



Date: August 2022



# Kernel Modeling Language (KerML)

*Version 1.0*

*Release 2022-07*

**Submitted in partial response to Systems Modeling Language (SysML®) v2 RFP (ad/2017-12-02) by:**

|                         |                               |
|-------------------------|-------------------------------|
| 88Solutions Corporation | Lockheed Martin Corporation   |
| Dassault Systèmes       | MITRE                         |
| GfSE e.V.               | Model Driven Solutions, Inc.  |
| IBM                     | PTC                           |
| INCOSE                  | Simula Research Laboratory AS |
| Intercax LLC            | Thematix Partners             |

Copyright © 2019-2022, 88Solutions Corporation  
Copyright © 2019-2022, Airbus  
Copyright © 2019-2022, Aras Corporation  
Copyright © 2019-2022, Association of Universities for Research in Astronomy (AURA)  
Copyright © 2019-2022, BigLever Software  
Copyright © 2019-2022, Boeing  
Copyright © 2021-2022, Commissariat à l'énergie atomique et aux énergies alternatives (CEA)  
Copyright © 2019-2022, Contact Software GmbH  
Copyright © 2019-2022, Dassault Systèmes (No Magic)  
Copyright © 2019-2022, DSC Corporation  
Copyright © 2020-2022, DEKonsult  
Copyright © 2020-2022, Delligatti Associates, LLC  
Copyright © 2019-2022, The Charles Stark Draper Laboratory, Inc.  
Copyright © 2020-2022, ESTACA  
Copyright © 2022, Galois, Inc.  
Copyright © 2019-2022, GfSE e.V.  
Copyright © 2019-2022, George Mason University  
Copyright © 2019-2022, IBM  
Copyright © 2019-2022, Idaho National Laboratory  
Copyright © 2019-2022, INCOSE  
Copyright © 2019-2022, Intercax LLC  
Copyright © 2019-2022, Jet Propulsion Laboratory (California Institute of Technology)  
Copyright © 2019-2022, Kenntnis LLC  
Copyright © 2020-2022, Kungliga Tekniska högskolan (KTH)  
Copyright © 2019-2022, LightStreet Consulting LLC  
Copyright © 2019-2022, Lockheed Martin Corporation  
Copyright © 2019-2022, Maplesoft  
Copyright © 2021-2022, MID GmbH  
Copyright © 2020-2022, MITRE  
Copyright © 2019-2022, Model Alchemy Consulting  
Copyright © 2019-2022, Model Driven Solutions, Inc.  
Copyright © 2019-2022, Model Foundry Pty. Ltd.  
Copyright © 2019-2022, On-Line Application Research Corporation (OAC)  
Copyright © 2019-2022, oose Innovative Informatik eG  
Copyright © 2019-2022, Østfold University College  
Copyright © 2019-2022, PTC  
Copyright © 2020-2022, Qualtech Systems, Inc.  
Copyright © 2019-2022, SAF Consulting  
Copyright © 2019-2022, Simula Research Laboratory AS  
Copyright © 2019-2022, System Strategy, Inc.  
Copyright © 2019-2022, Thematix Partners, LLC  
Copyright © 2019-2022, Tom Sawyer  
Copyright © 2022, Tucson Embedded Systems, Inc.  
Copyright © 2019-2022, Universidad de Cantabria  
Copyright © 2019-2022, University of Alabama in Huntsville  
Copyright © 2019-2022, University of Detroit Mercy  
Copyright © 2019-2022, University of Kaiserslautern  
Copyright © 2020-2022, Willert Software Tools GmbH (SodiusWillert)

Each of the entities listed above: (i) grants to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version, and (ii) grants to each member of the OMG a nonexclusive, royalty-free, paid up,

worldwide license to make up to fifty (50) copies of this document for internal review purposes only and not for distribution, and (iii) has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used any OMG specification that may be based hereon or having conformed any computer software to such specification.



# Table of Contents

|   |    |
|---|----|
| 0 Submission Introduction .....                   | 1  |
| 0.1 Submission Overview .....                     | 1  |
| 0.2 Submission Submitters .....                   | 1  |
| 0.3 Submission - Issues to be discussed .....     | 1  |
| 0.4 Language Requirement Tables .....             | 2  |
| 1 Scope .....                                     | 3  |
| 2 Conformance .....                               | 5  |
| 3 Normative References .....                      | 7  |
| 4 Terms and Definitions .....                     | 9  |
| 5 Symbols .....                                   | 11 |
| 6 Introduction .....                              | 13 |
| 6.1 Language Architecture .....                   | 13 |
| 6.2 Document Organization .....                   | 13 |
| 6.3 Acknowledgements .....                        | 14 |
| 7 Language Description .....                      | 17 |
| 7.1 Language Description Overview .....           | 17 |
| 7.2 Root .....                                    | 17 |
| 7.2.1 Root Overview .....                         | 17 |
| 7.2.2 Elements and Relationships .....            | 17 |
| 7.2.2.1 Elements and Relationships Overview ..... | 17 |
| 7.2.2.2 Elements .....                            | 18 |
| 7.2.2.3 Relationships .....                       | 19 |
| 7.2.3 Annotations .....                           | 20 |
| 7.2.3.1 Annotations Overview .....                | 21 |
| 7.2.3.2 Comments and Documentation .....          | 21 |
| 7.2.3.3 Textual Representations .....             | 22 |
| 7.2.4 Namespaces .....                            | 23 |
| 7.2.4.1 Namespaces Overview .....                 | 23 |
| 7.2.4.2 Namespace Declaration .....               | 24 |
| 7.2.4.3 Root Namespaces .....                     | 25 |
| 7.2.4.4 Imports .....                             | 25 |
| 7.3 Core .....                                    | 28 |
| 7.3.1 Core Overview .....                         | 28 |
| 7.3.2 Types .....                                 | 28 |
| 7.3.2.1 Types Overview .....                      | 28 |
| 7.3.2.2 Type Declaration .....                    | 29 |
| 7.3.2.3 Specialization .....                      | 30 |
| 7.3.2.4 Conjugation .....                         | 31 |
| 7.3.2.5 Disjoining .....                          | 32 |
| 7.3.2.6 Feature Membership .....                  | 32 |
| 7.3.3 Classifiers .....                           | 33 |
| 7.3.3.1 Classifiers Overview .....                | 33 |
| 7.3.3.2 Classifier Declaration .....              | 33 |
| 7.3.3.3 Subclassification .....                   | 33 |
| 7.3.4 Features .....                              | 34 |
| 7.3.4.1 Features Overview .....                   | 34 |
| 7.3.4.2 Feature Declaration .....                 | 34 |
| 7.3.4.3 Feature Typing .....                      | 36 |
| 7.3.4.4 Subsetting .....                          | 37 |
| 7.3.4.5 Redefinition .....                        | 38 |
| 7.3.4.6 Feature Chaining .....                    | 40 |

|  |    |
|--|----|
| 7.3.4.7 Feature Inverting .....                            | 40 |
| 7.3.4.8 Type Featuring .....                               | 41 |
| 7.4 Kernel .....   | 42 |
| 7.4.1 Kernel Overview .....                                | 42 |
| 7.4.2 Data Types .....                                     | 42 |
| 7.4.3 Classes .....  | 43 |
| 7.4.4 Structures .....                                     | 43 |
| 7.4.5 Associations .....                                   | 44 |
| 7.4.6 Connectors .....                                     | 45 |
| 7.4.6.1 Connectors Overview .....                          | 45 |
| 7.4.6.2 Connector Declaration .....                        | 46 |
| 7.4.6.3 Binding Connector Declaration .....                | 48 |
| 7.4.6.4 Succession Declaration .....                       | 49 |
| 7.4.7 Behaviors .....                                      | 50 |
| 7.4.7.1 Behaviors Overview .....                           | 50 |
| 7.4.7.2 Behavior Declaration .....                         | 50 |
| 7.4.7.3 Step Declaration .....                             | 51 |
| 7.4.8 Functions .....                                      | 52 |
| 7.4.8.1 Functions Overview .....                           | 52 |
| 7.4.8.2 Function Declaration .....                         | 52 |
| 7.4.8.3 Expression Declaration .....                       | 53 |
| 7.4.8.4 Predicate Declaration .....                        | 54 |
| 7.4.8.5 Boolean Expression and Invariant Declaration ..... | 55 |
| 7.4.9 Expressions .....                                    | 55 |
| 7.4.9.1 Expressions Overview .....                         | 55 |
| 7.4.9.2 Operator Expressions .....                         | 56 |
| 7.4.9.3 Primary Expressions .....                          | 58 |
| 7.4.9.4 Base Expressions .....                             | 60 |
| 7.4.9.5 Literal Expressions .....                          | 61 |
| 7.4.10 Interactions .....                                  | 61 |
| 7.4.10.1 Interactions Overview .....                       | 61 |
| 7.4.10.2 Interaction Declaration .....                     | 62 |
| 7.4.10.3 Item Flow Declaration .....                       | 62 |
| 7.4.11 Feature Values .....                                | 63 |
| 7.4.12 Multiplicities .....                                | 64 |
| 7.4.13 Metadata .....                                      | 65 |
| 7.4.14 Packages .....                                      | 68 |
| 8 Metamodel .....  | 71 |
| 8.1 Metamodel Overview .....                               | 71 |
| 8.2 Concrete Syntax .....                                  | 71 |
| 8.2.1 Concrete Syntax Overview .....                       | 72 |
| 8.2.2 Lexical Structure .....                              | 73 |
| 8.2.2.1 Line Terminators and White Space .....             | 73 |
| 8.2.2.2 Notes and Comments .....                           | 74 |
| 8.2.2.3 Names .....  | 74 |
| 8.2.2.4 Numeric Values .....                               | 75 |
| 8.2.2.5 String Value .....                                 | 76 |
| 8.2.2.6 Reserved Words .....                               | 76 |
| 8.2.2.7 Symbols .....                                      | 76 |
| 8.2.3 Root Concrete Syntax .....                           | 77 |
| 8.2.3.1 Elements and Relationships Concrete Syntax .....   | 77 |
| 8.2.3.1.1 Elements .....                                   | 77 |
| 8.2.3.1.2 Relationships .....                              | 78 |

|   |     |
|---|-----|
| 8.2.3.2 Annotations Concrete Syntax .....           | 78  |
| 8.2.3.2.1 Annotations .....                         | 79  |
| 8.2.3.2.2 Comments and Documentation .....          | 79  |
| 8.2.3.2.3 Textual Representation .....              | 80  |
| 8.2.3.3 Namespaces Concrete Syntax .....            | 80  |
| 8.2.3.3.1 Namespaces .....                          | 81  |
| 8.2.3.3.2 Imports .....                             | 82  |
| 8.2.3.3.3 Namespace Elements .....                  | 83  |
| 8.2.3.3.4 Qualified Names and Name Resolution ..... | 83  |
| 8.2.4 Core Concrete Syntax .....                    | 85  |
| 8.2.4.1 Types Concrete Syntax .....                 | 85  |
| 8.2.4.1.1 Types .....                               | 86  |
| 8.2.4.1.2 Specialization .....                      | 86  |
| 8.2.4.1.3 Conjugation .....                         | 87  |
| 8.2.4.1.4 Disjoining .....                          | 87  |
| 8.2.4.1.5 Feature Membership .....                  | 87  |
| 8.2.4.2 Classifiers Concrete Syntax .....           | 87  |
| 8.2.4.2.1 Classifiers .....                         | 88  |
| 8.2.4.2.2 Subclassification .....                   | 88  |
| 8.2.4.3 Features Concrete Syntax .....              | 88  |
| 8.2.4.3.1 Features .....                            | 89  |
| 8.2.4.3.2 Feature Typing .....                      | 91  |
| 8.2.4.3.3 Subsetting .....                          | 91  |
| 8.2.4.3.4 Redefinition .....                        | 91  |
| 8.2.4.3.5 Feature Chaining .....                    | 92  |
| 8.2.4.3.6 Feature Inverting .....                   | 92  |
| 8.2.4.3.7 Type Featuring .....                      | 92  |
| 8.2.5 Kernel Concrete Syntax .....                  | 92  |
| 8.2.5.1 Data Types Concrete Syntax .....            | 92  |
| 8.2.5.2 Classes Concrete Syntax .....               | 93  |
| 8.2.5.3 Structures Concrete Syntax .....            | 93  |
| 8.2.5.4 Associations Concrete Syntax .....          | 93  |
| 8.2.5.5 Connectors Concrete Syntax .....            | 93  |
| 8.2.5.5.1 Connectors .....                          | 93  |
| 8.2.5.5.2 Binding Connectors .....                  | 94  |
| 8.2.5.5.3 Successions .....                         | 94  |
| 8.2.5.6 Behaviors Concrete Syntax .....             | 94  |
| 8.2.5.6.1 Behaviors .....                           | 94  |
| 8.2.5.6.2 Steps .....                               | 94  |
| 8.2.5.7 Functions Concrete Syntax .....             | 94  |
| 8.2.5.7.1 Functions .....                           | 95  |
| 8.2.5.7.2 Expressions .....                         | 95  |
| 8.2.5.7.3 Predicates .....                          | 95  |
| 8.2.5.7.4 Boolean Expressions and Invariants .....  | 95  |
| 8.2.5.8 Expressions Concrete Syntax .....           | 95  |
| 8.2.5.8.1 Operator Expressions .....                | 96  |
| 8.2.5.8.2 Primary Expressions .....                 | 100 |
| 8.2.5.8.3 Base Expressions .....                    | 103 |
| 8.2.5.8.4 Literal Expressions .....                 | 105 |
| 8.2.5.9 Interactions Concrete Syntax .....          | 105 |
| 8.2.5.9.1 Interactions .....                        | 105 |
| 8.2.5.9.2 Item Flows .....                          | 106 |
| 8.2.5.10 Feature Values Concrete Syntax .....       | 107 |
| 8.2.5.11 Multiplicities Concrete Syntax .....       | 107 |

|  |     |
|--|-----|
| 8.2.5.12 Metadata Concrete Syntax .....                  | 108 |
| 8.2.5.13 Packages Concrete Syntax .....                  | 109 |
| 8.3 Abstract Syntax .....                                | 109 |
| 8.3.1 Abstract Syntax Overview .....                     | 109 |
| 8.3.2 Root Abstract Syntax .....                         | 111 |
| 8.3.2.1 Elements and Relationships Abstract Syntax ..... | 111 |
| 8.3.2.1.1 Overview .....                                 | 112 |
| 8.3.2.1.2 Element .....                                  | 112 |
| 8.3.2.1.3 Relationship .....                             | 115 |
| 8.3.2.2 Annotations Abstract Syntax .....                | 116 |
| 8.3.2.2.1 Overview .....                                 | 116 |
| 8.3.2.2.2 AnnotatingElement .....                        | 117 |
| 8.3.2.2.3 Annotation .....                               | 117 |
| 8.3.2.2.4 Comment .....                                  | 118 |
| 8.3.2.2.5 Documentation .....                            | 118 |
| 8.3.2.2.6 TextualRepresentation .....                    | 119 |
| 8.3.2.3 Namespace Abstract Syntax .....                  | 120 |
| 8.3.2.3.1 Overview .....                                 | 120 |
| 8.3.2.3.2 Import .....                                   | 120 |
| 8.3.2.3.3 Membership .....                               | 122 |
| 8.3.2.3.4 Namespace .....                                | 123 |
| 8.3.2.3.5 VisibilityKind .....                           | 126 |
| 8.3.2.3.6 OwningMembership .....                         | 126 |
| 8.3.3 Core Abstract Syntax .....                         | 127 |
| 8.3.3.1 Types Abstract Syntax .....                      | 127 |
| 8.3.3.1.1 Overview .....                                 | 128 |
| 8.3.3.1.2 Conjugation .....                              | 130 |
| 8.3.3.1.3 Disjoining .....                               | 131 |
| 8.3.3.1.4 FeatureDirectionKind .....                     | 131 |
| 8.3.3.1.5 FeatureMembership .....                        | 132 |
| 8.3.3.1.6 Specialization .....                           | 132 |
| 8.3.3.1.7 Multiplicity .....                             | 133 |
| 8.3.3.1.8 Type .....                                     | 134 |
| 8.3.3.2 Classifiers Abstract Syntax .....                | 138 |
| 8.3.3.2.1 Overview .....                                 | 138 |
| 8.3.3.2.2 Classifier .....                               | 138 |
| 8.3.3.2.3 Subclassification .....                        | 139 |
| 8.3.3.3 Features Abstract Syntax .....                   | 140 |
| 8.3.3.3.1 Overview .....                                 | 140 |
| 8.3.3.3.2 EndFeatureMembership .....                     | 142 |
| 8.3.3.3.3 Feature .....                                  | 143 |
| 8.3.3.3.4 FeatureChaining .....                          | 148 |
| 8.3.3.3.5 FeatureInverting .....                         | 149 |
| 8.3.3.3.6 FeatureTyping .....                            | 149 |
| 8.3.3.3.7 Redefinition .....                             | 150 |
| 8.3.3.3.8 Subsetting .....                               | 150 |
| 8.3.3.3.9 TypeFeaturing .....                            | 151 |
| 8.3.4 Kernel Abstract Syntax .....                       | 152 |
| 8.3.4.1 Data Types Abstract Syntax .....                 | 152 |
| 8.3.4.1.1 Overview .....                                 | 152 |
| 8.3.4.1.2 DataType .....                                 | 152 |
| 8.3.4.2 Classes Abstract Syntax .....                    | 153 |
| 8.3.4.2.1 Overview .....                                 | 153 |
| 8.3.4.2.2 Class .....                                    | 153 |



|   |     |
|---|-----|
| 8.3.4.3 Structures Abstract Syntax .....      | 154 |
| 8.3.4.3.1 Overview .....                      | 154 |
| 8.3.4.3.2 Structure .....                     | 154 |
| 8.3.4.4 Associations Abstract Syntax .....    | 154 |
| 8.3.4.4.1 Overview .....                      | 155 |
| 8.3.4.4.2 Association .....                   | 155 |
| 8.3.4.4.3 AssociationStructure .....          | 156 |
| 8.3.4.5 Connectors Abstract Syntax .....      | 157 |
| 8.3.4.5.1 Overview .....                      | 157 |
| 8.3.4.5.2 Binding Connector .....             | 158 |
| 8.3.4.5.3 Connector .....                     | 158 |
| 8.3.4.5.4 Succession .....                    | 160 |
| 8.3.4.6 Behaviors Abstract Syntax .....       | 161 |
| 8.3.4.6.1 Overview .....                      | 161 |
| 8.3.4.6.2 Behavior .....                      | 162 |
| 8.3.4.6.3 Step .....                          | 162 |
| 8.3.4.6.4 ParameterMembership .....           | 163 |
| 8.3.4.7 Functions Abstract Syntax .....       | 163 |
| 8.3.4.7.1 Overview .....                      | 164 |
| 8.3.4.7.2 BooleanExpression .....             | 164 |
| 8.3.4.7.3 Expression .....                    | 165 |
| 8.3.4.7.4 Function .....                      | 166 |
| 8.3.4.7.5 Invariant .....                     | 166 |
| 8.3.4.7.6 Predicate .....                     | 167 |
| 8.3.4.7.7 ResultExpressionMembership .....    | 167 |
| 8.3.4.7.8 ReturnParameterMembership .....     | 168 |
| 8.3.4.8 Expressions Abstract Syntax .....     | 168 |
| 8.3.4.8.1 Overview .....                      | 169 |
| 8.3.4.8.2 CollectExpression .....             | 169 |
| 8.3.4.8.3 FeatureChainExpression .....        | 170 |
| 8.3.4.8.4 FeatureReferenceExpression .....    | 170 |
| 8.3.4.8.5 InvocationExpression .....          | 171 |
| 8.3.4.8.6 LiteralBoolean .....                | 172 |
| 8.3.4.8.7 LiteralExpression .....             | 172 |
| 8.3.4.8.8 LiteralInteger .....                | 173 |
| 8.3.4.8.9 LiteralReal .....                   | 173 |
| 8.3.4.8.10 LiteralString .....                | 174 |
| 8.3.4.8.11 LiteralInfinity .....              | 174 |
| 8.3.4.8.12 NullExpression .....               | 175 |
| 8.3.4.8.13 OperatorExpression .....           | 175 |
| 8.3.4.8.14 SelectExpression .....             | 176 |
| 8.3.4.9 Interactions Abstract Syntax .....    | 176 |
| 8.3.4.9.1 Overview .....                      | 176 |
| 8.3.4.9.2 ItemFlow .....                      | 177 |
| 8.3.4.9.3 Interaction .....                   | 178 |
| 8.3.4.9.4 SuccessionItemFlow .....            | 178 |
| 8.3.4.10 Feature Values Abstract Syntax ..... | 179 |
| 8.3.4.10.1 Overview .....                     | 179 |
| 8.3.4.10.2 FeatureValue .....                 | 179 |
| 8.3.4.11 Multiplicities Abstract Syntax ..... | 180 |
| 8.3.4.11.1 Overview .....                     | 181 |
| 8.3.4.11.2 MultiplicityRange .....            | 181 |
| 8.3.4.12 Metadata Abstract Syntax .....       | 182 |
| 8.3.4.12.1 Overview .....                     | 182 |

|  |     |
|--|-----|
| 8.3.4.12.2 Metaclass .....               | 182 |
| 8.3.4.12.3 MetadataFeature .....         | 182 |
| 8.3.4.13 Packages Abstract Syntax .....  | 183 |
| 8.3.4.13.1 Overview .....                | 183 |
| 8.3.4.13.2 ElementFilterMembership ..... | 183 |
| 8.3.4.13.3 Package .....                 | 184 |
| 8.4 Semantics .....                      | 185 |
| 8.4.1 Semantics Overview .....           | 185 |
| 8.4.2 Core Semantics .....               | 186 |
| 8.4.2.1 Core Semantics Overview .....    | 186 |
| 8.4.2.2 Types Semantics .....            | 188 |
| 8.4.2.3 Classifiers Semantics .....      | 188 |
| 8.4.2.4 Features Semantics .....         | 189 |
| 8.4.3 Kernel Semantics .....             | 190 |
| 8.4.3.1 Kernel Semantics Overview .....  | 190 |
| 8.4.3.2 Data Types Semantics .....       | 191 |
| 8.4.3.3 Classes Semantics .....          | 191 |
| 8.4.3.4 Structures Semantics .....       | 191 |
| 8.4.3.5 Associations Semantics .....     | 191 |
| 8.4.3.6 Connectors Semantics .....       | 193 |
| 8.4.3.7 Behavior Semantics .....         | 195 |
| 8.4.3.8 Functions Semantics .....        | 196 |
| 8.4.3.9 Expressions Semantics .....      | 198 |
| 8.4.3.10 Interactions Semantics .....    | 201 |
| 8.4.3.11 Feature Values Semantics .....  | 202 |
| 8.4.3.12 Multiplicities Semantics .....  | 202 |
| 8.4.3.13 Metadata Semantics .....        | 203 |
| 8.4.3.14 Packages Semantics .....        | 204 |
| 9 Model Libraries .....                  | 205 |
| 9.1 Model Libraries Overview .....       | 205 |
| 9.2 Semantic Library .....               | 205 |
| 9.2.1 Semantic Library Overview .....    | 205 |
| 9.2.2 Base .....                         | 205 |
| 9.2.2.1 Base Overview .....              | 206 |
| 9.2.2.2 Elements .....                   | 206 |
| 9.2.2.2.1 Anything .....                 | 206 |
| 9.2.2.2.2 DataValue .....                | 206 |
| 9.2.2.2.3 dataValues .....               | 207 |
| 9.2.2.2.4 naturals .....                 | 207 |
| 9.2.2.2.5 SelfSameLifeLink .....         | 208 |
| 9.2.2.2.6 things .....                   | 208 |
| 9.2.3 Occurrences .....                  | 209 |
| 9.2.3.1 Occurrences Overview .....       | 209 |
| 9.2.3.2 Elements .....                   | 212 |
| 9.2.3.2.1 Causation .....                | 212 |
| 9.2.3.2.2 causations .....               | 212 |
| 9.2.3.2.3 HappensBefore .....            | 213 |
| 9.2.3.2.4 happensBeforeLinks .....       | 213 |
| 9.2.3.2.5 HappensDuring .....            | 214 |
| 9.2.3.2.6 HappensJustBefore .....        | 214 |
| 9.2.3.2.7 HappensLink .....              | 215 |
| 9.2.3.2.8 HappensWhile .....             | 215 |
| 9.2.3.2.9 InnerSpaceOf .....             | 216 |
| 9.2.3.2.10 InsideOf .....                | 216 |

|  |     |
|--|-----|
| 9.2.3.2.11 JustOutsideOf .....         | 217 |
| 9.2.3.2.12 Life .....                  | 217 |
| 9.2.3.2.13 MatesWith .....             | 218 |
| 9.2.3.2.14 Occurrence .....            | 218 |
| 9.2.3.2.15 occurrences .....           | 223 |
| 9.2.3.2.16 OutsideOf .....             | 223 |
| 9.2.3.2.17 PortionOf .....             | 224 |
| 9.2.3.2.18 SnapshotOf .....            | 225 |
| 9.2.3.2.19 SpaceShotOf .....           | 225 |
| 9.2.3.2.20 SpaceSliceOf .....          | 226 |
| 9.2.3.2.21 SurroundedBy .....          | 226 |
| 9.2.3.2.22 TimeSliceOf .....           | 227 |
| 9.2.3.2.23 Within .....                | 227 |
| 9.2.3.2.24 WithinBoth .....            | 228 |
| 9.2.3.2.25 Without .....               | 229 |
| 9.2.4 Links .....                      | 229 |
| 9.2.4.1 Links Overview .....           | 229 |
| 9.2.4.2 Elements .....                 | 230 |
| 9.2.4.2.1 BinaryLink .....             | 230 |
| 9.2.4.2.2 binaryLinks .....            | 230 |
| 9.2.4.2.3 Link .....                   | 231 |
| 9.2.4.2.4 links .....                  | 231 |
| 9.2.4.2.5 SelfLink .....               | 232 |
| 9.2.4.2.6 selfLinks .....              | 232 |
| 9.2.5 Objects .....                    | 233 |
| 9.2.5.1 Objects Overview .....         | 233 |
| 9.2.5.2 Elements .....                 | 234 |
| 9.2.5.2.1 BinaryLinkObject .....       | 234 |
| 9.2.5.2.2 binaryLinkObjects .....      | 234 |
| 9.2.5.2.3 Body .....                   | 235 |
| 9.2.5.2.4 Curve .....                  | 235 |
| 9.2.5.2.5 LinkObject .....             | 235 |
| 9.2.5.2.6 linkObjects .....            | 236 |
| 9.2.5.2.7 Object .....                 | 236 |
| 9.2.5.2.8 objects .....                | 237 |
| 9.2.5.2.9 Point .....                  | 238 |
| 9.2.5.2.10 StructuredSpaceObject ..... | 238 |
| 9.2.5.2.11 Surface .....               | 239 |
| 9.2.6 Performances .....               | 239 |
| 9.2.6.1 Performances Overview .....    | 240 |
| 9.2.6.2 Elements .....                 | 240 |
| 9.2.6.2.1 BooleanEvaluation .....      | 240 |
| 9.2.6.2.2 booleanEvaluations .....     | 241 |
| 9.2.6.2.3 Evaluation .....             | 241 |
| 9.2.6.2.4 evaluations .....            | 242 |
| 9.2.6.2.5 falseEvaluations .....       | 242 |
| 9.2.6.2.6 Involves .....               | 242 |
| 9.2.6.2.7 LiteralEvaluation .....      | 243 |
| 9.2.6.2.8 literalEvaluations .....     | 243 |
| 9.2.6.2.9 NullEvaluation .....         | 244 |
| 9.2.6.2.10 nullEvaluations .....       | 244 |
| 9.2.6.2.11 Performance .....           | 244 |
| 9.2.6.2.12 performances .....          | 245 |
| 9.2.6.2.13 Performs .....              | 246 |

|   |     |
|---|-----|
| 9.2.6.2.14 trueEvaluations .....                            | 246 |
| 9.2.7 Transfers.....  | 246 |
| 9.2.7.1 Transfers Overview .....                            | 246 |
| 9.2.7.2 Elements .....                                      | 247 |
| 9.2.7.2.1 Transfer .....                                    | 247 |
| 9.2.7.2.2 TransferBefore .....                              | 248 |
| 9.2.7.2.3 transfers .....                                   | 248 |
| 9.2.7.2.4 transfersBefore .....                             | 249 |
| 9.2.8 Feature Referencing Performances .....                | 249 |
| 9.2.8.1 Feature Referencing Performances Overview .....     | 249 |
| 9.2.8.2 Elements .....                                      | 249 |
| 9.2.8.2.1 BooleanEvaluationResultMonitorPerformance .....   | 249 |
| 9.2.8.2.2 BooleanEvaluationResultToMonitorPerformance ..... | 250 |
| 9.2.8.2.3 EvaluationResultMonitorPerformance .....          | 251 |
| 9.2.8.2.4 FeatureAccessPerformance .....                    | 252 |
| 9.2.8.2.5 FeatureMonitorPerformance .....                   | 252 |
| 9.2.8.2.6 FeatureReadEvaluation .....                       | 253 |
| 9.2.8.2.7 FeatureReferencingPerformance .....               | 254 |
| 9.2.8.2.8 FeatureWritePerformance .....                     | 254 |
| 9.2.9 Control Performances.....                             | 255 |
| 9.2.9.1 Control Performances Overview .....                 | 255 |
| 9.2.9.2 Elements .....                                      | 255 |
| 9.2.9.2.1 DecisionPerformance .....                         | 255 |
| 9.2.9.2.2 IfElsePerformance .....                           | 256 |
| 9.2.9.2.3 IfPerformance .....                               | 256 |
| 9.2.9.2.4 IfThenElsePerformance .....                       | 256 |
| 9.2.9.2.5 IfThenPerformance .....                           | 257 |
| 9.2.9.2.6 LoopPerformance .....                             | 257 |
| 9.2.9.2.7 MergePerformance .....                            | 258 |
| 9.2.10 State Performances.....                              | 258 |
| 9.2.10.1 State Performances Overview .....                  | 258 |
| 9.2.10.2 Elements .....                                     | 259 |
| 9.2.10.2.1 StatePerformance .....                           | 259 |
| 9.2.10.2.2 StateTransitionPerformance .....                 | 260 |
| 9.2.11 Transition Performances .....                        | 260 |
| 9.2.11.1 Transition Performances Overview .....             | 260 |
| 9.2.11.2 Elements .....                                     | 262 |
| 9.2.11.2.1 NonStateTransitionPerformance .....              | 262 |
| 9.2.11.2.2 TPCGuardConstraint .....                         | 262 |
| 9.2.11.2.3 TransitionPerformance .....                      | 262 |
| 9.2.12 Clocks.....  | 263 |
| 9.2.12.1 Clocks Overview .....                              | 263 |
| 9.2.12.2 Elements .....                                     | 263 |
| 9.2.12.2.1 BasicClock.....                                  | 263 |
| 9.2.12.2.2 BasicDurationOf .....                            | 264 |
| 9.2.12.2.3 BasicTimeOf .....                                | 264 |
| 9.2.12.2.4 Clock .....                                      | 265 |
| 9.2.12.2.5 DurationOf.....                                  | 265 |
| 9.2.12.2.6 TimeOf .....                                     | 266 |
| 9.2.12.2.7 universalClock.....                              | 266 |
| 9.2.13 Observation .....                                    | 267 |
| 9.2.13.1 Observation Overview .....                         | 267 |
| 9.2.13.2 Elements .....                                     | 267 |
| 9.2.13.2.1 CancelObservation .....                          | 267 |

|             |                                |     |
|-------------|--------------------------------|-----|
| 9.2.13.2.2  | changeCondition                | 267 |
| 9.2.13.2.3  | ChangeMonitor                  | 268 |
| 9.2.13.2.4  | ChangeSignal                   | 268 |
| 9.2.13.2.5  | defaultMonitor                 | 269 |
| 9.2.13.2.6  | ObserveChange                  | 269 |
| 9.2.13.2.7  | StartObservation               | 270 |
| 9.2.14      | Triggers                       | 270 |
| 9.2.14.1    | Triggers Overview              | 270 |
| 9.2.14.2    | Elements                       | 270 |
| 9.2.14.2.1  | TimeSignal                     | 270 |
| 9.2.14.2.2  | TriggerAfter                   | 271 |
| 9.2.14.2.3  | TriggerAt                      | 272 |
| 9.2.14.2.4  | TriggerWhen                    | 273 |
| 9.2.15      | SpatialFrames                  | 273 |
| 9.2.15.1    | SpatialFrames Overview         | 273 |
| 9.2.15.2    | Elements                       | 273 |
| 9.2.15.2.1  | CartesianCurrentDisplacementOf | 273 |
| 9.2.15.2.2  | CartesianCurrentPositionOf     | 274 |
| 9.2.15.2.3  | CartesianDisplacementOf        | 274 |
| 9.2.15.2.4  | CartesianPositionOf            | 275 |
| 9.2.15.2.5  | CartesianSpatialFrame          | 276 |
| 9.2.15.2.6  | CurrentDisplacementOf          | 276 |
| 9.2.15.2.7  | CurrentPositionOf              | 277 |
| 9.2.15.2.8  | defaultFrame                   | 277 |
| 9.2.15.2.9  | DisplacementOf                 | 278 |
| 9.2.15.2.10 | PositionOf                     | 278 |
| 9.2.15.2.11 | SpatialFrame                   | 279 |
| 9.2.16      | Metaobjects                    | 280 |
| 9.2.16.1    | Metaobjects Overview           | 280 |
| 9.2.16.2    | Elements                       | 280 |
| 9.2.16.2.1  | Metaobject                     | 280 |
| 9.2.16.2.2  | metaobjects                    | 280 |
| 9.2.16.2.3  | SemanticMetadata               | 281 |
| 9.2.17      | KerML                          | 281 |
| 9.2.17.1    | KerML Overview                 | 281 |
| 9.2.17.2    | Elements                       | 281 |
| 9.3         | Data Type Library              | 283 |
| 9.3.1       | Data Types Library Overview    | 283 |
| 9.3.2       | Scalar Values                  | 283 |
| 9.3.2.1     | Scalar Values Overview         | 283 |
| 9.3.2.2     | Elements                       | 283 |
| 9.3.2.2.1   | Boolean                        | 283 |
| 9.3.2.2.2   | Complex                        | 283 |
| 9.3.2.2.3   | Integer                        | 284 |
| 9.3.2.2.4   | Natural                        | 284 |
| 9.3.2.2.5   | Number                         | 285 |
| 9.3.2.2.6   | NumericalValue                 | 285 |
| 9.3.2.2.7   | Positive                       | 285 |
| 9.3.2.2.8   | Rational                       | 286 |
| 9.3.2.2.9   | Real                           | 286 |
| 9.3.2.2.10  | ScalarValue                    | 287 |
| 9.3.2.2.11  | String                         | 287 |
| 9.3.3       | Collections                    | 287 |
| 9.3.3.1     | Collections Overview           | 287 |

|  |     |
|--|-----|
| 9.3.3.2 Elements .....                     | 288 |
| 9.3.3.2.1 Array.....                       | 288 |
| 9.3.3.2.2 Bag.....                         | 288 |
| 9.3.3.2.3 Collection .....                 | 289 |
| 9.3.3.2.4 KeyValuePair .....               | 289 |
| 9.3.3.2.5 List.....                        | 290 |
| 9.3.3.2.6 Map.....                         | 290 |
| 9.3.3.2.7 OrderedCollection .....          | 290 |
| 9.3.3.2.8 OrderedMap.....                  | 291 |
| 9.3.3.2.9 OrderedSet.....                  | 291 |
| 9.3.3.2.10 Set.....                        | 292 |
| 9.3.3.2.11 UniqueCollection.....           | 292 |
| 9.3.4 Vector Values.....                   | 293 |
| 9.3.4.1 Vector Values Overview .....       | 293 |
| 9.3.4.2 Elements .....                     | 293 |
| 9.3.4.2.1 CartesianThreeVectorValue .....  | 293 |
| 9.3.4.2.2 CartesianVectorValue .....       | 293 |
| 9.3.4.2.3 NumericalVectorValue.....        | 294 |
| 9.3.4.2.4 ThreeVectorValue .....           | 294 |
| 9.3.4.2.5 VectorValue.....                 | 294 |
| 9.4 Function Library .....                 | 295 |
| 9.4.1 Function Library Overview.....       | 295 |
| 9.4.2 Base Functions .....                 | 295 |
| 9.4.2.1 Base Functions Overview.....       | 295 |
| 9.4.2.2 Elements .....                     | 295 |
| 9.4.3 Data Functions .....                 | 296 |
| 9.4.3.1 Data Functions Overview.....       | 296 |
| 9.4.3.2 Elements .....                     | 296 |
| 9.4.4 Scalar Functions .....               | 297 |
| 9.4.4.1 Scalar Functions Overview .....    | 297 |
| 9.4.4.2 Elements .....                     | 297 |
| 9.4.5 Boolean Functions.....               | 298 |
| 9.4.5.1 Boolean Functions Overview .....   | 298 |
| 9.4.5.2 Elements .....                     | 298 |
| 9.4.6 String Functions .....               | 298 |
| 9.4.6.1 String Functions Overview.....     | 298 |
| 9.4.6.2 Elements .....                     | 298 |
| 9.4.7 Numerical Functions .....            | 299 |
| 9.4.7.1 Numerical Functions Overview ..... | 299 |
| 9.4.7.2 Elements .....                     | 299 |
| 9.4.8 Complex Functions .....              | 300 |
| 9.4.8.1 Complex Functions Overview.....    | 300 |
| 9.4.8.2 Elements .....                     | 300 |
| 9.4.9 Real Functions.....                  | 301 |
| 9.4.9.1 Real Functions Overview .....      | 301 |
| 9.4.9.2 Elements .....                     | 301 |
| 9.4.10 Rational Functions .....            | 302 |
| 9.4.10.1 Rational Functions Overview.....  | 302 |
| 9.4.10.2 Elements .....                    | 302 |
| 9.4.11 Integer Functions.....              | 303 |
| 9.4.11.1 Integer Functions Overview .....  | 303 |
| 9.4.11.2 Elements .....                    | 303 |
| 9.4.12 Natural Functions .....             | 304 |
| 9.4.12.1 Natural Functions Overview .....  | 304 |

|  |     |
|--|-----|
| 9.4.12.2 Elements .....                      | 304 |
| 9.4.13 Trig Functions .....                  | 305 |
| 9.4.13.1 Trig Functions Overview .....       | 305 |
| 9.4.13.2 Elements .....                      | 305 |
| 9.4.14 Sequence Functions .....              | 306 |
| 9.4.14.1 Sequence Functions Overview .....   | 306 |
| 9.4.14.2 Elements .....                      | 306 |
| 9.4.15 Collection Functions .....            | 307 |
| 9.4.15.1 Collection Functions Overview ..... | 307 |
| 9.4.15.2 Elements .....                      | 307 |
| 9.4.16 Vector Functions .....                | 308 |
| 9.4.16.1 Vector Functions Overview .....     | 309 |
| 9.4.16.2 Elements .....                      | 309 |
| 9.4.17 Control Functions .....               | 313 |
| 9.4.17.1 Control Functions Overview .....    | 313 |
| 9.4.17.2 Elements .....                      | 313 |
| 10 Model Interchange .....                   | 317 |
| A Annex: Conformance Test Suite .....        | 319 |

# List of Tables

|  |     |
|--|-----|
| 1. Grammar Production Definitions.....           | 72  |
| 2. EBNF Notation Conventions .....               | 72  |
| 3. Abstract Syntax Synthesis Notation.....       | 72  |
| 4. Escape Sequences .....                        | 75  |
| 5. Operator Mapping.....                         | 98  |
| 6. Operator Precedence (highest to lowest) ..... | 99  |
| 7. Primary Expression Operator Mapping .....     | 102 |



# List of Figures

|                                       |     |
|---------------------------------------|-----|
| 1. KerML Syntax Layers.....           | 110 |
| 2. KerML Element Hierarchy .....      | 111 |
| 3. KerML Relationship Hierarchy ..... | 111 |
| 4. Elements.....                      | 112 |
| 5. Annotation.....                    | 116 |
| 6. Namespaces.....                    | 120 |
| 7. Types.....                         | 128 |
| 8. Specialization .....               | 129 |
| 9. Conjugation .....                  | 129 |
| 10. Disjointness.....                 | 130 |
| 11. Classifiers.....                  | 138 |
| 12. Features .....                    | 140 |
| 13. Subsetting.....                   | 141 |
| 14. Feature Chaining .....            | 141 |
| 15. Feature Inverting .....           | 142 |
| 16. End Feature Membership .....      | 142 |
| 17. Data Types .....                  | 152 |
| 18. Classes.....                      | 153 |
| 19. Structures .....                  | 154 |
| 20. Associations .....                | 155 |
| 21. Connectors .....                  | 157 |
| 22. Successions .....                 | 158 |
| 23. Behaviors .....                   | 161 |
| 24. Parameter Memberships.....        | 162 |
| 25. Functions.....                    | 164 |
| 26. Predicates .....                  | 164 |
| 27. Function Memberships.....         | 164 |
| 28. Expressions .....                 | 169 |
| 29. Literal Expressions.....          | 169 |
| 30. Interactions.....                 | 176 |
| 31. Item Flows .....                  | 177 |
| 32. Feature Values .....              | 179 |
| 33. Multiplicities .....              | 181 |
| 34. Metadata Annotation.....          | 182 |
| 35. Packages.....                     | 183 |
| 36. KerML Semantic Layers.....        | 186 |



# 0 Submission Introduction

## 0.1 Submission Overview

This document is the first of two documents submitted in response to the Systems Modeling Language (SysML®) v2 Request for Proposals (RFP) (ad/2017-11-04). This document defines a *Kernel Modeling Language (KerML)* that provides a syntactic and semantic foundation for creating application specific modeling languages. The second document specifies the *Systems Modeling Language (SysML)*, version 2.0, built on this foundation.

