
[/let]
```

Listing C.4: Providing methods for setting inputs of action sequences.

The following Listing C.5 completes the call of an action sequence, as contained in Listing 6.31, line 10, by considering inputs and outputs.

```
1  [if (outputActionValues(node.oclAsType(Action))->asOrderedSet()->size()=1)]
2  // one output
3    [node.name/][outputActionValues(node.oclAsType(Action))
4      ->asSequence()->first().name/] =
5  [elseif (outputActionValues(node.oclAsType(Action))
6      ->asOrderedSet()->size() > 1)]
7  // multiple outputs
8    [let act : Activity =
9              node.oclAsType(CallBehaviorAction).behavior.oclAsType(Activity)]
10     [containingPackages(act).name->sep('.')/].
11       [act.name/][node.oclAsType(Action).output.type.name/] =
12   [/let]
13 [else]
14 // no outputs
15 [/if]
16 [camelCase('ex', node.name)/](
17 [for (input : InputPin |
18               inputActionValues(node.oclAsType(Action))
19                 ->asOrderedSet()) separator (', ')]
20   [node.name/][input.name/]
21 [/for]
22 );
```

Listing C.5: Passing inputs of action sequences as parameters.

Listing C.6 is a modification of Listing 6.37, lines 6–9 considering inputs to sequences of actions and return values.

```
1  private [if (returnValues(node.behavior.oclAsType(Activity))->size()>0)]
2    [node.output.type.name/] [else] void [/if] [camelCase('ex', node.name)/](
3  [for (param : Parameter | inputValues(node.behavior.oclAsType(Activity))
4                            ->asOrderedSet()) separator (', ')]
5    [if (param.direction.toString() = 'in')] final [/if]
6    [if (param.upper>1)]List<[/if]
7    [param.type.name/]
8    [if (param.upper>1)]>[/if]
9    [param.name /]
10 [/for]
11 )
```

Listing C.6: Considering object flows declarations of methods implementing action sequences.

Listing C.7 is a modification of Listing 6.37, lines 19–21 considering object flows. Before calling a method on the context object, a local variable for the return value is declared, if applicable. If the method returns a value, the call becomes the right side of an assignment. The inputs are provided as parameters of the method call.

If an action is the last action of a sequence, the output of the method call is returned as a return value of the method implementing the action sequence (lines 21–25).

```
1  [if (action.oclIsKindOf(OpaqueAction))]      // execute an opaqueAction
2    [if (action.output->asSet()->size() > 0)] // prepare parameters
3      [for (output : OutputPin |
4                    action.oclAsType(OpaqueAction).output->asOrderedSet())]
5        [output.type.name/] [output.name/] = null;
6      [/for]
7    [/if]
8  // call [actions->indexOf(action)/]. operation
9    [if (not action.output->asSet()->isEmpty())]
10 [action.output.name/] =
11   [/if]
12 context.[action.name/]([action.input.incoming.source.name->sep(', ')/]);
13   [if (action.output->asSet()->size() > 1)]
14 // map output parameters from wrapper if we have more than one output pin:
15     [for (output : OutputPin |
16                    action.oclAsType(OpaqueAction).output->asSequence())]
17 [output.name/] = [action.output.name/].get[output.name/]();
18     [/for]
19   [/if]
20 [/if]
21 [if actions->indexOf(action) = actions->size()]
22   [if action.output->size() = 1]
23 return [output.name/];
24   [/if]
25 [/if]
```

Listing C.7: Providing output of actions as input for other actions.

## C.3 Considering Interruptible Activity Regions

Listing C.8 contains methods needed for the implementation of interruptible activity regions as specified in Listing 6.39.

```
[for (group: InterruptibleActivityRegion | a.group)]
/* Interruptible Activity Region implementation [group.node.name/] */
private List<ActivityThread> ir[group.node.name/] =
          new LinkedList<ActivityThread>();
[/for]

[if (a.group->size() > 0)]
/** By this method, an ActivityThread is registered
 * when entering an InterruptbleActivityRegion.
 * @param t, the ActivityThread entering the region.
 * @param region, a list of ActivityThreads
 * representing the Threads inside the region.
 */
private void register(ActivityThread t, List<ActivityThread> region){
  if (! region.contains(t)){
    region.add(t);
  }
}

/** By this method, an ActivityThread is un-registered
 * when leaving an InterruptbleActivityRegion.
 * @param t, the ActivityThread leaving the region.
 * @param region, a list of ActivityThreads
 * representing the Threads inside the region.
 */
private void unregister(ActivityThread t, List<ActivityThread> region){
  if (region.contains(t)){
    region.remove(t);
  }
}

/** By this method, an ActivityThread interrupts
 * an InterruptbleActivityRegion.
 * @param region, a list of ActivityThreads
 * representing the Threads inside the region.
 * All threads contained in this list will be interrupted.
 */
private void interruptRegion(List<ActivityThread> region){
  while (!region.isEmpty()){
    ActivityThread t = region.remove(0);
  t.interrupt();
  }
}
[/if]
```

Listing C.8: Generating methods for handling interruptible activity regions.

Since at fork node, new threads are created for concurrently executing the action sequences downstream of the fork node, these threads possibly must be registered into interruptible activity regions. This is achieved by modifications of the template generating code for fork node (see Listing 6.36) as given by Listing C.9: new threads are created but not started automatically. After registering threads in interruptible activity regions and supporting them with required input, if any, threads are started.

Modifications to the generation of constructors of threads executing action sequences are given in Listing C.10.

```
// implementation of a fork node
  synchronized (context) {
    while (![node.incoming->asSequence()->first().guard.stringValue()/]
      && (
[for (gedge : ControlFlow | node.outgoing->asSequence()) separator ('||')]
    ([gedge.guard.stringValue()/])
[/for]
    ))){
      try {
        context.wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
[for (gedge : ControlFlow | node.outgoing->asSequence())]
    if ([gedge.guard.stringValue()/]){
      ActivityThread at[seqNr->indexOf(gedge.target)/] = new
          ActivityThread([seqNr->indexOf(gedge.target)/], false);
  [for (ir : InterruptibleActivityRegion | node.inInterruptibleRegion)]
      registerRegion(at[seqNr->indexOf(gedge.target)/],ir[ir.node.name/]);
  [/for]
  [for (action : Action | gedge.target)]
    [for (input : InputPin | action.input)]
      [for (objectFlow: ObjectFlow | node.incoming)]
      at[seqNr->indexOf(gedge.target)/].set[input.name/]
        ([objectFlow.source.owner.oclAsType(Action).name/]
            [objectFlow.source.name/]);
      [/for]
    [/for]
  [/for]
      at[seqNr->indexOf(gedge.target)/].execute();
    }
[/for]
  fId = 0;
}
```

Listing C.9: Generating code for a fork node considering distribution of inputs to concurrently executing sequences of actions.

```
  /**
  * This constructor creates a Thread instance and starts the execution of
  * a code sequence.
  * @param id the code sequence to be executed.
  * @param start true, if the sequence should be started immediately,
  * false, if the sequence is started later by the caller.
  */
  private ActivityThread(int id, boolean start){
    this.fId = id;
    if (start)
      this.execute;
  }

  private void execute(){
    this.start();
  }
```

Listing C.10: Generating constructors supporting to set inputs before running the constructed thread.

## C.4 Queries

This part of the appendix contains all the queries called from Mof2Text templates which we specified for the transition from models to code.

```
[query public qualifiedName(c : Classifier) : String =
  containingPackages(c).name->sep('.')->
    including('.')->including(c.name)->toString() /]

[query public containingPackages(c : Classifier) : Sequence(Package) =
  c.ancestors(Package)->reject(oclIsKindOf(Model))->reverse() /]
```

Listing C.11: Queries for obtaining a qualified name for a given classifier.

```
[query public classFileName(c : Class) : String =
  qualifiedName(c).replaceAll('\\.', '/').concat('.java') /]
```

Listing C.12: Query returning a qualified file name for a given class.

```
[query public activityFileName(a : Activity) : String =
  qualifiedName(a).replaceAll('\\.', '/').concat('.java') /]
```

Listing C.13: Query returning a qualified file name for a given activity.

```
[query public visibilityString(v : VisibilityKind) : String =
  if(v.toString() = 'package') then
    ''
  else
    v.toString().concat(' ')
  endif /]
```

Listing C.14: Query mapping UML visibilities to Java visibility modifiers.

```
[query public superClass(c : Class) : Classifier =
  c.generalization->asOrderedSet()->first().general /]
```

Listing C.15: Query returning the general classifier of a class.

```
[query public returnType(c : Class, o : Operation) : String =
  if(o.name = c.name) then
    ''
  else
    if(not o.ownedParameter->exists(
      p : Parameter | p.direction.toString() = 'return')
    or o.ownedParameter->any(
      p : Parameter | p.direction.toString() = 'return').
        type.name.oclIsInvalid()) then
      'void '
    else
      if(o.ownedComment->exists(
        n : Comment | n._body.startsWith(
          'Operation#Kind:LinkMemberClassOperation#get'))) then
        typeString(false, o.type, -1)
      else
        typeString(true, o.type, o.upper)
      endif
    endif
  endif /]

[query public typeString(c: Boolean, t : Type, upper : Integer) : String =
  if(upper > 1 or upper = -1) then
    'List<'.concat(t.name).concat('> ')
  else
    if (c) then
      typeName(t).concat(' ')
    else
      t.name.concat(' ')
    endif
  endif /]

[query public typeName(t : Type) : String =
  if(t.name = 'Integer') then
    'int'
  else
    if(t.name = 'Boolean') then
      'boolean'
    else
      t.name
    endif
  endif /]
```

Listing C.16: Query returning a return type of an operation.

```
[query public inParameters(o : Operation) : OrderedSet(Parameter) =
  o.ownedParameter->select(p : Parameter | p.direction.toString() = 'in') /]
```

Listing C.17: Query returning input parameters of an operation.

```
[query public inSuperParameters(
    c : Class, o : Operation) : OrderedSet(Parameter) =
  inParameters(o)->reject(
    p : Parameter | c.ownedAttribute->exists(
    a : Property | a.name = p.name)) /]
```

Listing C.18: Query returning a subset of a set of parameters.