Even though both documents are being submitted together to fulfill the requirements of the RFP, the present document for KerML is proposed as a separate specification from SysML v2. KerML provides a common basis for creation of new modeling languages (or evolution of existing modeling languages). It moves beyond the syntactic interoperability offered by MOF to the possibility of diverse modeling languages that are tailored to specific applications while maintaining fundamental semantic interoperability.

## 0.2 Submission Submitters

The following OMG member organizations are jointly submitting this proposed specification:

- 88Solutions Corporation
- Dassault Systèmes
- GfSE e.V.
- IBM
- INCOSE
- Intercax LLC
- Lockheed Martin Corporation
- MITRE
- Model Driven Solutions, Inc.
- PTC
- Simula Research Laboratory AS
- Thematix Partners

The submitters also thankfully acknowledge the support of over 60 other organizations that participated in the SysML v2 Submission Team (SST).

## 0.3 Submission - Issues to be discussed

*6.7.1 Proposals shall describe a proof of concept implementation that can successfully execute the test cases that are required in 6.5.4.*

The SST is developing a pilot implementation of the full KerML abstract syntax and textual concrete syntax. This is publicly available under an open source license at <https://github.com/Systems-Modeling>.

**Implementation Note.** The pilot implementation is being incrementally developed along with each draft release of this document. Since the conformance test suite has not been developed yet, it is not possible to formally demonstrate the conformance of the implementation to the proposed specification. Nevertheless, the majority of this proposed specification describes the language as it has been implemented. For those specific areas in which the pilot implementation is known to not fully conform to the current draft specification, the deviations are identified in "implementation notes" in this document (such as this one).

*6.7.2 Proposals shall provide a requirements traceability matrix that demonstrates how each requirement in the RFP is satisfied. It is recognized that the requirements will be evaluated in more detail as part of the submission process. Rationale should be included in the matrix to support any proposed changes to these requirements.*

See subclause 0.4 in the proposed *Systems Modeling Language (SysML), Version 2.0* specification document submitted along with the present document.

*6.7.3 Proposals shall include a description of how OMG technologies are leveraged and what proposed changes to these technologies are needed to support the specification.*

As required in the SysML v2 RFP, the abstract syntax for KerML is defined as a model that is consistent with the OMG Meta Object Facility [MOF] as extended with MOF Support for Semantic Structures [SMOF] (see [Clause](#)). This also allows KerML models represented in the KerML abstract syntax to be interchanged using OMG XML Metadata Interchange [XMI].

The OMG MOF standard has been used to define many OMG-standardized modeling languages, and the KerML language definition is also built on it. However, MOF and XMI only standardize the means for specifying the abstract syntax of a modeling language and interchanging models so specified. Even SMOF provides only limited additional support for the syntactic structures required for so-called "semantic" languages.

The goal of KerML is to go beyond this and to become a new OMG standard providing application-independent syntax *and semantics* for creating more specific modeling languages (as described further in [Clause 1](#)). This will allow not only syntactic interchange between modeling tools, but also semantic interoperability. The KerML specification is being submitted as part of the SysML v2 submission, because the SST has built SysML v2 on KerML in exactly this way.

## **0.4 Language Requirement Tables**

See subclause 0.4 of the proposed *Systems Modeling Language (SysML), Version 2.0* specification document submitted along with the present document.

# 1 Scope

The Kernel Modeling Language (KerML) is a application-independent modeling language with a well-grounded formal semantics for modeling existing or planned systems. The language includes general syntactic constructs for structuring models, such as relationships, annotations and namespaces; core semantics constructs that have semantics based on classification; and additional constructs for commonly needed modeling capabilities, such as associations and behaviors.

System models are expressed in KerML using a textual concrete syntax. This can be parsed to an abstract syntax representation, which is then given a semantic interpretation for the system being modeled. The semantics for the KerML core constructs is grounded in formal mathematical logic, providing a consistent basis for mathematical reasoning about KerML models. However, beyond this, the semantics of KerML constructs are specified by the relationship of user model elements to the KerML Semantic Library.

The Semantic Library models, also expressed in KerML, provide an ontological model of the meaning of KerML models. Indeed, all KerML models can be semantically expressed using solely core modeling constructs referencing the appropriate semantic concepts defined in the Semantic Library. KerML semantic constructs beyond the core are essentially just syntactic conveniences for reusing specific library concepts: structures for modeling *objects*, behaviors for modeling *performances*, associations for modeling *links*, etc.

Indeed, the full KerML language can be considered to be simply a syntactic extension of the core, which is semantically extended using library models. By intent, this approach that can also be used to build on KerML to create more specific modeling languages. Application specific modeling languages can be built on KerML by extending the KerML abstract syntax, specializing its semantics, with concrete syntaxes similar to or entirely different from KerML's.

To support this, the KerML Semantic Library also includes additional library models beyond those directly providing semantics for KerML syntactic constructs, capturing typical semantic patterns (such as asynchronous transfers and state-based behavior) that can be reused by languages built on KerML. Specialized modeling languages can provide additional syntax for these libraries, tailored to their applications, with semantics based largely or entirely on the KerML libraries.

In this way, KerML can provide the kernel for a family of syntactically diverse but semantically integrated modeling languages.



## 2 Conformance

This specification defines the Kernel Modeling Language (KerML), a language used to construct *models* of (real or virtual, planned or imagined) things. The specification includes this document and the content of the machine-readable files listed on the cover page. If there are any conflicts between this document and the machine-readable files, the machine-readable files take precedence.

A *KerML model* shall conform to this specification only if it can be represented according to the syntactic requirements specified in [Clause](#). The model may be represented in a form consistent with the requirements for the KerML concrete syntax, in which case it can be parsed (as specified in [Clause](#)) into an abstract syntax form, or may be represented only in an abstract syntax form (see also [Clause](#) and [Clause](#)).

A *KerML modeling tool* is a software application that creates, manages, analyzes, visualizes, executes or performs other services on KerML models. A tool can conform to this specification in one or more of the following ways.

1. *Abstract Syntax Conformance.* A tool demonstrating Abstract Syntax Conformance provides a user interface and/or API that enables instances of KerML abstract syntax metaclasses to be created, read, updated, and deleted. The tool must also provide a way to validate the well-formedness of models that corresponds to the constraints defined in the KerML metamodel. A well-formed model represented according to the abstract syntax is syntactically conformant to KerML as defined above. (See [Clause](#).)
2. *Concrete Syntax Conformance.* A tool demonstrating Concrete Syntax Conformance provides a user interface and/or API that enables instances of KerML concrete syntax notation to be created, read, updated, and deleted. Note that a conforming tool may also provide the ability to create, read, update and delete additional notational elements that are not defined in KerML. Concrete Syntax Conformance implies Abstract Syntax Conformance, in that creating models in the concrete syntax acts as a user interface for the abstract syntax. However, a tool demonstrating Concrete Syntax Conformance need not represent a model internally in exactly the form modeled for the abstract syntax in this specification. (See [Clause](#).)
3. *Semantic Conformance.* A tool demonstrating Semantic Conformance provides a demonstrable way to interpret a syntactically conformant model (as defined above) according to the KerML semantics, e.g., via model execution, simulation, or reasoning, when and only when such interpretations are possible. Semantic Conformance implies Abstract Syntax Conformance, in that the semantics for KerML are only defined on models represented in the abstract syntax. (See [Clause](#) and [Clause 9](#). See also [6.1](#) for further discussion of the interpretation of models and their syntactic and semantic conformance.)
4. *Model Interchange Conformance.* A tool demonstrating model interchange conformance can import and/or export syntactically conformant KerML models (as defined above) in one or more of the formats specified in [Clause 10](#).

Every conformant KerML modeling tool shall demonstrate at least Abstract Syntax Conformance and Model Interchange Conformance. In addition, such a tool may demonstrate Concrete Syntax Conformance and/or Semantic Conformance, both of which are dependent on Abstract Syntax Conformance.

For a tool to demonstrate any of the above forms of conformance, it is sufficient that the tool pass the relevant tests from the Conformance Test Suite specified in [Annex A](#).





## 3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification.

[Alf] *Action Language for Foundational UML (Alf)*, Version 1.1  
<https://www.omg.org/spec/ALF/1.1>

[fUML] *Semantics of a Foundational Subset for Executable UML Models (fUML)*, Version 1.4  
<https://www.omg.org/spec/fUML/1.4>

[MOF] *Meta Object Facility*, Version 2.5.1  
<https://www.omg.org/spec/MOF/2.5.1>

[OCL] *Object Constraint Language*, Version 2.4  
<https://www.omg.org/spec/OCL/2.4>

[SMOF] *MOF Support for Semantic Structures*, Version 1.0  
<https://www.omg.org/spec/SMOF/1.0>

[SysAPI] *Systems Modeling Application Programming Interface (API) and Services*  
(as submitted contemporaneously with this proposed KerML specification)

[UUID] *A Universally Unique Identifier (UUID) URN Namespace*  
<https://tools.ietf.org/html/rfc4122>

[XMI] *XML Metadata Interchange*, Version 2.5.1  
<https://www.omg.org/spec/XMI/2.5.1>



## 4 Terms and Definitions

There are no terms and definitions specific to this specification.



## 5 Symbols

There are no symbols defined in this specification.



# 6 Introduction

## 6.1 Language Architecture

Developing systems generally involves creating a number of different specifications. For instance, a requirements specification gives the intended effects of a system, while a design specification determines how the system will bring about those effects. Many designs might be developed and evaluated against the same requirements. A test specification then describes test procedures that check whether requirements are met by real or virtual systems built and operated according to some design.

A *model* is a representation in some *modeling language* of all or part of any of the above kinds of system specification. The *semantics* of such models defines what it means for real or virtual things in a modeled system to conform to the specification given by the model. KerML is a foundational modeling language for expressing various kinds of system models with a consistent semantics.

Syntactically, KerML is divided into three layers, with each layer building increasingly more specific constructs on the previous layer. These layers are, from general to specific:

1. The *Root Layer* includes the most general syntactic constructs for structuring models, such as elements, relationships, annotations, and packaging.
2. The *Core Layer* includes the most general constructs that have semantics based on *classification*.
3. The *Kernel Layer* provides commonly needed modeling capabilities, such as associations and behavior.

The Core Layer grounds KerML semantics by interpreting it using mathematical logic. However, additional semantics are then specified through the relationship of Kernel abstract syntax constructs to model elements in the *Kernel Semantic Library*, which is written in KerML itself. Models expressed in KerML thus essentially reuse elements of the Semantic Library to give them semantics. The Semantic Library models give the basic conditions for the conformance of modeled things to the model, which are then augmented in the user model as appropriate.

Having a consistent specification of semantics helps people interpret models in the same way. In particular, because the Semantic Library models are expressed in the same language as user models, engineers and tool builders can inspect the library models to formally understand what real or virtual effects are actually being specified by their models for systems being modeled. More uniform model interpretation improves communication between everyone involved in modeling, including modelers and tool builders.

## 6.2 Document Organization

The remainder of this document is organized into four major clauses.

- [Clause 7](#) describes KerML from a user point of view, covering all the modeling constructs in the language. It is an informative reference for the normative language specification given in the following three subclauses.
- [Clause 8](#) specifies the normative metamodel for the KerML language. This includes the complete grammar for the concrete syntax, which is a textual notation (see [8.2](#)), the abstract syntax, which is a MOF model (see [8.3](#)), and formal semantics (see [8.4](#)).
- [Clause 9](#) specifies the normative Kernel Model Libraries, each of which is a set of *library models* available to be used in all KerML user models. They include the Semantic Library, which is a set of KerML models used to provide Kernel-layer semantics to user models (see [9.2](#)), the Data Type Library of standard data types (see [9.3](#)) and the Function Library of functions on those data types (see [9.4](#)).
- [Clause 10](#) describes each of the formats that can be used to provide standard interchange of KerML models between tools.

In addition, [Annex A](#) defines the suite of conformance tests that may be used to demonstrate the conformance of a modeling tool to this specification (see also [Clause 2](#)).

**Release Note.** A paragraph marked as a "release note" (like this one) is not to be considered part of the formal specification being proposed. Rather, it is a note describing either material that was not included at the time of this release of the proposed specification, or changes to the specification that are expected before the final submission of the proposal. Such notes will be removed in the final submission as the issues they address are resolved.

**Implementation Note.** A paragraph marked as an "implementation note" (like this one) is also not to be considered part of the formal specification being proposed. Rather, it describes an area in which the proof-of-concept pilot implementation being developed by the submission team is not fully consistent with what is being proposed in the specification as of the time of this submission. These notes will also be removed in the final submission.

## 6.3 Acknowledgements

This specification represents the work of many organizations and individuals. The Kernel Model Language concept, as developed for use with SysML v2, is based on earlier work of the KerML Working Group, which was led by:

- Conrad Bock, US National Institute of Standards and Technology (NIST)
- Charles Galey, Jet Propulsion Laboratory
- Bjorn Cole, Lockheed Martin Corporation

The primary authors of this specification document and the syntactic and library models described in it are:

- Ed Seidewitz, Model Driven Solutions
- Conrad Bock, US National Institute of Standards and Technology (NIST)
- Bjorn Cole, Lockheed Martin Corporation

The specification was formally submitted for standardization by the following organizations:

- 88Solutions Corporation
- Dassault Systèmes
- GfSE e.V.
- IBM
- INCOSE
- Intercax LLC
- Lockheed Martin Corporation
- MITRE
- Model Driven Solutions, Inc.
- PTC
- Simula Research Laboratory AS
- Thematrix Partners LLC

However, work on the specification was also supported by over 120 people in over 60 other organizations that participated in the SysML v2 Submission Team (SST). The following individuals had leadership roles in the SST:

- Manas Bajaj, Intercax LLC (API and services development lead)
- Yves Bernard, Airbus (v1 to v2 transformation co-lead)
- Bjorn Cole, Lockheed Martin Corporation (metamodel development co-lead)
- Sanford Friedenthal, SAF Consulting (SST co-lead, requirements V&V lead)
- Charles Galey, Lockheed Martin Corporation (metamodel development co-lead)
- Karen Ryan, Siemens (metamodel development co-lead)
- Ed Seidewitz, Model Driven Solutions (SST co-lead, pilot implementation lead)
- Tim Weikiens, oose (v1 to v2 transformation co-lead)



The specification was prepared using CATIA No Magic modeling tools and the OpenMBEE system for model publication (<http://www.openmbee.org>), with the invaluable support of the following individuals:

- Tyler Anderson, No Magic/Dassault Systèmes
- Christopher Delp, Jet Propulsion Laboratory
- Ivan Gomes, Twingineer
- Robert Karban, Jet Propulsion Laboratory
- Christopher Klotz, No Magic/Dassault Systèmes
- John Watson, Lightstreet Consulting



# 7 Language Description

(Informative)

## 7.1 Language Description Overview

This clause provides an informative description of KerML. [Clause 8](#) gives the full definition of the KerML metamodel, which is the normative specification for implementing the language. In contrast, the description in this clause focuses on how the various constructs of the language are used, along with the Kernel Model Library (see [Clause 9](#)), to construct models. While non-normative, it is intended to be precise and consistent with the normative specification of the language.

The following subclauses present the language features in each of the Root, Core and Kernel Layers of KerML (as described in 6.1). Each layer is then further subdivided, following a parallel structure to the packaging of the metamodel (see [8.1](#)). Each subclause within a layer includes references to the corresponding concrete syntax, abstract syntax and semantics subclauses from the normative metamodel specification. In this way, the clause can be used as a general reference for KerML as well as a guide for better understanding the formal specification of the metamodel.

This clause contains many examples of the KerML textual notation. In order to distinguish this text from normal body text, the following stylistic conventions are used in this clause.

1. Textual notation appears in "code" font. This includes references to individual element names from both example models (such as `Vehicle` and `wheels`) and the Kernel Model Library (such as `Performance` and `performances`), as well as more extensive model snippets.
2. Keywords appear in **boldface**, both when referenced in-line in body text ("Features are declared using the **feature** keyword.") and when used within complete notation examples.
3. Longer samples of textual notation are written in separate paragraphs, indented relative to body paragraphs.

## 7.2 Root

### 7.2.1 Root Overview

The Root layer provides the most general syntactic capabilities of the language: elements and relationships between them, annotations of elements, and membership of elements in namespaces. These capabilities are the syntactic foundation for structuring models in KerML, but they do not actually represent anything about a modeled system, and so have no semantic specification. The Core and Kernel layers build on the foundation provided by Root to provide constructs with modeling semantics (see [7.3](#) and [7.4](#)).

### 7.2.2 Elements and Relationships

#### 7.2.2.1 Elements and Relationships Overview

*Metamodel references:*

- Concrete syntax, [8.2.3.1](#)
- Abstract syntax, [8.3.2.1](#)
- Semantics, none

*Elements* are the constituents of a model. Some elements represent *relationships* between other elements, known as the *related elements* of the relationship. In general terms, a model is constructed as a graph structure in which relationships form the edges connecting non-relationship elements constituting the nodes. However, since

relationships are themselves elements, it is also possible in KerML for a relationship to be a related element in a relationship and for there to be relationships between relationships.

One of the related elements of a relationship may be the *owning* related element of the Relationship. If the owning related element of a relationship is deleted from a model, then the relationship is also be deleted. Some of the related elements of a relationship (distinct from the owning related element, if any) may be *owned* related elements. If a relationship has owned related elements, then, if the relationship is deleted from a model, all its owned related elements are also deleted.

The *owned relationships* of an element are all those relationships for which the element is the owning related element. The *owned elements* of an element are all those elements that are owned related elements of the owned relationships of the element (notice the extra level of indirection through the owned relationships). The *owning relationship* of an element (if any) is the relationship for which the element is an owned related element (of which the element can have at most one). The *owner* of an element (if any) is the owning related element of the owning relationship of the element (again, notice the extra level of indirection through the owning relationship).

The deletion rules for relationships imply that, if an element is deleted from a model, then all its owned relationships are also deleted and, therefore, all its owned elements. This may result in a further cascade of deletions until all deletion rules are satisfied. An element that has no owner acts as the *root element* of an *ownership tree structure*, such that all elements and relationships in the structure are deleted if the root element is deleted. Deleting any element other than the root element results in the deletion of the entire subtree rooted in that element.

### 7.2.2.2 Elements

Every element has a unique identifier known as its *element ID*. The properties of an element can change over its lifetime, but its element ID does not change after the element is created. An element may also have additional identifiers, its *alias IDs*, which may be assigned for tool-specific purposes.

The KerML notation, however, does not have any provision for specifying element or alias IDs, since these are expected to be managed by the underlying modeling tooling. Instead, an element may also have a *name* and/or a *short name*, by which it can be referenced in the notation. While the language makes no formal distinction between names and short names, the intent is that the name of an element should be fully descriptive, particularly in the context of the definition of the element, while the short name, if given, should be an abbreviated name useful for referring to the element. (For further discussion of naming, see also [7.2.4](#)).

An element in its simplest form, not representing any more specialized modeling construct, is notated using the keyword **element**. The declaration of an Element may also specify a short name or name for it, in that order. The short name is distinguished by being surrounded by the delimiting characters < and >. (The notation does not have any provision for specifying the element ID, since this is expected to be managed by the underlying modeling tooling.)

```
element <e145> MyName;
```

Note that it is not required to specify either a short name or a name for an element. However, unless at least one of these is given, it is not possible to reference the element from elsewhere in the textual notation.

**Release Note.** The KerML notation may be extended in the final submission to allow element and alias IDs to be specified in the notation and used to reference unnamed elements, in order to allow the textual notation to be used as more precise means of model interchange.

Names and short names have the same lexical structure, which has two variants.

1. A *basic name* is one that can be lexically distinguished in itself from other parts of the notation. The initial character of a basic name must be a lowercase letter, an uppercase letter or an underscore. The remaining

characters of a basic name can be any character allowed as an initial character or any digit. However, a reserved keyword may not be used as a name, even though it has the form of a basic name (see [8.2.2.6](#) for the list of reserved words).

```
Vehicle
power_line
```

2. An *unrestricted name* provides a way to represent a name that contains any character. It is represented as a non-empty sequence of characters surrounded by single quotes. The name consists of the characters *within* the single quotes – the single quotes are *not* included as part of the represented name. The characters within the single quotes may not include non-printable characters (including backspace, tab and newline). However, these characters may be included as part of the name itself through use of an escape sequence. In addition, the single quote character or the backslash character may only be included within the name by using an escape sequence.

```
'+'
'circuits in line'
'On/Off Switch'
```

An *escape sequence* is a sequence of two text characters starting with a backslash as an escape character, which actually denotes only a single character (except for the newline escape sequence, which represents however many characters is necessary to represent an end of line in a specific implementation). [Table 4](#) in subclause [8.2.2.3](#) shows the meaning of the allowed escape sequences.

In addition to the declaration notated as above, the representation for an Element may include a *body*, which is a list of *owned* elements delimited by curly braces { ... }. It is a general principle of the KerML textual concrete syntax that the representation of owned elements are nested inside the body of the representation of the owning element. In this way, when the notation for the owning Element is removed in its entirety from the representation of a model, the owned elements are also removed.

It is possible to specify the following owned Elements as part of the body of an Element:

- Owned (generic) relationships (see [7.2.2.3](#)), using the keyword **relationship**. The containing element becomes the owning related element and the sole source for the relationship, with one or more other elements identified as target elements.
- Owned comments (see [7.2.3.2](#)), using the keyword **comment** or the keyword **doc**. The containing element becomes the owning related element for an annotation relationship to the comment or documentation.
- Owned textual representations (see [7.2.3.3](#)), using the keyword **rep** or **language**. The containing element becomes the owning related element for an annotation relationship to the textual representation.
- Owned metadata features (see [7.4.13](#)), using the keyword **metadata** or the symbol @. The containing element becomes the owning related element for the annotation relationship to metadata feature. (Note that metadata annotations are a Kernel layer capability.)

```
element A {
    comment /* Element A is related to element B. */
    relationship to B;
}
element B {
    language "HTML"
    /* <a href="https://plm.elsewhere.com/part?id="1234"/> */
}
```

### 7.2.2.3 Relationships

The related elements of a relationship are divided into *source* and *target* elements. A relationship is said to be *directed* from its *source* elements to its *target* elements. It is allowed for a relationship to have only source or only

target elements. However, by convention, an *undirected* relationship is usually represented as having only target elements.

A relationship must have at least two related elements. A relationship with exactly two related elements is known as a *binary relationship*. A *directed binary relationship* is a binary relationship in which one related element is the source and one is the target. Most specialized kinds of relationship in KerML are directed binary relationships (the principal exceptions being associations and connectors, see [7.4.5](#) and [7.4.6](#)).

A relationship can be declared using the keyword **relationship**. As for a generic Element (see [7.2.2.2](#)), a short name and/or a name may be specified for the relationship. The (unowned) source elements of the relationship are then listed after the keyword **from**, while the target elements are listed after the keyword **to**. It is allowable for a relationship to have only source elements or only target elements, but there must be at least two elements specified across the source and target lists (though some of the target elements may be owned related elements, see below).

```
element <'1'> A;
element <'2'> B;
element <'3'> C;
relationship <'4'> R from '1' to B, C;
```

The top-level Elements of a model are implicitly declared within a *root namespace* (see [7.2.4](#)). If the model is further organized into a complete namespace structure, then elements may be identified using qualified names according to that structure (see [8.2.3.3.4](#) for the rules on the resolution of qualified names).

```
namespace N1 {
  element S;
}
namespace N2 {
  element T;
}
relationship from N1::S to N2::T;
```

A relationship may have a body that specifies the owned related elements of the relationship, which may include any kind of element other than an annotating element (see [7.2.3](#)). If an annotating element (i.e., a comment, textual representation or metadata feature) is included in the body of a relationship, then, rather than being directly an owned related element of the containing relationship, the annotating element is an owned related element of an annotation relationship owned by the containing relationship.

To specify that a relationship has an owning related element, use the nested owned relationship notation (see [7.2.2.2](#)).

```
element A;
element B {
  relationship x {
    element y; // Owned related element
    relationship from A to B; // Relationship as owned related element
  }
}
relationship R from A to B {
  /* This comment is owned via an annotation relationship,
   * so the containing relationship has no owned related elements.
   */
}
```

## 7.2.3 Annotations

### 7.2.3.1 Annotations Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.3.2](#)
- *Abstract syntax*, [8.3.2.2](#)
- *Semantics*, *none*

An *annotation* is a relationship between an *annotated element* and an *annotating element* that provides additional information about the element being annotated. Any kind of element may be annotated, but only certain kinds of elements may be annotating elements. Specific kinds of annotating elements include comments and textual representations (see [7.2.3.2](#) and [7.2.3.3](#)). A further kind of annotating element for user-defined metadata is defined in the Kernel layer (see [7.4.13](#)).

Each annotation relationship is between a single annotating element and a single annotated element, but an annotating element may have multiple annotation relationships with different annotated elements, and any element may have multiple annotations. The annotated element of an annotation can optionally be the owning related element of the annotation, in which case the annotation is an owned annotation of the owning annotated element. If an annotating element is an owned member of a namespace (see [7.2.4](#)) and is not involved in any annotation relationships, then its owning namespace is considered to be its annotated element without the need for an explicit annotation relationship.

### 7.2.3.2 Comments and Documentation

A *comment* is an annotating element with a textual `body` that in some way describes its annotated element. *Documentation* is a kind of comment that has the special status of documenting the annotated element, known in this case as the *documented element*. A documentation comment is always an owned element of its documented element.

The full declaration of a comment begins with the keyword **comment**, optionally followed by a short name and/or name (see [7.2.2.2](#)). One or more annotated elements are then identified for the comment after the keyword **about**, indicating that the comment has annotation relationships to each of the identified elements. The `body` of the comment is written lexically as regular comment text between `/*` and `*/` delimiters (see also [8.2.2.2](#)).

```
element A;  
element B;  
comment Comment1 about A, B  
    /* This is the comment body text. */
```

If the comment is an owned member of a namespace (see [7.2.4](#)), then the explicit identification of annotated elements can be omitted, in which case the annotated element is implicitly the containing namespace. Further, in this case, if no short name or name is given for the comment, then the **comment** keyword can also be omitted.

```
namespace N {  
    comment C /* This is a comment about N. */  
  
    /* This is also a comment about N. */  
}
```

A documentation comment is notated similarly to a regular comment, but using the keyword **doc** rather than **comment**. The documenting element of a documentation comment is always the owning element of the documentation.

```
element X {  
    doc X_Comment  
    /* This is a documentation comment about X. */
```

```

    doc /* This is more documentation about X. */
}
namespace P {
    doc P_Comment /* This is a documentation comment about P. */
}

```

The actual `body` text of a comment does not include the initial `/*` and final `*/` characters. Further, the written text is processed to allow formatting using `*` characters to delimit consistent initial indentation of a comment lines. For example, the comment notation in:

```

namespace CommentExample {
    /*
    * This is an example of multiline
    * comment text with typical formatting
    *   for readable display in a text editor.
    */
}

```

would result in the following `body` text in the comment element in the represented model:

```

This is an example of multiline
comment text with typical formatting
    for readable display in a text editor.

```

The `body` text of a comment can include markup information (such as HTML), and a tool may (but is not required to) display such text as rendered according to the markup. (See [8.2.3.2.2](#) for the complete rules for processing comment text.)

### 7.2.3.3 Textual Representations

A textual representation is an annotating element whose textual `body` represents its annotated element (known in this case as the *represented element*) in a given language. A textual representation is notated similarly to a documentation comment (see [7.2.3.2](#)), but with the keyword **rep** used instead of **comment**. As for documentation, a textual representation is always owned by its represented element. In particular, if the textual representation is an owned member of a namespace (see [7.2.4](#)), the represented element is the containing Namespace. A textual representation declaration must also specify the language used for the textual `body` as a literal string (see [8.2.2.5](#)) following the keyword **language**. If the textual representation has no short name or name, then the **rep** keyword can also be omitted.

```

class C {
    feature x: Real;
    inv x_constraint {
        rep inOCL language "ocl"
        /* self.x > 0.0 */
    }
}
behavior setX(c : C, newX : Real) {
    language "alf"
    /* c.x = newX;
    * WriteLine("Set new x");
    */
}

```

The lexical comment text given for a textual representation is processed as for regular comment text (see [7.2.3.2](#)), and it is the result after such processing that is the textual representation `body` expected to conform to the named language.



**Note.** Since the lexical form of a comment is used to specify the textual representation `body`, it is not possible to include comments of a similar form in the `body` text.

**Release Note.** The final submission may include a means to allow nested comments.

The language name in a textual representation is case insensitive. The name can be of a natural language, but will often be for a machine-parsable language. In particular, there are recognized standard language names.

If the language is `"kerm1"`, then the body of the textual representation must be a legal representation of the represented element in the KerML textual notation. A tool can use such a textual representation to record the original KerML notation text from which an element is parsed. Other standard language names that can be used in a textual representation include `"ocl"` and `"alf"`, in which case the body of the textual representation must be written in the Object Constraint Language [OCL] or the Action Language for fUML [Alf], respectively.

However, for any other language than `"kerm1"`, the KerML does not define how the `body` text is to be semantically interpreted as part of the model being represented. In particular, a generic element (e.g., as in [7.2.2.2](#)) with a textual representation in a language other than KerML is essentially a semantically "opaque" Element specified in the other language. Nevertheless, a conforming KerML tool may (be is not required to) interpret such an element consistently with the specification of the named language.

## 7.2.4 Namespaces

### 7.2.4.1 Namespaces Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.3.3](#)
- *Abstract syntax*, [8.3.2.3](#)
- *Semantics*, *none*

A *namespace* is an element that contains other elements via *membership* relationships with those elements. The namespace is the source element and owner of the membership. The target of a membership can be any kind of element, known as the *member element* of the membership. If the membership is an *owning* membership, then the member element is known as an *owned* member element, which is the only owned related element of the membership.

A namespace may also *import* memberships from other namespaces. Further, a type, which is kind of namespace, may *inherit* memberships from other types that it specializes (see [7.3.2](#)).

The *members* of a namespace are the member elements of all the memberships of the namespace (whether owned, imported or inherited). The *owned members* of a namespace are the owned member elements of all the owned memberships of the namespace that are owning memberships.

If an element is a member of a namespace, then any name for that element relative to the namespace is known as an *unqualified name* for that element in the namespace. If the containing namespace is not a root namespace (see [7.2.4.3](#)), then the *qualified name* for the member element consists of a name for the containing namespace, known as the *qualifier*, followed by an unqualified name for the element. Since a namespace is an element that may itself be a member of another namespace, a qualifier may be a qualified name. Therefore, a qualified name of an element, in general, has the form of a list of unqualified names of namespaces, each relative to the previous one, followed by the unqualified name of the element in the final namespace.

A qualified name is notated as a sequence of *segment names* separated by `" : "` punctuation. An *unqualified* name can be considered the degenerate case of a qualified name with a single segment name. A qualified name is used in the KerML textual concrete syntax to identify an element that is being referred to in the representation of another

element. A qualified name used in this way does not appear in the corresponding abstract syntax—instead, the abstract syntax representation contains an actual reference to the identified element. *Name resolution* is the process of determining the element that is identified by a qualified name (see [8.2.3.3.4](#)).

Since namespaces their members may have aliases (see [7.2.4.2](#)), it is possible for there to be multiple qualified names for an element even if it does not itself have aliases. On the other hand, if a namespace does not have any name, then its members will have no qualified names, even if they are themselves named.

#### 7.2.4.2 Namespace Declaration

A namespace that is not a root namespace (see [7.2.4.3](#)), and does not represent any more specialized modeling construct (such as a type—see [7.3.2](#)) is declared using the keyword **namespace**, optionally followed by a short name and/or name (see [7.2.2.2](#)). The *body* of the namespace is notated as a list of representations of the content of the namespace delimited between curly braces {...}. If the namespace is empty, then the body may be omitted and the declaration ended instead with a semicolon.

```
namespace <'1.1'> N1; // This is an empty namespace.
namespace <'1.2'> N2 {
    doc /* This is an example of a namespace body. */
    class C;
    datatype D;
    feature f : C;
    namespace N3; // This is a nested namespace.
}
```

Declaring an element within the body of a namespace denotes that the element is an owned member of the namespace—that is, that there is an owning membership relationship between the namespace and the member element.

The *visibility* of the membership can be specified by placing one of the keywords **public**, **protected** or **private** before the public element declaration. If the membership is **public** (the default), then it is visible outside of the namespace. If it is **private**, then it is not visible. For namespaces other than types, **protected** visibility is equivalent to **private**. For types, **protected** visibility has a special meaning relating to member inheritance (see [7.3.2](#)).

```
namespace N {
    public class C;
    private datatype D;
    feature f : C; // public by default
}
```

An *alias* for an element is non-owning membership of the element in a namespace, which may or may not be the same namespace that owns the element. An alias name or short name are determined only relative to its membership in the namespace, and can therefore be different than the name or short name defined on the element itself. Note that the same element may be related to a namespace by multiple alias memberships, allowing the element to have multiple, different names relative to that namespace.

An alias is declared using the keyword **alias** followed by the alias short name and/or name, with a qualified name identifying the element given after the keyword **for**. The alias declaration may optionally include a body with the same syntax as for a generic **relationship** declaration (see [7.2.2.3](#)). The visibility of the alias membership can be specified as for an owned member.

```
namespace N1 {
    class A;
    class B;
    alias <C> CCC for B {
```

```

        doc /* Documentation of the alias. */
    }
    private alias D for B;
}

```

A comment (see [7.2.3.2](#)), including documentation, declared within a namespace body also becomes an owned member of the namespace. If no annotated elements are specified for the comment (with an **about** clause), then, by default, the comment is considered to be about the containing namespace.

```

namespace N9 {
    class A;
    comment Comment1 about A
        /* This is a comment about class A. */

    comment Comment 2
        /* This is a comment about namespace N9. */

    /* This is also a comment about namespace N9. */

    doc N9_Doc
        /* This is documentation about namespace N9. */
}

```

With the ability to specify names, short names and aliases for elements, any element can potentially have several names relative to a namespace. However, the set of names provided for any one member of a namespace must be distinct from the set of names provided for any other member of the namespace. That is, a namespace effectively provides a "space" of names, each one of which uniquely identifies a single member element of the namespace (though there may be multiple names that identify the same element).

### 7.2.4.3 Root Namespaces

A *root namespace* is a namespace that has no owner. The owned members of a root namespace are known as *top-level elements*. Any element that is not a root namespace has an owner and, therefore, must be in the ownership tree of a top-level element of some root namespace.

The declaration of a root namespace is implicit and no identification of it is provided in the KerML textual notation. Instead, the body of a root namespace is given simply by the list of representations of its top-level elements.

```

doc /* This is a model notated in KerML concrete syntax. */
element A {
    relationship B to C;
}
class C;
datatype D;
feature f: C;
package P;

```

Since the notation does not provide a means for naming a root namespace, the name of a top-level element is *not* qualified by the name of its containing root namespace. The name resolution rules consider all top-level elements to be directly and globally visible without qualification (see [8.2.3.3.4](#)). Therefore, the *fully qualified* name of an element relative to a root namespace always begins with the name of a top-level element in the root namespace, without regard to the name (if any) of the root namespace.

### 7.2.4.4 Imports

A namespace may *import* the visible memberships from other namespaces. The namespace that is the source of an import relationship also owns it. The namespace that is the target of an import relationship is known as

the *imported* namespace. Visible memberships of an imported namespace become *imported* memberships of the owning namespace of the import relationship.

The complete set of memberships of a namespace include all its owned memberships and all its imported memberships, and the member elements of imported memberships are included in the set of members of the namespace. Various kinds of namespaces may also define additional memberships to be included in the set of memberships of that kind of namespace (for instance, the memberships of a type also include its *inherited* members – see [7.3.2](#)) and which of those are visible (e.g., public inherited memberships).

An import is denoted using the keyword **import** followed by a qualified name. This specifies an import whose imported namespace is identified by the qualification part of the qualified name and whose *imported member name* is given by the the unqualified name. If the name given for the **import** is unqualified, then there is no imported namespace, and the given name shall be resolved in the scope of the namespace owning the import (see [8.2.3.3.4](#)).

Such an import results in the membership of the imported namespace whose member name or short name is the given imported member name becoming an imported membership of the importing namespace. That is, the member element of this membership becomes an imported member of the imported namespace. Note that the imported member name may be an alias of the import element in the imported namespace, in which case the element is still imported with that name.

```
namespace N2 {
  import N1::A;
  import N1::C; // Imported with name "C".
  namespace M {
    import C; // "C" is re-imported from N2 into M.
  }
}
```

If the qualified name in an import is followed by " : : \*", then the entire qualified name identifies the imported namespace. In this case, all visible memberships of the imported namespace become imported memberships of the importing namespace.

```
namespace N3 {
  // Memberships A, B and C are all imported from N1.
  import N1::*;
}
```

If the qualified name of an **import**, with or without a " : : \*", is further followed by " : : \*\*", then the import is *recursive*. Such an import is equivalent to importing all memberships as described above, followed by further recursively importing from each imported member that is itself a namespace.

```
namespace N4 {
  class A;
  class B;
  namespace M {
    class C;
  }
}
namespace N5 {
  import N4::**;
  // The above recursive import is equivalent to all
  // of the following taken together:
  //   import N4;
  //   import N4::*;
  //   import N4::M::*;
}
```

```

namespace N6 {
  import N4::*::*;
  // The above recursive import is equivalent to all
  // of the following taken together:
  //   import N4::*;
  //   import N4::M::*;
  // (Note that N4 itself is not imported.)
}

```

The *visibility* of an import can be specified by placing the keyword **public** or **private** before the import declaration. If the import is **public** (the default), then all the imported memberships become public for the importing namespace. If import is **private**, then the imported memberships become private relative to the importing namespace. An import declaration may optionally have a body, with the same syntax as for a generic **relationship** declaration (see [7.2.2.3](#)).

```

namespace N7 {
  public import N1::A {
    /* The imported membership is visible outside N7. */
  }

  private import N4::* {
    doc /* None of the imported memberships are visible
       * outside of N7. */
  }
}

```

An import may also be declared with one or more *filter conditions*. Given as model-level evaluable Boolean expressions (see [7.4.9](#)), listed after the imported namespace specification, each surrounded by square brackets [...]. Such a filtered import is equivalent to importing an implicit package that then both imports the given imported namespace and has all the given filter conditions. The effect is such that, for a filtered import, memberships are imported from if and only if they satisfy all the given filter conditions. (While filtered imports may be used in any namespace, packages and filter conditions are actually Kernel-layer concepts, because expressions are only defined in that layer. See [7.4.14](#).)

```

namespace N8 {
  import Annotations::*;

  // Only import elements of NA that are annotated as Approved.
  import NA::*[@Approved];
}

```

If the member name or member short name of any imported membership conflicts with the name of any owned member, or with the name of any visible membership from any other imported namespace, then the conflicting membership is *hidden* and is not included in the set of imported memberships of the importing namespace. As a result of this rule and the distinguishability rule for owned members, the names of all owned and imported members will always be distinct from each other. Any specialized kind of namespace that adds further kinds of memberships (e.g., inherited memberships of types) always maintains the property that the names of all memberships of a namespace are distinct from each other.

**Implementation Note.** The pilot implementation does not current check to see if one imported membership is hidden by another imported membership. Instead, if there are two imported memberships with the same name, and they are not hidden by an owned membership (or inherited membership for a type), name resolution will find first of the imported memberships with that name.

**Release Note.** The current rules for membership distinguishability in a namespace require that all names be distinct from each other. This may be loosened in the final submission to allow overloading of behavioral elements with the same name when these can be distinguished by having different parameter signatures.

## 7.3 Core

### 7.3.1 Core Overview

The Core layer builds on the Root layer to add the minimum constructs for modeling systems as designed, built and operated. *Semantics* is about how models are interpreted as giving conditions on how things should be (i.e., as a *specification* of a modeled system) or as a reflection of how things are (i.e., as a *description* of a modeled system). KerML semantics are based on *classification*: a model has elements that classify things in the modeled system.

A *type* is the most general kind of model element that classifies things (see [8.2.4.1.1](#)). *Classifiers* are types that classify things, such as cars, people and processes being carried out, as well as how they are related by features (see [7.3.3](#)). *Features* are also types, classifying relations between things (see [8.2.4.3.1](#)). In addition to simple relations between two things, KerML allows features to classify longer *chains* of relations. For example, cars owned by people who live in a particular city might be required to be registered. These cars are identified by a chain of two relations, first ownership of the car, then the residence of the owner.

KerML also supports taxonomies of classifications using *specialization* relationships between types. All the things classified by a specialized type are also classified by the general types it is related to via specialization relationships. This means that all the things classified by a specialized type have all the features of its general types, referred to as *inheriting* features from general to specific types. KerML includes several special kinds of specialization, including *subclassification* between classifiers, *subsetting* and *redefinition* between features, and *feature typing* between a feature and another type.

### 7.3.2 Types

#### 7.3.2.1 Types Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.4.1](#)
- *Abstract syntax*, [8.3.3.1](#)
- *Semantics*, [8.4.2.2](#)

*Types* classify things in a modeled system. The set of things classified by a type is the *extent* of the type, each member of which is an *instance* of the type. Everything being modeled is an instance of the type `Anything` from the `Base` library model (see [9.2.2](#)).

Types give conditions for what things must be in their extent and what must not be (*sufficient* and *necessary* conditions, respectively). The simplest conditions directly identify instances that must be in or not in the extent. Other conditions can give characteristics of instances indicating they must be in or not in the extent. These conditions apply to all procedures that determine the extents of types, including logical solving, inference, and execution

For example, a type `Car` could require every instance in its extent (everything it classifies) to have four wheels, which means anything that does not have four wheels is not in its extent (necessary condition). It does not mean all four wheeled things are in the extent (are cars), however. (Note that necessary conditions are usually stated as what must be true of all instances in the extent, even though they really only determine what is not.) Alternatively, `Car` could require all four wheeled things to be in its extent (sufficient condition).

Types are namespaces, enabling them to have members via membership relationships to other elements identified as their members (see [7.2.4](#)). These include *inherited* memberships, which are certain memberships from the general types of their *owned specializations* (see [7.3.2.3](#)). The member names of all inherited memberships must be distinct from each other and from the member names of all owned memberships. A membership that would otherwise be imported is also hidden by an inherited memberships with the same member name, just as in the case of an owned membership (see [7.2.4](#)).

**Note.** Name conflicts due to inherited memberships can be resolved by redefining them to give non-conflicting member names (see [7.3.4](#)).

### 7.3.2.2 Type Declaration

A type is declared using the keyword **type**, optionally followed by a short name and/or name. In addition, a type declaration defines either one or more *owned specializations* for the type (see [7.3.2.3](#)) or a *conjugator* for the type (see [7.3.2.4](#)). This may optionally be followed by the definition of one or more *owned disjointings* (see [7.3.2.5](#)).

```
type A specializes Base::Anything disjoint from B;
type C conjugates
```

A type is specified as *abstract* by placing the keyword **abstract** before the keyword **type**, which means that all instances of a type must also be instances of at least one (possibly indirect) specialized type (which must not be abstract, that is, must be *concrete*).

```
abstract type A specializes Base::Anything;
type A1 specializes A;
type A2 specializes A;
```

The multiplicity constrains the number of instances in the extent of a type (the *cardinality* of the extent). A multiplicity is a feature whose values are natural numbers (extended with infinity, see [9.3.2.1](#)) that are the only ones allowed for the cardinality of its featuring type (each multiplicity is the feature of exactly one Type). A type can have at most one feature that is its multiplicity. Cardinality for classifiers is the number of things it classifies. For features that are not end features (see below), cardinality is the number of values of the feature for a specific instance of its featuring types.

**Note.** The semantics of multiplicity is different for features that are identified as *end features*. End Features are used primarily in the definition of associations and connectors, and the semantics of end features is discussed in conjunction with them (see [7.4.5](#) and [7.4.6](#), respectively).

The multiplicity of a type can be specified as a *range* after any identification of the Type, between square brackets [...]. (See [7.4.12](#) for a complete description of multiplicity ranges, including declaring named multiplicity features.)

```
// This Type has exactly one instance.
type Singleton[1] specializes Base::Anything;
```

The body of a type is specified as for a generic namespace, by listing the members between curly braces {...} (see [7.2.4.2](#)). However, for types, *protected* members, indicated using the keyword **protected** instead of **public** or **private**, have special visibility rules for inheritance (see [7.3.2.3](#)). A feature declared as an owned member of a type is automatically considered to be an *owned feature* of the type, related by a *feature membership*, unless its declaration is preceded by the keyword **member**, in which case it is related by regular membership (see [7.3.2.6](#) for details).

```
type Super specializes Base::Anything {
  private namespace N {
    type Sub specializes Super;
  }
}
```



```

    protected feature f : N::Sub;
    member feature f1 : Super featured by N::Sub;
}

```

The conditions that a type places on its instances (e.g., what feature it has) are always considered *necessary*. They can be indicated as *sufficient* by placing the keyword **all** after the keyword **type**. In this case, the type places additional sufficiency conditions on its instances corresponding to all the necessary conditions. For example, if `Car` requires all instances to be four-wheeled (necessary), and then is also indicated as sufficient, its extent will include all four wheeled things and no others. (See also the discussion in [7.3.2.1](#).)

```

type all Car specializes MaterialThing {
    feature wheels[4] : Wheel;
}

```

### 7.3.2.3 Specialization

*Specializations* are relationships between types, identified as *specific* and *general*, indicating that all instances of the specific type are instances of the *general* one (that is, the extent of the specific type is a subset of the extent of the general one, which might be the same set). This means instances of the specific type have all the features of the general one, referred to syntactically as *inheriting* features from general to specific types. A type may participate in multiple specialization relationships, both as specific and general types.

A specialization relationship is declared using the keyword **specialization**, optionally followed by a short name and/or a name. The qualified name of the specific type, or a feature chain (see [7.3.4.6](#)) if the specific type is such a feature, is then given after the keyword **subtype**, followed by the qualified name of the general type, or a feature chain if the general type is such a feature, after the keyword **specializes**. The symbol `:>` can be used interchangeably with the keyword **specializes**. A specialization declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```

specialization Gen subtype A specializes B;
specialization subtype x :> Base::things {
    doc /* This specialization is unnamed. */
}

```

If no `shortName` or `name` is given, then the keyword **specialization** may be omitted.

```

subtype C specializes A;
subtype C specializes B;

```

The *direct supertypes* of a type are all the general types in specializations for which the type is the specific type, and the *direct subtypes* of a type are all the specific types in specializations for which the type is the general type. *Indirect supertypes* include, recursively, the supertypes of the direct supertypes of a type, and similarly for *indirect subtypes*.

Specialization relationships can form cycles, which means all types in the cycle have the same instances (same extent). However, since all types are required to specialize the base type `Anything` (directly or indirectly), no cycle of valid types can be entirely closed, unless it includes the type `Anything`.

The *owned specializations* of a type are those specializations that are owned relationships of the type (see [Clause](#)), for which the type is the *specific* type. An owned specification of a type is defined as part of the declaration of the type, rather than in a separate declaration, by including the qualified name or feature chain of the general type in a list after the keyword **specializes** (or the symbol `:>`).

```

type C specializes A, B;
type f :> Base::things;

```



A type *inherits* all visible and protected memberships of the general types of its owned specializations. *Protected* memberships are all owned and inherited memberships of the general type whose visibility declared as **protected** (see also [7.3.2.2](#) on **protected** visibility; for imported memberships, protected visibility is equivalent to private). This means protected memberships are memberships that are only visible to their owning type and to (direct or indirect) specializations of it.

```
type A specializes Base::Anything {
  feature f; // Public by default.
  protected feature g;
  private feature h;
}
type B specializes A {
  // B inherits feature memberships for
  // f and g, but not h.
}
```

### 7.3.2.4 Conjugation

*Conjugation* is a relationship between types, identified as the *original* type and the *conjugated* type, indicating the conjugated type inherits visible and protected memberships from the original type, except the direction of input and output features is reversed (see [7.3.4.1](#) on features with direction). Features with direction **in** relative to the original type are treated as having direction **out** relative to the conjugated type, and vice versa for direction **in** treated as **out**. Features with no direction or direction **inout** in the original type are inherited without change.

A conjugation relationship is declared using the keyword **conjugation**, followed by a short name and/or a name. The qualified name of the conjugated type, or a feature chain (see [7.3.4.6](#)) if the conjugated type is such a feature, is then given after the keyword **conjugate**, followed by the qualified name of the original type, or a feature chain if the original type is such a feature, after the keyword **conjugates**. The symbol ~ can be used interchangeably with the keyword **conjugates**. A conjugation declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```
type Original specializes Base::Anything {
  in feature Input;
}
type Conjugate1 specializes Base::Anything;
type Conjugate2 specializes Base::Anything;
conjugation c1 conjugate Conjugate1 conjugates Original;
conjugation c2 conjugate Conjugate2 ~ Original {
  doc /* This conjugation is equivalent to c1. */
}
```

If no short name or name is given, then the keyword **conjugation** may be omitted.

```
conjugate Conjugate1 conjugates Original;
conjugate Conjugate2 ~ Original;
```

An *owned conjugation* is an owned relationship of a type ([7.2.2](#)) that is a conjugation relationship, for which the type is the *conjugated* type. An owned conjugation for a type is defined as part of the declaration of the type, rather than in a separate declaration, by including the qualified name or feature chain of the original type after the keyword **conjugates** (or the symbol ~).

```
type Conjugate1 conjugates Original;
type Conjugate2 ~ Conjugate1;
```

A type can be the conjugated type of at most one conjugation relationship, and a conjugated type cannot be the specific type in any specialization relationship.

### 7.3.2.5 Disjoining

Types related by *disjoining* do not share instances (instances cannot be in more than one of the extents; the extents are *disjoint*). For example, a classifier for mammals is disjoint from a classifier for minerals, and a feature for people's parents is disjoint from a feature for their children.

A disjoining relationship is declared using the keyword **disjoining**, optionally followed by a short name and/or a name. The qualified name of the first type, or a feature chain (see [7.3.4.6](#)) if the type is such a feature, is then given after the keyword **disjoint**, followed by the qualified name of the second type, or a feature chain, if the the type is such a feature, after the keyword **from**. A disjoining declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```
disjoining Disj disjoint A from B;
disjoining disjoint Mammal from Mineral;
disjoining disjoint Person::parents from Person::children {
  doc /* No Person can be their own parent. */
}
```

If no short name or name is given, then the keyword **disjoining** may be omitted.

```
disjoint A from B;
disjoint Mammal from Mineral;
disjoint Person::parents from Person::children;
```

An *owned disjoining* of a type is an owned relationship of the type (see [7.2.2](#)) that is a disjoining relationship. An owned disjoining is defined as part of the declaration of the type, rather than in a separate declaration, by including the qualified name or feature chain of the disjoining type in a list after the keyword **disjoint from**.

```
type C disjoint from A, B;
type Mammal disjoint from Mineral;
```

### 7.3.2.6 Feature Membership

A *feature membership* is a relationship between a type and a feature that is both a kind of owning membership and a kind of *type featuring* (see [7.3.4.8](#)). Features related to a type via feature membership are identified as *owned features of the type*. The owning type is one of the feature's featuring types, meaning that the feature specifies a relation between the owning type and the type of the feature.

A feature that is declared within the body of a type is normally an owned feature of that type, so it automatically has that type as a featuring type (because feature membership is a kind of type featuring). This also applies to the bodies of classifiers (see [7.3.3](#)) and features (see [7.3.4](#)), since they are kinds of types. A feature may also be aliased in a type like any other Element (see [7.2.4](#)), in which case it is related to the aliasing type by a regular membership relationship, not a feature membership, and, so, does not become one of the owned features of the type.

```
feature person[*] : Person;
classifier Person {
  // This declares an owned feature using a feature membership.
  feature age[1] : ScalarValues::Integer;

  // This is not a feature membership.
  alias personAlias for person;
}
```

However, if a feature declaration in the body of type is preceded by the keyword **member**, then the feature is owned by the containing type via a membership relationship, not a feature membership. In this case, the feature is *not* an owned feature of the containing type, and it does *not* automatically have the containing type as a featuring type,

though it may have featuring types declared in its **featured by** list (see [7.3.4.1](#) on declaring the owned typings of a feature).

```
classifier A;
classifier B {
  // Feature f has B as its featuring type.
  feature f;

  // Feature g has A as its featuring type, not B.
  member feature g featured by A;
}
```

## 7.3.3 Classifiers

### 7.3.3.1 Classifiers Overview

*Metamodel references:*

- Concrete syntax, [8.2.4.2](#)
- Abstract syntax, [8.3.3.2](#)
- Semantics, [8.4.2.3](#)

*Classifiers* are types that classify things in the modeled system, as distinct from features, which model the relations between them (see [7.3.4](#)). *Subclassification* is a kind of specialization that specifically relates classifiers.

### 7.3.3.2 Classifier Declaration

The notation for a classifier is the same as the generic notation for a type (see [7.3.2.2](#)), except using the keyword **classifier** rather than **type**. However, any general types referenced in a **specializes** list must be classifiers, and the specializations defined are specifically *subclassifications* (see [7.3.3.3](#)).

A classifier is also not required to have any owned subclassifications explicitly specified. If no explicit subclassification is given for a classifier, and the classifier is not conjugated, then the classifier is given a default subclassification to the most general base classifier `Anything` from the Base library model (see [9.2.2](#)).

```
classifier Person { // Default superclassifier is Base::Anything.
  feature age : ScalarValues::Integer;
}
classifier Child specializes Person;
```

The declaration of a classifier may also specify that the classifier is a conjugated type (see [7.3.2.4](#)), in which case the original type must also be a classifier.

```
classifier FuelInPort {
  in feature fuelFlow : Fuel;
}
classifier FuelOutPort conjugates FuelInPort;
```

### 7.3.3.3 Subclassification

A subclassification relationship is declared using the keyword **specialization**, optionally followed by a short name and/or a name. The qualified name of the *subclassifier* is then given after the keyword **subclassifier**, followed by the qualified name of the *superclassifier* after the keyword **specializes**. The symbol `:>` can be used interchangeably with the keyword **specializes**. A subclassification declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```

specialization Super subclassifier A specializes B;
specialization subclassifier B :> A {
    /* This subclassification is unnamed. */
}

```

If no short name or name is given, then the keyword **specialization** may be omitted.

```

subclassifier C specializes A;
subclassifier C specializes B;

```

An owned subclassification of a Classifier is defined as part of the declaration of the Classifier, rather than in a separate declaration, by including the qualified name of the `superclassifier` in a list after the keyword **specializes** (or the symbol `:>`).

```

classifier C specializes A, B;

```

## 7.3.4 Features

### 7.3.4.1 Features Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.4.3](#)
- *Abstract syntax*, [8.3.3.3](#)
- *Semantics*, [8.4.2.4](#)

*Features* are types that classify how things in a modeled system are related, including by chains of relations. Relations between things can also be treated as things, allowing relations between relations, recurring as many times as needed. A feature relates instances in the intersection of the extents of its *featuring types* (the *domain*) with instances in the intersection of the extents of its *featured types* (the *co-domain*). Instances in the domain of a feature are said to "have values" that are instances of the co-domain. The domain of features with no explicit featuring types is the type `Anything` from the `Base` library model (see [9.2.2](#)).

*Type featuring* is a relationship between a feature and a type that identifies the type as a featuring type of the feature. *Feature membership* is both a kind of owning membership and a kind of type featuring, by which a type owns a feature and becomes a featuring type of that feature (see [7.3.2.6](#)).

There are also several forms of specialization that apply specifically to features.

- *Feature typing* is a relationship between a feature and a type that identifies the type as a featured type of the feature.
- *Subsetting* is a relationship between a specific feature (the *subsetting feature*) and a more general feature (the *subsetted feature*), where the specific feature may further constrain the featuring types, featured types and multiplicity of the general feature.
- *Redefinition* is a kind of subsetting in which the specific feature (the *redefining feature*) also replaces an otherwise inherited general feature (the *redefined feature*) in the context of the owning type of the specific feature.

### 7.3.4.2 Feature Declaration

The notation for a feature is similar to the generic notation for a type (see [7.3.2.2](#)), except using the keyword **feature** rather than **type**. Further, a feature can have any of three kinds of specialization, each identified by a specific keyword or equivalent symbol:

- **typed by** or **:** – Specifies `FeatureTyping` (see [7.3.4.3](#)).

- **subsets** or **:>** – Specifies Subsetting (see [7.3.4.4](#)).
- **redefines** or **:>>** – Specifies Redefinition (see [7.3.4.5](#)).

In general, clauses for the different kinds of Specialization can appear in any order in a Feature declaration.

```
feature x typed by A, B subsets f redefines g;

// Equivalent declaration:
feature x redefines g typed by A subsets f typed by B;
```

If no subsetting (or redefinition) is explicitly specified for a feature, and the feature is not conjugated, then the feature is given a default subsetting of the most general base feature `things` from the `Base` library model (see [9.2.2](#)). This is true even if a feature typing is given for the feature.

```
abstract feature person : Person; // Default subsets Base::things.
feature child subsets person;
```

The declaration of a feature may also specify that the feature is a conjugated type (see [7.3.2.4](#)), in which case the original type must also be a feature. In this case, the feature must not have any owned specializations.

```
classifier Tanks {
  port feature fuelInPort {
    in feature fuelFlow : Fuel;
  }
  port feature fuelOutPort ~ fuelInPort;
}
```

As for any type, the multiplicity of a feature can be given in square brackets [...] after any identification of the feature (see also [7.3.2.2](#)). However, the multiplicity for a feature can also be placed *after* one of the specialization clauses in the feature declaration (but, in all cases, only one multiplicity may be specified). In particular, this allows a notation style for multiplicity consistent with that used in previous modeling languages (such as [UML]). It is also useful when redefining a Feature without giving an explicit name (see [7.3.4.5](#)).

```
feature parent[2] : Person;
feature mother : Person[1] :> parent;
feature redefines children[0];
```

In addition to, or instead of, an explicit multiplicity, a feature declaration can include either or both of the following keywords (in either order). The properties flagged by these keywords are only meaningful if the feature has a multiplicity upper bound greater than one.

- **nonunique** – If a feature is *non-unique*, then, for any domain instance, the same co-domain instance may appear more than once as a value of the feature. The default is that the feature is *unique*.
- **ordered** – If a feature is *ordered*, then for any domain instance, the values of the featured can be placed in order, indexed from 1 to the number of values. The default is that the feature is *unordered*.

```
feature sensorReadings : ScalarValues::Real [*] nonunique ordered;
```

There are four other kinds of relationships that can be declared as owned relationships of a feature, each indicated by a specific keyword:

- **disjoint from** – Specifies disjoining (see [7.3.2.5](#)).
- **chains** – Specifies feature chaining (see [7.3.4.6](#)).
- **inverse of** – Specifies feature inverting (see [7.3.4.7](#)).
- **featured by** – Specifies type featuring (see [7.3.4.7](#)).

The clauses for these relationships must appear after any specialization or conjugation part, but can otherwise appear in any order.

```
feature cousins : Person[*] chains parents.siblings.children featured by Person;  
feature children : Person[*] featured by Person inverse of parents;
```

There are a number of additional properties of a feature that can be flagged by adding specific keywords to its declaration. If present, these are always specified in the following order, before the keyword **feature**:

1. **in**, **out**, **inout** – Specifies the *direction* of a feature, which determines what is allowed to change its values on instances of its domain:
  - **in** – Things "outside" the instance. These features identify things input to an instance.
  - **out** – The instance itself or things "inside" it. These features identify things output by an instance.
  - **inout** – Both things "outside" and "inside" the instance. These features identify things that are both input to and output by an instance.
2. **abstract** – Specifies that the feature is *abstract* (see [7.3.2.2](#) on abstract types in general).
3. **composite** or **portion** – Specifies that the feature is either a *composite* or *portion* feature (specifying both is not allowed).
  - Values of composite features on each instance of their domain cannot exist after that instance does. This only applies to values at the time the instance goes out of existence, not to other things in the co-domain that might have been values before that.
  - Portion features (re composite features where the values cannot exist without the whole, because they are the “same thing” as the whole. (For example, the portion of a person's life when they are a child cannot be added or removed from that person's life.)
4. **readonly** – Specifies that the feature is *read only*. Values of read only features on each instance of their domain are the same during the entire existence of that instance.
5. **derived** – Specifies that the feature is *derived*. Such a feature is typically expected to have a bound feature value expression that completely determines its value at all times (see [7.4.11](#) on feature values, which is a kernel concept).
6. **end** – Specifies that the feature is an *end feature*. Any kind of type can have end features, but they are mostly used in associations (see [7.4.5](#)) and connectors (see [7.4.6](#)), and they are further described in those contexts.

(Note that the semantics of **composite**, **portion**, and **readonly** require a model of things existing in time, which is provided in the Kernel layer, see [7.4.3](#)).

```
classifier Fuel {  
  portion feature fuelPortion : Fuel;  
}  
  
classifier Tank {  
  in feature fuelFlow: Fuel;  
  composite feature fuel : Fuel;  
}  
  
assoc VehicleRegistration {  
  end feature owner[1] : Person;  
  end feature vehicle[*] : Vehicle;  
}
```

### 7.3.4.3 Feature Typing

A feature typing relationship is declared using the keyword **specialization**, optionally followed by a short name and/or a name. The qualified name of the typed feature is then given after the keyword **typing**, followed by the

qualified name of the type, or a feature chain (see [7.3.4.6](#)), after the keyword **typed by**. The symbol **:** can be used interchangeably with the keyword **typed by**. A feature typing declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```
specialization t1 typing customer typed by Person;
specialization t2 typing employer : Organization {
    doc /* An employer is an Organization. */
}
```

If no short name or name is given, then the keyword **specialization** may be omitted.

```
typing customer typed by Person;
typing employer : Organization;
```

An *owned feature typing* is a feature typing that is an owned relationship of its type feature. An owned feature typing is defined as part of the declaration of the typed feature, rather than in a separate declaration, by including the qualified name or feature chain for the type in a list after the keyword **typed by** (or the symbol **:**).

```
feature foodItem typed by Food, InventoryItem;
```

#### 7.3.4.4 Subsetting

Subsetting is a kind of specialization between two features. This means that the values of the subsetting feature are also values of the subsetted feature on each instance (separately) of the domain of the subsetting feature.

A subsetting relationship is declared using the keyword **specialization**, optionally followed by a short name and/or a name. The qualified name of the subsetting feature, or a feature chain (see [7.3.4.6](#)), is then given after the keyword **subset**, followed by the qualified name of the subsetted feature, or a feature chain, after the keyword **subsets**. The symbol **:>** can be used interchangeably with the keyword **subsets**. A subsetting declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```
specialization Sub subset parent subsets person;
specialization subset mother subsets parent {
    doc /* All mothers are parents. */
}
```

If no short name or name is given, then the keyword **specialization** may be omitted.

```
subset rearWheels subsets wheels;
subset rearWheels subsets driveWheels;
```

An *owned subsetting* is a subsetting that is an owned relationship of the subsetting feature. An owned subsetting is defined as part of the declaration of the subsetting feature, rather than in a separate declaration, by including the qualified name or feature chain of the subsetted feature in a list after the keyword **subsets** (or the symbol **:>**).

```
feature rearWheels subsets wheels, driveWheels;
```

A subsetting feature can restrict aspects of the subsetted feature, otherwise it will, by default, have the same properties as the subsetted feature. In particular, a subsetting feature can constrain its featured types to be specializations of those of the subsetted feature and add additional feature types. A subsetting feature can also restrict the multiplicity of its subsetted feature to allow cardinalities that are smaller than those of the subsetted feature (e.g., by specifying smaller lower and/or upper bounds).

```
classifier Wheel;
classifier DriveWheel specializes Wheel;
feature anyWheels[*] : Wheel;
```

```

classifier Automobile {
  // Restricts multiplicity
  composite feature wheels[4] subsets anyWheels;
  // Restricts multiplicity and type.
  composite feature driveWheels[2] : DriveWheel subsets wheels;
}

```

If a subsetted feature is ordered, then the subsetting feature must also be ordered. If the subsetted feature is unordered, then the subsetting feature will be unordered by default, unless explicitly flagged as **ordered**.

```

classifier Automobile {
  composite feature wheels[4] ordered subsets anyWheels;
  // driveWheels must be ordered because wheels is ordered.
  composite feature driveWheels[2] ordered : DriveWheel subsets wheels;
}

```

If a subsetted feature is unique, then the subsetting feature must not be specified as non-unique. If the subsetted feature is non-unique, then the subsetting feature will still be unique by default, unless specifically flagged as **nonunique**.

```

feature urls[*] nonunique : URL;
classifier Server {
  feature accessibleURLs subsets urls; // Unique by default.
  feature visibleURLs subsets accessibleURLs; // Cannot be nonunique.
}

```

#### 7.3.4.5 Redefinition

Redefinition is a kind of subsetting that requires the values of the redefining feature and the redefined feature to be the same on each instance (separately) of the domain of the redefining feature. This means any restrictions on the values of redefining feature relative to redefined feature, such as typing or multiplicity, also apply to the values of redefined feature, and vice versa.

A redefinition relationship is declared using the keyword **specialization**, optionally followed by a short Name and/or a name. The qualified name of the redefining feature, or a feature chain (see [7.3.4.6](#)), is then given after the keyword **redefinition**, followed by the qualified name of the redefined feature, or a feature chain, after the keyword **redefines**. The symbol **:>>** can be used interchangeably with the keyword **redefines**. A redefinition declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```

specialization Redef redefinition LegalRecord::guardian redefines parent;
specialization redefinition Vehicle::vin redefines RegisteredAsset::identifier {
  doc /* A "vin" is a Vehicle Identification Number. */
}

```

If no short name or name is given, then the keyword **specialization** may be omitted.

```

redefinition Vehicle::vin redefines RegisteredAsset::identifier;
redefinition Vehicle::vin redefines legalIdentification;

```

A feature can only be redefined once for any featuring type. A feature without any feature types is considered to be implicitly featured by the most general base type **Anything** (see [7.3.4.1](#)). It is therefore allowable to redefine such a feature by a redefining feature that does have some other featuring type. It is, however, illegal for one such feature to redefine another, because that would correspond to a semantically inconsistent redefinition of one feature of **Anything** by another.



The restrictions on the specification of the multiplicity, ordering and uniqueness of a subsetting feature (see [7.3.4.4](#)) also apply to a redefining feature. In addition, the multiplicity of a redefining feature must only allow cardinalities that are consistent with the multiplicity of the redefined feature (e.g., it cannot have a multiplicity lower bound that is less than that of the redefined feature).

An *owned redefinition* is a redefinition that is an owned relationship of its redefining feature. An owned redefinition of a feature is defined as part of the declaration of the feature, rather than in a separate declaration, by including the qualified name or feature chain of the redefined feature in a list after the keyword **redefines** (or the symbol **:>>**).

```
feature vin redefines RegisteredAsset::identifier, legalIdentification;
```

If a redefining feature is declared as an owned feature of a type (see [7.3.2.6](#)), then each of the redefined features of its owned redefinitions must be features that would otherwise be inherited from supertypes of its owning type. When redefined, however, these otherwise inheritable features are *not* inherited and are, instead, replaced by the redefining feature. This enables the redefining feature to have the same name as a redefined feature, if desired. (Note, however, that even though a redefined feature is not in the namespace of the owning type of the redefining feature, the redefined feature still has values on instances of that type, particularly when they are considered as instances of the supertype that owns the redefined feature. The values will be the same as for the redefining feature, as described above.)

In general, the resolution of a qualified name begins with the namespace in which the name appears and proceeds outwards from there to containing namespaces (see [8.2.3.3.4](#)). However, the resolution of the qualified names of redefined features of owned redefinitions follow special rules. In particular, the local namespace of the owning type of the redefining feature is *not* included in the name resolution of the redefined features, with resolution beginning instead with the direct supertypes of the owning type. Since redefined features are not inherited, they would not be included in the local namespace of the owning type and, therefore, could not be referenced by an unqualified name. The special rules for redefined features, however, allow such a reference, because the name resolution begins with the namespaces of the supertypes of the owning type, one of which must contain the redefined feature.