```
[query public inParameterString(c : Boolean, p : Parameter) : String =
  typeString(c, p.type, p.upper).concat(p.name) /]
```

Listing C.19: Query returning a string containing type and name for a set of parameters.

```
[query public defaultStringValue(p : Property) : String =
  if(p.type.name = 'String') then
    '"'.concat(p.default).concat('"')
  else
    if(p.type.name = 'Integer' and p.default = '*') then
      'Integer.MAX_VALUE'
    else
      p.default
    endif
  endif /]
```

Listing C.20: Query translating a default value into s string representation.

```
[query public defaultReturnValue(
    c : Class, o : Operation, p : String) : String =
  if(o.name = c.name) then
    ''
  else
    if(p = 'byte ' or p = 'short ' or p = 'int ' or p = 'long '
    or p = 'float ' or p = 'double ' or p = 'char ') then
      '// TODO\nreturn 0;'
    else
      if(p = 'boolean ') then
        '// TODO\nreturn false;'
      else
        if(p <> 'void ') then
          '// TODO\nreturn null;'
        else
          '// TODO\n'
        endif
      endif
    endif
  endif /]
```

Listing C.21: Query returning a return statement with a default return value.

```
[query public paramString(c : Action) : String =
  paramString_(c, 0, '')/]

[query public paramString_(c : Action, i : Integer, s : String) : String =
  if(i < c.input->asOrderedSet()->size()-1) then
    paramString_(c, i+1, s.concat(
      c.input->asOrderedSet()->at(i).type.name).concat(
      ' ').concat(c.input->asOrderedSet()->at(i).name).concat(', '))
  else
    if(i < c.input->asOrderedSet()->size()) then
      paramString_(c, i+1, s.concat(
        c.input->asOrderedSet()->at(i).type.name).concat(
        ' ').concat(c.input->asOrderedSet()->at(i).name))
    else
      s
    endif
  endif
/]
```

Listing C.22: Query returning a string of comma separated pairs of input pin types and names.

```
[query public camelCase(str1 : String, str2 : String) : String =
  str1.concat(str2.toUpperFirst())/]
```

Listing C.23: Query building a camel case string concatenation of two given strings.

```
[query public getSequence(initial: InitialNode) : OrderedSet(Action) =
  getSequence(
    initial.outgoing->asSequence()->first().target.oclAsType(Action),
    initial.outgoing->asSequence()->first().target.oclAsType(Action)->
      asOrderedSet())/]
```

Listing C.24: Query returning a sequence of actions starting at an initial node.

```
[query public getSequence(
    from: Action, result: OrderedSet(Action)) : OrderedSet(Action) =
  if (isAction(from.outgoing.target->asSequence()->first())) then
    getSequence(
      from.outgoing.target->asSequence()->first().oclAsType(Action),
      result->append(from))
  else
    result
  endif /]
```

Listing C.25: Query returning a sequence of actions starting at a given action.

```
[query public isAction(node: ActivityNode) : Boolean =
  node.oclIsKindOf(Action) /]
```

Listing C.26: Query checking whether an activity node is an Action.

```
[query public ifLiteral(count : Integer) : String =
  if(count = 1) then
    'if'
  else
    'else if'
  endif /]
```

Listing C.27: Query returning the string *if* or *else if*.

```
[query public hasCallBehavior(a : Activity) : Boolean =
  a.node->exists(cba : ActivityNode | cba.oclIsTypeOf(CallBehaviorAction))/]
```

Listing C.28: Query returning true if a given activity contains a *CallBehaviorAction*.

```
[query public returnValues(a: Activity) : OrderedSet(Parameter) =
  a.ownedParameter->select(p : Parameter |
    ((p.direction.toString() = 'return')
    or (p.direction.toString() = 'out')
    or (p.direction.toString() = 'inout'))) /]
```

Listing C.29: Query returning output parameters of a given activity.

```
[query public inputValues(a: Activity) : OrderedSet(Parameter) =
  a.ownedParameter->select(p : Parameter |
    ((p.direction.toString() = 'in')
    or (p.direction.toString() = 'inout'))) /]
```

Listing C.30: Query returning input parameters of a given activity.

```
[query public inputActionValues(a: Action) : OrderedSet(InputPin) =
  a.input->select(p : InputPin | not p.type.name->isEmpty()) /]
```

Listing C.31: Query returning all typed input pins of a given Action.

```
[query public outputActionValues(a: Action) : OrderedSet(OutputPin) =
  a.output->select(p : OutputPin | not p.type.name->isEmpty()) /]
```

Listing C.32: Query returning typed output pins of a given action.

```
[query public isLinkMemberClass(c: Class) : Boolean =
  c.ownedComment->exists(n : Comment | n._body.startsWith(
    'Class#Kind:LinkMemberClass')) /]
```

Listing C.33: Query checking whether a class participates in an association.

```
[query public isLinkManager(c: Class) : Boolean =
  c.ownedComment->exists(n : Comment | n._body.startsWith(
    'Class#Kind:LinkManager')) /]
```

Listing C.34: Query checking whether a class is a link manager class of an association.

```
[query public isLinkMemberClassProperty(p: Property) : Boolean =
  p.ownedComment->exists(n : Comment | n._body.startsWith(
    'Property#Kind:LinkMemberClassProperty#LinkManager')) /]
```

Listing C.35: Query checking whether a property is an owned attribute referencing a link manager class.

```
[query public substringName(e : Element, s : String) : String =
  e.ownedComment->any(
    c : Comment | c._body.startsWith(s))._body.substring(s.size()+2) /]
```

Listing C.36: Query returning the rest of the string value of a comment starting with a given string.

## C.5 Snippets Processing Structures

Fig. C.1 gives a survey over the complete composition of the template snippets processing static structures as presented in Chapter 6.

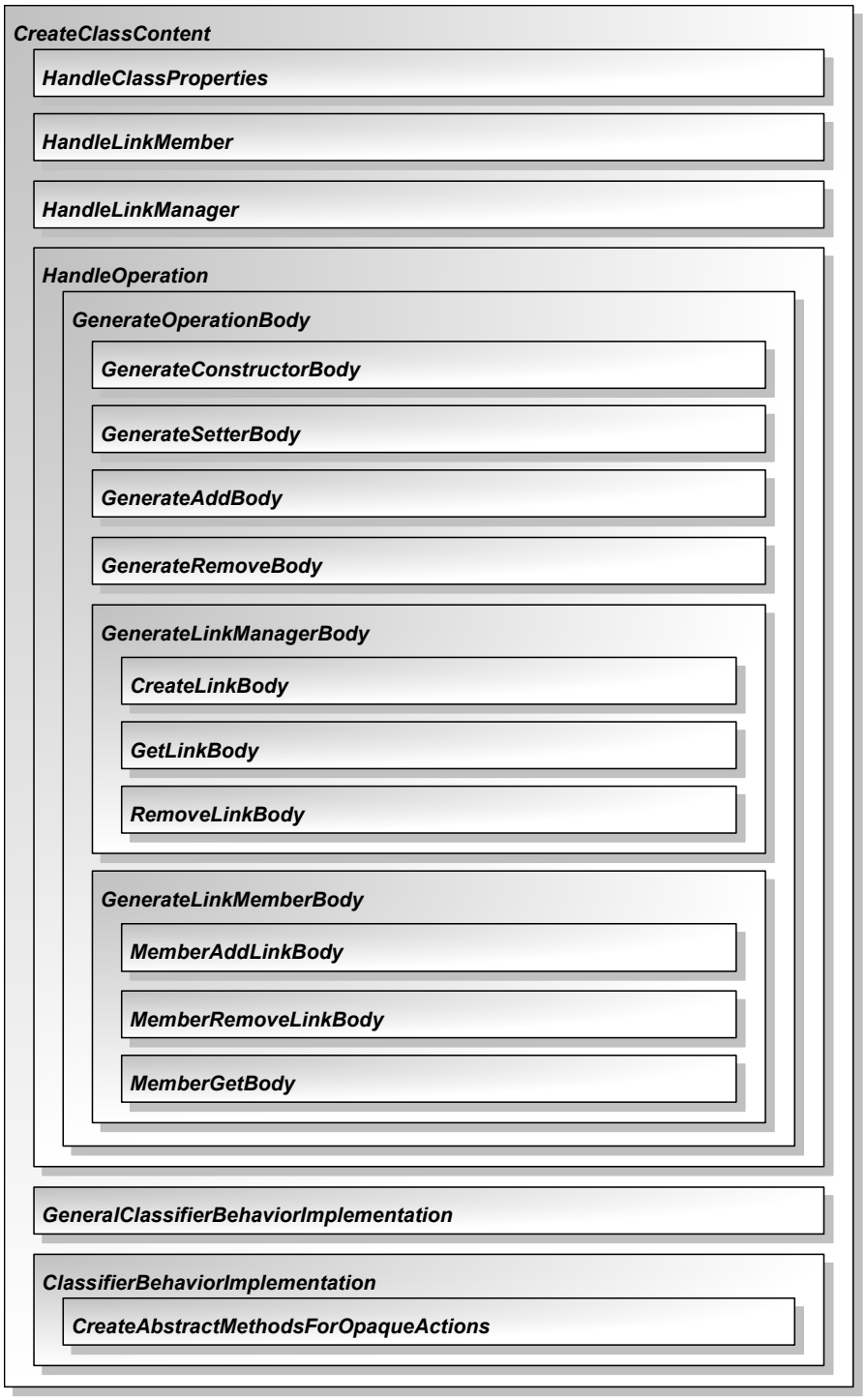



Figure C.1: Composition of snippets for class code generation.

## C.6 Snippets Processing Behavior

Fig. C.2 gives a survey over the complete composition of the template snippets processing dynamic behavior as presented in Chapter 6.

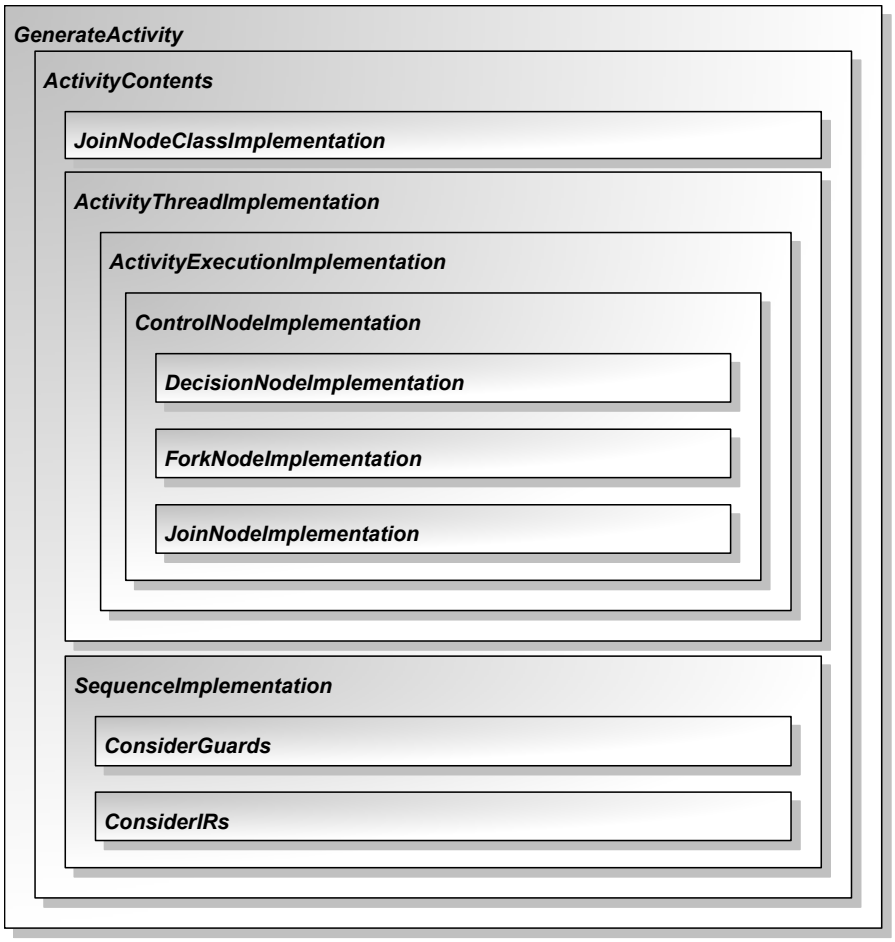



Figure C.2: Composition of snippets for code generation of dynamic behavior.

# Appendix D

# Evaluation

For evaluation purposes, we modified the code generation for actions and activity edges by inserting some debug outputs.

The implementation given in Listing D.1 is a modification to the processing of actions as in Listing 6.27. It replaces lines 17–20. In line 5, `r` is an instance of `java.util.Random`. In line 6, it is checked if the action is to be aborted due to the interruption of the executing thread. The executing thread may be interrupted because of a token passing an interrupting edge of an interruptible activity region or because a flow reached an activity final node.