```
classifier RegisteredAsset {
    feature identifier : Identifier;
}
classifier Vehicle : RegisteredAsset { // Owing type.
    // Legal even though "identifier" is not inherited.
    feature vin redefines identifier;
}
```

If a name is not given in the declaration of a feature with an owned redefinition, then it receives an implicit *effective name* that is the same as the name of the first redefined feature (which may itself be an implicit name, if the redefined feature is itself a redefining feature). Even though an effective name is implicit, it is the name used in name resolution, just as an explicit name would be. This is useful for constraining a redefined feature, while maintaining the same naming.

```
classifier WheeledVehicle {
    composite feature wheels[1..*] : Wheel;
}
classifier MotorizedVehicle specializes WheeledVehicle {
    composite feature redefines wheels[2..4];
}
classifier Automobile specializes MotorizedVehicle {
    composite feature redefines wheels[4] : AutomobileWheel;
}
```

### 7.3.4.6 Feature Chaining

*Feature chaining* is an owned relationship between the owning *chained feature* and a *chaining feature*. If a feature has any chaining features, then it must have at least two. The list of chained features of a chained feature is called its *feature chain*.

The meaning of a chained feature depends on its feature chain. The values of a chained feature are the same as the values of the last feature in the chain. These can be found by starting with the values of the first feature (for each instance of the original features domain), then, on each of those, finding the values of the second feature in the chain, and so on, to values of the last feature.

A feature chain is notated as a sequence of two or more qualified names separated by dot (.) symbols. Each qualified name in a feature chain must resolve to a feature. The first qualified name in a feature chain is resolved in the local namespace as usual (see [8.2.3.3.4](#)). Subsequent qualified names are then resolved using the previously resolved feature as the context namespace (but considering only visible memberships). This notation specifies a list of chaining features, as given by the resolution of the qualified names in the chain, in order.

The feature chain notation can be placed after the keyword **chains** in the declaration of the Feature, appearing after any specialization or conjugation part, but before any disjoining or type featuring part (see also [7.3.4.2](#)).

```
feature cousins chains parents.siblings.children;
```

The featuring types of the chaining feature are implicitly considered to include the featuring types of the first chaining feature. Similarly, the featured types of the chaining feature are implicitly considered to include the featured types of the last chaining feature.

The feature chain notation may also be used to specify a related element in the declaration of any of the following relationships:

1. Specialization (see [7.3.2.3](#))
2. Disjoining (see [7.3.2.5](#))
3. Subsetting (see [7.3.4.4](#))
4. Redefinition (see [7.3.4.5](#))
5. Feature inverting (see [7.3.4.7](#))
6. Connector (see [7.4.6](#), in the Kernel layer)

In this case, the related element specified using the feature chain notation becomes an owned related feature of the relationship with the feature chain as notated.

```
feature uncles subsets parents.siblings;  
feature cousins redefines parents.siblings.children;  
connector vehicle.wheelAssembly.wheels to vehicle.road;
```

**Note.** A similar dot notation is also used for the related Kernel-layer concept of a feature chain expression (see [7.4.9.3](#)). However, it is always syntactically unambiguous as to whether the notation should be parsed as a plain feature chain or as a feature chain expression.

### 7.3.4.7 Feature Inverting

*Feature inverting* is a relationship between two features that have interpretations as relations that are the reverse of each other. For example, a feature identifying each person's parents is the inverse of a feature identifying each person's children. A person identified as a parent of another will identify that other as one of their children.

A feature inverting relationship is declared using the keyword **inverting**, optionally followed by a short name and/or a name. The qualified name of the first feature, or a feature chain (see [7.3.4.6](#)), is then given after the

keyword **inverse**, followed by the qualified name of the second feature, or a feature chain, after the keyword **of**. A feature inverting declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```
inverting parent_child inverse Person::parent of Person::child {  
  doc /* A Person is the parent of their children. */  
}
```

If no short name or name is given, then the keyword **inverting** may be omitted.

```
inverse Person::parents of Person::children;
```

An *owned feature inverting* is a feature inverting that is an owned relationship of its first feature. An owned feature inverting is defined as part of the declaration of the inverted feature, rather than in a separate declaration, by giving the qualified name or feature chain of the other feature after the keyword **inverse of**.

```
classifier Person {  
  feature children : Person[*];  
  feature parents : Person[*] inverse of children;  
}
```

Note that only a single feature identification is allowed after **inverse of**. While it is possible to declare multiple feature inverting relationships for a single feature, this is generally not useful.

Inverse features can be arbitrarily nested. However, while it is allowable to use feature chains in the declaration of a feature inverting relationship, note that a feature chain is a separate feature from any of the features it chains. In order to indicate that two declared features are inverses, one should use qualified names rather than feature chains.

```
classifier A {  
  feature b: B {  
    feature c: C;  
  }  
}  
classifier C {  
  feature b: B {  
    feature a: A inverse of A::b::c;  
  }  
}
```

### 7.3.4.8 Type Featuring

*Type featuring* is a relationship between a feature and a type, identifying the type as a featuring type of the feature (see also [7.3.4.1](#)). Feature membership is a kind of type featuring that also makes the feature an owned member of the featuring type (see [7.3.2.6](#)).

A type featuring relationship is declared using the keyword **featuring**, optionally followed by a short name and/or a name, and the keyword **of**. The qualified name of the featured feature is then given, followed by the qualified name of the featuring type after the keyword **featured by**. A type featuring declaration can also optionally have a relationship body (see [7.2.2.3](#)) for, e.g., nested annotations.

```
featuring engine_by_Vehicle of engine featured by Vehicle;  
featuring power featured by engine {  
  doc /* The engine of a Vehicle has power. */  
}
```

An *owned type featuring* is a type featuring that is an owned relationship of the featured feature. An owned type featuring is defined as part of the declaration of the feature, rather than in a separate declaration, by including the qualified name of the featuring type in a list after the keyword **featured by**.

```
classifier Vehicle;  
classifier PoweredComponent;  
feature engine : Engine featured by Vehicle, PoweredComponent;
```

Note that the domain of a feature is given by the *intersection* of its featuring types. That is, in the above example, an instance in the domain of `engine` must be *both* a `Vehicle` *and* a `PoweredComponent`.

## 7.4 Kernel

### 7.4.1 Kernel Overview

The Kernel layer completes KerML. It extends the Core layer to add modeling capabilities beyond basic classification. These include specialized classifiers for things that have the semantics of data values (*data types*) from others that have an independent existence over time and space (*classes*), and for reified relationships between things (*associations*).

Classes have instances that exist or happen in time and space. They are divided into those for *structure* and *behavior*. Structures typically limit how things and relations between them might change over time, while behaviors specify changes within those limits. Structures and behaviors do not overlap, but structures can be involved in, perform, and own behaviors. Behaviors can coordinate other behaviors via *steps* (usages of behaviors). *Functions* are behaviors that yield a single result, which can be used to form trees of *expressions*. Interactions combine behaviors and associations. Some associations are also structures.

The Kernel layer adds semantics beyond the Core primarily by specifying how model elements use the Kernel model library (see [Clause 9](#)), rather than be specified mathematically as in the Core. In the simplest case, The Kernel textual syntax introduces keywords that translate to patterns of using Core abstract syntax and library models, acting as syntactic "markers" for modeling patterns tying Kernel to the Core. In the simplest case, this involves introducing implicit specializations of model library types. For example, classes must directly or indirectly subclassify the library class `Object`, while behaviors must directly or indirectly sub classify the library class `Performance`. Sometimes more complicated reuse patterns are needed. For example, binary associations (with exactly two ends) specialize `BinaryLink` from the library, and additionally require the ends of the association to redefine the `source` and `target` ends of `BinaryLink`.

This is also how other modeling languages can be built on KerML. Domain-specific metamodels and libraries can also reuse Kernel metamodel and libraries, inheriting the patterns of library reuse above, as well as the mathematical semantics they inherit from Core. This enables domain-specific modelers to use terms and syntax familiar to them and still benefit from automated assistance based on mathematically-defined semantics.

### 7.4.2 Data Types

*Metamodel references:*

- *Concrete syntax*, [8.2.5.1](#)
- *Abstract syntax*, [8.3.4.1](#)
- *Semantics*, [8.4.3.2](#)

*Data types* are classifiers that classify *data values* (see [9.2.2.2.2](#)). Certain *primitive* data types have have specified extents of values, such as the numerical and other types from the `ScalarValues` library model (see [9.3.2](#)). Other data types have features whose values can distinguish one instance of the data type from another. But, otherwise, different data values are not distinguishable.

This means that data types cannot also be classes or associations, or share instances with them. It also means that data types classify things that do not exist in time or space, because they require changing relations to other things. The feature values of a data value cannot change over time, because different feature values would inherently identify a different data value.

A data type is declared as a classifier (see [7.3.3](#)), using the keyword **datatype**. If no owned superclassing is explicitly given for the data type, then it is implicitly given a default superclassing to the data type `DataValue` from the `Base` library model (see [9.2.2](#)).

If any of the types of a feature are data types, then all of them must be. If a feature has data types as its types, and no owned subsetting or owned redefinition is explicitly given in the feature declaration, then the feature is implicitly given a default subsetting to the Feature `dataValues` from the `Base` model library (see [9.2.2](#)).

```
datatype IdNumber specializes ScalarValues::Integer;
datatype Reading { // Subtypes Base::DataValue by default
    feature sensorId : IdNumber; // Subsets Base::dataValues by default.
    feature value : ScalarValues::Real;
}
```

### 7.4.3 Classes

*Metamodel references:*

- Concrete syntax, [8.2.5.2](#)
- Abstract syntax, [8.3.4.2](#)
- Semantics, [8.4.3.3](#)

*Classes* are classifiers that classify *occurrences*, which exist in time and space (see [9.2.3.2.14](#)). Relations between an occurrence and other things can change over time and space, while the occurrence still maintains an independent identity.

A class is declared as a classifier (see [7.3.3](#)), using the keyword **class**. If no owned superclassing is explicitly given for the class, then it is implicitly given a default superclassing to the class `Occurrence` from the `Occurrences` model library (see [9.2.3](#)).

If any of the types of a feature are classes, then all of them must be. If a feature has class types, and no owned subsetting or owned redefinition is explicitly given in the feature declaration, then the feature is implicitly given a default subsetting to the feature `occurrences` from the `Occurrences` library model (see [9.2.3](#)), unless at least one of the types is an association structure, in which case the default subclassing is as described in [7.4.5](#).

```
class Situation { // Specializes Occurrences::Occurrence by default.
    feature condition : ConditionCode;
    feature soundAlarm : ScalarValues::Boolean;
}
class SituationStatusMonitor specializes StatusMonitor {
    feature currentSituation[*] : Situation; // Subsets Occurrences::occurrences by default.
}
```

### 7.4.4 Structures

*Metamodel references:*

- Concrete syntax, [Clause](#)
- Abstract syntax, [8.3.4.3](#)
- Semantics, [8.4.3.4](#)

*Structures* are classes that classify *objects*, which are kinds of occurrences. Structures typically limit how their instances and relations between them can change over time, as opposed to Behaviors, which indicate how objects and their relations change. Structures and behaviors do not overlap, but structures can own behaviors, and the objects they classify can be involved in and perform behaviors.

A structure is declared as a classifier (see [7.3.3](#)), using the keyword **struct**. If no owned superclassing is explicitly given for the structure, then it is implicitly given a default superclassing to the structure `Object` from the `Objects` library model (see [9.2.5](#)).

If any of the types of a feature are structures, then all of them must be. If a feature has structure types, and no owned subsetting or owned redefinition is explicitly given in the feature declaration, then the feature is implicitly given a default subsetting to the feature `objects` from the `Objects` library model (see [9.2.5](#)), unless at least one of the types is an association structure, in which case the default subsetting shall be as specified in [7.4.5](#).

```
struct Sensor { // Specializes Objects::Object by default.
  feature id : IdNumber;
  feature currentReading : ScalarValues::Real;
  step updateReading { ... } // Performed behavior
}
struct SensorAssembly specializes Assembly {
  composite feature sensors[*] : Sensor; // Subsets Objects::objects by default.
}
```

## 7.4.5 Associations

*Metamodel references:*

- *Concrete syntax*, [8.2.5.4](#)
- *Abstract syntax*, [8.3.4.4](#)
- *Semantics*, [8.4.3.5](#)

*Associations* are classifiers that classify *links* between things (see [9.2.4.1](#)). At least two owned features of an association must be end features (see [7.3.4.2](#)), its *association ends*, which identify the things being linked by (at the "ends" of) each link (exactly one thing per end, which might be the same thing). Associations with exactly two association ends are called *binary associations*. Associations can also have features that are not end features, which characterize each instance of the association separately from the things it links.

An association is also a relationship between the types of its association ends, which might be the same type, and are identified by its *related types*. Links are between instances of an association's related types. For binary associations, the two related types are identified as the *source type* and the *target type*, which might be the same. Associations with more than two association ends ("n-ary") have only target types, no source types.

An association is declared as a classifier (see [7.3.3](#)), using the keyword **assoc**. If no owned superclassing is explicitly given for the association, then it is implicitly given a default superclassing to either the association `BinaryLink` (if it is a binary association) or the association `Link` (otherwise), both of which are from the `Links` library model (see [9.2.4](#)).

If an association has a single superclass that is an association, it may inherit association ends from this superclass association. However, if it declares any owned association ends, then each of these must redefine an association end of the superclass association, in order, up to the number of association ends of the superclassifier. If no redefinition is given explicitly for an owned association end, then it is considered to implicitly redefine the association end at the same position, in order, of the superclassifier Association (including implicit defaults), if any.

```
assoc Ownership { // Specializes Objects::BinaryLink by default.
  feature valuationOnPurchase : MonetaryValue;
```

```

    end feature owner[1..*] : LegalEntity; // Redefines BinaryLink::source.
    end feature ownedAsset[*] : Asset;      // Redefines BinaryLink::target.
}
assoc SoleOwnership specializes Ownership {
    end feature owner[1]; // Redefines Ownership::owner.
    // ownedAsset is inherited.
}

```

If an association has more than one superclassifier that is an association, then the association *must* declare a number of owned association ends at least equal to the maximum number of association ends of any of its superclassifier associations. Each of these owned association ends must then redefine the corresponding association end (if any) at the same position, in order, of each of the superclassifier associations.

*Association structures* are both associations and classes (see [7.4.3](#) on classes), classifying *link objects*, which are both links and objects (see [9.2.5.1](#) on objects). As objects, link objects can be created and destroyed, and their non-end features can change over time. However, the values of the end features of a link object are fixed and cannot change over its lifetime.

An association structure is declared like a regular association, but using the keyword **assoc struct**. If no owned superclassing is explicitly given for the association structure, then it is implicitly given a default superclassing to either the association structure `BinaryLinkObject` (if it is a binary association structure) or the association structure `LinkObject` (otherwise), both of which are from the `Objects` library model (see [9.2.5](#)). The same rules on association ends described above for associations also apply to association structures. An association structure may specialize an association that is not an association structure, but all subclassifications of an association structure must be association structures.

```

assoc struct ExtendedOwnership specializes Ownership {
    // End features are inherited from Ownership.
    // The values of the feature "revaluations" may change over time.
    feature revaluations[*] ordered : MonetaryValue;
}

```

If a feature has one or more associations as types, then these associations must all have the same number of association ends. If the feature defines owned end features in its body, then it can have no more than the number of association ends of its association types. The owned end features of such a feature follow the same rules for redefinition of the association ends of its association types as described above for the redefinition of the association ends of superclassifier associations by a subclassifier association.

If a feature declaration has no explicit owned subsettings or owned redefinitions, and any of its types are binary associations, then the feature is implicitly given a default subsetting to the feature `binaryLinks` from the `Links` library model (see [9.2.4](#)) or to the feature `binaryLinkObjects` from the `Objects` library model (see [9.2.5](#)), if any of the associations are association structures. If some of the types are associations, but not binary associations, then it is given a default subsetting to the feature `links` from the `Links` library model (see [9.2.4](#)) or to the feature `linkObjects` from the `Objects` library model (see [9.2.5](#)), if any of the associations are association structures.

## 7.4.6 Connectors

### 7.4.6.1 Connectors Overview

*Metamodel references:*

- Concrete syntax, [8.2.5.5](#)
- Abstract syntax, [8.3.4.5](#)
- Semantics, [8.4.3.6](#)



*Connectors* are features that are typed by associations (see [7.4.5](#)), having values that are links (see [9.2.4.2.3](#)). Like an association, a connector has end features, known as its *connector ends*. Each connector end redefines an association end from each of the associations that type the connector and subsets a feature that becomes a *related feature* of the connector. Connectors typed by binary associations are called *binary connectors*.

A connector is also a relationship between its related features. For binary connectors, the two related features are identified as the *source feature* and the *target feature*, which might be the same. Connectors with more than two connector ends ("n-ary") have only target features, no source features.

Connectors can be thought of as "instance-specific" associations, because their values (which are links) are each limited to linking things identified via related features on the same instance of the connector's domain (or by things identified by that instance, recursively, see below). For example, an association could be used to model an engine driving wheels, and to type a connector in the car model. This connector specifies an engine driving wheels only in the same car, not in another car, as would be allowed with just the association.

Specifically, the values (links) of a connector are restricted to those that link things

1. classified by the types of its association ends, regardless of the domain of the connector
2. identified by its related features for the same instance of the domain of the connector (or by things identified by that instance, recursively).

For example, if the wheels in a car are taken to be part of its drive train, rather than part of the car directly, then the engine in each car will drive wheels identified by that car's drive train, rather than a feature of the car directly. This requires that each related feature of a connector have some featuring type of the connector as a direct or indirect featuring type (where a feature with no featuring type is treated as if the classifier `Anything` was its featuring type). This condition is satisfied if a connector has an owned type for which its related features are either direct or features reached by chaining. Otherwise, explicit owned type featureings (see [7.3.4.8](#)) should be used to ensure that the connector has a sufficiently general domain.

*Binding connectors* are binary connectors that require their source and target features to have the same values on each instance of their domain. They are typed by the library association `SelfLink` (which only links things in the modeled universe to themselves, see [9.2.4.1](#)) and have end multiplicities of exactly 1. This requires a `SelfLink` to exist between each value of the source feature and exactly one value of the target feature, and vice-versa.

To be meaningful, the declared co-domains of the related features of a binding connector must at last overlap. Since the interpretations of data types are disjoint from those of classes, this means that a feature typed by data types can only be bound to another feature typed by data types. In the determination of the equivalence of such features, indistinguishable data values are considered equivalent. The binding of features typed by classes to another feature typed by classes, on the other hand indicates that the same occurrences play the roles represented by each of the related features.

*Successions* are binary connectors requiring their source and target features to identify *Occurrences* that are ordered in time. They are typed by the library association `HappensBefore` (see [9.2.3](#)), which links occurrences that happen completely separately in time, with the connector's source feature being the earlier occurrence and the target feature being the later occurrence

### 7.4.6.2 Connector Declaration

A connector is declared as a feature (see [7.3.4.2](#)) using the keyword `connector`. In addition, a connector declaration includes a list of qualified names of the related features of the connector, between parentheses (...), after the regular feature declaration part and before the body of the connector (if any). All associations typing a connector must have the same number of association ends, which must be the same as the number of related features of the connector.



If no owned subsetting or owned redefinition is explicitly given, then the connector is implicitly given a default subsetting to the feature `binaryLinks` from the `Links` library model (see [9.2.4](#)), if it is a binary connector, or to the feature `links` from the `Links` model library, if it is not a `binaryConnector`, and none of its types are association structures. If at least one of the types of a connector is an association structure, then the default subsetting is `linkObjects` from the `Objects` library model (see [9.2.5](#)) instead of `links`, and, if it is a binary Connector, the default subsetting is to `binaryLinkObjects`.

```
// Specializes Objects::BinaryLinkObject by default.
assoc struct Mounting {
    end feature mountingAxle[1] : Axle;
    end feature mountedWheel[2] : Wheel;
}

struct WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;

    // Subsets Objects::binaryLinkObjects by default.
    connector mount[2] : Mounting (axle, wheels);
}
```

By default, the connector ends of a connector are declared in the same order as the association ends of the types of the connector. However, if the connector has a single type, then the related features can be given in any order, with each related feature paired with an association end of the type using a notation of the form  $e \rightarrow f$ , where  $e$  is the name of an association end and  $f$  is the qualified name of a related feature. In this case, the name of each association end must appear exactly once in the list of connector end declarations.

```
struct WheelAssembly {
    composite feature axle[0..1] : Axle;
    composite feature wheels[0..2] : Wheel;
    connector mount[2] : Mounting (
        mountedWheel  $\rightarrow$  wheels,
        mountingAxle  $\rightarrow$  axle);
}
```

A special notation can be used for a binary connector, in which the source related feature is referenced after the keyword **from**, and the target related feature is referenced after the keyword **to**.

```
struct WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;
    connector mount[2] : Mounting from axle to wheels;
}
```

If a binary connector declaration includes only the related features part, then the keyword **from** can be omitted.

```
struct WheelAssembly {
    composite feature axle[1] : Axle;
    composite feature wheels[2] : Wheel;
    connector axle to wheels;
}
```

If a binary connector has a single type, then the names of the association ends of the type can also be used in the declaration of the connector ends in the special notation for binary connectors. However, since the connector ends are always declared in order from source to target in this notation, the association end names given must match those from the type in the order they are declared for that type.

```

struct WheelAssembly {
  composite feature axle[1] : Axle;
  composite feature wheels[2] : Wheel;
  connector mount[2] : Mounting
    from mountingAxle :> axle
    to mountedWheel :> wheels;
}

```

In any of the above notations, a multiplicity can be specified for a connector end, after the qualified name of the related feature for that end. In this case, the given multiplicity redefines the multiplicity that would otherwise be inherited from the association end corresponding to the connector end.

```

struct WheelAssembly {
  composite feature halfAxles[2] : Axle;
  composite feature wheels[2] : Wheel;

  // Connects each one of the halfAxles to a different one of the wheels.
  connector mount : Mounting from halfAxles[1] to wheels[1];
}

```

Note that, if a connector is an owned feature of a type (as above), the context consistency condition for the related features of the connector (see [7.4.6.1](#)) requires that these features also be directly or indirectly nested within the owning type. The feature chain dot notation (see [7.3.4.6](#)) should be used when connecting so-called "deeply nested" features.

While the resolution of a feature chain is similar to a qualified name, the feature path contextualizes the resolution of the final feature. Thus, for example, while the qualified name `axle::halfAxles` statically resolves to `Axle::halfAxles`, in the Feature chain `axle.halfAxles`, `halfAxles` is understood to be specifically the feature as nested in `axle`.

```

struct Axle {
  composite feature halfAxles[2] : HalfAxle;
}
struct Wheel {
  composite feature hub : Hub[1];
  composite feature tire : Tire[1];
}
struct WheelAssembly {
  composite feature axle[1] : Axle;
  composite feature wheels[2] : Wheel;

  connector mount : Mounting from axle.halfAxles to wheels.hub;
}

```

### 7.4.6.3 Binding Connector Declaration

A binding connector is declared as a feature (see [7.3.4.2](#)) using the keyword **binding**. In addition, a binding connector declaration gives, after the keyword **of**, the qualified names of the two related features that are bound by the binding connector, separated by the symbol `=`, after the regular feature declaration part and before the body of the binding connector (if any). If no owned subsetting or owned redefinition is explicitly given, then the binding connector is implicitly given a default subsetting to the feature `selfLinks` from the `Links` library model (see [9.2.4](#)). Note that, due to this default subsetting, if no `type` is explicitly given for a binding connector, then it will implicitly have the `type SelfLink` (the type of `selfLinks`).

```

struct WheelAssembly {
  composite feature fuelTank {
    out feature fuelFlowOut : Fuel;
  }
}

```

```

    }

    composite feature engine {
        in feature fuelFlowIn : Fuel;
    }

    // Subsets Links::selfLinks by default.
    binding fuelFlowBinding of          fuelTank.fuelFlowOut = engine.fuelFlowIn;
}

```

If a binding connector declaration includes only the related features part, then the keyword **of** can be omitted.

```

struct WheelAssembly {
    composite feature fuelTank {
        out feature fuelFlowOut : Fuel;
    }

    composite feature engine {
        in feature fuelFlowIn : Fuel;
    }

    binding fuelTank.fuelFlowOut = engine.fuelFlowIn;
}

```

The connector ends of a binding connector always have multiplicity 1..1.

(See also [7.4.11](#) on the use of binding connectors with feature values.)

#### 7.4.6.4 Succession Declaration

A succession is declared as a feature (see [7.3.4.2](#)) using the keyword **succession**. In addition, the succession declaration gives the qualified name of the source feature after the keyword **first** and the qualified name of the target feature after the keyword **then**. If no owned subsetting or owned redefinition is explicitly given, then the succession is implicitly given a default subsetting to the feature `happensBeforeLinks` from the `Occurrences` library model (see [9.2.3](#)). Note that, due to this default subsetting, if no `type` is explicitly given for a succession, then it will implicitly have the `type HappensBefore` (the type of `happensBeforeLinks`).

```

behavior TakePicture {
    composite step focus : Focus;
    composite step shoot : Shoot;
    succession controlFlow first focus then shoot;
}

```

If a succession declaration includes only the related features part, then the keyword **first** can be omitted.

```

behavior TakePicture {
    composite step focus : Focus;
    composite step shoot : Shoot;
    succession focus then shoot;
}

```

As for connector ends on regular connectors, constraining multiplicities can also be defined for the connector ends of successions.

```

behavior TakePicture {
    composite step focus[*] : Focus;
    composite step shoot[1] : Shoot;
}

```

```

    // A focus may be preceded by a previous focus.
    succession focus[0..1] then focus[0..1];
    // A shoot must follow a focus.
    succession focus[1] then shoot[0..1];
}

```

## 7.4.7 Behaviors

### 7.4.7.1 Behaviors Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.5.6](#)
- *Abstract syntax*, [8.3.4.6](#)
- *Semantics*, [8.4.3.7](#)

*Behaviors* are classes that classify *performances*, which are kinds of occurrences that can be spread out in disconnected portions of space and time (see [9.2.6](#)). The performance of behaviors can cause effects on other things, including their existence and relations, some of which might be accepted as input to or provided as output from the behavior.

Behaviors can have *steps*, which are features typed by behaviors, allowing the containing behavior to coordinate the performance of other behaviors. Steps can be ordered in time using succession connectors (see [7.4.6.4](#)). They can also be connected by item flows to model things flowing between the output of one step and the input of another. Steps can also nest other steps to augment or redefine steps inherited from their behavior types.

### 7.4.7.2 Behavior Declaration

A behavior is declared as a classifier (see [7.3.3](#)), using the keyword **behavior**. If no owned superclassing is explicitly given for the behavior, then it is implicitly given a default superclassing to the behavior *Performance* from the *Performances* library model (see [9.2.6](#)).

Features declared in the body of a behavior with a non-null direction (see [7.3.4.2](#)) are considered to be the owned *parameters* of the behavior. Features with direction **in** are input parameters, those with direction **out** are output parameters, and those with direction **inout** are both input and output parameters.

```

// Specializes Performances::Performance by default.
behavior TakePicture {
    in scene : Scene;
    out picture : Picture;
}

```

Parameters are ordered in the lexical order they are declared in the body of a behavior. They may appear at any location within the body.

If a behavior has owned subclassifications whose superclassifiers are behaviors, then each of the owned parameters of the subclassifier behavior must, in order, redefine the parameter at the same position of each of the superclassifier behaviors. The redefining parameters shall have the same direction as the redefined parameters.

```

behavior A { in a1; out a2; }
behavior B { in b1; out b2; }
behavior C specializes A, B {
    in c1 redefines a1, b1;
    out c2 redefines a2, b2;
}

```

If there is a single superclassifier behavior, then the subclassifier behavior can declare fewer owned parameters than the superclassifier behavior, inheriting any additional parameters from the superclassifier (which are considered to be ordered after any owned parameters). If there is more than one superclassifier behavior, then every parameter from every superclassifier must be redefined by an owned parameter of the subclassifier. If every superclassifier parameter is redefined, then the subclassifier behavior may also declare additional parameters, ordered after the redefining parameters. If no redefinitions are given explicitly for a parameter, then the parameter is implicitly given owned redefinitions of superclassifier parameters sufficient to meet the previously stated requirements.

```
behavior A1 :> A { in aa; } // aa redefines A::a1, A::a2 is inherited.
behavior B1 :> B { in b1; out b2; inout b3; } // Redefinitions are implicit.
behavior C1 :> A1, B1 { in c1; out c2; inout c3; }
```

Steps (see [7.4.7.3](#)) declared in the body of a behavior are the owned steps of the containing behavior. A behavior can also inherit or redefine non-private steps from any superclassifier Behavior.

```
behavior Focus { in scene : Scene; out image : Image; }
behavior Shoot { in image : Image; out picture : Picture; }
behavior TakePicture {
  in scene : Scene;
  out picture : Picture;
  composite step focus : Focus;
  composite step shoot : Shoot;
}
```

Though the performance of a behavior takes place over time, the order in which its steps are declared has no implication for temporal ordering of the performance of those steps. Any restriction on temporal order, or any other connections between the steps, must be modeled explicitly.

```
behavior TakePicture {
  in scene : Scene;
  out picture : Picture;

  binding focus.scene = scene;
  composite step focus : Focus;
  succession focus then shoot;
  composite flow focus.image to shoot.image;
  composite step shoot : Shoot;
  binding picture = focus.picture;
}
```

### 7.4.7.3 Step Declaration

A step is declared as a feature (see [7.3.4.2](#)) using the keyword **step**. If no owned subsetting or owned redefinition is explicitly given, then the step is implicitly given a default subsetting to the feature performances from the Performances library model (see [9.2.6](#)).

As for a behavior, directed features declared in the body of a step are considered to be *parameters* of the step (see [7.4.7.2](#)). If a step has owned specializations (including all feature typings, subsettings, and redefinitions), whose general type is a behavior or a step, then the rules for the redefinition of parameters of the behaviors and steps are the same as for the redefinition of the parameters of superclassifier behaviors by a subclassifier behavior (see [7.4.7.2](#)).

```
step focus : Focus {
  // Parameters redefine parameters of Focus.
  in scene;
  out image;
}
```

```
// Parameters are inherited.
step refocus subsets focus;
```

A step can also have a body, which may have steps in it. A step can inherit or redefine steps from its behavior types or any other steps it subsets.

```
step takePictureWithAutoFocus : TakePicture {
  in feature unfocusedScene redefines scene;
  step redefines focus : AutoFocus;
  out feature focusedPicture redefines picture;
}
```

## 7.4.8 Functions

### 7.4.8.1 Functions Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.5.7](#)
- *Abstract syntax*, [8.3.4.7](#)
- *Semantics*, [8.4.3.8](#)

*Functions* are behaviors (see [7.4.7](#)) with one out parameter designated as the *result parameter*. Functions classify *evaluations* (see [9.2.6.2.3](#)), which are kinds of performances that produce *results* as values of the result parameter. Like all behaviors, functions can change things, often referred to as "side effects". A *pure* function is one that has no side effects and always produces the same results given the same input values, similarly to a function in the mathematical sense. The numerical functions in the Kernel Model Library (see [Clause 9](#)), for example, are pure functions.

*Expressions* are steps (see [7.4.7](#)) typed by only a single function, which means that their values are evaluations. An expression whose value is an evaluation with results is said to *evaluate to* those results. They can be steps in any behavior, but a function, in particular, can designate one of its expression steps as the *result expression* that gives the value of its result parameter. Expressions can have their own nested parameters, to augment or redefine those of their functions, including the result parameter. They can also own other expressions and designate a result expression, similarly to a function. (See also [7.4.9](#) for more on expressions).

*Predicates* are functions whose result is a single Boolean value (that is, true or false). A predicate determines whether the values of its input parameters meet particular conditions at the time of its evaluation, resulting in true if they do, and false otherwise. Predicates classify *boolean evaluations*, which are specialized evaluations giving a Boolean result (see [9.2.6.2.1](#)).

*Boolean expressions* are expressions whose function is a predicate and, so, evaluate to a Boolean result. A boolean expression might, in general, evaluate to true at some times and false at other times. An *invariant*, though, is a boolean expression that must always evaluate to either true at all times or false at all times. By default, an invariant is asserted to always evaluate to true, while a *negated invariant* is asserted to always evaluate to false.

### 7.4.8.2 Function Declaration

A function is declared as a behavior (see [7.4.7.2](#)), using the keyword **function**. If no owned superclassing is explicitly given for a function, then it is implicitly given a default subclassification to the function `Evaluation` from the `Performances` library model (see [9.2.6](#)). As for a behavior, any feature declared in the body of a function with an explicit direction is considered to be a parameter of the function. In addition, the result parameter of a function may be declared in its body by beginning the declaration with the keyword **return** (instead of a direction keyword).

```
// Specializes Performances::Evaluation by default.
function Velocity {
  in v_i : VelocityValue;
  in a : AccelerationValue;
  in dt : TimeValue;
  return v_f : VelocityValue = v_i + a * dt;
}
```

If a function has owned subclassifications that are behaviors, then the rules for redefinition or inheritance of non-result parameters are the same as for a behavior (see [7.4.7.2](#)). If some of the superclassifier behaviors are functions, then the result parameter of the subclassifier function must redefine the result parameters of the superclassifier functions. If, in this case, the result parameter of the subclassifier function has no owned redefinitions, then it is implicitly given redefinitions of the result parameter of each of the superclassifier functions.

```
abstract function Dynamics {
  in initialState : DynamicState;
  in time : TimeValue;
  return : DynamicState;
}
function VehicleDynamics specializes Dynamics {
  // Each parameter redefines the corresponding superclassifier parameter
  in initialState : VehicleState;
  in time : TimeValue;
  return : VehicleState;
}
```

The body of a function is like the body of a behavior (see [7.4.7.2](#)), with the optional addition of the declaration of a result expression at the end. A result expression is always be written using the Expression notation described in [7.4.9](#), *not* using the Expression declaration notation from [7.4.8.3](#). The result of the result expression is implicitly bound to the result parameter of the containing function.

```
function Average {
  in scores[1..*] : Rational;
  return : Rational;

  Sum(scores) / Length(scores)
}
```

**Note.** A result expression is written *without* a final semicolon.

The result of a function can also be explicitly bound, either using a binding connector (see [7.4.6.3](#)) or a feature value on the result parameter declaration (see [7.4.11](#)). In this case, the body of the function should *not* include a result expression.

```
function Average {
  in scores[1..*] : Rational;
  return : Rational = Sum(scores) / Length(scores);
}
```

### 7.4.8.3 Expression Declaration

An expression can be declared as a step (see [7.4.7.3](#)) using the keyword **expr** (see also [7.4.9](#) for more traditional expression notation). If no owned subsetting or owned redefinition is explicitly given, then the expression is implicitly given a default subsetting to the feature evaluations from the Performances library model (see [9.2.6](#)).

As for a step, directed features declared in the body of an expression are considered to be parameters of the expression (see [7.4.7.3](#)). If an expression has owned specializations (including all feature typings, subsettings, and redefinitions) whose general type is a behavior (including a function) or a step (including an expression), then the rules for the redefinition of the parameters of those behaviors and steps are the same as for the redefinition of the parameters of superclassifier behaviors by a subclassifier function (see [7.4.8.2](#)).

```

expr computation : ComputeDynamics {
  // Parameters redefined parameters of ComputeDynamics.
  in state;
  in dt;
  return result;
}
expr vehicleComputation subsets computation {
  // Input parameters are inherited, result is redefined.
  return : VehicleState;
}

```

Like a function body, an expression body can also specify a result expression.

```

expr : VehicleDynamics {
  in initialState;
  in time;
  return result;

  vehicleComputation(initialState, time)
}

```

Or the result can be explicitly bound.

```

expr : Dynamics {
  in initialState;
  in time;
  return result : VehicleState =
    vehicleComputation(initialState, time);
}

```

#### 7.4.8.4 Predicate Declaration

A predicate is declared as a function (see [7.4.8](#)), using the keyword **predicate**. If no owned subclassification is explicitly given for a predicate, then it is implicitly given a default subclassification to the predicate `BooleanEvaluation` from the `Performances` library model (see [9.2.6](#)). If a predicate has owned subclassifications that are behaviors, then the rules for redefinition or inheritance of non-result parameters are the same as for a function (see [7.4.8.2](#)). Since a predicate must always return a `Boolean` result, it is not necessary to explicitly declare a result parameter for it. However, if a result parameter is declared, then it must have type `Boolean` from the `ScalarValues` library model (see [9.3.2](#)) and multiplicity `1..1` (see [7.4.12](#)).

```

predicate isAssembled {
  in assembly : Assembly;
  in subassemblies[*] : Assembly;
}

```

The body of a predicate is the same as a function body (see [7.4.8](#)). If a result expression is included, then it must have a `Boolean` result.

```

predicate isFull {
  in tank : FuelTank;

```



```

    tank.fuelLevel == tank.maxFuelLevel
}

```

### 7.4.8.5 Boolean Expression and Invariant Declaration

A boolean expression is declared as an expression (see [7.4.8.3](#)), using the keyword **bool**. If no owned subsetting or owned redefinition is explicitly given, then the boolean expression is implicitly given a default subsetting to the feature `booleanEvaluations` from the `Performances` library model (see [9.2.6](#)).