```
1  ...
2  {
3    // simulate some time consuming action implementation...
4    System.out.print(" [action.name/]".toUpperCase());
5    for (int c = 0; c < r.nextInt(500)*1000; c++) {
6      if (Thread.currentThread().isInterrupted()) {
7        System.out.print(" !");
8        break;
9      }
10   }
11   System.out.print(" [action.name/]".toLowerCase());
12 }
```

Listing D.1: Modified code generation for actions in test activities.

An execution of action $A$ produces the output `A a`, if it is normally finished, or `A ! a`, if it is aborted.

If actions $A$ and $B$ are executed concurrently, valid outputs are
`A a B b`,
`A B a b`,
`A B b a`,
`B b A a`,
`B A b a`,
`B A a b`.

For each test activity, a test runner is started which creates instances of a class owning the test activity as its classifier behavior. The integer variable `i` of the context object is set and the output `Execution i=` followed by the value of `i` is written to the console.

If the value of `i` is changed in order to continue an execution that is paused due to failing guards, the output *<old value>*`->`*<new value>* , e. g. `5->10` is written.

If the execution of an activity is interrupted while waiting for a guard to evaluate to true, the output *!<source.name>*`->`*<target.name>*, e. g. `!X->Y` is written.

Note that this output not necessarily occurs when a path is terminated. It depends on the scheduler whether a waiting threat will be interrupted, or possibly be resumed normally. If being continued normally, the guard will still not evaluate to true, but since the thread is marked being discontinued, it will end anyway, but without producing the debug output.

## D.1 Testing Control Flows and Activity Termination

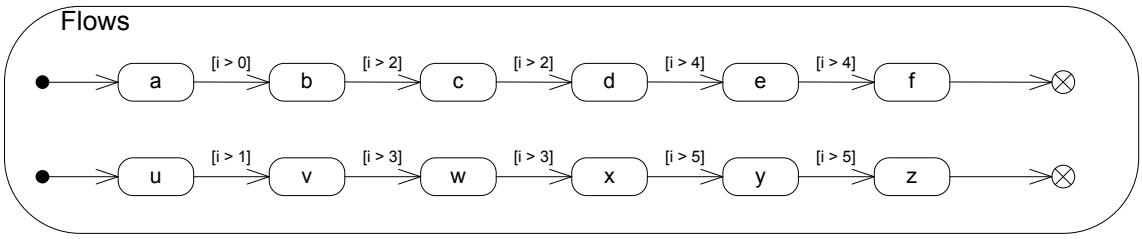



Figure D.1: Test activity for guards and flow final nodes.

```
Execution i=1
 A U a B u b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 A U u a B b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A a B u b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A u a B b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A u a B b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A u a B b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z

Execution i=2
 A U a B b u V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 A U u V v a B b 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 A U u V v a B b 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A a B b u V v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A a B u V b v 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A u V v a B b 2->3 C c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z

Execution i=3
 A U a B b C u V v c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 A U u V a B b C c D v d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 A U u V a B b C v c D d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A a B b C c D u V d v 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A a B u V b C c D v d 3->4 W w X x 4->5 E e F f 5->6 Y y Z z
 U A u V a B b C c D d v 3->4 W w X x 4->5 E e F f 5->6 Y y Z z

Execution i=4
 A U a B u V b C v W c D d w X x 4->5 E e F f 5->6 Y y Z z
 A U u V a B v W b C w X x c D d 4->5 E e F f 5->6 Y y Z z
 A U u V v W a B w X b C c D x d 4->5 E e F f 5->6 Y y Z z
 A U u V v W a B w X b C c D x d 4->5 E e F f 5->6 Y y Z z
 A U u V v W a B w X x b C c D d 4->5 E e F f 5->6 Y y Z z
 U A a B u V v W b C w X c D d x 4->5 E e F f 5->6 Y y Z z

Execution i=5
 A U a B b C u V v W c D w X d E e F f x 5->6 Y y Z z
 A U a B u V v W b C c D w X x d E e F f 5->6 Y y Z z
 A U u V a B v W w X x b C c D d E e F f 5->6 Y y Z z
 U A a B u V b C c D v W w X d E x e F f 5->6 Y y Z z
 U A a B u V v W b C w X x c D d E e F f 5->6 Y y Z z
 U A u V a B v W w X x b C c D d E e F f 5->6 Y y Z z

Execution i=6
 A U a B b C c D u V d E v W w X e F x Y y Z f z
 A U u V v W w X a B b C x Y y Z c D z d E e F f
 U A a B b C u V c D d E e F f v W w X x Y y Z z
 U A u V a B b C c D d E v W w X e F f x Y y Z z
 U A u V a B v W b C c D w X x Y d E y Z z e F f
 U A u V v W a B b C w X c D x Y d E y Z e F z f

Test completed.
```

Listing D.2: Output produced by executing test activity *Flows* of Fig. D.1

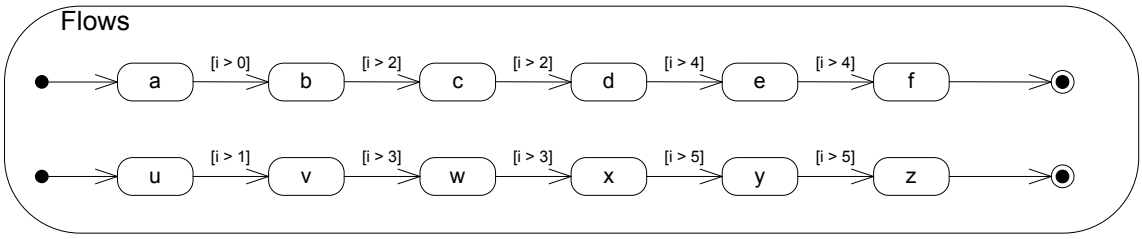



Figure D.2: Test activity for guards and activity final nodes.

```
Execution i=1
 A U a B b u 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f
 A U u a B b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f !X->Y
 A U u a B b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f
 A U u a B b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f
 U A a B u b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f
 U A a B u b 1->2 V v 2->3 C c D d 3->4 W w X x 4->5 E e F f

Execution i=2
 A U a B u V b v 2->3 C c D d 3->4 W w X x 4->5 E e F f
 A U a B u V v b 2->3 C c D d 3->4 W w X x 4->5 E e F f
 A U u V a B b v 2->3 C c D d 3->4 W w X x 4->5 E e F f
 A U u V a B v b 2->3 C c D d 3->4 W w X x 4->5 E e F f
 A U u V a B v b 2->3 C c D d 3->4 W w X x 4->5 E e F f
 U A a B u V b v 2->3 C c D d 3->4 W w X x 4->5 E e F f

Execution i=3
 A U u V a B b C v c D d 3->4 W w X x 4->5 E e F f
 A U u V a B b C v c D d 3->4 W w X x 4->5 E e F f
 A U u V v a B b C c D d 3->4 W w X x 4->5 E e F f
 U A u V v a B b C c D d 3->4 W w X x 4->5 E e F f
 U A u V a B v b C c D d 3->4 W w X x 4->5 E e F f
 U A u V a B v b C c D d 3->4 W w X x 4->5 E e F f

Execution i=4
 A U u V v W a B w X b C x c D d 4->5 E e F f !X->Y
 A U u V v W w X a B x b C c D d 4->5 E e F f
 U A a B b C u V c D v W d w X x 4->5 E e F f
 U A a B u V b C c D v W d w X x 4->5 E e F f
 U A u V a B v W w X b C x c D d 4->5 E e F f
 U A u V v W a B w X x b C c D d 4->5 E e F f

Execution i=5
 A U u V a B b C c D v W w X d E x e F f !X->Y
 A U u V a B b C v W w X c D d E e F x f
 A U u V a B v W w X b C c D x d E e F f !X->Y
 A U u V v W a B w X b C c D x d E e F f
 A U u V v W a B w X b C c D x d E e F f
 U A u V a B v W w X b C x c D d E e F f

Execution i=6
 A U a B u V b C c D v W d E e F w X f ! x
 A U a B u V b C c D v W w X d E x Y y Z e F f ! z
 A U a B u V b C v W c D d E e F w X f ! x
 A U a B u V v W b C c D w X d E x Y e F y Z f ! z
 A U u V a B v W b C w X x Y c D d E y Z e F f ! z
 U A a B u V b C c D v W d E e F w X f ! x

Test completed.
```

Listing D.3: Output produced by executing test activity *Flows* of Fig. D.2

## D.2 Testing Control Flows with Control Nodes

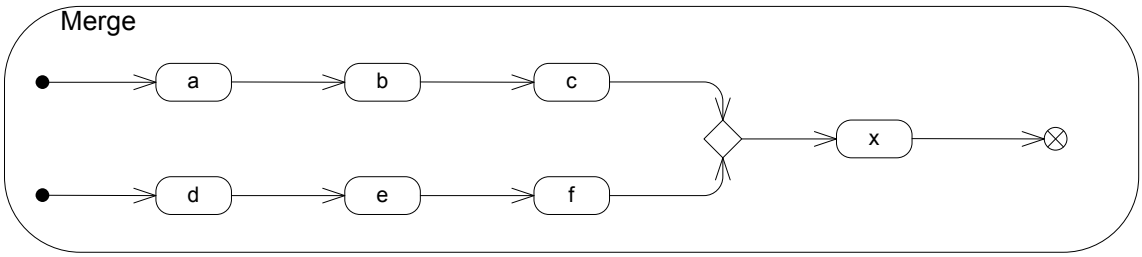



Figure D.3: Test activity for merge semantics without guards.

```
Execution i=2
 A D a B b C d E c X e F x f X x
 A D a B b C d E e F c X x f X x
 A D a B b C d E e F f X c X x x
 D A a B d E b C e F c X x f X x
 D A d E e F f X a B x b C c X x

Execution i=3
 A D a B d E b C e F c X f X x x
 A D a B d E e F f X b C x c X x
 D A a B b C d E c X e F x f X x
 D A a B d E b C e F f X x c X x
 D A a B d E b C e F c X x f X x

Execution i=5
 A D d E a B e F b C c X f X x x
 A D d E a B e F f X b C c X x x
 D A d E a B e F f X b C c X x x
 D A d E a B b C e F c X f X x x
 D A d E e F a B b C f X c X x x

Execution i=10
 A D a B d E b C e F c X f X x x
 A D d E a B b C e F c X f X x x
 A D d E e F a B f X b C x c X x
 D A a B d E b C e F c X f X x x
 D A d E a B e F f X b C c X x x

Execution i=15
 A D a B d E b C c X x e F f X x
 A D d E a B e F b C c X x f X x
 D A a B b C c X x d E e F f X x
 D A d E a B e F b C f X c X x x
 D A d E e F a B f X x b C c X x

Execution i=30
 A D a B b C c X d E e F f X x x
 A D a B d E b C c X e F x f X x
 A D d E a B e F b C f X c X x x
 D A a B b C d E c X x e F f X x
 D A d E a B b C e F c X x f X x

Test completed.
```

Listing D.4: Output produced by executing test activity *Merge* of Fig. D.3

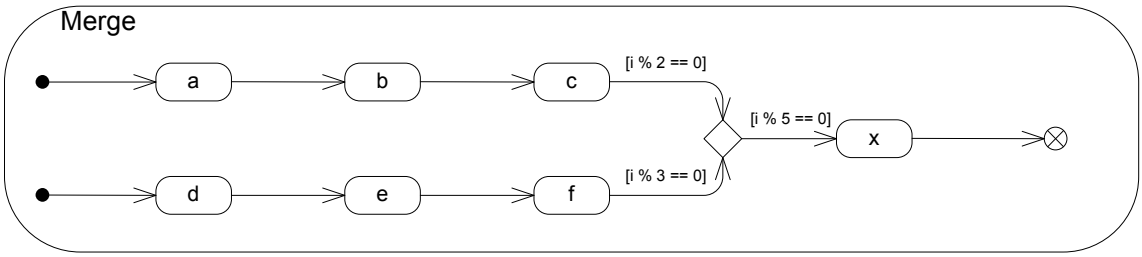



Figure D.4: Test activity for merge semantics with guarded flows.

```
Execution i=2
 A D d E a B e F b C c f
 A D d E a B e F f b C c
 D A a B d E b C c e F f
 D A d E a B e F f b C c
 D A d E e F a B b C f c

Execution i=3
 A D a B b C d E e F c f
 A D a B b C d E e F f c
 A D a B d E e F b C c f
 D A a B b C c d E e F f
 D A a B b C d E c e F f

Execution i=5
 A D a B b C d E c e F f
 A D d E a B e F b C f c
 D A a B b C c d E e F f
 D A d E a B b C e F c f
 D A d E a B e F f b C c

Execution i=10
 A D a B b C d E e F c X f x
 A D a B d E b C c X e F x f
 A D d E a B b C e F f c X x
 D A a B d E b C e F c X x f
 D A a B d E e F f b C c X x

Execution i=15
 A D a B b C d E e F f X x c
 A D d E a B e F b C f X x c
 D A a B b C d E c e F f X x
 D A a B b C d E e F c f X x
 D A a B d E e F b C c f X x

Execution i=30
 A D d E e F a B b C c X f X x x
 A D d E e F f X a B b C x c X x
 D A a B b C d E e F c X f X x x
 D A a B d E b C c X x e F f X x
 D A d E a B e F f X b C x c X x

Test completed.
```

Listing D.5: Output produced by executing test activity *Merge* of Fig. D.4

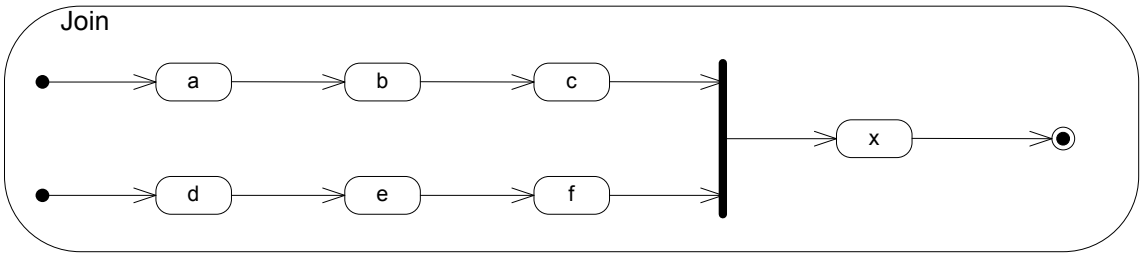



Figure D.5: Test activity for join semantics without guards.

```
Execution i=2
 A D d E a B e F b C c f X x
 D A d E a B b C e F c f X x
 D A d E a B b C e F f c X x
 D A d E a B e F b C f c X x
 D A d E e F a B f b C c X x

Execution i=3
 A D d E e F a B b C c f X x
 D A d E a B b C e F c f X x
 D A d E a B e F f b C c X x
 D A d E e F a B b C f c X x
 D A d E e F a B f b C c X x

Execution i=5
 A D d E a B e F b C f c X x
 D A d E a B b C e F f c X x
 D A a B b C d E c e F f X x
 D A a B b C d E c e F f X x
 D A d E e F a B b C f c X x

Execution i=10
 A D a B b C d E c e F f X x
 A D d E a B e F f b C c X x
 D A a B b C d E c e F f X x
 D A a B d E e F b C f c X x
 D A d E e F a B f b C c X x

Execution i=15
 A D d E a B e F b C c f X x
 D A a B d E b C c e F f X x
 D A a B d E e F f b C c X x
 D A d E a B b C e F c f X x
 D A d E a B e F f b C c X x

Execution i=30
 A D a B b C c d E e F f X x
 A D d E a B e F b C f c X x
 A D d E a B e F b C f c X x
 A D d E e F a B b C c f X x
 A D d E e F f a B b C c X x

Test completed.
```

Listing D.6: Output produced by executing test activity *Join* of Fig. D.5

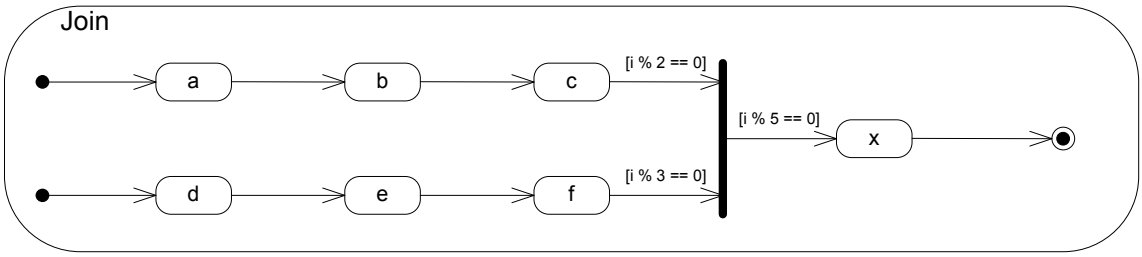



Figure D.6: Test activity for join semantics with guarded flows.