As for an expression, directed features declared in the body of a boolean expression are considered to be parameters of the boolean expression (see [7.4.8.3](#)). If a boolean expression has owned specializations (including all feature typings, subsettings, and redefinitions) whose general type is a behavior or step, then the rules for the redefinition of the parameters of those behaviors and steps are the same as for a regular expression declaration (see [7.4.8.3](#)). The requirements on and default for the result parameter of a boolean expression are the same as for a predicate (see [7.4.8.4](#)).

```

// All input parameters are inherited.
bool assemblyChecks[*] : isAssembled;

```

Like a predicate body (see [7.4.8.4](#)), a boolean expression body can specify a `Boolean` result expression.

```

class FuelTank {
    feature fuelLevel : Real;
    feature readonly maxFuelLevel : Real;
    bool isFull { fuelLevel == maxFuelLevel }
}

```

An invariant is declared like any other boolean expression, except using the keyword **inv** instead of **bool**, and, additionally, this keyword may be optionally followed by one of the keywords **true** or **false**, to indicate whether the invariant is asserted to be true or false (i.e., is negated). The default is **true**.

```

class FuelTank {
    feature fuelLevel : Real;
    feature readonly maxFuelLevel : Real;
    // The invariant is asserted true by default.
    inv { fuelLevel >= 0 & fuelLevel <= maxFuelLevel }
    // The invariant is explicitly asserted false, that is, it is negated.
    inv false { fuelLevel > maxFuelLevel }
}

```

## 7.4.9 Expressions

### 7.4.9.1 Expressions Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.5.8](#)
- *Abstract syntax*, [8.3.4.8](#)
- *Semantics*, [8.4.3.9](#)

As described in [7.4.8](#), expressions are steps typed by functions, and [7.4.8.3](#) covers the general notation for declaring an expression as a step. However, expressions are commonly organized into tree structures, with expressions as the nodes, and the input parameters of each expression bound to the result of each of its child expressions. KerML includes extensive textual notation for constructing expression trees, including traditional operator notations for functions in the Kernel Model Library (see [Clause 9](#)).

These expression notations map entirely to an abstract syntax involving just a few specialized kinds of expressions:

- The non-leaf nodes of an expression tree are *invocation expressions*, a kind of expression that specifies its input values as the results of other expressions (its *argument expressions*), one for each of the input parameters of its *invoked* function.
- The edges of the tree are binding connectors between the input parameters of an invocation expression (redefining those of its *function*) and the results of its argument expressions.
- The leaf nodes are these kinds of expressions:
  - *Feature reference expressions* evaluate to values of a referenced feature that is not part of the expression tree.
  - *Literal expressions* evaluate to the literal value of one of the primitive data types from the `ScalarValues` model library (see [9.3.2](#)).
  - *Null expressions* evaluate to the empty set.

An expression can also be the referent of a feature reference expression in an expression tree, as above. This enables the evaluation of the referent expression to be taken as the value of the argument of an invocation, rather than passing the value of the *result* of the evaluation. As a shorthand for doing this, the concrete syntax for an expression body (as described in [7.4.8.3](#)) can be used as a leaf node in the expression syntax tree.

A *model-level evaluable* expression is an expression that refers to metadata, which is data about model elements, rather than the things being modeled. Model-level evaluable expressions can give values to the features of a metadata (see [7.4.13](#)) and be used as element filtering conditions in packages (see [7.4.14](#)). The expressiveness of model-level evaluable expressions is restricted to support this:

- All null expressions, literal expressions and feature reference expressions are model-level evaluable.
- An invocation expression is model-level evaluable if and only if it meets the following conditions:
  1. All its argument expressions are model-level evaluable.
  2. It invokes a function that is listed as being model-level evaluable in [Table 5](#) (in [8.2.5.8.1](#)) or [Table 7](#) (in [8.2.5.8.2](#)).

**Release Note.** The Functions allowed in model-level evaluable expressions may be expanded in the final submission.

### 7.4.9.2 Operator Expressions

*Operator expression* notation provides a shorthand for invocation expressions that invoke a library function represented as an *operator symbol*. ([Table 5](#) in [8.2.5.8.1](#) shows the mapping from operator symbols to the functions they represent from the Kernel Model Library.) An operator expression contains subexpressions called its *operands* that generally correspond to the argument expressions of the invocation expression, except in the case of operators representing *control functions*, in which case the evaluation of certain operands is as determined by the function.

Operator expressions include the following:

- *Conditional expression.* The *conditional test* operator `?` is a ternary operator that evaluates to the value of its second or third operand, depending on whether the result of its first operand is true or false. Note that only one of the second or third operand is actually evaluated. There are two forms of conditional expressions, both of which place the `?` operator after the first operand. The first form separates the second and third operands with a `:` symbol, while the second form begins with the keyword `if` and separates the second and third operands with the keyword `else`.

```
x >= 0? x: -x
if x >= 0? x else -x
```

- *Binary operator expression.* A *binary operator* is one that has two operands. The binary operators include numerical operators (+, -, \*, /, %, ^, \*\*), logical operators (&, |, ^^), comparison operators (==, !=, <, >, <=, >=), and the range construction operator (. .). In general, both operands become arguments of the invocation expression, with their results being passed to the invocation of the function represented by the operator. However, the null-coalescing (??), conditional and (&&), conditional or (||) and implication (=>) operators all correspond to control functions in which their second operand is only evaluated depending on a certain condition of the value of their first operand (whether it is null, true, false, or true, respectively). The keywords **and**, **or**, **xor**, and **implies** can be used as synonyms for the &&, ||, ^^, and => operators, respectively.

```
x + y
list[i] ?? default
i > 0 && sensor[i] != null
sensor == null or sensor.reading > 0
```

- *Unary operator expressions.* A *unary operator* is one that has a single operand. The result of evaluating the operand is passed to the invocation of the Function represented by the operator. The unary operators include the numerical operators + and - and the logical operator !. The keyword **not** can be used as a synonym for the ! operator.

```
-x
!isOutOfRange(sensor)
not completed
```

- *Classification expression.* The *classification operators* are syntactically similar to binary operators, but, instead of an expression as their second operand, they take a type name. The classification operators **istype** and **hastype** test whether the value of their first operand is classified by the named type (either including or not including subtypes, respectively). The symbol @ can be used as a synonym for **istype**.

```
sensor istype ThermalSensor
sensor @ ThermalSensor
person hastype Administrator
```

The classification operator **as**, known as the *cast operator*, performs an **isType** test of whether each of the values of its first operand is classified by the named type, and then it selects only those values that pass the test to include in its result. The result values of such a cast expression (if any) are always guaranteed to be instances of the named type.

```
allSensors as ThermalSensor
person as Administrator
```

The classification operators may also be used without a first operand, in which case the first operand is implicitly `Anything::self` (see [9.2.2.2.1](#)). This is useful, in particular, when used as a test within an element filter condition expression (see [7.4.14](#)).

```
istype ThermalSensor
@ThermalSensor
hastype Administrator
as Supervisor
```

- *Extent expression.* The *extent operator* **all** is syntactically similar to a unary operator, but, instead of an expression as its operand, it takes a type name. An extent expression evaluates to a sequence of all instances of the named type.

```
all Sensor
```

In an operator expression containing nested operator expressions, the nested expressions are implicitly grouped according to the *precedence* of the operators involved, as given in [Table 6](#) (in [8.2.5.8.1](#)). Operator expressions with higher precedence operators are grouped more tightly than those with lower precedence operators. For example, the operator expression

```
-x + y * z
```

is considered equivalent to

```
( (-x) + (y * z) )
```

### 7.4.9.3 Primary Expressions

*Primary expression* notation provides additional shorthands for certain kinds of invocation expressions. For those cases in which the invoked function is represented by an operator symbol, the symbol is mapped to the appropriate library function as given in [Table 7](#) (in [8.2.5.8.2](#)).

Primary expressions include the following:

- *Index expression.* An index expression specifies the invocation of the indexing function ' `[]` ' from the `BaseFunctions` library model (see [9.4.2](#)). The default behavior for this function is given by the specialization `SequenceFunctions::[]`, for which the first operand is expected to evaluate to a sequence of values, and the second operand is expected to evaluate to an index into that sequence. Default indexing is from 1 using `Natural` numbers. However, the functionality of the `BaseFunctions::[]` operator may be specialized differently for domain-specific types, as is already the case for the library `Array` data type (see [9.3.3.2.1](#)).

```
sensors[activeSensorIndex]
```

- *Sequence expression.* A sequence expression consists of a list of one or more expressions separated by comma ( `,` ) symbols, optionally terminated by a final comma, all surrounded by parentheses ( `()` ). Such an expression specifies sequential invocations of the sequence concatenation function ' `,` ' from the `BaseFunctions` library model (see [9.4.2](#)). The default behavior for this Function is given by the specialization `SequenceFunctions::','`, which concatenates the sequence of values resulting from evaluating its two arguments. With this behavior, a sequence expression concatenates, in order, the results of evaluating all the listed expressions.

```
(temperatureSensor, windSensor, precipitationSensor)
( 1, 3, 5, 7, 11, 13, )
```

A sequence expression with a single constituent expression simply evaluates to the value of the contained expression, as would be expected for a parenthesized expression. The empty sequence ( `()` ) is not actually a sequence expression, but, rather, an alternative notation for a null expression (see [7.4.9.4](#)).

```
(highValue + lowValue) / 2
```

Sequences of values are *not* themselves values. Therefore, sequences are "flat", with no element of a sequence itself being a sequence. For example, `((1, 2, 3), 4)`, `(1, (2, 3), 4)` and `(1, null, (2, 3, 4))` all evaluate to the same sequence of values as `(1, 2, 3, 4)`. To model nested collection values, use the data types from the `Collections` library model (see [9.3.3](#)).

- *Feature chain expression.* A feature chain expression consists of a primary expression and a feature qualified name or a feature chain ([Clause](#)), separated by a dot ( `.` ) symbol. The referenced feature is evaluated in the context of each of the result values of the primary expression, in order. The resulting feature values are then collected into a sequence in order of evaluation. The qualified name for the referent

feature is resolved using the result parameter of the primary expression as the context namespace (see [8.2.3.3.4](#)), but considering only visible memberships.

```
// The primary expression is "getPlatform(id)".
// The feature chain is "sensors.isActive".
// Results in a sequence of Boolean values,
// one for each platform sensor.
getPlatform(id).sensors.isActive
```

To avoid ambiguity, the primary expression of a feature chain expression cannot be itself a feature chain expression. To read a list of features sequentially, rather than in a single evaluation, delimit nested feature chain expressions using parentheses

```
// First evaluate "getPlatform(id).sensors",
// then evaluate ".isActive" on the result of that.
(getPlatform(id).sensors).isActive
```

- *Collect expression.* A collect expression consists of a primary expression and an expression body (see [7.4.9.4](#)) separated by a dot (.) symbol. The expression body must have a single input parameter. The expression body is evaluated on each of the result values from the primary expression, in order, and each of the results are collected into a sequence in order of evaluation (that is, a collect expression is a shorthand for invoking the `ControlFunctions::collect` Function).

```
sensors.{in s: Sensor; s.reading} // results in a sequence of
                                // readings of each of the sensors
```

- *Select expression.* A select expression consists of a primary expression and an expression body (see [7.4.9.4](#)) separated by a dot-question-mark (.) symbol. The expression body must have a single input parameter and a Boolean result. The expression body is evaluated on each of the result values from the primary expression, in order, and those for which the expression body evaluates to true are selected for inclusion in the result of the select expression (that is, a select expression is a shorthand for invoking the `ControlFunctions::select` Function).

```
sensors.?(in s: Sensor; s.isActive) // results in the subsequence of
                                    // sensors that are active
```

- *Function operation expression.* A function operation expression is a special syntax for an invocation expression in which the first argument is given before the arrow (->) symbol, which is followed by the name of the function to be invoked and an argument list for any remaining arguments (see [7.4.9.4](#)). This is useful for chaining invocations in an effective data flow.

```
sensors -> selectSensorsOver(limit) -> computeCriticalValue()
```

If the invoked function has exactly two input parameters, and the second input parameter is an expression, then an expression body (see [7.4.9.4](#)) can be used as the argument for the second argument without surrounding parentheses. The argument expression body should declare parameters consistent with those on the parameter expression (if any). This is particularly useful when invoking functions from the `ControlFunctions` library model (see [9.4.17](#)).

```
sensors -> select {in s: Sensor; s::isActive}
members -> reject {in member: Member; !member->isInGoodStanding()}
factors -> reduce {in x: Real; in y: Real; x * y}
```

If the argument expression is simply the direct invocation of another function, then the argument expression may be specified using simply the name of the invoked function.

```
factors -> reduce RealFunctions::'*'
```

#### 7.4.9.4 Base Expressions

*Base expression* notation includes representations for literal expressions, null expressions, invocation expressions, feature reference expressions (including using expression bodies as base expressions).

- *Literal expressions* are described in [7.4.9.5](#).
- A *null expression* is notated by the keyword **null**. A null expression always evaluates to a result of "no values", which is equivalent to the empty sequence `()`.
- An *invocation expression* can be directly represented by giving the qualified name for the function to be invoked followed by a list of argument expressions, surrounded by parentheses `()` and separated by commas. The parentheses must be included, even if the argument list is empty.

```
IntegerFunctions::'+'(i, j)
isInGoodStanding(member)
Computation()
```

If the qualified name given for an invocation expression resolves to an expression instead of a function, then the invocation expression is considered to subset the named expression, meaning that, effectively, the invocation is taken to be for the function of the named expression, as specialized by that expression.

```
function UnaryFunction {in x : Anything; return: Anything;}
function apply {
  in expr fn : UnaryFunction;
  in value : Anything;

  // Invokes UnaryFunction as specified by parameter fn.
  return : Anything = fn(value);
}
```

It is also possible to specify an expression to be invoked using a feature chain (see [7.3.4.6](#)).

```
class Stats {
  feature vales[1..*] : Real;
  expr avg { sum(values)/size(values) }
}
feature myStats : States {
  redefines feature values = (1.0, 2.0, 3.0);
}
feature myAvg = myStats.avg();
```

- A *feature reference expression* is represented simply by the qualified name of the feature being referenced.

```
member
spacecraft::mainAssembly::sensors
sensor::isActive
```

Note that the referenced feature may be an expression. The notation for a reference to an expression is distinguished from the notation for an invocation by not having following parentheses.

```
expr addOne : UnaryFunction {
  x istype Integer? (x as Integer) + 1: 0
}
feature two = apply(addOne, 1); // "addOne" is a reference to expr addOne
```

Rather than declaring a named expression in order to pass it as an argument, an *expression body* may be used directly as a base expression. In this case, any parameters must be declared as features with direction

within the expression body (see [7.4.8.3](#)). Such body expressions are particularly useful when used for the second argument of a function operation expression (see [7.4.9.3](#)).

```
feature two =  
  apply({in x; x istype Integer? (x as Integer) + 1: 0}, 1);  
feature incrementedValues =  
  values -> collect {in x: Number; x + 1};
```

### 7.4.9.5 Literal Expressions

A *literal expression* is represented by giving a lexical literal for the value of the expression.

- A *literal Boolean* is represented by either of the keywords **true** or **false**.
- A *literal string* is represented by a lexical string value surrounded by double quotes `"",` as specified in [Clause](#).

```
"This is a string literal."
```

- A *literal integer* is represented by a lexical decimal value as specified in [8.2.2.4](#). Note that notation is only provided for non-negative integers (i.e., natural numbers). Negative integers can be represented by applying the unary negation operator `-` (see [7.4.9.2](#)) to an unsigned decimal literal.

```
0  
1234
```

- A *literal real* is represented with a syntax constructed from lexical decimal values and exponential values (see [8.2.2.4](#)). The full real number notation allows for a literal with a decimal point, with or without an exponential part, as well as an exponential value without a decimal point.

```
3.14  
.5  
2.5E-10  
1E+3
```

- A *literal infinity* is represented by the symbol `*`.

## 7.4.10 Interactions

### 7.4.10.1 Interactions Overview

*Metamodel references:*

- *Concrete syntax*, [8.2.5.9](#)
- *Abstract syntax*, [8.3.4.9](#)
- *Semantics*, [8.4.3.10](#)

*Interactions* are behaviors that are also associations (see [7.4.7](#) and [7.4.5](#), respectively), classifying performances that are also links between occurrences (see [9.2.4](#) through [9.2.6](#)). They specify how the linked participants affect each other and collaborate.

*Transfers* are interactions between two participants (see [9.2.7](#)). They specify when things provided by one occurrence (via its output features) are accepted by another (via input features).

*Item flows* are steps that are also binary connectors (see [7.4.7](#) and [7.4.6](#), respectively), whose values are transfers. An item flow ensures that values from a nested output feature of the source feature will fill into a nested input feature of the target feature. *Succession item flows* are item flows that are also successions (see [7.4.6](#)). They identify transfers

that happen after their source (that is, after the end of the occurrence that provides the things being transferred) and before their target (that is, before the start of the occurrence accepting those things).

#### 7.4.10.2 Interaction Declaration

An interaction is declared as a behavior (see [7.4.7](#)), using the keyword **interaction**. If no owned subclassification is explicitly given for the interaction, then it is implicitly given default subclassifications to *both* the behavior `Performance` from the `Performances` library model (see [9.2.6](#)) and the association `BinaryLink` or the association `Link` from the `Links` library model (see [9.2.4](#)), depending on whether it is a binary interaction or not.

As a kind of behavior, if the interaction has owned subclassifications whose superclasses are behaviors, then the rules related to their parameters are the same as for any subclassifier behavior (see [7.4.7](#)). As a kind of association, the body of an interaction must declare at least two association ends. If the interaction has owned subclassifications whose superclassifiers are associations, the rules related to their association ends are the same as for any association that is a subclassifier (see [7.4.5](#)).

```
interaction Authorization {
  end feature client[*] : Computer;
  end feature server[*] : Computer;
  composite step login;
  composite step authorize;
  composite succession login then authorize;
}
```

#### 7.4.10.3 Item Flow Declaration

An item flow declaration is syntactically similar to a binary connector declaration (see [7.4.6](#)), using the keyword **flow**, or **succession flow** for a succession item flow. If no owned subsetting or owned redefinition is explicitly given, then the item flow is implicitly given a default subsetting to the item flow `transfers` from the `Transfers` model library (see [9.2.7](#)), or to the succession item flow `transfersBefore`, if a succession item flow is being declared. If an item flow has owned specializations (including all feature typings, subsettings, and redefinitions) whose general type is a behavior or a step, then the rules for the redefinition of the parameters of those behaviors and steps are the same as for the redefinition of the parameters of general behavior or step by a specializing step (see [7.4.7.3](#)).

Unlike a regular binary connector declaration, though, an item flow declaration does not directly specify the related features for the item flow. Instead, the declaration gives the *source output feature* for the transfer after the keyword **from** and the *target input Feature* for the transfer after the keyword **to**. The related features are then determined as the owning features of the features given in the item flow declaration. It is these related features that are constrained to have a common context with the item flow (see [7.4.6](#)), not the features actually given in the declaration.

```
class Vehicle {
  composite feature fuelTank {
    out feature fuelOut : Fuel;
  }
  composite feature engine {
    in feature fuelIn : Fuel;
  }
  // The item flow actually connects the fuelTank to the engine.
  // The transfer moves Fuel from fuelOut to fuelIn.
  flow fuelFlow from fuelTank::fuelOut to engine::fuelIn;
}
```

The source output and target input features of an item flow can also be specified using feature chains (see [7.3.4.6](#)). In this case, the related features are determined as the features identified by the chains, excluding the last feature. This is particularly useful when the desired related features are inherited features.



```

class Vehicle {
  composite feature fuelTank {
    out feature fuelOut : Fuel;
  }
  composite feature engine {
    in feature fuelIn : Fuel;
  }
}

feature vehicle : Vehicle {
  // The item flow actually connects the inherited fuelTank
  // feature to the inherited engine feature.
  flow fuelFlow from fuelTank.fuelOut to engine.fuelIn;
}

```

An item flow declaration can also include an explicit declaration of the type and/or multiplicity of the items that are flowing, after the keyword **of**. This asserts that any items transferred by the item flow have the declared type. In the absence of an item declaration, any values may flow across the item flow, consistent with the types of the source output and target input features.

```

flow of flowingFuel : Fuel from fuelTank.fuelOut to engine.fuelIn;

```

If no feature declaration or item declaration details are included in an item flow declaration, then the keyword **from** may also be omitted.

```

flow fuelTank.fuelOut to engine.fuelIn;

```

**Note.** Item flow are also commonly used to move data from the output parameters of one step to the input parameters of another step.

```

behavior TakePicture {
  composite step focus : Focus { out image : Image; }
  composite step shoot : Shoot { in image : Image; }
  // The use of a succession item flow means that focus must complete before
  // the image is transferred, after which shoot can begin.
  succession flow focus.image to shoot.image;
}

```

## 7.4.11 Feature Values

*Metamodel references:*

- Concrete syntax, [8.2.5.10](#)
- Abstract syntax, [8.3.4.10](#)
- Semantics, [8.4.3.11](#)

A *feature value* is a membership relationship (see [7.2.4](#)) between an owning feature and a *value expression*, whose result provides the value of the feature. The value is specified as either a *bound value* or an *initial value*, and as either a *concrete value* or a *default value*. A feature can have at most one feature value.

A concrete, bound feature value is declared using the symbol = followed by a representation of the value expression using the concrete syntax described in [7.4.9](#). This notation is appended to the declaration of the owning feature of the feature value.

```

feature monthsInYear : Natural = 12;
struct TestRecord {
  feature scores[1..*] : Integer;
}

```

```

    derived feature averageScore[1] : Rational = sum(scores)/size(scores);
}

```

Features that have a feature value of this form implicitly have a nested binding connector (see [7.4.6](#)) between the feature and the result of the value expression.

**Note.** The semantics of binding mean that such a feature value asserts that a feature is *equivalent* to the result of the value expression. To highlight this, a feature with such a feature value can be flagged as **derived** (though this is not required, nor is it required that the value of a **derived** feature be computed using a feature value – see also [7.3.4.2](#)).

A concrete, initial feature value is declared as above but using the symbol `:=` instead of `=`.

```

feature count : Natural := 0;

```

In this case, the feature also has an implicit nested step typed by a `FeatureWritePerformance` (see [9.2.8.2.8](#)) used to initialize the feature to the result of the value expression. Unlike in the case of a bound value, an initial value may be changed using subsequent `FeatureWritePerformances`.

A default feature value is declared similarly to the above, but with the keyword **default** preceding the symbol `=` or `:=`, depending on whether it is bound or initial. However, for a default, bound value, the symbol `=` may be elided.

```

struct Vehicle {
    feature mass : Real default 1500.0;
    feature engine : Engine default := standardEngine;
}
struct TestWithCutoff :> TestRecord {
    feature cutoff : Rational default = 0.75 * averageScore;
}

```

For a default value, no binding connector or initialization step is added to the feature declaration.

A feature value can be included with the following kinds of feature declaration:

- Feature (see [7.3.4.2](#))
- Step (see [7.4.7.3](#))
- Expression (see [7.4.8.3](#))
- Boolean expression and invariant (see [7.4.8.5](#))

```

behavior ProvidePower {
    in cmd : Command;
    out wheelTorque : Torque;

    composite step generate : GenerateTorque {
        in cmd = ProvidePower::cmd;
        out generatedTorque;
    }
    composite step apply : ApplyTorque {
        in generatedTorque = generate.generatedTorque;
        out appliedTorque = ProvidePower::wheelTorque;
    }
}

```

## 7.4.12 Multiplicities

*Metamodel references:*

- *Concrete syntax*, [8.2.5.11](#)
- *Abstract syntax*, [8.3.4.11](#)
- *Semantics*, [8.4.3.12](#)

Multiplicity is defined in the Core layer as a feature for specifying cardinalities (number of instances) of a type by enumerating all numbers the cardinality might be (see [7.3.2.2](#)). The Kernel layer provides a specific way to do this by specifying a *range* of cardinalities. A multiplicity range has *lower bound* and *upper bound* expressions that are evaluated to determine the lowest and highest cardinalities, with both expression evaluating to natural numbers (that is, of type `Natural` from the `ScalarValues` library model, see [9.3.2](#)). An upper bound value of `*` (infinity) means that the cardinality includes all numbers greater than or equal to the lower bound value.

A multiplicity range is written in the form `[lowerBound..upperBound]`, where each of *lowerBound* and *upperBound* is either a literal expression or a feature reference expression represented in the notation described in [7.4.9](#). Literal expressions can be used to specify a multiplicity range with fixed lower and/or upper bounds. If the result of the *lowerBound* expression is `*`, then the meaning of the multiplicity range is not defined.

A multiplicity range can also be written without the lower bound (or `..`). In this case, the result of the single expression is used as both the lower and upper bound of the range, unless the result is the infinite value `*`, in which case the lower bound is taken to be 0.

Multiplicity ranges can be used in the declaration of types, particularly features (see [7.3.4.2](#)).

```
class Automobile {
  feature n : Positive;
  composite feature wheels : Wheel[n]; // Equivalent to [n..n] for n < *
  feature driveWheels[2..n] subsets wheels;
}
feature autoCollection : Automobile[*]; // Equivalent to [0..*]
```

It is also possible to declare a multiplicity feature using the keyword **multiplicity**, optionally followed by a short name and/or name, and including either a multiplicity range or a subsetting of another multiplicity. A multiplicity declaration is a kind of feature declaration, and it can optionally include a body as in a generic feature declaration (see [7.3.4.2](#)).

```
multiplicity zeroOrMore [0..*];
multiplicity m subsets zeroOrMore;
```

If a multiplicity feature is declared in the body of a type, then this becomes the multiplicity of the type. A type can have at most one multiplicity, whether this is given in the declaration or the body of the type.

```
feature driveWheels subsets wheels {
  multiplicity [2..n];
}
feature autoCollection {
  multiplicity subsets zeroOrMore;
}
```

## 7.4.13 Metadata

*Metamodel references:*

- *Concrete syntax*, [8.2.5.12](#)
- *Abstract syntax*, [8.3.4.12](#)
- *Semantics*, [8.4.3.13](#)

*Metadata* is additional information on elements of a model that does not have any instance-level semantics (in the sense described in Core Overview). In general, metadata is specified in annotating elements (including comments and textual representations) attached to annotated elements (see [7.2.3](#)). A *metadata feature* is a kind of annotating element that allows for the definition of structured metadata with modeler-specified features. This may be used, for example, to add tool-specific information to a model that can be relevant to the function of various kinds of tooling that may use or process a model, or domain-specific information relevant to a certain project or organization.

A metadata feature is syntactically a feature (see [7.3.4](#)) that is typed by a single *metaclass*, which is a kind of structure (see [7.4.4](#)), with implicit multiplicity 1..1. If the metaclass has no features, then the metadata feature simply acts as a user-defined syntactic tag on the annotated element. If the metaclass has features, then the metadata feature must have nested features that redefine each of the features of its type, binding them to the results of model-level evaluable expressions (see [7.4.9](#)), which provide the values of the specified attributive metadata for the annotated element.

A metaclass is declared like a structure (see [7.4.4](#)), but using the keyword **metaclass**. If no owned subclassification is explicitly given for the metaclass, then it is implicitly given a default subclassification to the metaclass `Metaobject` from the `Metaobjects` library model (see [9.2.16](#)).

```
metaclass SecurityRelated;

metaclass ApprovalAnnotation {
    feature approved : Boolean;
    feature approver : String;
}
```

A metadata feature is declared using the keyword **metadata** (or the symbol @), optionally followed by a short name and/or name, followed by the keyword **typed by** (or the symbol :) and the qualified name of exactly one metaclass. If no short name or name is given, then the keyword **typed by** (or the symbol :) may also be omitted. One or more annotated elements are then identified for the metadata feature after the keyword **about**, indicating that the metadata feature has annotation relationships to each of the identified elements (see [7.2.3](#)).

```
metadata securityDesignAnnotation : SecurityRelated about SecurityDesign;
```

If the specified metaclass has features, then a body must be given for the metadata feature that declares features that redefine each of the features of the metaclass and binds them to the result of model-level evaluable expressions (see [7.4.9](#)). The nested features of a metadata feature must always have the same names as the names of the typing metaclass, so the shorthand prefix redefines notation (see [7.3.4.5](#)) is always used.

```
metadata ApprovalAnnotation about Design {
    feature redefines approved = true;
    feature redefines approver = "John Smith";
}
```

The keywords **feature** and/or **redefines** (or the equivalent symbol :>>) may be omitted in the declaration of a metadata feature.

```
metadata ApprovalAnnotation about Design {
    approved = true;
    approver = "John Smith";
}
```

If the metadata feature is an owned member of a namespace (see [7.2.4](#)), then the explicit identification of annotated elements can be omitted, in which case the annotated element is implicitly the containing namespace (see [7.2.3](#)).

```
class Design {
    // This metadata feature is implicitly about the class Design.
```

```

    @ApprovalAnnotation {
        approved = true;
        approver = "John Smith";
    }
}

```

If a metadata feature has one or more concrete features that directly or indirectly subset `Metaobject::annotatedElement`, then, for each annotated element of the metadata feature, there must be at least one such feature for which the metaclass of the annotated element conforms to all the types of the feature (which must all be specializations of the reflective metaclass `KerML::Element`, see [9.2.17](#)).

```

metaclass Command {
    // A metadata feature of this metaclass may annotate
    // a behavior or a step.
    subsets annotatedElement : KerML::Behavior;
    subsets annotatedElement : KerML::Step;
}

behavior Save specializes UserAction {
    @Command; // This is valid.
    redefine step doAction {
        @Command; // This is valid.
    }
}

struct Options {
    @Command; // This is INVALID.
}

```

If the metaclass of a metadata feature is a direct or indirect specialization of `Metaobjects::SemanticMetadata` (see [9.2.16.2.3](#)), then the annotated elements must all be types and the feature `SemanticMetadata::baseType` must be bound to a value of type `KerML::Type` (see [9.2.17](#)). Each type annotated by such semantic metadata has an implicit specialization added to a type determined from the `baseType` value as follows:

- If the annotated type is neither a classifier nor a feature, then the annotated type implicitly specializes the `baseType`.
- If the annotated type is a classifier and the `baseType` is a classifier, then annotated classifier implicitly subclassifies the `baseType`.
- If the annotated type is a classifier and the `baseType` is a feature, then the annotated classifier implicitly subclassifies each type of the `baseType`.
- If the annotated type is a feature and the `baseType` is a feature, then the annotated feature shall implicitly subset the `baseType`.
- In all other cases, no implicit specialization is added.

When evaluated in a model-level evaluable expression, the cast operator **as** (see [7.4.9.2](#)) may be used to cast a feature referenced as its first operand to the actual reflective metaclass value for this feature, which may then be bound to the `baseType` feature of `SemanticMetadata`.

```

behavior UserAction;
step userActions : UserAction[*] nonunique;

metaclass Command specializes SemanticMetadata {
    // The cast operation "userAction as KerML::Feature" has
    // type KerML::Feature, which conforms to the type Type of
    // baseType. Since userActions is a step, the expression
    // evaluates at model level to a value of type KerML::Step.
    baseType = userActions as KerML::Feature;
}

```

```

// Save implicitly subclassifies UserAction (which is the
// type of userActions).
behavior Save {
    @Command;
}

// previousAction implicitly subsets userActions.
step previousAction[1] {
    @Command;
}

```

**Release Note.** This use of the cast operator for "meta-casting" is a workaround until a more general notation for reflection is introduced.

## 7.4.14 Packages

*Metamodel references:*

- *Concrete syntax*, [8.2.5.13](#)
- *Abstract syntax*, [8.3.4.13](#)
- *Semantics*, [8.4.3.14](#)

*Packages* are namespaces used to group elements, without any instance-level semantics (as opposed to *Types*, which are namespaces with classification semantics, see [7.3.2](#)). A package is notated like a generic namespace (see [7.2.4.2](#)), but using the keyword **package** instead of **namespace**.

```

package AddressBooks {
    datatype Entry {
        feature name: String;
        feature address: String;
    }
    struct AddressBook {
        composite feature entries[*]: Entry;
    }
}

```

A package may also have one or more *filter conditions* for selecting a subset of its imported memberships. A filter condition is a Boolean-valued, model-level evaluable expression (see [7.4.9](#)) that must evaluate to true for any imported member of the package. These are notated using the keyword **filter** followed by the filter condition expression.

```

package Annotations {
    datatype ApprovalAnnotation {
        feature approved : Boolean;
        feature approver : String;
        feature level : Natural;
    }
    ...
}

package DesignModel {
    import Annotations::*;
    struct System {
        @ApprovalAnnotation {
            approved = true;
            approver = "John Smith";
        }
    }
}

```

```

        level = 2;
    }
}
...
}

package UpperLevelApprovals {
    // This package imports all direct or indirect members
    // of the DesignModel package that have been approved
    // at a level greater than 1.
    import DesignModel::*;
    filter @Annotations::ApprovalAnnotation &&
        Annotations::ApprovalAnnotation::approved &&
        Annotations::ApprovalAnnotation::level > 1;
}

```

A filter condition can operate on metadata on elements (see [7.4.13](#)), such as checking for a metadata feature of a particular type or accessing the values of the features of a metadata feature. For the purposes of filter condition expressions, every element is also considered to have an implicit metadata feature that is typed by a metaclass from the reflective library model of the KerML abstract syntax (see [9.2.17](#)). This enables filter conditions to test for the abstract syntax metaclass of an element and to access the values of abstract syntax meta-attributes.

**Implementation Note.** The implemented KerML library model currently contains the declaration of all abstract syntax metaclasses, but does not yet include any meta-attributes.

Note that a filter condition in a package will filter *all* imports of that Package. That is why full qualification is used for `Annotations::ApprovalAnnotation` in the example above, since imported elements of the `Annotations` package would be filtered out by the very filter condition in which the elements are intended to be used. This may be avoided by combining one or more filter conditions with a specific import, using the filtered import notation described in [7.2.4.4](#)).

```

package UpperLevelApprovals {
    // Recursively import all annotation data types and all
    // features of those types.
    import Annotations::*;

    // The filter condition for this import applies only to
    // elements imported from the DesignModel package.
    import DesignModel::*[@ApprovalAnnotation && approved && level > 1];
}

```

The KerML library package contains a complete model of the KerML abstract syntax represented in KerML itself. When a filter condition is evaluated on an element, abstract syntax metadata for the element can be tested as if the element had an implicit metadata feature typed by the type from the KerML package corresponding to the metaclass of the element.

```

package PackageApprovals {
    import Annotations::*;
    import KerML::*;

    // This imports all structures from the DesignModel that have
    // at least one owned feature and have been marked as approved.
    import DesignModel::*[@Structure &&
        @Structure::ownedFeature != null &&
        @ApprovalAnnotation &&
        ApprovalAnnotation::approved];
}

```





# 8 Metamodel

## 8.1 Metamodel Overview

This clause presents the normative specification of the *metamodel* for KerML, which includes the KerML concrete syntax, abstract syntax and semantics (though the complete semantics depends on the *model library* specified in [Clause 9](#)).

1. *Concrete syntax* specifies how the language appears to modelers. Modelers construct and review models using a textual notation that conforms to the concrete syntax specification (see [8.2](#)).
2. *Abstract syntax* specifies linguistic terms and relations between them (as opposed to library model terms), which may be expressed in the concrete syntax (see [8.3](#)). The abstract syntax omits aspects of the concrete syntax, such as delimiters and formatting, that do not affect what modelers are trying to expression. A concrete syntax representation of a model can be *parsed* into an abstract syntax representation, or an abstract syntax representation can be *serialized* into the concrete syntax notation. The mapping between the concrete and abstract syntax is given as part of the *grammar* specification for the concrete syntax (see [8.2.1](#) on the conventions for this).
3. *Semantics* specifies the interpretation of models as representations of or specifications for modeled systems (see [8.4](#)). The semantics for a *core* subset of the abstract syntax are specified using mathematical logic. Semantics for the rest of KerML are specified by mapping complicated abstract syntax constructs into equivalent models using the core subset, and, in particular, introducing *implicit* relationships to required elements from the KerML model library (see [8.4.1](#) on this approach).

As described in 6.1, KerML is divided into Root, Core and Kernel Layers, which cut across each of the above facets. The subclauses on Concrete Syntax ([8.2](#)) and Abstract Syntax ([8.3](#)) are each further subdivided into subclauses on the three layers, and then, within each layer, into subclauses following the package structure of the abstract syntax. Subclause [8.4](#) on Semantics only covers the Core and Kernel Layers, because Root Layer constructs do not have model-level semantics.

Throughout this clause, the names of elements from the KerML abstract syntax model appear in a "code" font. Further:

1. Names of metaclasses appear exactly as in the abstract syntax, including capitalization, except possibly with added pluralization. When used as English common nouns, e.g., "an `Element`", "multiple `FeatureTypings`", they refer to instances of the metaclass. E.g., "Elements can own other Elements" refers to instances of the metaclass `Element` that reside in models. This can be modified with the term "metaclass" as necessary to refer to the metaclass itself instead of its instances, e.g., "The `Element` metaclass is contained in the `Elements` package."
2. Names of properties of metaclasses, when used as English common nouns, e.g., "an `ownedRelatedElement`", "multiple `featuringTypes`", refer to values of the properties. This can be modified using the term "metaproperty" as necessary to refer to the metaproperty itself instead of its values, e.g., "The `ownedRelatedElement` metaproperty is contained in the `Elements` package."

Similar stylistic conventions apply to text about KerML models, except that an "*italic code*" front is used.

1. Convention 1 above applies to KerML *Types* (e.g., *Performance*), using "type" (or a more specialized term) instead of "metaclass" (e.g., "the *Performance* behavior").
2. Convention 2 above applies to KerML *Features* (e.g., *performances*), using "feature" (or a more specialized term) instead of "metaproperty" (e.g., "the *performances* step").

## 8.2 Concrete Syntax

## 8.2.1 Concrete Syntax Overview

The concrete syntax for KerML is a textual notation that can be used to express or construct an abstract syntax representation of a model. The *lexical structure* of the KerML textual notation defines how the string of characters in a text is divided into a set of *lexical elements*. Such lexical elements can be categorized as *whitespace*, *notes*, or *tokens*. Only tokens are significant for the mapping of the notation to the abstract syntax. The *syntactic structure* of the KerML textual notation defines how lexical tokens are grouped and mapped to an abstract syntax representation of a model.

Both the lexical syntactic structures are specified as *grammars* consisting of productions for lexical elements or non-terminal syntactic elements (see [Table 1](#)). The body of a production is specified using an Extended Backus Naur Form (EBNF) notation (see [Table 2](#)). The syntactic grammar includes further notations to describe how the concrete syntax maps to the abstract syntax element being synthesized (see [Table 3](#)).

Subclause [8.2.2](#) presents the lexical grammar for KerML. Subclauses [8.2.3](#), [8.2.3](#), and [8.2.5](#) then each present the portion of the syntactic grammar for KerML covering the Root, Core and Kernel Layers of KerML (see 6.1). Each of these subclauses is further divided into subclauses corresponding to each of the packages from the abstract syntax model (see [8.3](#)).

**Table 1. Grammar Production Definitions**

|   |   |
|---|---|
| LEXICAL_ELEMENT = ...                               | Define a production for the LEXICAL_ELEMENT.  |
| NonterminalElement :<br>AbstractSyntaxElement = ... | Define a production for the NonterminalElement that synthesizes the AbstractSyntaxElement. If the NonterminalElement has the same name as the AbstractSyntaxElement, then ": AbstractSyntaxElement" may be omitted. |

**Table 2. EBNF Notation Conventions**

|   |                     |
|---|---------------------|
| <b>Lexical element</b>                  | LEXICAL_ELEMENT     |
| <b>Terminal element</b>                 | 'terminal'          |
| <b>Non-terminal element</b>             | NonterminalElement  |
| <b>Sequential elements</b>              | Element1 Element2   |
| <b>Alternative elements</b>             | Element1   Element2 |
| <b>Optional elements (zero or one)</b>  | Element ?           |
| <b>Repeated elements (zero or more)</b> | Element *           |
| <b>Repeated elements (one or more)</b>  | Element +           |
| <b>Grouping</b>                         | ( Elements ... )    |

**Table 3. Abstract Syntax Synthesis Notation**

|                            |             |   |
|----------------------------|-------------|---|
| <b>Property assignment</b> | p = Element | Assign the result of parsing the concrete syntax Element to abstract syntax property p. |
|----------------------------|-------------|---|

|                                    |   |   |
|------------------------------------|---|---|
| <b>List property construction</b>  | <code>p += Element</code>               | Add the result of parsing the concrete syntax <code>Element</code> to the abstract syntax list property <code>p</code> .  |
| <b>Boolean property assignment</b> | <code>p ?= Element</code>               | If the concrete syntax <code>Element</code> is parsed, then set the abstract Boolean property <code>p</code> to true.   |
| <b>Non-parsing assignment</b>      | <pre>{ p = value } { p += value }</pre> | Assign (or add) the given <code>value</code> to the abstract syntax property <code>p</code> , without parsing any input. The <code>value</code> may be a literal or a reference to another abstract syntax property. The symbol "this" refers to the element being synthesized. |
| <b>Name resolution</b>             | <code>[QualifiedName]</code>            | Parse a <code>QualifiedName</code> , then resolve that name to an <code>Element</code> reference (see <a href="#">8.2.3.3.4</a> ) for use as a value in an assignment as above.   |

## 8.2.2 Lexical Structure

### 8.2.2.1 Line Terminators and White Space

```

LINE_TERMINATOR =
    implementation defined character sequence
LINE_TEXT =
    character sequence excluding LINE_TERMINATORS
WHITE_SPACE =
    space | tab | form_feed | LINE_TERMINATOR

```

#### Notes.

1. Notation text is divided up into lines separated by *line terminators*. A line terminator may be a single character (such as a line feed) or a sequence of characters (such as a carriage return/line feed combination). This specification does not require any specific encoding for a line terminator, but any encoding used must be consistent throughout any specific input text.
2. Any characters in text line that are not a part of the line terminator are referred to as *line text*.
3. A *white space* character is a space, tab, form feed or line terminator. Any contiguous sequence of white space characters can be used to separate tokens that would otherwise be considered to be part of a single

token. It is otherwise ignored, with the single exception that a line terminator is used to mark the end of a single-line note (see [8.2.2.2](#)).

### 8.2.2.2 Notes and Comments

```
SINGLE_LINE_NOTE =  
    '/' '/' LINE_TEXT  
  
MULTILINE_NOTE =  
    '/' '/' '*' COMMENT_TEXT '*' '/'  
  
REGULAR_COMMENT =  
    '/' '*' COMMENT_TEXT '*' '/'  
  
COMMENT_TEXT =  
    ( COMMENT_LINE_TEXT | LINE_TERMINATOR ) *  
  
COMMENT_LINE_TEXT =  
    LINE_TEXT excluding the sequence '*' '/'
```

### 8.2.2.3 Names

```
NAME =  
    BASIC_NAME | UNRESTRICTED_NAME  
  
BASIC_NAME =  
    BASIC_INITIAL_CHARACTER BASIC_NAME_CHARACTER *  
  
UNRESTRICTED_NAME =  
    single_quote ( NAME_CHARACTER | ESCAPE_SEQUENCE ) * single_quote  
    (see Note 1)  
  
BASIC_INITIAL_CHARACTER =  
    ALPHABETIC_CHARACTER | '_'  
  
BASIC_NAME_CHARACTER =  
    BASIC_INITIAL_CHARACTER | DECIMAL_DIGIT  
  
ALPHABETIC_CHARACTER =  
    any character 'a' through 'z' or 'A' through 'Z'  
  
DECIMAL_DIGIT =  
    any character '0' through '9'  
  
NAME_CHARACTER =  
    any printable character other than backslash or single_quote  
  
ESCAPE_SEQUENCE =  
    see Note 2
```

## Notes.

1. The `single_quote` character is `'`. The name represented by an `UNRESTRICTED_NAME` shall consist of the characters *within* the single quotes, with escape characters resolved as described below. The surrounding single quote characters are *not* part of the represented name.
2. An `ESCAPE_SEQUENCE` is a sequence of two text characters starting with a backslash that actually denotes only a single character, except for the newline escape sequence, which represents however many characters is necessary to represent an end of line in a specific implementation (see also [8.2.2.1](#)). [Table 4](#) shows the meaning of the allowed escape sequences. The `ESCAPE_SEQUENCES` in an `UNRESTRICTED_NAME` shall be replaced by the characters specified as their meanings in the actual represented name.

**Table 4. Escape Sequences**

| Escape Sequence | Meaning         |
|-----------------|-----------------|
| <code>\'</code> | Single Quote    |
| <code>\"</code> | Double Quote    |
| <code>\b</code> | Backspace       |
| <code>\f</code> | Form Feed       |
| <code>\t</code> | Tab             |
| <code>\n</code> | Line Terminator |
| <code>\\</code> | Backslash       |

### 8.2.2.4 Numeric Values

```
DECIMAL_VALUE =  
    DECIMAL_DIGIT+  
  
EXPONENTIAL_VALUE =  
    DECIMAL_VALUE ('e' | 'E') ('+' | '-')? DECIMAL_VALUE
```

## Notes.

1. A `DECIMAL_VALUE` may specify a natural literal, or it may be part of the specification of a real literal (see [8.2.5.8.4](#)). Note that a `DECIMAL_VALUE` does not include a sign, because negating a literal is an operator in the KerML Expression syntax.
2. An `EXPONENTIAL_VALUE` may be used in the specification of a real literal (see [8.2.5.8.4](#)). Note that a decimal point and fractional part are not included in the lexical structure of an exponential value. They are handled as part of the syntax of real literals.

### 8.2.2.5 String Value

```
STRING_VALUE =  
    "'" ( STRING_CHARACTER | ESCAPE_SEQUENCE ) * "'"
```

```
STRING_CHARACTER =  
    any printable character other than backslash or "'"
```

#### Notes.

1. ESCAPE\_SEQUENCE is specified in [8.2.2.3](#).

### 8.2.2.6 Reserved Words

A *reserved keyword* is a token that has the lexical structure of a basic name but cannot actually be used as a basic name. The following keywords are so reserved in KerML.

**about abstract alias all and as assign assoc behavior binding bool by chains  
class classifier comment composite conjugate conjugates conjugation connector  
datatype default derived disjoining disjoint doc element else end expr false  
feature featured featuring filter first flow for from function generalization  
hastype if implies import in inout interaction inv inverse inverting istype  
language member metaclass metadata multiplicity namespace nonunique not null  
of or ordered out package portion predicate private protected public readonly  
redefines redefinition relationship rep return specialization specializes step  
struct subclassifier subset subsets subtype succession then to true type typed  
typing xor**

Tooling for the KerML textual notation should generally highlight keywords relative to other text, for example by using boldface and/or distinctive coloring. However, while keywords are shown in boldface in this specification, the specification does not require any specific highlighting (or any highlighting at all), and KerML textual notation documents are expected to be interchanged as plain text (see also Clause 10 on Model Interchange).

### 8.2.2.7 Symbols

The *symbols* shown below are non-name tokens composed entirely of characters that are not alphanumeric. In some cases these symbols have no meaning themselves, but are used to allow unambiguous separation between other tokens that do have meaning. In other cases, they are distinguished notations in the KerML Expression sublanguage (see [8.2.5.8](#)) that map to particular library Functions or symbolic shorthand for meaningful relationships.

```
( ) { } [ ] ; , != % & * ** + - -> .. / : ::  
> :>> < <= = := == => > >= ? ?? @ ^ | ~
```

Some symbols are made of multiple characters that may themselves individually be valid symbol tokens. Nevertheless, a multi-symbol token is not considered a combination of the individual symbol tokens. For example, “: :” is considered a single token, not a combination of two “:” tokens. Input characters shall be grouped from left to right to form the longest possible sequence of characters to be grouped into a single token. So “a : : b” would be analyzed into four tokens: “a”, “: :”, “:” and “b” (which, as it turns out, is not a valid sequence of tokens in the KerML textual concrete syntax).

Certain keywords in the concrete syntax have an equivalent symbolic representation. For convenience, the concrete syntax grammar uses the following special lexical terminals, which match either the symbol or the corresponding keyword.

```
TYPED_BY      = ':' | 'typed' 'by'
SPECIALIZES  = ':>' | 'specializes'
SUBSETS      = ':>' | 'subsets'
REDEFINES    = ':>>' | 'redefines'
CONJUGATES   = '~' | 'conjugates'
```

## 8.2.3 Root Concrete Syntax

### 8.2.3.1 Elements and Relationships Concrete Syntax

#### 8.2.3.1.1 Elements

```
Element =
    'element' Identification ElementBody

Identification : Element =
    ( '<' shortName = NAME '>' )? ( name = NAME )?

ElementBody : Element =
    ';' | '{' OwnedElement* '}'

OwnedElement : Element =
    ownedRelationship += OwnedRelationship
    { ownedRelationship.source = this }
    | ownedRelationship += OwnedAnnotation
```

### 8.2.3.1.2 Relationships

```
Relationship =
  'relationship' Identification
  RelationshipRelatedElements
  RelationshipBody

OwnedRelationship : Relationship =
  'relationship' Identification
  ( 'to' RelationshipTargetList )?
  RelationshipBody

RelationshipRelatedElements : Relationship =
  ( 'from' RelationshipSourceList )?
  ( 'to' RelationshipTargetList )?

RelationshipSourceList : Relationship =
  RelationshipSource ( ',' RelationshipSource )*

RelationshipSource : Relationship =
  source += [QualifiedName]

RelationshipTargetList : Relationship =
  RelationshipTarget ( ',' RelationshipTarget )*

RelationshipTarget : Relationship =
  target += [QualifiedName]

RelationshipBody : Relationship =
  ';' | '{' RelationshipOwnedElement* '}'

RelationshipOwnedElement : Relationship =
  ownedRelatedElement += OwnedRelatedElement
  | ownedRelationship += OwnedAnnotation

OwnedRelatedElement : Element =
  NonFeatureElement | FeatureElement

OwnedRelatedRelationship : Relationship =
  'relationship' Identification
  RelationshipRelatedElements
  RelationshipBody
```

### 8.2.3.2 Annotations Concrete Syntax



### 8.2.3.2.1 Annotations

```
Annotation =
    annotatedElement = [Qualified Name]

OwnedAnnotation : Annotation =
    annotatingElement = AnnotatingElement
    { ownedRelatedElement += annotatingElement }

AnnotatingElement =
    Comment
    | Documentation
    | TextualRepresentation
    | MetadataFeature
```

### 8.2.3.2.2 Comments and Documentation

```
Comment =
    'comment' Identification
    ( 'about' annotation += Annotation
      { ownedRelationship += annotation }
      ( ',' annotation += Annotation
        { ownedRelationship += annotation } ) *
    ) ?
    body = REGULAR_COMMENT

Documentation =
    'doc' Identification
    body = REGULAR_COMMENT
```

#### Notes.

1. The text of a lexical `REGULAR_COMMENT` or `PREFIX_COMMENT` shall be processed as follows before it is included as the body of a `Comment` or `Documentation`:
  1. Remove the initial `/*` and final `*/` characters.
  2. Remove any white space immediately after the initial `/*`, up to and including the first line terminator (if any).
  3. On each subsequent line of the text:
    1. Strip initial white space other than line terminators.
    2. Then, if the first remaining character is `"`, remove it.
    3. Then, if the first remaining character is now a space, remove it.
2. The body text of a `Comment` can include markup information (such as HTML), and a conforming tool may display such text as rendered according to the markup. However, marked up "rich text" for a `Comment` written using the KerML textual concrete syntax shall be stored in the `Comment` body in plain text including all mark up text, with all line terminators and white space included as entered, other than what is removed according to the rules above.

### 8.2.3.2.3 Textual Representation

```
TextualRepresentation =  
  ( 'rep' Identification )?  
  'language' language = STRING_VALUE  
  body = REGULAR_COMMENT
```

#### Notes.

1. The lexical text of a REGULAR\_COMMENT shall be processed as specified in [8.2.3.2.2](#) for Comments before being included as the body of a TextualRepresentation.

### 8.2.3.3 Namespaces Concrete Syntax

### 8.2.3.3.1 Namespaces

```
Namespace =
    NamespaceDeclaration NamespaceBody

NamespaceDeclaration : Namespace =
    'namespace' Identification

NamespaceBody : Namespace =
    ';' | '{' NamespaceBodyElement* '}'

NamespaceBodyElement : Namespace =
    ownedRelationship += NamespaceMember
    | ownedRelationship += AliasMember
    | ownedRelationship += Import

MemberPrefix : Membership =
    ( visibility = VisibilityIndicator )?

VisibilityIndicator : VisibilityKind =
    'public' | 'private' | 'protected'

NamespaceMember : OwningMembership =
    NonFeatureMember
    | NamespaceFeatureMember

NonFeatureMember : OwningMembership =
    MemberPrefix
    ownedMemberElement = MemberElement

NamespaceFeatureMember : Membership =
    MemberPrefix
    ownedMemberElement = FeatureElement

AliasMember : Membership =
    MemberPrefix
    'alias' ( '<' memberShortName = NAME '>' )?
    ( memberName = NAME )?
    'for' memberElement = [Qualified Name]
    RelationshipBody

RootNamespace : Namespace =
    NamespaceBodyElement*
```

### 8.2.3.3.2 Imports

```
Import : Import =
  ( visibility = VisibilityIndicator )?
  'import' ( isImportAll ?= 'all' )?
  ( ImportedNamespace
    | ImportedFilterPackage )
  RelationshipBody

ImportedNamespace : Import =
  ( importedNamespace = [QualifiedName] '::' )?
  ( importedName = NAME | '*' )
  ( '::' isRecursive ?= '**' )?

ImportedFilterPackage : Import =
  importedNamespace = FilterPackage
  { ownedRelatedElement += importedNamespace }

FilterPackage : Package =
  ownedRelationship += FilterPackageImport
  ( ownedRelationship += FilterPackageMember )+

FilterPackageImport : Import =
  ImportedNamespace

FilterPackageMember : ElementFilterMembership =
  '[' condition = OwnedExpression ']'
  { visibility = 'private' }
```

### 8.2.3.3 Namespace Elements

```
MemberElement : Element =
    AnnotatingElement | NonFeatureElement

NonFeatureElement : Element =
    Element
    | Relationship
    | Namespace
    | Type
    | Classifier
    | DataType
    | Class
    | Structure
    | Metaclass
    | Association
    | AssociationStructure
    | Interaction
    | Behavior
    | Function
    | Predicate
    | Multiplicity
    | Package
    | Specialization
    | Conjugation
    | Subclassification
    | Disjoining
    | FeatureInverting
    | FeatureTyping
    | Subsetting
    | Redefinition
    | TypeFeaturing

FeatureElement : Feature =
    Feature
    | Step
    | Expression
    | BooleanExpression
    | Invariant
    | Connector
    | BindingConnector
    | Succession
    | ItemFlow
    | SuccessionItemFlow
```

### 8.2.3.4 Qualified Names and Name Resolution

```
QualifiedName =
    NAME ( '::' NAME ) *
```

A qualified name is notated as a sequence of *segment names* separated by "::" punctuation. An *unqualified* name can be considered the degenerate case of a qualified name with a single segment name. A qualified name is used in the KerML textual concrete syntax to identify an Element that is being referred to in the representation of another Element. A qualified name used in this way does not appear in the corresponding abstract syntax—instead, the

abstract syntax representation contains an actual reference to the identified Element. *Name resolution* is the process of determining the Element that is identified by a qualified name.

Qualified name resolution uses Namespace memberships to map simple names to named Elements. Every Namespace other than a root Namespace is nested in a containing Namespace called its *owningNamespace*. A root Namespace has an implicit containing namespace known as its *global namespace*. The global namespace for a root Namespace includes all the visible Memberships of all other root Namespaces that are *available* to the first Namespace, which shall include at least all the KerML Model Libraries (see [Clause 9](#)). A conforming tool can provide means for making additional Namespaces available to a root Namespace, but this specification does not define any standard mechanism for doing so.

An Element is considered to be *directly contained* in a Namespace if it is an *ownedElement* of the Namespace or if it is indirectly owned by the Namespace without any other intervening Namespace (e.g., if the Element is an *ownedRelatedElement* of a Relationship that is not a Membership but is an *ownedMember* of the Namespace). A Namespace defines a mapping from names to Elements directly contained in the Namespace, known as the *local resolution* of those names.

1. For each Element that is directly contained in a Namespace, but is *not* a member of the Namespace, the *shortName* and *effectiveName* of the Element, if non-null, locally resolve to that Element.
2. For each membership of a Namespace, the *memberShortName* and *memberName* of the Membership, if non-null, locally resolve to the *memberElement* of the Membership.

**Note.** If the Namespace is well formed, then there can be at most one Element that locally resolves to any given name.

The *visible resolution* of a name restricts the memberships in the second step to those that are visible outside the Namespace. (Note that resolution of names in the first step above is not restricted by visibility.) The *visible* Memberships of a Namespace shall comprise the following:

- All *ownedMemberships* of the Namespace with *visibility* = *public*.
- All *importedMemberships* of the Namespace that are derived from Import Relationships with *visibility* = *public*.
- If the Namespace is a Type, then all *inheritedMemberships* of the Type with *visibility* = *public*.

In general, the *full resolution* of a simple name relative to a Namespace then proceeds as follows:

1. If the name locally resolves to an Element directly contained in the Namespace, then it fully resolves to that Element.
2. If there is no such Element, then:
  - If the Namespace is *not* a root Namespace, then the name resolution continues with the *owningNamespace* of the Namespace.
  - If the Namespace *is* a root Namespace, then the name resolution continues with the global namespace.

The resolution of a simple name in the global namespace proceeds as follows:

1. If there is a Membership in the global namespace that has a *shortMemberName* or *memberName* equal to the simple name, then the name resolves to the *memberElement* of that Membership.
2. If there is no such Membership, then the name has no resolution.

**Note.** It is possible that there will be more than one Membership that resolve a given simple name. In this case, one of these Memberships is chosen for the resolution of the name, but which one is chosen is not otherwise determined by this specification.

**Implementation Note.** The pilot implementation currently only resolves the `ownedMemberNames` of `OwningMemberships` in the global namespace. Short names and aliases are *not* resolved.

A qualified name is always used to identify an Element that is a `target` Element of some *context* Relationship. The *context* Namespace is the nearest Namespace that directly or indirectly owns that Relationship. The *local namespace* for resolving the qualified name is then determined as follows:

- If the context Relationship is *not* a Membership or an Import, then the local namespace is the context Namespace.
- If the context Relationship *is* a Membership or an Import, then
  - If the context Namespace is *not* a root Namespace, then the local namespace is the `owningNamespace` of the context Namespace.
  - If the context Namespace *is* a root Namespace, then the local namespace is the global namespace for the context Namespace.

**Note.** Membership and Import Relationships are treated as a special case in order to avoid possible infinite recursion in the name resolution process.

The resolution of a qualified name begins with the full resolution of its first segment name with respect to the local namespace for the qualified name. If the qualified name has only one segment name, then the qualified name resolves to the resolution of its first segment name. Otherwise, each segment name of the qualified name, other than the last, must resolve to a Namespace that is the visible resolution of the name relative to the Namespace identified by the previous segment. The qualified name then resolves to the resolution of its last segment name.

**Note.** In the concrete syntax grammar productions, the notation `[QualifiedName]` is used to signify that a `Qualified Name` shall be parsed, then that name shall be resolved into a reference to an Element, per the rules given in this subclause, and that reference shall be inserted into the abstract syntax as specified in the production, not the `Qualified Name` itself (see [8.2.1](#)).

## 8.2.4 Core Concrete Syntax

### 8.2.4.1 Types Concrete Syntax

### 8.2.4.1.1 Types

```
Type =
  ( isAbstract ?= 'abstract' )? 'type'
  TypeDeclaration TypeBody

TypeDeclaration : Type =
  (t.isSufficient ?= 'all' )? Identification
  ( t.ownedRelationship += OwnedMultiplicity )?
  ( SpecializationPart | ConjugationPart )+
  DisjoiningPart?

SpecializationPart : Type =
  SPECIALIZES ownedRelationship += OwnedSpecialization
  ( ',' ownedRelationship += OwnedSpecialization )*

ConjugationPart : Type =
  CONJUGATES ownedRelationship += OwnedConjugation

DisjoiningPart : Type =
  'disjoint' 'from' ownedRelationship += OwnedDisjoining
  ( ',' ownedRelationship += OwnedDisjoining )*

TypeBody : Type =
  ';' | '{' TypeBodyElement* '}'

TypeBodyElement : Type =
  ownedRelationship += NonFeatureMember
  | ownedRelationship += FeatureMember
  | ownedRelationship += AliasMember
  | ownedRelationship += Import
```

### 8.2.4.1.2 Specialization

```
Specialization =
  ( 'specialization' Identification )?
  'subtype' SpecificType
  SPECIALIZES GeneralType
  RelationshipBody

OwnedSpecialization : Specialization =
  GeneralType

SpecificType : Specialization :
  specific = [QualifiedName]
  | specific += OwnedFeatureChain
  { ownedRelatedElement += specific }

GeneralType : Specialization =
  general = [QualifiedName]
  | general += OwnedFeatureChain
  { ownedRelatedElement += general }
```



#### 8.2.4.1.3 Conjugation

```
Conjugation =
  ( 'conjugation' Identification )?
  'conjugate'
  ( conjugatedType = [QualifiedName]
  | conjugatedType = FeatureChain
    { ownedRelatedElement += conjugatedType } )
  CONJUGATES
  ( originalType = [QualifiedName]
  | originalType = FeatureChain
    { ownedRelatedElement += originalType } )
  RelationshipBody

OwnedConjugation : Conjugation =
  originalType = [QualifiedName]
  | originalType = FeatureChain
    { ownedRelatedElement += originalType }
```

#### 8.2.4.1.4 Disjoining

```
Disjoining =
  ( 'disjoining' Identification )?
  'disjoint'
  ( typeDisjoined = [QualifiedName]
  | typeDisjoined = FeatureChain
    { ownedRelatedElement += typeDisjoined } )
  'from'
  ( disjoiningType = [QualifiedName]
  | disjoiningType = FeatureChain
    { ownedRelatedElement += disjoiningType } )
  RelationshipBody

OwnedDisjoining : Disjoining =
  disjoiningType = [QualifiedName]
  | disjoiningType = FeatureChain
    { ownedRelatedElement += disjoiningType }
```

#### 8.2.4.1.5 Feature Membership

```
FeatureMember : OwingMembership =
  TypeFeatureMember
  | OwnedFeatureMember

TypeFeatureMember : OwingMembership =
  MemberPrefix 'member' ownedMemberElement = FeatureElement

OwnedFeatureMember : FeatureMembership =
  MemberPrefix ownedMemberFeature = FeatureElement
```

#### 8.2.4.2 Classifiers Concrete Syntax

#### 8.2.4.2.1 Classifiers

```
Classifier =  
  ( isAbstract ?= 'abstract' ) 'classifier'  
  ClassifierDeclaration TypeBody  
  
ClassifierDeclaration : Classifier =  
  ( isSufficient ?= 'all' )? Identification  
  ( ownedRelationship += OwnedMultiplicity )?  
  ( SuperclassingPart | ConjugationPart )?  
  DisjoiningPart?  
  
SuperclassingPart : Classifier =  
  SPECIALIZES ownedRelationship += OwnedSubclassification  
  ( ',' ownedRelationship += OwnedSubclassification )*
```

#### 8.2.4.2.2 Subclassification

```
Subclassification =  
  ( 'specialization' Identification )?  
  'subclassifier' subclassifier = [QualifiedName]  
  SPECIALIZES superclassifier = [QualifiedName]  
  RelationshipBody  
  
OwnedSubclassification : Subclassification =  
  superclassifier = [QualifiedName]
```

#### 8.2.4.3 Features Concrete Syntax

#### 8.2.4.3.1 Features

```

Feature =
  FeaturePrefix
  'feature'? FeatureDeclaration
  ValuePart? TypeBody

FeaturePrefix : Feature =
  ( direction = FeatureDirection )?
  ( isAbstract ?= 'abstract' )?
  ( isComposite ?= 'composite' | isPortion ?= 'portion' )?
  ( isReadOnly ?= 'readonly' )?
  ( isDerived ?= 'derived' )?
  ( isEnd ?= 'end' )?

FeatureDirection : FeatureDirectionKind =
  'in' | 'out' | 'inout'

FeatureDeclaration : Feature =
  ( isSufficient ?= 'all' )? Identification
  ( FeatureSpecializationPart | ConjugationPart )?
  FeatureRelationshipPart*

FeatureRelationshipPart : Feature =
  ChainingPart
  | DisjoiningPart
  | InvertingPart
  | TypeFeaturingPart

ChainingPart : Feature =
  'chains' FeatureChain

InvertingPart : Feature =
  'inverse' 'of' ownedRelationship += OwnedFeatureInverting

TypeFeaturingPart : Feature =
  'featured' 'by' ownedRelationship += OwnedTypeFeaturing
  ( ',' ownedTypeFeaturing += OwnedTypeFeaturing )*

FeatureSpecializationPart : Feature =
  FeatureSpecialization+ MultiplicityPart? FeatureSpecialization*
  | MultiplicityPart FeatureSpecialization*

MultiplicityPart : Feature =
  ownedRelationship += OwnedMultiplicity
  | ( ownedRelationship += OwnedMultiplicity )?
  ( isOrdered ?= 'ordered' ( {isUnique = false} 'nonunique' )?
  | {isUnique = false} 'nonunique' ( isOrdered ?= 'ordered' )? )

FeatureSpecialization : Feature =
  Typings | Subsettings | Redefinitions

Typings : Feature =
  TypedBy ( ',' ownedRelationship += OwnedFeatureTyping )*

TypedBy : Feature =
  TYPED_BY ownedRelationship += OwnedFeatureTyping

Subsettings : Feature =

```

```

    Subsets ( ',' ownedRelationship += OwnedSubsetting ) *

Subsets : Feature =
    SUBSETS ownedRelationship += OwnedSubsetting

Redefinitions : Feature =
    Redefines ( ',' ownedRelationship += OwnedRedefinition ) *

Redefines : Feature =
    REDEFINES ownedRelationship += OwnedRedefinition

```

#### 8.2.4.3.2 Feature Typing

```

FeatureTyping =
    ( 'specialization' Identification )?
    'typing' typedFeature = [Qualified Name]
    TYPED_BY GeneralType
    RelationshipBody

OwnedFeatureTyping : FeatureTyping =
    GeneralType

```

#### 8.2.4.3.3 Subsetting

```

Subsetting =
    ( 'specialization' Identification )?
    'subset' SpecificType
    SUBSETS GeneralType
    RelationshipBody

OwnedSubsetting : Subsetting =
    GeneralType

```

#### 8.2.4.3.4 Redefinition

```

Redefinition =
    ( 'specialization' Identification )?
    'redefinition' SpecificType
    REDEFINES GeneralType
    RelationshipBody

OwnedRedefinition : Redefinition =
    GeneralType

```

#### 8.2.4.3.5 Feature Chaining

```
OwnedFeatureChain : Feature =  
    FeatureChain  
  
FeatureChain : Feature =  
    ownedRelationship += OwnedFeatureChaining  
    ( '.' ownedRelationship += OwnedFeatureChaining )+  
  
OwnedFeatureChaining : FeatureChaining =  
    chainingFeature = [QualifiedName]
```

#### 8.2.4.3.6 Feature Inverting

```
FeatureInverting =  
    ( 'inverting' Identification? )?  
    'inverse'  
    ( featureInverted = [QualifiedName]  
    | featureInverted = OwnedFeatureChain  
      { ownedRelatedElement += featureInverted } )  
    'of'  
    ( invertingFeature = [QualifiedName]  
    | ownedRelatedElement += OwnedFeatureChain  
      { ownedRelatedElement += invertingFeature } )  
    RelationshipBody  
  
OwnedFeatureInverting : FeatureInverting =  
    invertingFeature = [QualifiedName]  
    | invertingFeature = OwnedFeatureChain  
      { ownedRelatedElement += invertingFeature }
```

#### 8.2.4.3.7 Type Featuring

```
TypeFeaturing =  
    'featuring' ( Identification 'of')?  
    featureOfType = [QualifiedName]  
    'by' featuringType = [QualifiedName]  
    RelationshipBody  
  
OwnedTypeFeaturing : TypeFeaturing =  
    featuringType = [QualifiedName]
```

### 8.2.5 Kernel Concrete Syntax

#### 8.2.5.1 Data Types Concrete Syntax

```
DataType =  
    ( isAbstract ?= 'abstract' )? 'datatype'  
    ClassifierDeclaration TypeBody
```

### 8.2.5.2 Classes Concrete Syntax

```
Class =  
  ( isAbstract ?= 'abstract' )? 'class'  
  ClassifierDeclaration TypeBody
```

### 8.2.5.3 Structures Concrete Syntax

```
Structure =  
  ( isAbstract ?= 'abstract' )? 'struct'  
  ClassifierDeclaration TypeBody
```

### 8.2.5.4 Associations Concrete Syntax

```
Association =  
  ( isAbstract ?= 'abstract' )? 'assoc'  
  ClassifierDeclaration TypeBody  
  
AssociationStructure =  
  ( isAbstract ?= 'abstract' )? 'assoc' 'struct'  
  ClassifierDeclaration TypeBody
```

### 8.2.5.5 Connectors Concrete Syntax

#### 8.2.5.5.1 Connectors

```
Connector =  
  FeaturePrefix 'connector'  
  ConnectorDeclaration TypeBody  
  
ConnectorDeclaration : Connector =  
  BinaryConnectorDeclaration | NaryConnectorDeclaration  
  
BinaryConnectorDeclaration : Connector =  
  ( FeatureDeclaration? 'from' | isSufficient ?= 'all' 'from'? )?  
  ownedRelationship += ConnectorEndMember 'to'  
  ownedRelationship += ConnectorEndMember  
  
NaryConnectorDeclaration : Connector =  
  FeatureDeclaration  
  ( '(' ownedRelationship += ConnectorEndMember ','  
    ownedRelationship += ConnectorEndMember  
    ( ',' ownedRelationship += ConnectorEndMember )* ')' )?  
  
ConnectorEndMember : EndFeatureMembership =  
  ownedMemberFeature = ConnectorEnd  
  
ConnectorEnd : Feature =  
  ( name = NAME ';>' )?  
  ownedRelationship += OwnedSubsetting  
  ( ownedRelationship += OwnedMultiplicity )?
```

#### 8.2.5.5.2 Binding Connectors

```
BindingConnector =  
    FeaturePrefix 'binding'  
    BindingConnectorDeclaration TypeBody  
  
BindingConnectorDeclaration : BindingConnector =  
    ( FeatureDeclaration 'of' | isSufficient ?= 'all' 'of'? )?  
    ownedRelationship += ConnectorEndMember '='  
    ownedRelationship += ConnectorEndMember
```

#### 8.2.5.5.3 Successions

```
Succession =  
    FeaturePrefix 'succession'  
    SuccessionDeclaration TypeBody  
  
SuccessionDeclaration : Succession =  
    ( FeatureDeclaration? 'first' | s.isSufficient ?= 'all' 'first'? )?  
    ownedRelationship += ConnectorEndMember 'then'  
    ownedRelationship += ConnectorEndMember
```

#### 8.2.5.6 Behaviors Concrete Syntax

##### 8.2.5.6.1 Behaviors

```
Behavior =  
    ( isAbstract ?= 'abstract ')? 'behavior'  
    ClassifierDeclaration TypeBody
```

##### 8.2.5.6.2 Steps

```
Step =  
    FeaturePrefix 'step'  
    StepDeclaration TypeBody  
  
StepDeclaration : Step =  
    FeatureDeclaration ValuePart?
```

#### 8.2.5.7 Functions Concrete Syntax



### 8.2.5.7.1 Functions

```
Function =
  ( isAbstract ?= 'abstract' )? 'function'
  ClassifierDeclaration FunctionBody

FunctionBody : Type =
  ';' | '{' FunctionBodyPart '}'

FunctionBodyPart : Type =
  ( TypeBodyElement
  | ownedRelationship += ReturnFeatureMember
  )*
  ( ownedRelationship += ResultExpressionMember )?

ReturnFeatureMember : ReturnParameterMembership =
  MemberPrefix 'return'
  ownedMemberParameter = FeatureElement

ResultExpressionMember : ResultExpressionMembership =
  MemberPrefix
  ownedResultExpression = OwnedExpression
```

### 8.2.5.7.2 Expressions

```
Expression =
  FeaturePrefix 'expr'
  ExpressionDeclaration FunctionBody

ExpressionDeclaration : Expression =
  FeatureDeclaration ValuePart?
```

### 8.2.5.7.3 Predicates

```
Predicate =
  ( isAbstract ?= 'abstract' )? 'predicate'
  ClassifierDeclaration FunctionBody
```

### 8.2.5.7.4 Boolean Expressions and Invariants

```
BooleanExpression =
  FeaturePrefix 'bool'
  ExpressionDeclaration FunctionBody

Invariant =
  FeaturePrefix 'inv' ( 'true' | isNegated ?= 'false' )?
  ExpressionDeclaration FunctionBody
```

### 8.2.5.8 Expressions Concrete Syntax

#### 8.2.5.8.1 Operator Expressions

```

OwnedExpressionReferenceMember : FeatureMembership =
    ownedRelationship += OwnedExpressionReference

OwnedExpressionReference : FeatureReferenceExpression =
    ownedRelationship += OwnedExpressionMember

OwnedExpressionMember : FeatureMembership =
    ownedFeatureMember = OwnedExpression

OwnedExpression : Expression =
    ConditionalExpression
    | BinaryOperatorExpression
    | UnaryOperatorExpression
    | ClassificationExpression
    | ExtentExpression
    | PrimaryExpression

ConditionalExpression : OperatorExpression =
    'if' ownedRelationship += OwnedExpressionMember
    operator = '?'
    ownedRelationship += OwnedExpressionReferenceMember 'else'
    ownedRelationship += OwnedExpressionReferenceMember

ConditionalBinaryOperatorExpression : OperatorExpression =
    ownedRelationship += OwnedExpressionMember
    operator = ConditionalBinaryOperator
    ownedRelationship += OwnedExpressionReferenceMember

ConditionalBinaryOperator =
    '??' | 'or' | 'and' | 'implies'

BinaryOperatorExpression : OperatorExpression =
    ownedRelationship += OwnedExpressionMember
    operator = BinaryOperator
    ownedRelationship += OwnedExpressionMember

BinaryOperator =
    '|' | '&' | '^' | 'xor' | '==' | '!='
    | '..' | '<' | '>' | '<=' | '>=' | '+'
    | '-' | '*' | '/' | '%' | '^' | '**'

UnaryOperatorExpression : OperatorExpression =
    operator = UnaryOperator
    ownedRelationship += OwnedExpressionMember

UnaryOperator =
    '+' | '-' | '~' | 'not'

ClassificationExpression : OperatorExpression =
    ( ownedRelationship += OwnedExpressionMember )?
    operator = ClassificationOperator
    ownedRelationship += TypeReferenceMember

ClassificationOperator =
    'istype' | 'hastype' | '@' | 'as'

ExtentExpression : OperatorExpression =

```

```

operator = 'all'
ownedRelationship += TypeReferenceMember

TypeReferenceMember : FeatureMembership =
    ownedMemberFeature = TypeReference

TypeReference : Feature =
    ownedRelationship += ReferenceTyping

ReferenceTyping : FeatureTyping =
    type = [QualifiedNames]

```

## Notes.

1. *Operator expressions* provide a shorthand notation for InvocationExpressions that invoke a library Function represented as an *operator symbol*. [Table 5](#) shows the mapping from operator symbols to the Functions they represent from the Kernel Model Library (see [Clause 9](#)). An operator expression contains subexpressions called its *operands* that generally correspond to the *argument* Expressions of the InvocationExpression, except in the case of operators representing *control Functions*, in which case the evaluation of certain operands is as determined by the Function (see [8.4.3.9](#) for details).
2. Though not directly expressed in the syntactic productions given above, in any operator expression containing nested operator expressions, the nested expressions shall be implicitly grouped according to the *precedence* of the operators involved, as given in [Table 6](#). Operator expressions with higher precedence operators shall be grouped more tightly than those with lower precedence operators.