```
Execution i=2
 A D a B b C d E c e F f
 A D a B d E b C e F f c
 A D d E a B e F b C c f
 D A a B d E b C e F f c
 D A d E e F a B b C c f

Execution i=3
 A D a B d E b C e F c f
 A D d E a B b C e F c f
 A D d E e F a B f b C c
 D A a B d E e F b C f c
 D A a B d E e F b C f c

Execution i=5
 A D a B b C c d E e F f
 A D a B d E e F f b C c
 A D d E a B e F f b C c
 D A a B d E e F b C c f
 D A d E a B e F f b C c

Execution i=10
 A D a B b C d E c e F f
 A D d E a B b C e F c f
 A D d E e F a B b C c f
 D A d E a B e F b C f c
 D A d E e F a B f b C c

Execution i=15
 D A a B d E b C e F c f
 D A a B d E e F b C f c
 D A d E a B e F b C f c
 D A d E e F a B b C f c
 D A d E e F f a B b C c

Execution i=30
 D A d E a B b C e F c f X x
 D A d E a B b C e F f c X x
 D A d E a B e F b C c f X x
 D A d E a B e F b C f c X x
 D A d E a B e F b C f c X x

Test completed.
```

Listing D.7: Output produced by executing test activity *Join* of Fig. D.6

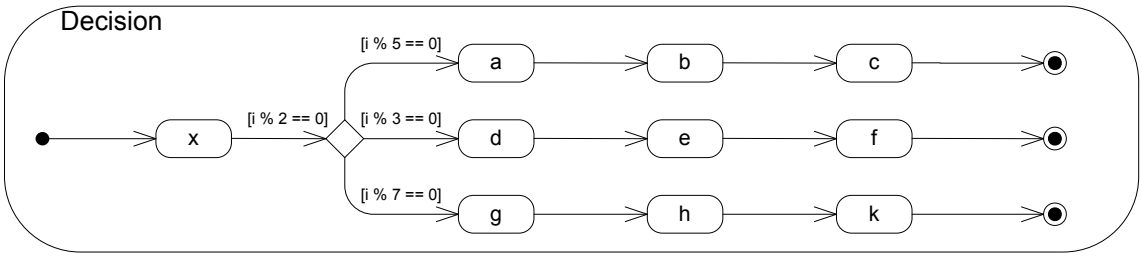



Figure D.7: Test activity for decision node.

```
Execution i=2
 X x 2->6 D d E e F f
 X x 2->10 A a B b C c
 X x 2->14 G g H h K k

Execution i=3
 X x 3->6 D d E e F f
 X x 3->10 A a B b C c
 X x 3->14 G g H h K k

Execution i=5
 X x 5->6 D d E e F f
 X x 5->10 A a B b C c
 X x 5->14 G g H h K k

Execution i=6
 X x D d E e F f

Execution i=7
 X x 7->6 D d E e F f
 X x 7->10 A a B b C c
 X x 7->14 G g H h K k

Execution i=10
 X x A a B b C c

Execution i=14
 X x G g H h K k

Execution i=30
 X x D d E e F f

Execution i=42
 X x D d E e F f

Execution i=70
 X x G g H h K k

Execution i=210
 X x D d E e F f

Test completed.
```

Listing D.8: Output produced by executing test activity *Decision* of Fig. D.7

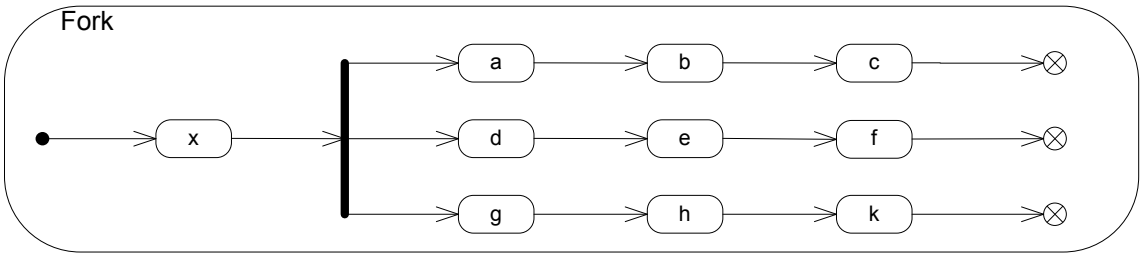



Figure D.8: Test activity for fork semantics without guards.

```
Execution i=2
 X x G A D d E a B g H e F b C f c h K k
 X x G A D a B b C g H d E e F f c h K k

Execution i=3
 X x G A D a B d E b C g H e F h K k c f
 X x G A D a B g H h K b C c k d E e F f

Execution i=5
 X x D G A d E e F a B g H f b C h K c k
 X x G D A g H d E a B h K k e F b C f c

Execution i=6
 X x A G D a B d E e F g H h K b C k f c
 X x A G D g H a B b C h K d E k c e F f

Execution i=7
 X x G A D g H d E e F a B h K f b C c k
 X x G D A d E e F a B f b C g H h K c k

Execution i=10
 X x G A D a B d E b C g H e F h K k c f
 X x G A D g H a B h K d E b C k e F c f

Execution i=14
 X x G D A d E g H a B h K e F k b C c f
 X x A G D d E g H a B h K b C e F c k f

Execution i=30
 X x A G a B b C D g H h K c d E e F k f
 X x G A D a B d E g H h K b C e F c f k

Execution i=42
 X x G D A g H d E a B e F h K k f b C c
 X x G A D a B g H d E e F f b C c h K k

Execution i=70
 X x G A D d E g H e F f a B h K k b C c
 X x G A D d E a B g H e F h K f k b C c

Execution i=210
 X x G A D d E a B g H e F b C c h K f k
 X x A G D d E a B g H b C e F h K c f k

Test completed.
```

Listing D.9: Output produced by executing test activity *Fork* of Fig. D.8

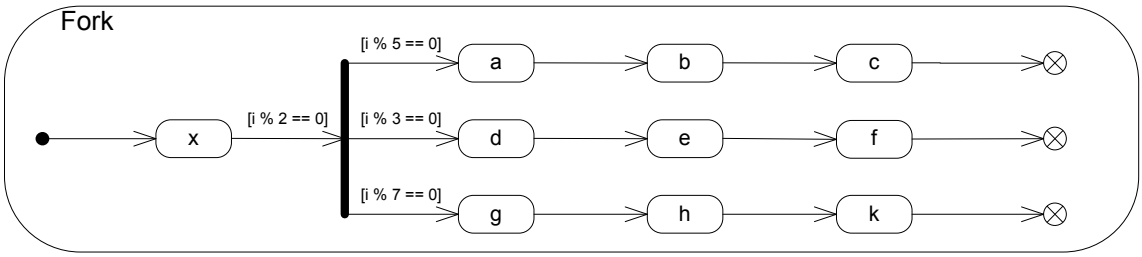



Figure D.9: Test activity for fork semantics with guarded flows.

```
1  Execution i=2 / i=3 / i=5 / i=7
2   X x 2->6 D d E e F f
3   X x 2->10 A a B b C c
4   X x 2->14 G g H h K k
5
6  Execution i=6
7   X x D d E e F f
8
9  Execution i=10
10  X x A a B b C c
11
12 Execution i=14
13  X x G g H h K k
14
15 Execution i=30
16  X x D A a B b C d E c e F f
17  X x D A a B d E e F f b C c
18  X x D A d E e F a B b C f c
19
20 Execution i=42
21  X x D G g H d E h K k e F f
22  X x D G g H d E h K k e F f
23  X x D G g H h K d E e F f k
24
25 Execution i=70
26  X x G A a B g H h K b C c k
27  X x G A g H a B h K k b C c
28  X x G A g H h K a B k b C c
29
30 Execution i=210
31  X x D G A a B d E b C g H h K e F c k f
32  X x G A D a B d E g H b C e F c f h K k
33  X x G D A g H h K k a B d E b C e F f c
34
35 Test completed.
```

Listing D.10: Output produced by executing test activity *Fork* of Fig. D.9

In Listing D.10, lines 2–4, the outputs of activity executions with different values of `i` are combined in a single output block, since the executions create identical outputs.

## D.3 Testing Interruptible Activity Regions

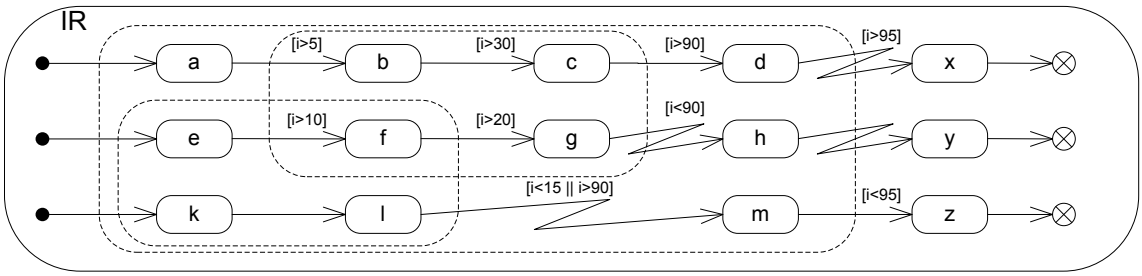



Figure D.10: Test activity for interruptible activity region.

```
Execution i=3
 A K E a e k L l M !E->F m Z z
 E K A k L e a l M !E->F m Z z
 E A K k L l !M e a m Z z
 E K A e k L a l M !E->F m Z z
 E K A a k L e l M !E->F m Z z
 E A K e a k L l M !E->F m Z z
Execution i=8
 E A K a B k L l !e M b m Z z
 E K A k L a B e b l M !E->F m Z z
 E K A k L e l M !E->F m Z a B z b
 E A K a B k L e l M !E->F b m Z z
 K A E k L e a B b l M !E->F m Z z
 A K E k L a B e b l M !E->F m Z z
Execution i=12
 E K A k L e F l !f M a B m Z b z
 A K E a B k L e F l !f M b m Z z
 E A K k L a B b l !e M m Z z
 E K A a B k L b e F l !f M m Z z
 E K A k L e F a B f l M !F->G m Z b z
 E A K a B e F f k L b l M !F->G m Z z
Execution i=25
 E K A a B e F k L f G b g H !B->C l h Y !L->M y
 E A K a B k L e F l b f G g H !B->C h Y !L->M y
 E A K a B e F k L l f G g !b H h Y !L->M y
 E A K a B e F f G g !H b k L l h Y !L->M y
 E K A k L a B b e F l f G g H !B->C h Y !L->M y
 A K E k L l e F a B f G b g H !B->C h Y !L->M y
Execution i=92
 E A K e F a B k L b C l M !f c D m Z z d
 E A K e F k L l !f M a B m Z b C z c D d
 E K A e F f G a B k L g b C l M m Z c D z d
 E A K e F f G k L a B l M g m Z b C c D z d
 E A K a B e F f G k L b C g c D l M d m Z z
 E K A a B e F k L l !f M b C c D m Z d z
Execution i=99
 E A K k L a B e F b C l !f M m c D d X !M->Z x
 E A K a B e F k L f G b C c D g l M d !m X !G->H x
 E A K k L e F a B f G g l M m b C c D d !G->H !M->Z X x
 E A K e F a B k L l !f M b C c D d !m X x
 E A K a B k L e F f G b C l M g c D d !m X !G->H x
 E A K k L l !e M a B b C m c D d X !M->Z x
Test completed.
```

Listing D.11: Output produced by executing test activity *IR* of Fig. D.10

## D.4 Testing Object Flows

For testing object flows, we designed a simple class model for the data types to be passed between actions. If types of input pins are not identical to or subtypes of the types of output pins, the generated code is not compilable.

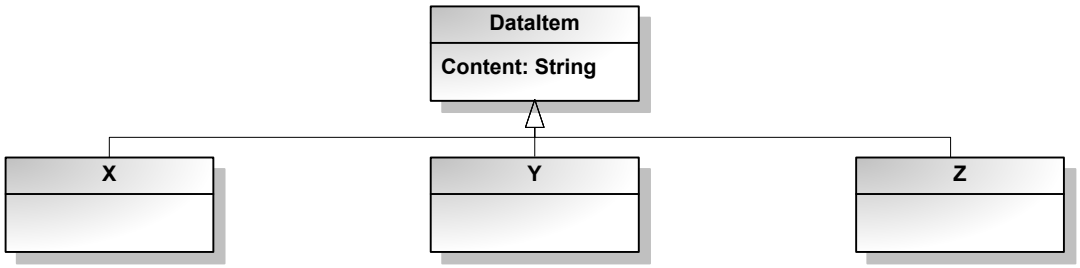



Figure D.11: Data model used for test activities with object flows.

Since the extraction of sequences of actions is currently implemented on the basis of control flow analysis only, activities must have control flows in addition to the object flows. Control flows specify the execution semantics. Hence, control nodes are applied to control flows. Object flows directly connect output pins and input pins. In Fig. D.12, Fig. D.14 and Fig. D.16, object flows are colored blue, the control flows are colored black.

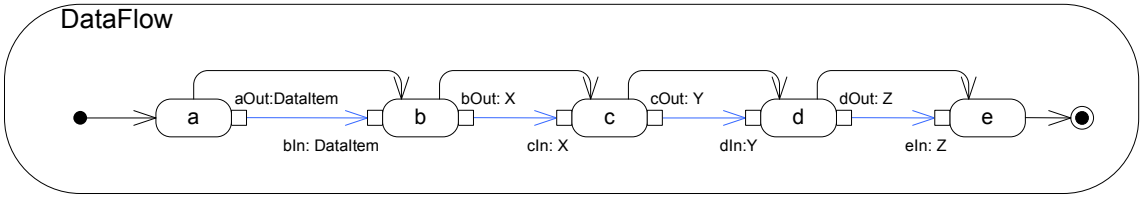



Figure D.12: Simple test activity for object flows.

For traceability of data items, actions that receive data and provide this data as input to other actions, change the content of the data object. The action creating the data sets the content to its name. Other actions append their name, as shown in Listing D.12. An action that only receives data but does not provide any output writes the content into its debug output between the indications of action execution start and end.

Listing D.13 shows the output of test activity *DataFlow* of Fig. D.12.

```
1 public X B(DataItem bIn) {
2   System.out.print(" B".toUpperCase());
3   X x = new XImpl("");
4   x.setContent(bIn.getContent() + "_B");
5   System.out.print(" B".toLowerCase());
6   return x;
7 }
```

Listing D.12: Modified action implementation for test activities with data flows.

```
1 Execution i=1
2  A a B b C c D d E A_B_C_D e
```

Listing D.13: Output produced by executing test activity *DataFlow* of Fig. D.12

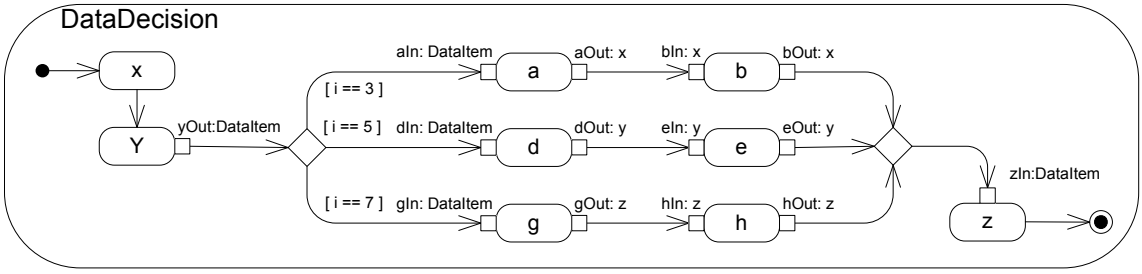



Figure D.13: Activity with object flows and decision node.

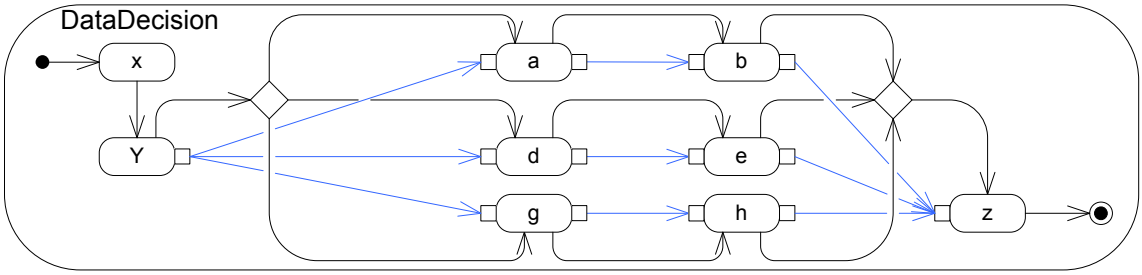



Figure D.14: Activity of Fig. D.13 with object flows and additional control flows.

```
Execution i=2
 X x Y y

Execution i=3
 X x Y y A a B b Z X_A_B_Z_ z

Execution i=4
 X x Y y

Execution i=5
 X x Y y D d E e Z X_D_E_Z_ z

Execution i=6
 X x Y y

Execution i=7
 X x Y y G g H h Z X_G_H_Z_ z

Test completed.
```

Listing D.14: Output produced by executing test activity *DataFlow* of Fig. D.13

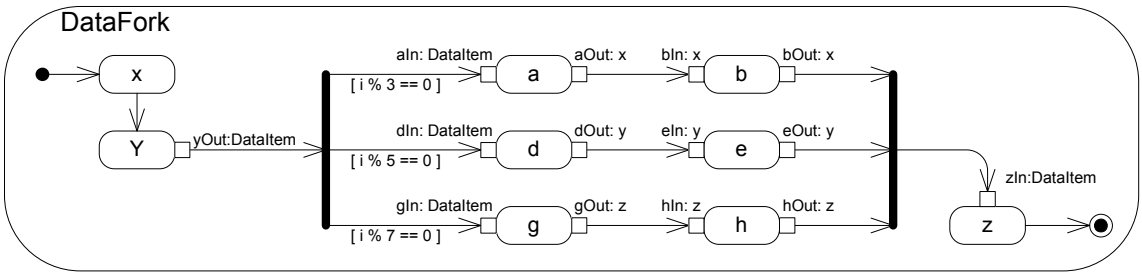



Figure D.15: Activity with object flows and fork node.

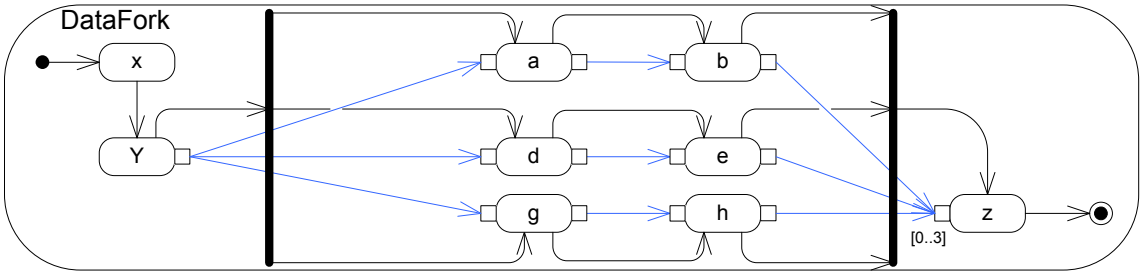



Figure D.16: Activity of Fig. D.15 with object flows and additional control flows.