**Table 5. Operator Mapping**

| Operator       | Library Function         | Description                                | Model-Level Evaluable? |
|----------------|--------------------------|--|------------------------|
| <b>all</b>     | BaseFunctions::'all'     | Type extent                                | No                     |
| <b>istype</b>  | BaseFunctions::'istype'  | Is directly or indirectly instance of type | Yes                    |
| <b>hastype</b> | BaseFunctions::'hastype' | Is directly instance of type               | Yes                    |
| <b>as</b>      | BaseFunctions::as        | Select instances of type (cast)            | Yes                    |
| @              | BaseFunctions::'@'       | Same as 'istype'                           | Yes                    |
| ==             | BaseFunctions::'=='      | Equality                                   | Yes                    |
| !=             | BaseFunctions::'!='      | Inequality                                 | Yes                    |
| <b>xor</b>     | DataFunctions::'xor'     | Logical "exclusive or"                     | Yes                    |
| <b>not</b>     | DataFunctions::'not'     | Logical "not"                              | Yes                    |
|                | DataFunctions::' '       | Logical "inclusive or"                     | Yes                    |
| &              | DataFunctions::'&'       | Logical "and"                              | Yes                    |
| <              | DataFunctions::'<'       | Less than                                  | Yes                    |
| >              | DataFunctions::'>'       | Greater than                               | Yes                    |
| <=             | DataFunctions::'<='      | Less than or equal to                      | Yes                    |
| >=             | DataFunctions::'>='      | Greater than or equal to                   | Yes                    |

| Operator | Library Function            | Description                | Model-Level Evaluable? |
|----------|-----------------------------|----------------------------|------------------------|
| +        | DataFunctions::'+'          | Addition                   | Yes                    |
| -        | DataFunctions::'-'          | Subtraction                | Yes                    |
| *        | DataFunctions::'+'          | Multiplication             | Yes                    |
| /        | DataFunctions::'/'          | Division                   | Yes                    |
| %        | DataFunctions::'+'          | Remainder                  | Yes                    |
| ^ **     | DataFunctions::'^'          | Exponentiation             | Yes                    |
| ..       | DataFunctions::'..'         | Range construction         | Yes                    |
| ??       | ControlFunctions::'??'      | Null coalescing            | Yes                    |
| if       | ControlFunctions::'if'      | Conditional test (ternary) | Yes                    |
| or       | ControlFunctions::'or'      | Conditional "or"           | Yes                    |
| and      | ControlFunctions::'and'     | Conditional "and"          | Yes                    |
| implies  | ControlFunctions::'implies' | Logical "implication"      | Yes                    |

**Table 6. Operator Precedence (highest to lowest)**

|                              |
|------------------------------|
| <i>Unary</i>                 |
| <b>all</b>                   |
| <b>+ - ~ not</b>             |
| <i>Binary</i>                |
| <b>^ **</b>                  |
| <b>* / %</b>                 |
| <b>+ -</b>                   |
| <b>..</b>                    |
| <b>&lt; &gt; &lt;= &gt;=</b> |
| <b>istype hastype as @</b>   |
| <b>== !=</b>                 |
| <b>&amp; and</b>             |
| <b>xor</b>                   |
| <b>  or</b>                  |
| <b>implies</b>               |
| <b>??</b>                    |
| <i>Ternary</i>               |
| <b>if</b>                    |

#### 8.2.5.8.2 Primary Expressions

```

PrimaryExpression : Expression =
    FeatureChainExpression
    | NonFeatureChainPrimaryExpression

PrimaryExpressionMember : FeatureMembership =
    ownedMemberFeature = PrimaryExpression

NonFeatureChainPrimaryExpression : Expression =
    IndexExpression
    | SequenceExpression
    | SelectExpression
    | CollectExpression
    | FunctionOperationExpression
    | BaseExpression

NonFeatureChainPrimaryExpressionMember : FeatureMembership =
    ownedMemberFeature = NonFeatureChainPrimaryExpression

IndexExpression : OperatorExpression =
    ownedRelationship += PrimaryExpressionMember
    operator = '['
    ownedRelationship += OwnedExpressionMember ']'

SequenceExpression : Expression =
    '(' ( OwnedExpression | SequenceExpressionList ) ',' '?' ')'

SequenceExpressionList : OperatorExpression =
    ownedRelationship += OwnedExpressionMember
    operator = ','
    ( ownedRelationship += SequenceExpressionListMember
    | ownedRelationship += OwnedExpressionMember )

SequenceExpressionListMember : FeatureMembership =
    ownedMemberFeature = SequenceExpressionList

FeatureChainExpression : FeatureChainExpression =
    ownedRelationship += NonFeatureChainPrimaryExpressionMember '.'
    ownedRelationship += FeatureChainMember

CollectExpression : CollectExpression =
    ownedRelationship += PrimaryExpressionMember '.'
    ownedRelationship += BodyExpressionMember

SelectExpression : SelectExpression =
    ownedRelationship += PrimaryExpressionMember '.*'
    ownedRelationship += BodyExpressionMember

FunctionOperationExpression : InvocationExpression =
    ownedRelationship += PrimaryExpressionMember '->'
    ownedRelationship += ReferenceTyping
    ( ownedRelationship += BodyExpressionMember
    | ownedRelationship += FunctionReferenceExpressionMember
    | ArgumentList )

BodyExpressionMember : FeatureMembership =
    ownedMemberFeature = BodyExpression

```

```

FunctionExpressionMember : FeatureMembership =
    ownedMemberFeature = FunctionReferenceExpression

FunctionReferenceExpression : FeatureReferenceExpression =
    ownedRelationship += FunctionReferenceMember

FunctionReferenceMember : FeatureMembership =
    ownedMemberFeature = FunctionReference

FunctionReference : Expression =
    ownedRelationship += ReferenceTyping

FeatureChainMember : Membership =
    FeatureReferenceMember
    | OwnedFeatureChainMember

OwnedFeatureChainMember : OwningMembership =
    ownedMemberElement = FeatureChain

```

#### Notes.

1. Primary expressions provide additional shorthand notations for certain kinds of InvocationExpressions. For those cases in which the InvocationExpression is an OperatorExpression, its `operator` shall be resolved to the appropriate library function as given in [Table 7](#).

**Table 7. Primary Expression Operator Mapping**

| Operator | Library Function                        | Description                              | Model-level Evaluable? |
|----------|---|--|------------------------|
| [        | BaseFunctions:: <code>'['</code>        | Indexing                                 | Yes                    |
| ,        | BaseFunctions:: <code>','</code>        | Sequence construction                    | Yes                    |
| .        | ControlFunctions:: <code>'.'</code>     | Feature chaining                         | Yes                    |
| .        | ControlFunctions:: <code>collect</code> | Sequence collection (collect expression) | Yes                    |
| .?       | ControlFunctions:: <code>select</code>  | Sequence selection (select expression)   | Yes                    |



#### 8.2.5.8.3 Base Expressions

```

BaseExpression : Expression =
    NullExpression
    | LiteralExpression
    | FeatureReferenceExpression
    | InvocationExpression
    | BodyExpression

NullExpression : NullExpression =
    'null' | '(' ')'

FeatureReferenceExpression : FeatureReferenceExpression =
    ownedRelationship += FeatureReferenceMember

FeatureReferenceMember : Membership =
    memberElement = FeatureReference

FeatureReference : Feature =
    [QualifiedName]

InvocationExpression : InvocationExpression =
    ownedRelationship += OwnedSpecialization
    ArgumentList(this)

ArgumentList : InvocationExpression =
    '(' ( PositionalArgumentList | NamedArgumentList )? ')'

PositionalArgumentList : InvocationExpression =
    e.ownedRelationship += ArgumentMember
    ( ',' e.ownedRelationship += ArgumentMember ) *

ArgumentMember : ParameterMembership =
    ownedMemberParameter = Argument

Argument : Feature =
    ownedRelationship += ArgumentValue

NamedArgumentList : InvocationExpression =
    ownedRelationship += NamedArgumentMember
    ( ',' ownedRelationship += NamedArgumentMember ) *

NamedArgumentMember : FeatureMembership =
    ownedMemberFeature = NamedArgument

NamedArgument : Feature =
    ownedRelationship += ParameterRedefinition '='
    ownedRelationship += ArgumentValue

ParameterRedefinition : Redefinition =
    redefinedFeature = [QualifiedName]

ArgumentValue : FeatureValue =
    value = OwnedExpression

BodyExpression : FeatureReferenceExpression =
    ownedRelationship += ExpressionBodyMember

ExpressionBodyMember : FeatureMembership =

```

```

        ownedMemberFeature = ExpressionBody

ExpressionBody : Expression =
    '{' FunctionBodyPart '}'

```

#### 8.2.5.8.4 Literal Expressions

```

LiteralExpression : LiteralExpression =
    LiteralBoolean
    | LiteralString
    | LiteralInteger
    | LiteralReal
    | LiteralInfinity

LiteralBoolean : LiteralBoolean =
    value = BooleanValue

BooleanValue : Boolean =
    'true' | 'false'

LiteralString : LiteralString =
    value = STRING_VALUE

LiteralInteger : LiteralInteger =
    value = DECIMAL_VALUE

LiteralReal : LiteralReal =
    value = RealValue

RealValue : Real =
    DECIMAL_VALUE? '.' ( DECIMAL_VALUE | EXPONENTIAL_VALUE )
    | EXPONENTIAL_VALUE

LiteralInfinity : LiteralInfinity =
    '*'

```

#### 8.2.5.9 Interactions Concrete Syntax

##### 8.2.5.9.1 Interactions

```

Interaction =
    ( isAbstract ?= 'abstract' )? 'interaction'
    ClassifierDeclaration TypeBody

```

### 8.2.5.9.2 Item Flows

```
ItemFlow =
  FeaturePrefix 'flow'
  ItemFlowDeclaration TypeBody

SuccessionItemFlow =
  FeaturePrefix 'succession' 'flow'
  ItemFlowDeclaration TypeBody

ItemFlowDeclaration : ItemFlow =
  ( FeatureDeclaration
    ( 'of' ownedRelationship += ItemFeatureMember
      | ownedRelationship += EmptyItemFeatureMember )
    'from'
    | ( isSufficient ?= 'all' )?
      ownedRelationship += EmptyItemFeatureMember
    )
    ownedRelationship += ItemFlowEndMember 'to'
    ownedRelationship += ItemFlowEndMember

ItemFeatureMember : FeatureMembership =
  ownedMemberFeature = ItemFeature

ItemFeature : Feature =
  ( name = NAME ':' )?
  ( ownedTyping += OwnedFeatureTyping
    ( ownedRelationship += OwnedMultiplicity )?
    | ownedRelationship += OwnedMultiplicity
      ( ownedTyping += OwnedFeatureTyping )?
    )

EmptyItemFeatureMember : FeatureMembership =
  ownedMemberFeature = EmptyItemFeature

EmptyItemFeature : Feature =
  {}

ItemFlowEndMember : FeatureMembership =
  ownedMemberFeature = ItemFlowEnd

ItemFlowEnd : Feature =
  ( ownedRelationship += Subsetting '.' )?
  ownedRelationship += ItemFlowFeatureMember

ItemFlowFeatureMember : FeatureMembership =
  ownedMemberFeature = ItemFlowFeature

ItemFlowFeature : Feature =
  ownedRelationship += ItemFlowRedefinition

ItemFlowRedefinition : Redefinition =
  redefinedFeature = [Qualified Name]
```

### 8.2.5.10 Feature Values Concrete Syntax

```
ValuePart : Feature =
    ownedRelationship += FeatureValue

FeatureValue =
    ( '='
    | isInitial ?= ':= '
    | isDefault ?= 'default' ( '=' | isInitial ?= ':= ' )?
    )
    value = OwnedExpression
```

### 8.2.5.11 Multiplicities Concrete Syntax

```
Multiplicity =
    MultiplicitySubset | MultiplicityRange

MultiplicitySubset : Multiplicity =
    'multiplicity' Identification Subsets
    TypeBody

MultiplicityRange =
    'multiplicity' Identification MultiplicityBounds
    TypeBody

OwnedMultiplicity : OwingMembership =
    ownedMemberElement = OwnedMultiplicityRange

OwnedMultiplicityRange : MultiplicityRange =
    MultiplicityBounds

MultiplicityBounds : MultiplicityRange =
    '[' ( ownedRelationship += MultiplicityExpressionMember '..' )?
        ownedRelationship += MultiplicityExpressionMember ']'

MultiplicityExpressionMember : OwingMembership =
    ownedMemberElement = ( LiteralExpression | FeatureReferenceExpression )
```

### 8.2.5.12 Metadata Concrete Syntax

```
Metaclass =
  ( isAbstract ?= 'abstract' )? 'metaclass'
  ClassifierDeclaration TypeBody

OwnedMetadataFeatureAnnotation : Annotation =
  ownedRelatedElement += OwnedMetadataFeature

OwnedMetadataFeature : MetadataFeature
  ( '@' | 'metadata' )
  ( Identification ( ':' | 'typed' 'by' ) )?
  ownedRelationship += MetadataTyping
  TypeBody

MetadataFeature =
  ( '@' | 'metadata' )
  MetadataFeatureDeclaration
  ( 'about' annotation += Annotation
    { ownedRelationship += annotation }
    ( ',' annotation += Annotation
      { ownedRelationship += annotation } ) *
  )?
  MetadataBody

MetadataFeatureDeclaration : MetadataFeature =
  ( Identification ( ':' | 'typed' 'by' ) )?
  ownedRelationship += ownedFeatureTyping

MetadataBody : Feature =
  ';' | '{' ( ownedRelationship += MetadataBodyElement ) * '}'

MetadataBodyElement : Membership =
  NonFeatureMember
  | MetadataBodyFeatureMember
  | AliasMember
  | Import

MetadataBodyFeatureMember : FeatureMembership =
  ownedMemberFeature = MetadataBodyFeature

MetadataBodyFeature : Feature =
  'feature'? ( ':>' | 'redefines'? ownedRelationship += OwnedRedefinition
  FeatureSpecializationPart? ValuePart?
  MetadataBody
```

### 8.2.5.13 Packages Concrete Syntax

```
Package =  
    PackageDeclaration PackageBody  
  
PackageDeclaration =  
    'package' Identification  
  
PackageBody =  
    ';' |  
    | '{' ( NamespaceBodyElement  
        | ownedRelationship += ElementFilterMember  
        ) *  
    '}'  
  
ElementFilterMember : ElementFilterMembership =  
    MemberPrefix  
    'filter' condition = OwnedExpression ';' ;
```

## 8.3 Abstract Syntax

### 8.3.1 Abstract Syntax Overview

The KerML abstract syntax is specified as a UML model conforming to the CMOF conformance point of the Meta Object Facility Core Specification [MOF]. As shown in [Fig. 1](#), this model is divided into three top-level packages corresponding to the three layers of KerML (see [8.1](#)). Each top-level package contains nested packages for the modeling areas it addresses. Further, the Core package imports the Root package and the Kernel package imports the Core package, so that the Kernel package contains (as owned or imported members) all abstract syntax elements. [Fig. 2](#) shows the generalization hierarchy for all abstract syntax elements, other than those that represent KerML Relationships, and [Fig. 3](#) shows a similar hierarchy for all abstract syntax elements that represent Relationships.

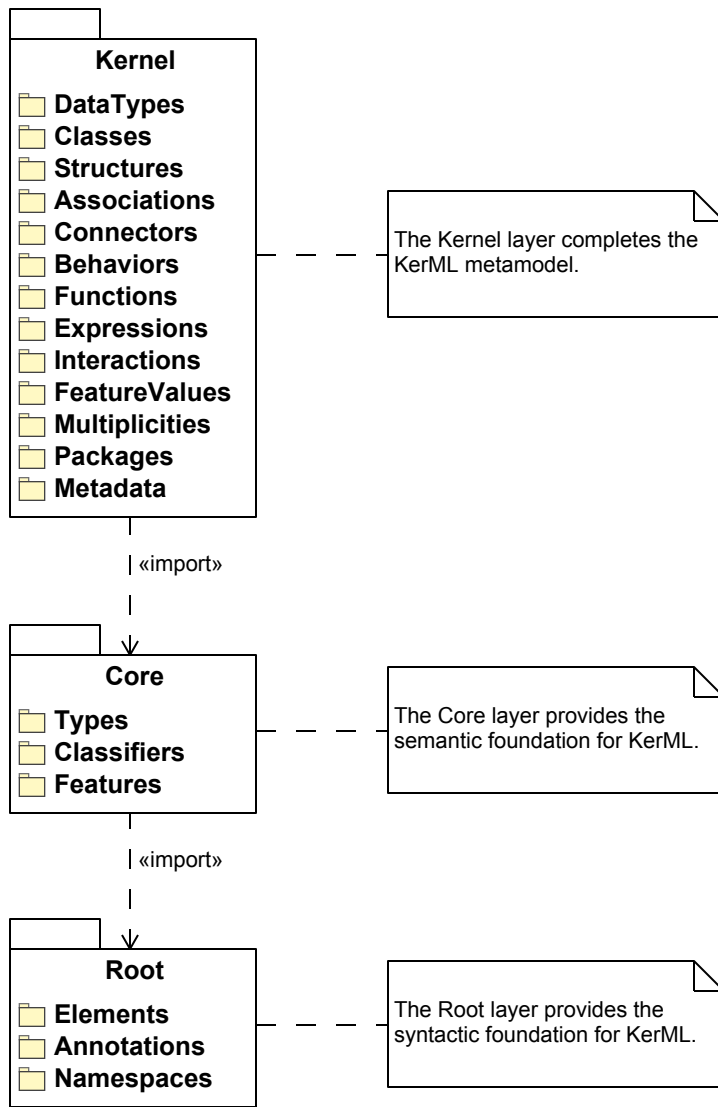


Figure 1. KerML Syntax Layers



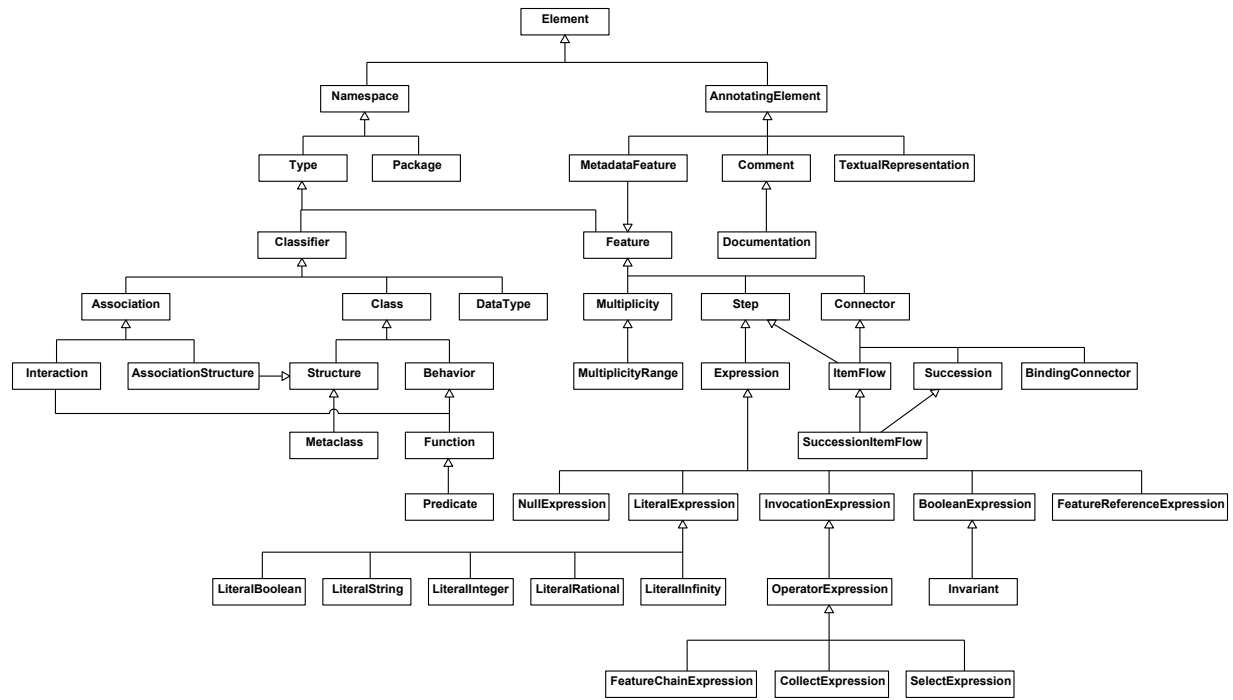


Figure 2. KerML Element Hierarchy

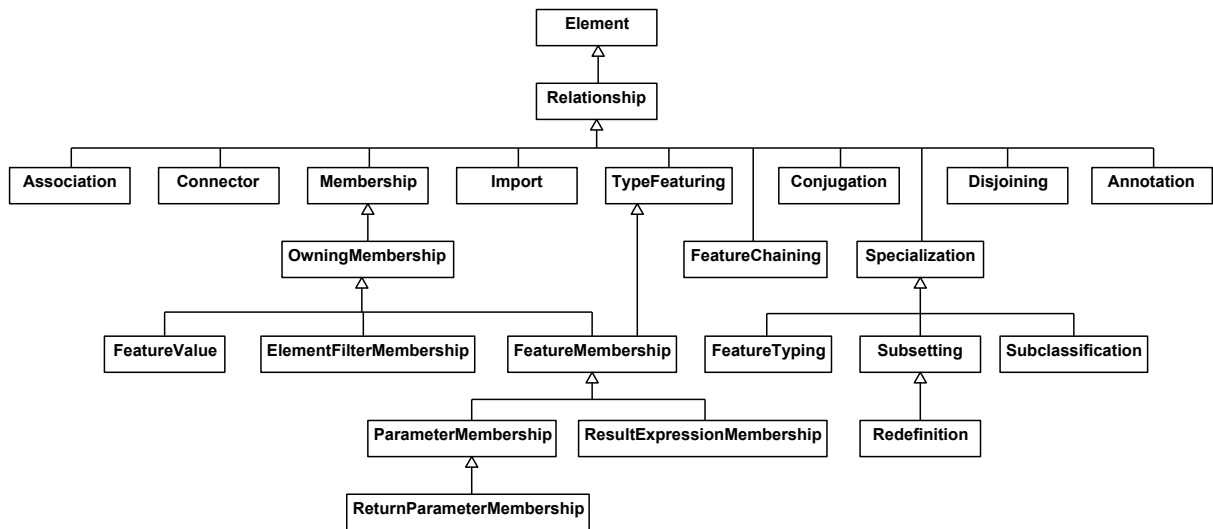
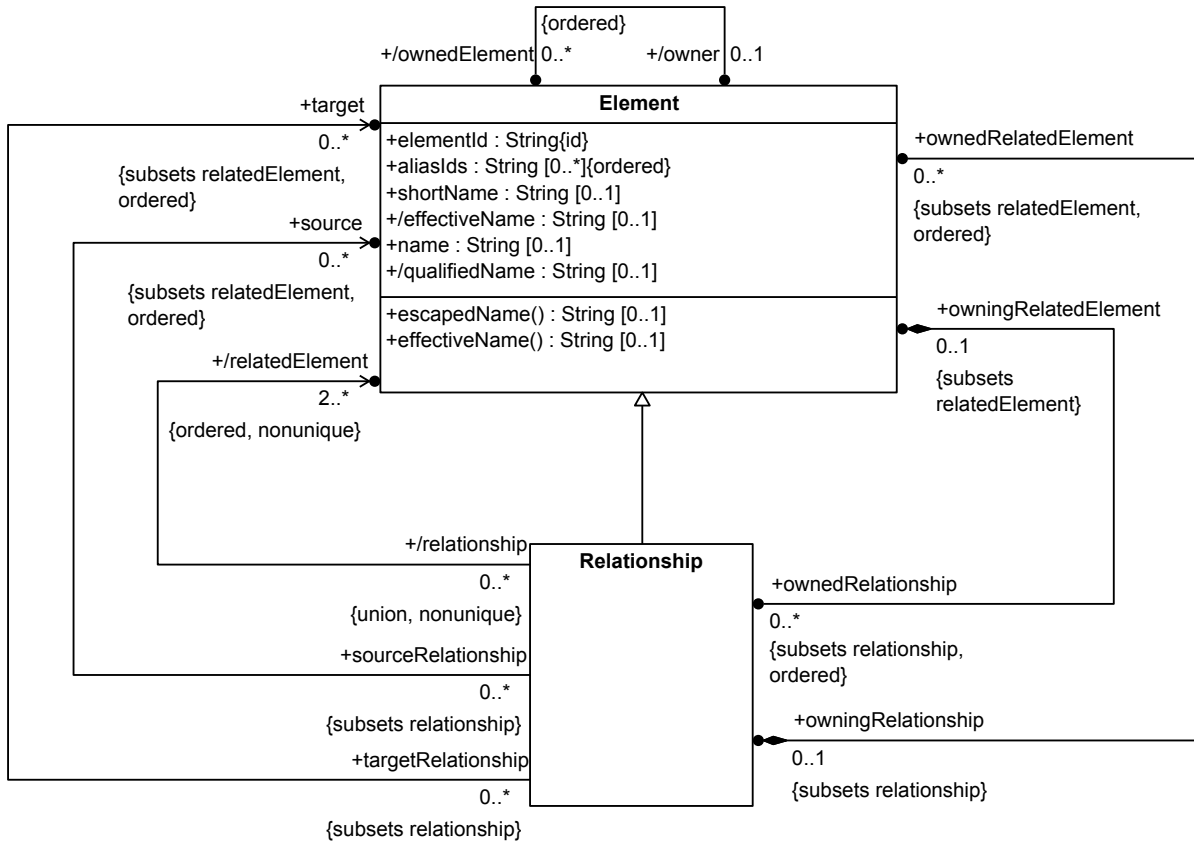


Figure 3. KerML Relationship Hierarchy

## 8.3.2 Root Abstract Syntax

### 8.3.2.1 Elements and Relationships Abstract Syntax

### 8.3.2.1.1 Overview



**Figure 4. Elements**

It is a general design principle of the KerML abstract syntax that non-Relationship Elements are related only by reified instances of Relationships. All other meta-associations between Elements are derived from these reified Relationships. For example, the `owningRelatedElement/ownedRelationship` meta-association between an Element and a Relationship is fundamental to establishing the structure of a model. However, the `owner/ownedElement` meta-association between two Elements is derived, based on the Relationship structure between them.

### 8.3.2.1.2 Element

#### Description

An Element is a constituent of a model that is uniquely identified relative to all other Elements. It can have Relationships with other Elements. Some of these Relationships might imply ownership of other Elements, which means that if an Element is deleted from a model, then so are all the Elements that it owns.

#### General Classes

None.

#### Attributes

`aliasIds : String [0..*] {ordered}`

Various alternative identifiers for this Element. Generally, these will be set by tools.

`/documentation : Documentation [0..*] {subsets ownedElement, annotatingElement, ordered}`

The Documentation owned by this Element.

`/effectiveName : String [0..1]`

The effective name to be used for this Element during name resolution within its `owningNamespace`.

`elementId : String`

The globally unique identifier for this Element. This is intended to be set by tooling, and it must not change during the lifetime of the Element.

`name : String [0..1]`

The primary name of this Element.

`/ownedAnnotation : Annotation [0..*] {subsets ownedRelationship, annotation, ordered}`

The `ownedRelationships` of this Element that are Annotations, for which this Element is the `annotatedElement`.

`/ownedElement : Element [0..*] {ordered}`

The Elements owned by this Element, derived as the `ownedRelatedElements` of the `ownedRelationships` of this Element.

`ownedRelationship : Relationship [0..*] {subsets relationship, ordered}`

The Relationships for which this Element is the `owningRelatedElement`.

`/owner : Element [0..1]`

The owner of this Element, derived as the `owningRelatedElement` of the `owningRelationship` of this Element, if any.

`owningMembership : OwningMembership [0..1] {subsets owningRelationship, membership}`

The `owningRelationship` of this Element, if that Relationship is a Membership.

`/owningNamespace : Namespace [0..1] {subsets namespace}`

The Namespace that owns this Element, derived as the `membershipOwningNamespace` of the `owningMembership` of this Element, if any.

`owningRelationship : Relationship [0..1] {subsets relationship}`

The Relationship for which this Element is an `ownedRelatedElement`, if any.

`/qualifiedName : String [0..1]`

The full ownership-qualified name of this Element, represented in a form that is valid according to the KerML textual concrete syntax for qualified names (including use of unrestricted name notation and escaped characters, as

necessary). The `qualifiedName` is null if this `Element` has no `owningNamespace` or if there is not a complete ownership chain of named `Namespaces` from a root `Namespace` to this `Element`.

`shortName` : `String` [0..1]

An optional alternative name for the `Element` that is intended to be shorter or in some way more succinct than its primary `name`. It may act as a modeler-specified identifier for the `Element`, though it is then the responsibility of the modeler to maintain the uniqueness of this identifier within a model or relative to some other context.

`/textualRepresentation` : `TextualRepresentation` [0..\*] {subsets `ownedElement`, `annotatingElement`, ordered}

The `textualRepresentations` that annotate this `Element`.

## Operations

`effectiveName()` : `String` [0..1]

Return the effective name for this `Element`. By default this is the same as its `name`, but, for certain kinds of `Elements`, this may be overridden if the `Element` `name` is empty (e.g., for redefining `Features`).

**body**: `name`

`escapedName()` : `String` [0..1]

Return `effectiveName`, if that is not null, otherwise `shortName`, if that is not null, otherwise null. If the returned name is non-null, it is returned as-is if it has the form of a basic name, or, otherwise, represented as a restricted name according to the lexical structure of the KerML textual notation (i.e., surrounded by single quote characters and with special characters escaped).

## Constraints

`elementDocumentation`

The documentation of an `Element` are its `ownedElements` that are `Documentation`.

```
documentation = ownedElement->selectByKind(Documentation)
```

`elementOwnedElements`

The `ownedElements` of an `Element` are the `ownedRelatedElements` of its `ownedRelationships`.

```
ownedElement = ownedRelationship.ownedRelatedElement
```

`elementQualifiedName`

If this `Element` does not have an `owningNamespace`, then its `qualifiedName` is empty. If the `owningNamespace` of this `Element` is a root `Namespace`, then the `qualifiedName` of the `Element` is the escaped name of the `Element` (if any). If the `owningNamespace` is non-empty but not a root `Namespace`, then the `qualifiedName` of this `Element` is constructed from the `qualifiedName` of the `owningNamespace` and the escaped name of the `Element`, unless the `qualifiedName` of the `owningNamespace` is empty, in which case the `qualifiedName` of this `Element` is also empty.

```
qualifiedName =  
    if owningNamespace = null then null  
    else if owningNamespace.owner = null then escapedName()
```

```

else if owningNamespace.qualifiedName = null then null
else owningNamespace.qualifiedName + '::' + escapedName()
endif endif endif

```

#### elementOwnedAnnotation

The `ownedAnnotations` of an `Element` are its `ownedRelationships` that are `Annotations`.

```

ownedAnnotation = ownedRelationship->selectByKind(Annotation)->
  select(a | a.annotatedElement = self)

```

#### elementOwner

The `owner` of an `Element` is the `owningRelatedElement` of its `owningRelationship`.

```

owner = owningRelationship.owningRelatedElement

```

#### elementEffectiveName

The `effectiveName` of an `Element` is given by the result of the `effectiveName()` operation.

```

effectiveName()

```

### 8.3.2.1.3 Relationship

#### Description

A `Relationship` is an `Element` that relates two or more other `Elements`. Some of its `relatedElements` may be owned, in which case those `ownedRelatedElements` will be deleted from a model if their `owningRelationship` is. A `Relationship` may also be owned by another `Element`, in which case the `ownedRelatedElements` of the `Relationship` are also considered to be transitively owned by the `owningRelatedElement` of the `Relationship`.

The `relatedElements` of a `Relationship` are divided into `source` and `target` `Elements`. The `Relationship` is considered to be directed from the `source` to the `target` `Elements`. An undirected `Relationship` may have either all `source` or all `target` `Elements`.

A "relationship `Element`" in the kernel abstract syntax is generically any `Element` that is an instance of either `Relationship` or a direct or indirect specialization of `Relationship`. Any other kind of `Element` is a "non-relationship `Element`". It is a convention of the kernel abstract syntax that non-relationship `Elements` are *only* related via reified relationship `Elements`. Any meta-associations directly between non-relationship `Elements` must be derived from underlying reified `Relationships`.

#### General Classes

`Element`

#### Attributes

`ownedRelatedElement` : `Element` [0..\*] {subsets `relatedElement`, ordered}

The `relatedElements` of this `Relationship` that are owned by the `Relationship`.

`owningRelatedElement` : `Element` [0..1] {subsets `relatedElement`}

The `relatedElement` of this `Relationship` that owns the `Relationship`, if any.

/relatedElement : Element [2..\*] {ordered, nonunique}

The Elements that are related by this Relationship, derived as the union of the source and target Elements of the Relationship. Every Relationship must have at least two relatedElements.

source : Element [0..\*] {subsets relatedElement, ordered}

The relatedElements from which this Relationship is considered to be directed.

target : Element [0..\*] {subsets relatedElement, ordered}

The relatedElements to which this Relationship is considered to be directed.

## Operations

No operations.