```
Execution i=2
 X x Y y

Execution i=3
 X x Y y A a B b

Execution i=5
 X x Y y D d E e

Execution i=7
 X x Y y G g H h

Execution i=15
 X x Y y D d E A e a B b

Execution i=21
 X x Y y G g A a H h B b

Execution i=35
 X x Y y G g D H h d E e

Execution i=105
 X x Y y D A a B b d E e G g H h Z X_A_B_:X_D_E_:X_G_H_:Z z

Test completed.
```

Listing D.15: Output produced by executing test activity *DataFork* of Fig. D.16

# Bibliography


[1] D. Akehurst, G. Howells, and K. McDonald-Maier. Implementing associations: UML 2.0 to Java 5. In *Software and Systems Modeling (SoSyM), Volume 6, Number 1, pp. 3-35(33)*. Springer Verlag, 2007.

[2] P. Avgeriou, N. Guelfi, and N. Medvidovic. Software architecture description and uml. In N. Jardim Nunes, B. Selic, A. Rodrigues da Silva, and A. Toval Alvarez, editors, *UML Modeling Languages and Applications*, volume 3297 of *Lecture Notes in Computer Science*, pages 23–32. Springer Berlin / Heidelberg, 2005.

[3] O. Badreddin, A. Forward, and T. C. Lethbridge. Improving code generation for associations: Enforcing multiplicity constraints and ensuring referential integrity. In R. Lee, editor, *Software Engineering Research, Management and Applications*, pages 129–149, Heidelberg, 2014. Springer International Publishing.

[4] H. G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *SIGPLAN OOPS Mess.*, 4(4):2–27, Oct. 1993.

[5] S. Balzer. *RUMER: A Programming Language and Modular Verification Technique Based on Relationships*. PhD thesis, ETH Zurich, 2011.

[6] C. Bensoussan, M. Schöttle, and J. Kienzle. Associations in mde: A concern-oriented, reusable solution. In A. Wąsowski and H. Lönn, editors, *Modelling Foundations and Applications*, pages 121–137, Cham, 2016. Springer International Publishing.

[7] A. Bhattacharjee and R. Shyamasundar. Validated Code Generation for Activity Diagrams. In *Second International Conference, ICDCIT 2005*, volume 3816 of *Lecture Notes in Computer Science*, pages 508–521. Springer, 2005.

[8] B. Biafore. *Visio 2003 Bible*. Bible Series. Wiley, 2004.

[9] G. Bierman and A. Wren. First-class relationships in an object-oriented language. In *19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 262–286. Springer-Verlag, 2005.

[10] C. Bock. UML 2 Activity and Action Models Part 2: Actions. *Journal of Object Technology*, 2(5):41–56, 2003.

[11] M. Broy and M. Cengarle. Uml formal semantics: lessons learned. *Software and Systems Modeling*, 10:441–446, 2011.

[12] M. Broy, M. L. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2nd uml 2 semantics symposium: formal semantics for uml. In *Proceedings of the 2006 international conference on Models in software engineering*, MoDELS'06, pages 318–323, Berlin, Heidelberg, 2006. Springer-Verlag.

[13] A. Bulach. Untersuchungen zur Laufzeitverbesserung des ActiveCharts-Interpreters. Diploma thesis, Ulm University, January 2008.

[14] S. Burmester, H. Giese, J. Niere, M. Tichy, J. P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level: the fujaba approach. *International Journal on Software Tools for Technology Transfer*, 6(3):203–218, Aug 2004.

[15] H. Chavez, W. Shen, R. France, and B. Mechling. An approach to testing java implementation against its uml class model. In A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, editors, *Model-Driven Engineering Languages and Systems*, volume 8107 of *Lecture Notes in Computer Science*, pages 220–236. Springer Berlin Heidelberg, 2013.

[16] P. P.-S. Chen. The Entity-Relationship Model
Toward a Unified View of Data. *ACM Trans. Database Syst.*, 1(1):9–36, Mar. 1976.

[17] C. M. Chiao, V. Kuenzle, and M. Reichert. A tool for supporting object-aware processes. In *Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW), 2014 IEEE 18th International*, pages 410–413, Sept 2014.

[18] M. L. Crane. *Slicing UML's Three-layer Architecture: A Semantic Foundation for Behavioural Specification.* PhD thesis, Queen's University, Kingston, Ontario, Canada, January 2009.

[19] M. L. Crane, J. Dingel, and Z. Diskin. Class Diagramms: Abstract Syntax and Mapping to System Model, (Draft - Version 1.7), 2006. School of Computing, Queen's University, Kingston, Ontario, Canada.

[20] David J. Pearce and James Noble. Relationship aspect patterns. In *11th European Conference on Pattern Languages of Programs (EuroPLoP'06)*, pages 531–546, 2006.

[21] M. Derakhshanmanesh, J. Ebert, T. Iguchi, and G. Engels. Model-integrating software components. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 386–402. Springer International Publishing, 2014.

[22] Z. Diskin and J. Dingel. Mappings, Maps and Tables: Towards Formal Semantics for Associations in UML2. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *MODELS*, volume 4199 of *Lecture Notes in Computer Science*, pages 230–244. Springer, 2006.

[23] The Eclipse Modeling Framework (EMF) Overview. https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.emf.doc%2Freferences%2Foverview%2FEMF.html, 2005-06-16.

[24] Eclipse Foundation. ATL Transformation Language, 2012. http://www.eclipse.org/atl/.

[25] H. Eichelberger, Y. Eldogan, and K. Schmid. *A Comprehensive Analysis of UML Tools, their Capabilities and their Compliance.* Software Systems Engineering, Institut für Informatik, Universität Hildesheim, 2009.

[26] T. Frühwirth. *Constraint Handling Rules.* Cambridge University Press, 2009.

[27] T. Frühwirth and F. Raiser, editors. *Constraint Handling Rules: Compilation, Execution, and Analysis.* Books On Demand, 2011.

[28] Fujaba Associations Specification.
http://web.archive.org/web/20101011174023/http://www.se.eecs.uni-kassel.de:80/~fujabawiki/index.php/Fujaba_Associations_Specification.

[29] E. Gamma, R. Helm, P. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 2003.

[30] G. Génova, J. Llorens, and C. R. del Castillo. Mapping UML Associations into Java Code. In *Journal of Object Technology, vol.2, no. 5, pp. 135-162*, 2003.

[31] G. Génova, J. Lloréns, and P. Martínez. Semantics of the Minimum Multiplicity in Ternary Associations in UML. In *UML*, pages 329–341, 2001.

[32] D. Gessenharter. Visualisierung der Simulation von graphischen Prototypen als Möglichkeit des interaktiven Debuggings von UML 2.0 Aktivitätasdiagrammen. Diploma thesis, Ulm University, Institute of Sofware Engineering and Compiler Construction, 2005.

[33] D. Gessenharter. Mapping the UML2 Semantics of Associations to a Java Code Generation Model. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Model Driven Engineering Languages and Systems*, volume 5301 of *Lecture Notes in Computer Science*, pages 813–827. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-87875-9_56.

[34] D. Gessenharter. Implementing UML Associations in Java: A Slim Code Pattern for a Complex Modeling Concept. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, RAOOL '09, pages 17–24, New York, NY, USA, 2009. ACM.

[35] D. Gessenharter. Extending The UML Semantics For A Better Support of Model-Driven Software Development. In *Software Engineering Research and Practice*, pages 45–51. CSREA Press, U. S. A., 2010.

[36] D. Gessenharter. UML Activities at Runtime. In J. Trujillo, G. Dobbie, H. Kangassalo, S. Hartmann, M. Kirchberg, M. Rossi, I. Reinhartz-Berger, E. Zimányi, and F. Frasincar, editors, *Advances in Conceptual Modeling — Applications and Challenges*, volume 6413 of *Lecture Notes in Computer Science*, pages 275–284. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-16385-2_34.

[37] D. Gessenharter. Model-Driven Software Development with ActiveCharts — A Case Study. Ulmer Informatik-Berichte 2011-08, Ulm University, December 2011.

[38] D. Gessenharter, A.-M. Merten, A. Raschke, and N. F. Porta. Experiences on using software experiments in the validation of industrial research questions. In J. Cuadrado-Gallego, R. Braungarten, R. Dumke, and A. Abran, editors, *Software Process and Product Measurement*, volume 4895 of *Lecture Notes in Computer Science*, pages 86–94. Springer Berlin Heidelberg, 2008.

[39] D. Gessenharter and M. Rauscher. Code Generation for UML 2 Activity Diagrams — Towards a Comprehensive Model-Driven Development Approach. In R. B. France, J. M. Kuester, B. Bordbar, and R. F. Paige, editors, *ECMFA'11: Proceedings of the 7th European Conference on Modelling Foundations and Applications*, pages 205–220, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[40] M. Gogolla. Employing the object constraint language in model-based engineering. In P. Van Gorp, T. Ritter, and L. Rose, editors, *Modelling Foundations and Applications*, volume 7949 of *Lecture Notes in Computer Science*, pages 1–2. Springer Berlin Heidelberg, 2013.

[41] V. Gruhn, D. Pieper, and C. Röttgers. *MDA: Effektives Software-Engineering mit UML2 und Eclipse*. Springer-Verlag, $1^{st}$ edition, 2006.

[42] Y. Gurevich. Specification and validation methods. chapter Evolving algebras 1993: Lipari guide, pages 9–36. Oxford University Press, Inc., New York, NY, USA, 1995.

[43] B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451 –461, 2006.

[44] D. Harel. From Play-In Scenarios to Code: An Achievable Dream. *Computer*, 34:53–60, 2001.

[45] C. Kecher. *UML 2.0 - Das umfassende Handbuch*. Galileo Computing, Bonn, 2006.

[46] E. Kindler and R. Wagner. Triple graph grammars: Concepts, extensions, implementations, and application scenarios. Technical report, Technical Report tr-ri-07-284, University of Paderborn, 2007.

[47] A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[48] N. Koch and A. Kraus. The Expressive Power of UML-based Web Engineering, 2002.

[49] N. Koch, G. Zhang, and H. Baumeister. UML-Based Web Engineering: An Approach Based on Standards. *Web Engineering: Modelling and Implementing Web Applications*, pages 157–191, 2008.

[50] J. Kohlmeyer. *Eine formale Semantik für die Verknüpfung von Verhaltensbeschreibungen in der UML 2*. PhD thesis, Ulm University, 2009.

[51] J. Kohlmeyer and W. Guttmann. Unifying the Semantics of UML 2 State, Activity and Interaction Diagrams. In *Ershov Memorial Conference*, pages 206–217, 2009.

[52] V. Künzle. *Object-Aware Process Management*. PhD thesis, Ulm University, Institute of Databases and Information Systems, 2013.

[53] V. Künzle and M. Reichert. A Modeling Paradigm for Integrating Processes and Data at the Micro Level. In T. Halpin, S. Nurcan, J. Krogstie, P. Soffer, E. Proper, R. Schmidt, and I. Bider, editors, *Enterprise, Business-Process and Information Systems Modeling*, volume 81 of *Lecture Notes in Business Information Processing*, pages 201–215. Springer Berlin Heidelberg, 2011.

[54] Y. Laurent, R. Bendraou, S. Baarir, and M.-P. Gervais. Alloy4spv : A formal framework for software process verification. In J. Cabot and J. Rubin, editors, *Modelling Foundations and Applications*, volume 8569 of *Lecture Notes in Computer Science*, pages 83–100. Springer International Publishing, 2014.

[55] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson. Assessing the state-of-practice of model-based engineering in the embedded systems domain. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 166–182. Springer International Publishing, 2014.

[56] LMU — Ludwig-Maximilians-Universität München, Institute for Informatics Programming and Software Engineering. *UWE Examples*, 12 2009. http://uwe.pst.ifi.lmu.de/exampleAddressBookWithContentUpdates.html.

[57] J. Lowy. Create elegant code with anonymous methods, iterators, and partial classes. *The Microsoft Journal for Developers*, May 2004. http://msdn.microsoft.com/en-us/magazine/cc163970.aspx.

[58] D. Markulin, K. Musa, and M. Kunstic. Using uml 2 activity diagram for visual business management modeling. In *MIPRO, 2011 Proceedings of the 34th International Convention*, pages 356 –361, May 2011.

[59] S. J. Mellor and M. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. Foreword By-Jacoboson, Ivar.

[60] D. Milićev. On the Semantics of Associations and Association Ends in UML. *IEEE Transactions on Software Engineering*, 33:238–251, 2007.

[61] G. Mussbacher, D. Amyot, R. Breu, J.-M. Bruel, B. Cheng, P. Collet, B. Combemale, R. France, R. Heldal, J. Hill, J. Kienzle, M. Schöttle, F. Steimann, D. Stikkolorum, and J. Whittle. The relevance of model-driven engineering thirty years from now. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 183–200. Springer International Publishing, 2014.

[62] S. Nelson, J. Noble, and D. J. Pearce. Implementing First-Class Relationships in Java. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL)*, 2008.

[63] S. Nelson, D. J. Pearce, and J. Noble. Implementing relationships using affinity. In *Proceedings of the Workshop on Relationships and Associations in Object-Oriented Languages*, RAOOL '09, pages 5–8, New York, NY, USA, 2009. ACM.

[64] U. Nickel, J. Niere, and A. Zündorf. The fujaba environment. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 742–745, New York, NY, USA, 2000. ACM.

[65] J. Noble. Basic relationship patterns. In *EuroPLOP Proceedings*. Addison-Wesley, 1997.

[66] J. Noble and J. Grundy. Explicit relationships in object-oriented development. In *Proceedings of TOOLS*, volume 18, pages 211–225, 1995.

[67] Object Management Group. Introduction to OMG's Unified Modeling Language (UML), July 2005. http://www.omg.org/gettingstarted/what_is_uml.htm.

[68] Object Management Group. MOF Model to Text Transformation Language, v1.0, 2011. OMG Document Number: formal/2008-01-16.

[69] Object Management Group. Unified Modeling Language (OMG UML), Superstructure Version 2.4.1, 2011. OMG Document Number: formal/2011-08-06.

[70] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification 1.3, 2016. OMG Document Number: formal/2016-06-03.

[71] Object Management Group. Action Language for Foundational UML (Alf), Concrete Syntax for a UML Action Language, Version 1.1, 2017. OMG Document Number: formal/2017-07-04.

[72] Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), v1.3, 2017. OMG Document Number: formal/2017-07-02.

[73] Object Management Group. Unified Modeling Language (OMG UML), Superstructure Version 2.5.1, 2017. OMG Document Number: formal/2017-12-06.

[74] B. Ogunyomi, L. Rose, and D. Kolovos. Property access traces for source incremental model-to-text transformation. In G. Taentzer and F. Bordeleau, editors, *Modelling Foundations and Applications*, volume 9153 of *Lecture Notes in Computer Science*, pages 187–202. Springer International Publishing, 2015.

[75] K. Østerbye. Design of a Class Library for Association Relationships. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD'07, pages 67–75, New York, NY, USA, 2007. ACM.

[76] D. J. Pearce and J. Noble. Relationship aspects. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 75–86, New York, NY, USA, 2006. ACM.

[77] M. Rauscher. Code Generation for Distributed Activities. Diploma thesis, Ulm University, Institute of Sofware Engineering and Compiler Construction, 2011.

[78] G. Reggio, M. Leotta, and F. Ricca. Who knows/uses what of the uml: A personal opinion survey. In J. Dingel, W. Schulte, I. Ramos, S. Abrahão, and E. Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 149–165. Springer International Publishing, 2014.

[79] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. *SIGPLAN Not.*, 22(12):466–481, Dec. 1987.

[80] C. Rupp, J. Hahn, B. Zengler, and S. Queins. *UML 2 glasklar*. Hanser, München Wien, 2007.

[81] S. Safdar, M. Iqbal, and M. Khan. Empirical evaluation of uml modeling tools–a controlled experiment. In G. Taentzer and F. Bordeleau, editors, *Modelling Foundations and Applications*, volume 9153 of *Lecture Notes in Computer Science*, pages 33–44. Springer International Publishing, 2015.

[82] S. Sarstedt. Model-Driven Development with ActiveCharts - Tutorial. Technical report, Ulm University, Institute of Software Engineering and Complier Construction, 2006.

[83] S. Sarstedt. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. PhD thesis, Ulm University, 2006.

[84] S. Sarstedt, D. Gessenharter, J. Kohlmeyer, A. Raschke, and M. Schneiderhan. ActiveChartsIDE: An Integrated Software Development Environment comprising a Component for Simulating UML 2 Activity Charts. In A. E. C. B. J. M. Feliz-Teixeira, editor, *Proceedings of the 2005 European Simulation and Modelling Conference (ESM'05)*, pages 66–73, October 2005.

[85] S. Sarstedt, J. Kohlmeyer, A. Raschke, and M. Schneiderhan. A New Approach to Combine Models and Code in Model Driven Development. In *International Conference on Software Engineering Research and Practice, International Workshop on Applications of UML/MDA to Software Systems*, pages 396–402, 2005.

[86] S. Sarstedt, J. Kohlmeyer, A. Raschke, and M. Schneiderhan. Targeting System Evolution by Explicit Modeling of Control Flows Using UML 2 Activity Charts. In *Proceedings of the International Conference on Programming Languages and Compilers (PLC '05), Technical Session on Support for Unanticipated Software Evolution*, pages 237–244, 2005.

[87] T. Schlecht. Formalisierung der Transformationen von UML2 Klassen- und Verhaltensmodellen in Java-Code. Diploma thesis, Ulm University, Institute of Sofware Engineering and Compiler Construction, 2012.

[88] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin / Heidelberg, 1995.

[89] B. Selic. Model-Driven Development: Its Essence and Opportunities. In *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, page 7 pp., April 2006.

[90] B. Selic. A Short Catalogue of Abstraction Patterns for Model-Based Software Engineering. *Int. J. Software and Informatics*, 5(1-2):313–334, 2011.

[91] B. Selic. The Theory and Practice of Modeling Language Design for Model-Based Software Engineering—A Personal Perspective. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 290–321. Springer Berlin / Heidelberg, 2011.

[92] S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *Software, IEEE*, 20(5):42 – 45, sept.-oct. 2003.

[93] M. Shroff and R. France. Towards a formalization of UML class structures in Z. In *Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International*, pages 646 –651, aug 1997.

[94] S. Sulistyo and A. Prinz. Recursive Modeling for Completed Code Generation. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, BM-MDA '09, pages 6:1–6:7, New York, NY, USA, 2009. ACM.

[95] University of Ottawa. Umple Language, Decembre 2017. https://cruise.eecs.uottawa.ca/umple/.

[96] M. Usman and A. Nadeem. Automatic Generation of Java Code from UML Diagrams using UJECTOR. *International Journal of Software Engineering and Its Applications*, 3(2):21–37, April 2009.

[97] T. van Tricht. Konzeption und Realisierung eines Persistenz-Frameworks für ActiveCharts. Master's thesis, Ulm University, 2007.

[98] M. Walker and N. Eaton. *Microsoft Office Visio 2003 Inside Out*. Inside Out. Microsoft Press, 2003.

[99] C. Waniek. Konzepte zur Codeerzeugung aus UML2.0 Klassendiagrammen unter spezieller Berücksichtigung semantischer Aspekte. Diploma thesis, Ulm University, Institute of Sofware Engineering and Compiler Construction, 2007.

[100] A. I. Wasserman. Toward a discipline of software engineering. *Software, IEEE*, 13(6):23 –31, Nov. 1996.

[101] Wikipedia, SSLV ascent, Ray Gomez, Retrieved 29 June 2018. https://de.wikipedia.org/wiki/Datei:SSLV_ascent.jpg.

[102] World Wide Web Consortium (W3C). Xsl transformations (xslt), 2007. Version 2.0, W3C Recommendation (2007).

[103] A. Wren. Relationships for object-oriented programming languages. Technical report, University of Cambridge, Computer Laboratory, 2007.

[104] Wright Brothers Aeroplane Company, Retrieved 12 March 2018. http://www.wright-brothers.org/Information_Desk/Help_with_Homework/Wright_Plans/Wright_Plans.htm

# Software.

[105] Altova Inc., ALTOVA umodel 2018 sp1, Enterprise Edition, 2018. https://shop.altova.com/UModel.

[106] Gentleware AG, Apollo for Eclipse, Version 3.0, 2008. http://www.gentleware.com/apollo.html.

[107] ARTiSAN Studio, (Version 6.1.21),, 2006. http://www.artisansw.com/.

[108] Change Vision Inc., Astah Professional 7.2.0, Model Version:37, 2018. http://astah.net/editions/professional.

[109] XTUML.org, BridgePoint UML Suite 5.5.0, 2016. https://xtuml.org.

[110] Sparx Systems, Enterprise Architect,Version 13.5.1352 (Build: 1352), 2017. http://www.sparxsystems.com.

[111] Fujaba Development Group, Fujaba Tool Suite 4.3.2, 2007. http://www.fujaba.de/.

[112] No Magic, Inc., MagicDraw Professional Java 19.0 beta, 2017. http://www.magicdraw.com.

[113] Micro Focus IP Development Limited, Micro Focus Together 13, 2018. https://www.microfocus.com/products/requirements-management/together/.

[114] Papyrus, Open Source Tool, Version 3.2.0.201712060842, 2011. http://www.papyrusuml.org.

[115] Gentleware AG, Poseidon for UML Professional Edition 8.0.0, 2010. http://www.gentleware.com/new-poseidon-for-uml-8-0.html.

[116] Telelogic, Rhapsody 7.2, 2008. http://www.telelogic.com/products/rhapsody/.

[117] IBM Corp., IBM Rational Software Architect, Version 7.0.0, 2008. http://www-306.ibm.com/software/awdtools/architect/swarchitect/index.html.

[118] Visual Paradigm for UML, Enterprise Edition, Version 14.2, 2017. http://www.visual-paradigm.com/.