## Constraints

relationshipRelatedElement

The relatedElements of a Relationship consist of all of its source Elements followed by all of its target Elements.

relatedElement = source->union(target)

## 8.3.2.2 Annotations Abstract Syntax

### 8.3.2.2.1 Overview

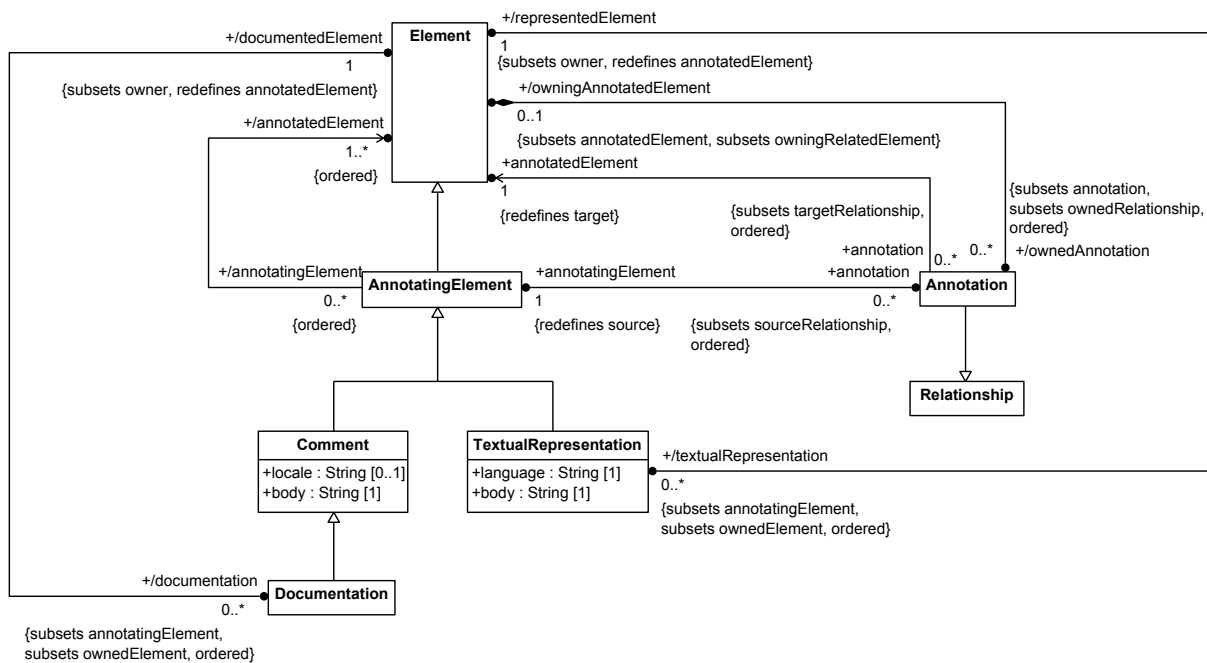


Figure 5. Annotation

### 8.3.2.2.2 AnnotatingElement

#### Description

An AnnotatingElement is an Element that provides additional description of or metadata on some other Element. An AnnotatingElement is attached to its `annotatedElement` by an Annotation Relationship.

#### General Classes

Element

#### Attributes

`/annotatedElement : Element [1..*] {ordered}`

The Elements that are annotated by this AnnotatingElement. If `annotation` is not empty, this is derived as the `annotatedElements` of the annotations. If `annotation`, then it is derived as the `owningNamespace` of the AnnotatingElement.

`annotation : Annotation [0..*] {subsets sourceRelationship, ordered}`

The Annotations that relate this AnnotatingElement to its `annotatedElements`.

#### Operations

No operations.

#### Constraints

`annotatingElementAnnotatedElement`

[no documentation]

```
annotatedElement =  
  if annotation->notEmpty() then annotation.annotatedElement  
  else owningNamespace endif
```

### 8.3.2.2.3 Annotation

#### Description

An Annotation is a Relationship between an AnnotatingElement and the Element that is annotated by that AnnotatingElement.

#### General Classes

Relationship

#### Attributes

`annotatedElement : Element {redefines target}`

The Element that is annotated by the `annotatingElement` of this Annotation.

`annotatingElement : AnnotatingElement {redefines source}`

The `AnnotatingElement` that annotates the `annotatedElement` of this `Annotation`.

`/owningAnnotatedElement : Element [0..1] {subsets annotatedElement, owningRelatedElement}`

The `annotatedElement` of this `Annotation`, when it is also its `owningRelatedElement`.

### Operations

No operations.

### Constraints

None.

#### 8.3.2.2.4 Comment

### Description

A `Comment` is an `AnnotatingElement` whose `body` in some way describes its `annotatedElements`.

### General Classes

`AnnotatingElement`

### Attributes

`body : String`

The annotation text for the `Comment`.

`locale : String [0..1]`

Identification of the language of the `body` text and, optionally, the region and/or encoding. The format shall be a POSIX locale conformant to ISO/IEC 15897, with the format `[language[_territory][.codeset][@modifier]]`.

### Operations

No operations.

### Constraints

None.

#### 8.3.2.2.5 Documentation

### Description

`Documentation` is a `Comment` that specifically documents a `documentedElement`, which must be its `owner`.

### General Classes

`Comment`

### Attributes



/documentedElement : Element {subsets owner, redefines annotatedElement}

The Element that is documented by this Documentation.

### Operations

No operations.

### Constraints

None.

## 8.3.2.2.6 TextualRepresentation

### Description

A TextualRepresentation is an AnnotatingElement whose `body` represents the `representedElement` in a given language. The `representedElement` must be the owner of the TextualRepresentation. The named `language` can be a natural language, in which case the `body` is an informal representation, or an artificial language, in which case the `body` is expected to be a formal, machine-parsable representation.

If the named `language` of a TextualRepresentation is machine-parsable, then the `body` text should be legal input text as defined for that `language`. The interpretation of the named language string shall be case insensitive. The following `language` names are defined to correspond to the given standard languages:

|                    |                            |
|--------------------|----------------------------|
| <code>kerml</code> | Kernel Modeling Language   |
| <code>ocl</code>   | Object Constraint Language |
| <code>alf</code>   | Action Language for fUML   |

Other specifications may define specific `language` strings, other than those shown in [Table](#), to be used to indicate the use of languages from those specifications in KerML TextualRepresentations.

If the `language` of a TextualRepresentation is "kerml", then the `body` text shall be a legal representation of the `representedElement` in the KerML textual concrete syntax. A conforming tool can use such a TextualRepresentation Annotation to record the original KerML concrete syntax text from which an Element was parsed. In this case, it is a tool responsibility to ensure that the `body` of the TextualRepresentation remains correct (or the Annotation is removed) if the annotated Element changes other than by re-parsing the `body` text.

An Element with a TextualRepresentation in a language other than KerML is essentially a semantically "opaque" Element specified in the other language. However, a conforming KerML tool may interpret such an element consistently with the specification of the named language.

### General Classes

AnnotatingElement

### Attributes

`body` : String

The textual representation of the `representedElement` in the given `language`.

`language` : String



`isImportAll = false` (the default), then only public Memberships are considered "visible". If `isImportAll = true`, then all Memberships are considered "visible", regardless of their declared visibility.

If no `importedMemberName` is given, then all visible Memberships are imported from the `importedNamespace`. If `isRecursive = true`, then visible Memberships are also recursively imported from all visible `ownedMembers` of the Namespace that are also Namespaces.

If an `importedMemberName` is given, then the Membership whose `effectiveMemberName` is that name is imported from the `importedNamespace`, if it is visible. If `isRecursive = true` and the imported `memberElement` is a Namespace, then visible Memberships are also recursively imported from that Namespace and its owned sub-Namespaces.

## General Classes

### Relationship

#### Attributes

`importedMemberName` : String [0..1]

The `effectiveMemberName` of the Membership of the `importedNamespace` to be imported. If not given, all public Memberships of the `importedNamespace` are imported.

`importedNamespace` : Namespace {redefines target}

The Namespace whose visible `members` are imported by this Import.

`/importOwningNamespace` : Namespace {subsets `owningRelatedElement`, redefines source}

The Namespace into which `members` are imported by this Import, which must be the `owningRelatedElement` of the Import.

`isImportAll` : Boolean

Whether to import memberships without regard to declared visibility.

`isRecursive` : Boolean

Whether to recursively import Memberships from visible, owned sub-namespaces.

`visibility` : VisibilityKind

The visibility level of the imported `members` from this Import relative to the `importOwningNamespace`.

## Operations

`importedMembership(excluded : Namespace [0..*])` : Membership [0..\*]

Returns the Memberships of the `importedNamespace` whose `memberElements` are to become imported members of the `importOwningNamespace`. By default, this is the set of publicly visible Memberships of the `importedNamespace`, but this may be overridden in specializations of Import. (The `excluded` parameter is used to handle the possibility of circular Import Relationships.)

```
body: let exclusions : Set(Namespace) =  
    excluded->including(importOwningNamespace) in
```

```

let visibleMemberships : Sequence(Membership) =
    importedNamespace.visibleMemberships(exclusions, false, isImportAll) in
let memberships : Sequence(Membership) =
    if importedMemberName = null then visibleMemberships
    else visibleMemberships->select(effectiveMemberName = importedMemberName)
    endif in
if not isRecursive then memberships
else memberships->union(
    memberships.ownedMember->selectAsKind(Namespace).
    visibleMemberships(exclusions, true, isImportAll))
endif

```

## Constraints

None.

### 8.3.2.3.3 Membership

#### Description

Membership is a Relationship between a Namespace and an Element that indicates the Element is a `member` of (i.e., is contained in) the Namespace. Any `memberNames` specify how the `memberElement` is identified in the Namespace and the `visibility` specifies whether or not the `memberElement` is publicly visible from outside the Namespace.

If a Membership is an `OwningMembership`, then it owns its `memberElement`, which becomes an `ownedMember` of the `membershipOwningNamespace`. Otherwise, the `memberNames` of a Membership are effectively aliases within the `membershipOwningNamespace` for an Element with a separate `OwningMembership` in the same or a different Namespace.

#### General Classes

Relationship

#### Attributes

`memberElement` : Element {redefines target}

The Element that becomes a member of the `membershipOwningNamespace` due to this Membership.

`/memberElementId` : String

The `elementId` of the `memberElement`.

`memberName` : String [0..1]

The name of the `memberElement` relative to the `membershipOwningNamespace`.

`/membershipOwningNamespace` : Namespace {subsets `membershipNamespace`, `owningRelatedElement`, redefines source}

The Namespace of which the `memberElement` becomes a member due to this Membership.

`memberShortName` : String [0..1]

The short name of the `memberElement` relative to the `membershipOwningNamespace`.

visibility : VisibilityKind

Whether or not the Membership of the `memberElement` in the `membershipOwningNamespace` is publicly visible outside that Namespace.

## Operations

isDistinguishableFrom(other : Membership) : Boolean

Whether this Membership is distinguishable from a given other Membership. By default, this is true if this Membership has no `memberShortName` or `memberName`; or each of the `memberShortName` and `memberName` are different than both of those of the other Membership; or neither of the metaclasses of the `memberElement` of this Membership and the `memberElement` of the other Membership conform to the other. But this may be overridden in specializations of Membership.

```
body: not (memberElement.oclKindOf(other.memberElement.oclType()) or
            other.memberElement.oclKindOf(memberElement.oclType())) or
(shortMemberName = null or
 (shortMemberName <> other.shortMemberName and
  shortMemberName <> other.memberName)) and
(memberName = null or
 (memberName <> other.shortMemberName and
  memberName <> other.memberName))
```

## Constraints

None.

### 8.3.2.3.4 Namespace

#### Description

A Namespace is an Element that contains other Elements, known as its `members`, via Membership Relationships with those Elements. The `members` of a Namespace may be owned by the Namespace, aliased in the Namespace, or imported into the Namespace via Import Relationships with other Namespaces.

A Namespace can provide names for its `members` via the `memberNames` specified by the Memberships in the Namespace. If a Membership specifies a `memberName`, then that is the name of the corresponding `memberElement` relative to the Namespace. Note that the same Element may be the `memberElement` of multiple Memberships in a Namespace (though it may be owned at most once), each of which may define a separate alias for the Element relative to the Namespace.

#### General Classes

Element

#### Attributes

/importedMembership : Membership [0..\*] {subsets membership, ordered}

The Memberships in this Namespace that result from Import Relationships between the Namespace and other Namespaces.

/member : Element [0..\*] {ordered}

The set of all member Elements of this Namespace, derived as the `memberElements` of all memberships of the Namespace.

`/membership : Membership [0..*] {ordered, union}`

All Memberships in this Namespace, including (at least) the union of `ownedMemberships` and `importedMemberships`.

`/ownedImport : Import [0..*] {subsets sourceRelationship, ownedRelationship, ordered}`

The `ownedRelationships` of this Namespace that are Imports, for which the Namespace is the `importOwningNamespace`.

`/ownedMember : Element [0..*] {subsets member, ordered}`

The `owned members` of this Namespace, derived as the `ownedMemberElements` of the `ownedMemberships` of the Namespace.

`/ownedMembership : Membership [0..*] {subsets membership, sourceRelationship, ownedRelationship, ordered}`

The `ownedRelationships` of this Namespace that are Memberships, for which the Namespace is the `membershipOwningNamespace`.

## Operations

`importedMemberships(excluded : Namespace [0..*]) : Membership [0..*]`

Derive the imported Memberships of this Namespace as the `importedMembership` of all `ownedImports`, excluding those Imports whose `importOwningNamespace` is in the `excluded set`, and excluding Memberships that have distinguishability collisions with each other or with any `ownedMembership`.

```
body: ownedImport->
  excluding(excluded->contains(importOwningNamespace)) .
  importedMembership(excluded)
```

`namesOf(element : Element) : String [0..*]`

Return the names of the given `element` as it is known in this Namespace.

```
body: let elementMemberships : Sequence(Membership) =
  memberships->select(memberElement = element)
in
  memberships.memberShortName->
    union(memberships.memberName)->
    asSet()
```

`visibilityOf(mem : Membership) : VisibilityKind`

Returns this visibility of `mem` relative to this Namespace. If `mem` is an `importedMembership`, this is the `visibility` of its Import. Otherwise it is the `visibility` of the Membership itself.

```
body: if importedMembership->includes(mem) then
  ownedImport->any(importedMembership(Set{})->includes(mem)) .visibility
else if memberships->includes(mem) then
  mem.visibility
else
```

```

    VisibilityKind::private
endif

```

`visibleMemberships(excluded : Namespace [0..*], isRecursive : Boolean, includeAll : Boolean) : Membership [0..*]`

If `includeAll = true`, then return all the Memberships of this Namespace. Otherwise, return only the publicly visible Memberships of this Namespace (which includes those `ownedMemberships` that have a visibility of public and those `importedMemberships` imported with a visibility of public). If `isRecursive = true`, also recursively include all visible Memberships of any visible owned Namespaces.

```

body: let publicMemberships : Sequence(Membership) =
    ownedMembership->
        select(visibility = VisibilityKind::public)->
        union(ownedImport->
            select(visibility = VisibilityKind::public).
            importedMembership(excluded)) in
if not isRecursive then publicMemberships
else publicMemberships->union(publicMemberships->
    selectAsKind(Namespace).
    publicMembership(excluded->including(this), true))
endif

```

## Constraints

### `namespaceMembers`

The members of a Namespace are the `memberElements` of all its memberships.

```
member = membership.memberElement
```

### `namespaceOwnedMember`

The `ownedMembers` of a Namespace are the `ownedMemberElements` of all its `ownedMemberships` that are `OwningMemberships`.

```
ownedMember = ownedMembership->selectByKind(OwningMembership).ownedMemberElement
```

### `namespaceOwnedMembership`

The `ownedMemberships` of a Namespace are all its `ownedRelationships` that are `Memberships`.

```
ownedMembership = ownedRelationship->selectByKind(Membership)
```

### `namespaceOwnedImport`

The `ownedImports` of a Namespace are all its `ownedRelationships` that are `Imports`.

```
ownedImport = ownedRelationship->selectByKind(Import)
```

### `namespaceDistinguishability`

All memberships of a Namespace must be distinguishable from each other.

```
membership->forAll(m1 | membership->forAll(m2 | m1 <> m2 implies m1.isDistinguishableFrom(m2)))
```

### `namespaceImportedMembership`

The `importedMemberships` of a Namespace are derived using the `importedMemberships()` operation, with no initially excluded Namespaces.

```
importedMembership = importedMemberships(Set{})
```

### 8.3.2.3.5 VisibilityKind

#### Description

VisibilityKind is an enumeration whose literals specify the visibility of a Membership of an Element in a Namespace outside of that Namespace. Note that "visibility" specifically restricts whether an Element in a Namespace may be referenced by name from outside the Namespace and only otherwise restricts access to an Element as provided by specific constraints in the abstract syntax (e.g., preventing the import or inheritance of private Elements).

#### General Classes

None.

#### Literal Values

`private`

Indicates a Membership is not visible outside its owning Namespace.

`protected`

An intermediate level of visibility between `public` and `private`. By default, it is equivalent to `private` for the purposes of normal access to and import of Elements from a Namespace. However, other Relationships may be specified to include Memberships with `protected` visibility in the list of `memberships` for a Namespace (e.g., Generalization).

`public`

Indicates that a Membership is publicly visible outside its owning Namespace.

### 8.3.2.3.6 OwningMembership

#### Description

An OwningMembership is a Membership that owns its `memberElement` as a `ownedRelatedElement`. The `ownedMemberElementM` becomes an `ownedMember` of the `membershipOwningNamespace`.

#### General Classes

Membership

#### Attributes

`ownedMemberElement` : Element {subsets `ownedRelatedElement`, redefines `memberElement`}

The Element that becomes an `ownedMember` of the `membershipOwningNamespace` due to this OwningMembership.

`/ownedMemberElementId` : String {redefines `memberElementId`}



The `elementId` of the `ownedMemberElement`.

```
/ownedMemberName : String [0..1] {redefines memberName}
```

The `effectiveName` of the `ownedMemberElement`.

```
/ownedMemberShortName : String [0..1] {redefines memberShortName}
```

The `shortName` of the `ownedMemberElement`.

### **Operations**

No operations.

### **Constraints**

`owningMembershipOwnedMemberShortName`

The `ownedMemberName` of an `OwningMembership` is the `effectiveName` of its `ownedMemberElement`.

```
ownedMemberShortName = ownedMemberElement.shortName
```

`owningMembershipOwnedMemberName`

The `ownedMemberName` of an `OwningMembership` is the `effectiveName` of its `ownedMemberElement`.

```
ownedMemberName = ownedMemberElement.effectiveName
```

## **8.3.3 Core Abstract Syntax**

### **8.3.3.1 Types Abstract Syntax**

### 8.3.3.1.1 Overview

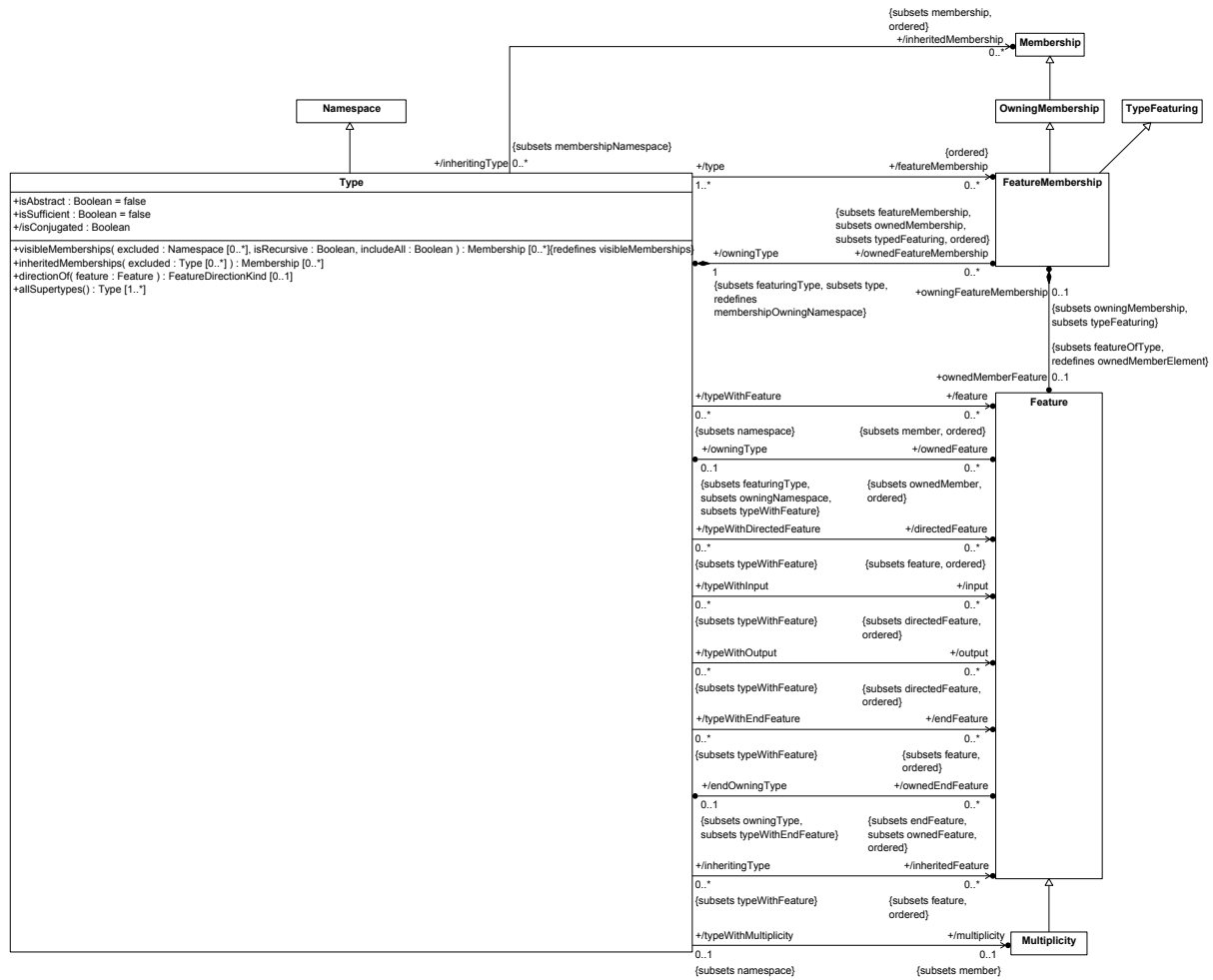


Figure 7. Types

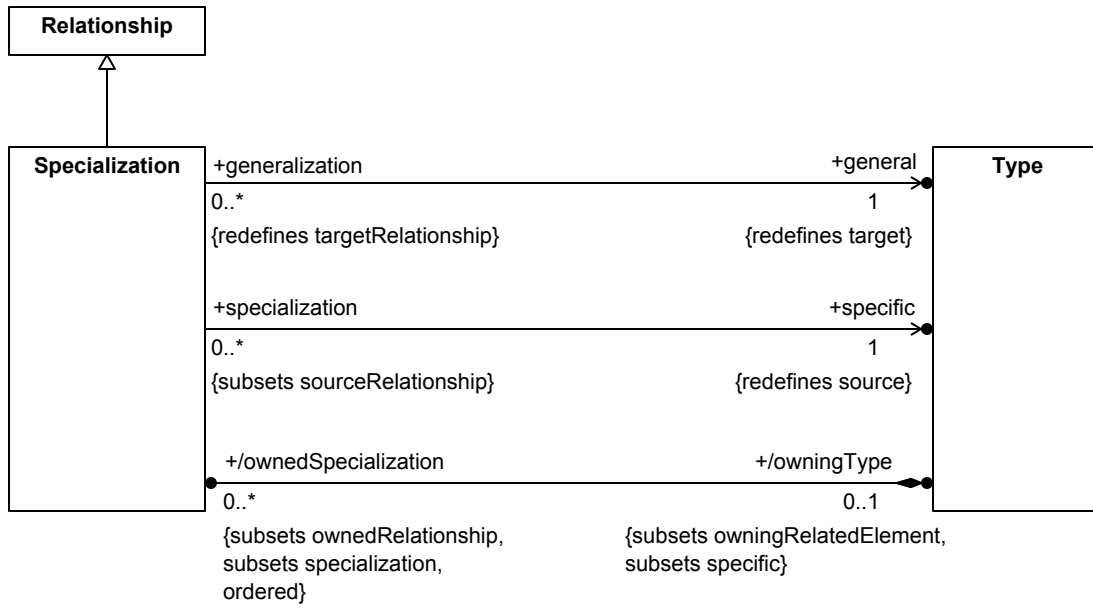


Figure 8. Specialization

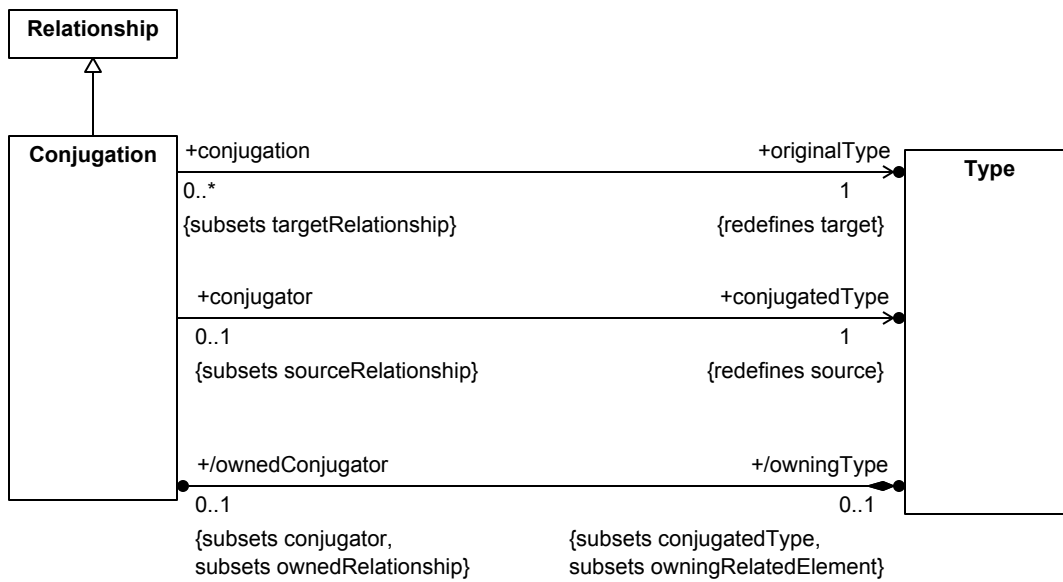
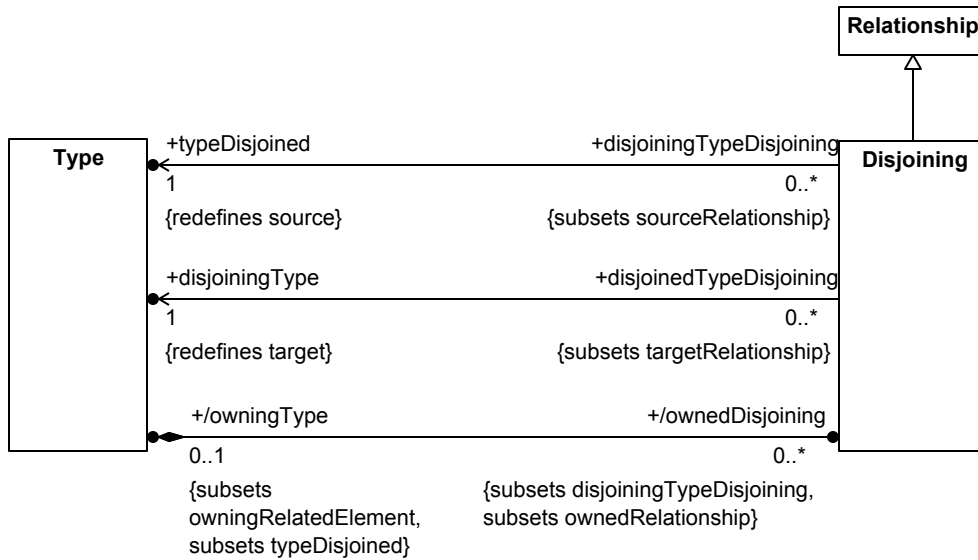


Figure 9. Conjugation



**Figure 10. Disjointness**

### 8.3.3.1.2 Conjugation

#### Description

Conjugation is a Relationship between two types in which the `conjugatedType` inherits all the Features of the `originalType`, but with all input and output Features reversed. That is, any Features with a FeatureMembership with direction *in* relative to the `originalType` are considered to have an effective direction of *out* relative to the `conjugatedType` and, similarly, Features with direction *out* in the `originalType` are considered to have an effective direction of *in* in the `originalType`. Features with direction *inout*, or with no direction, in the `originalType`, are inherited without change.

A Type may participate as a `conjugatedType` in at most one Conjugation relationship, and such a Type may not also be the `specific` Type in any Generalization relationship.

#### General Classes

Relationship

#### Attributes

`conjugatedType` : Type {redefines source}

The Type that is the result of applying Conjugation to the `originalType`.

`originalType` : Type {redefines target}

The Type to be conjugated.

`/owningType` : Type [0..1] {subsets conjugatedType, owningRelatedElement}

The `conjugatedType` of this Type that is also its `owningRelatedElement`.

#### Operations

No operations.

#### Constraints

None.

### 8.3.3.1.3 Disjoining

#### Description

A Disjoining is a Relationship between Types asserted to have interpretations that are not shared (disjoint) between them, identified as `typeDisjoined` and `disjoiningType`. For example, a Classifier for mammals is disjoint from a Classifier for minerals, and a Feature for people's parents is disjoint from a Feature for their children.

#### General Classes

Relationship

#### Attributes

`disjoiningType` : Type {redefines target}

Type asserted to be disjoint with the `typeDisjoined`.

`/owningType` : Type [0..1] {subsets `typeDisjoined`, `owningRelatedElement`}

A `typeDisjoined` that is also an `owningRelatedElement`.

`typeDisjoined` : Type {redefines source}

Type asserted to be disjoint with the `disjoiningType`.

#### Operations

No operations.

#### Constraints

None.

### 8.3.3.1.4 FeatureDirectionKind

#### Description

`FeatureDirectionKind` enumerates the possible kinds of `direction` that a Feature may be given as a member of a Type.

#### General Classes

None.

#### Literal Values

in

Values of the Feature on each instance of its domain are determined externally to that instance and used internally.

inout

Values of the Feature on each instance are determined either as *in* or *out* directions, or both.

out

Values of the Feature on each instance of its domain are determined internally to that instance and used externally.

### 8.3.3.1.5 FeatureMembership

#### Description

FeatureMembership is an OwningMembership for a Feature in a Type that is also a TypeFeaturing Relationship between the Feature and the Type.

#### General Classes

TypeFeaturing  
OwningMembership

#### Attributes

ownedMemberFeature : Feature [0..1] {subsets featureOfType, redefines ownedMemberElement}

The Feature that this FeatureMembership relates to its `owningType`, making it an `ownedFeature` of the `owningType`.

`/owningType` : Type {subsets featuringType, type, redefines membershipOwningNamespace}

The Type that owns this FeatureMembership.

#### Operations

No operations.

#### Constraints

None.

### 8.3.3.1.6 Specialization

#### Description

Specialization is a Relationship between two Types that requires all instances of the `specific` type to also be instances of the `general` Type (i.e., the set of instances of the `specific` Type is a *subset* of those of the `general` Type, which might be the same set).

#### General Classes

Relationship

#### Attributes

`general : Type {redefines target}`

A Type with a superset of all instances of the `specific` Type, which might be the same set.

`/owningType : Type [0..1] {subsets specific, owningRelatedElement}`

The Type that is the `specific` Type of this Specialization and owns it as its `owningRelatedElement`.

`specific : Type {redefines source}`

A Type with a subset of all instances of the `general` Type, which might be the same set.

### Operations

No operations.

### Constraints

`generalizationSpecificNotConjugated`

The `specific` Type of a Generalization cannot be a conjugated Type.

`not specific.isConjugated`

#### 8.3.3.1.7 Multiplicity

##### Description

A Multiplicity is a Feature whose co-domain is a set of natural numbers that includes the number of sequences determined below, based on the kind of `typeWithMultiplicity`:

- Classifiers: minimal sequences (the single length sequences of the Classifier).
- Features: sequences with the same feature-pair head. In the case of Features with Classifiers as domain and co-domain, these sequences are pairs, with the first element in a single-length sequence of the domain Classifier (head of the pair), and the number of pairs with the same first element being among the Multiplicity co-domain numbers.

Multiplicity co-domains (in models) can be specified by Expression that might vary in their results. If the `typeWithMultiplicity` is a Classifier, the domain of the Multiplicity shall be *Anything*. If the `typeWithMultiplicity` is a Feature, the Multiplicity shall have the same domain as the `typeWithMultiplicity`.

##### General Classes

Feature

##### Attributes

None.

##### Operations

No operations.

##### Constraints

None.

### 8.3.3.1.8 Type

#### Description

A Type is a Namespace that is the most general kind of Element supporting the semantics of classification. A Type may be a Classifier or a Feature, defining conditions on what is classified by the Type (see also the description of `isSufficient`).

#### General Classes

Namespace

#### Attributes

`/directedFeature : Feature [0..*] {subsets feature, ordered}`

The `features` of this Type that have a non-null `direction`.

`/endFeature : Feature [0..*] {subsets feature, ordered}`

All `features` related to this Type by `EndFeatureMemberships`.

`/feature : Feature [0..*] {subsets member, ordered}`

The `ownedMemberFeatures` of the `featureMemberships` of this Type.

`/featureMembership : FeatureMembership [0..*] {ordered}`

The `FeatureMemberships` for `features` of this Type, which include all `ownedFeatureMemberships` and those `inheritedMemberships` that are `FeatureMemberships` (but does *not* include any `importedMemberships`).

`/inheritedFeature : Feature [0..*] {subsets feature, ordered}`

All the `memberFeatures` of the `inheritedMemberships` of this Type.

`/inheritedMembership : Membership [0..*] {subsets membership, ordered}`

All `Memberships` inherited by this Type via Generalization or Conjugation. These are included in the derived union for the `memberships` of the Type.

`/input : Feature [0..*] {subsets directedFeature, ordered}`

All `features` related to this Type by `FeatureMemberships` that have `direction in` or `inout`.

`isAbstract : Boolean`

Indicates whether instances of this Type must also be instances of at least one of its specialized Types.

`/isConjugated : Boolean`

Indicates whether this Type has an `ownedConjugator`. (See Conjugation.)

`isSufficient : Boolean`



Whether all things that meet the classification conditions of this Type must be classified by the Type.

(A Type gives conditions that must be met by whatever it classifies, but when `isSufficient` is false, things may meet those conditions but still not be classified by the Type. For example, a Type `Car` that is not sufficient could require everything it classifies to have four wheels, but not all four wheeled things would need to be cars. However, if the type `Car` were sufficient, it would classify all four-wheeled things.)

`/multiplicity : Multiplicity [0..1] {subsets member}`

The one `member` (at most) of this Type that is a Multiplicity, which constrains the cardinality of the Type. A `multiplicity` can be owned or inherited. If it is owned, the `multiplicity` must redefine the `multiplicity` (if it has one) of any general Type of a Generalization of this Type.

`/output : Feature [0..*] {subsets directedFeature, ordered}`

All `features` related to this Type by `FeatureMemberships` that have `direction out` or `inout`.

`/ownedConjugator : Conjugation [0..1] {subsets ownedRelationship, conjugator}`

A `Conjugation` owned by this Type for which the Type is the `originalType`.

`/ownedDisjoining : Disjoining [0..*] {subsets ownedRelationship, disjoiningTypeDisjoining}`

The `ownedRelationships` of this Type that are `Disjoinings`, for which the Type is the `typeDisjoined` Type.

`/ownedEndFeature : Feature [0..*] {subsets endFeature, ownedFeature, ordered}`

All `endFeatures` of this Type that are `ownedFeatures`.

`/ownedFeature : Feature [0..*] {subsets ownedMember, ordered}`

The `ownedMemberFeatures` of the `ownedFeatureMemberships` of this Type.

`/ownedFeatureMembership : FeatureMembership [0..*] {subsets ownedMembership, featureMembership, typedFeaturing, ordered}`

The `ownedMemberships` of this Type that are `FeatureMemberships`, for which the Type is the `owningType`. Each such `FeatureMembership` identifies an `ownedFeature` of the Type.

`/ownedSpecialization : Specialization [0..*] {subsets specialization, ownedRelationship, ordered}`

The `ownedRelationships` of this Type that are `Specializations`, for which the Type is the `specific` Type.

## Operations

`allSupertypes() : Type [1..*]`

Return all Types related to this Type as supertypes directly or transitively by Generalization Relationships.

```
body: ownedGeneralization->
  closure(general.ownedGeneralization).general->
  including(self)
```

```
post: result = let g : Bag = generalization.general in
  g->union(g->collect(allSupertypes()))->flatten()->asSet()->including(self)
```

directionOf(feature : Feature) : FeatureDirectionKind [0..1]

If the given feature is a feature of this type, then return its direction relative to this type, taking conjugation into account.

```
body: if input->includes(feature) and output->includes(feature) then
    FeatureDirectionKind::inout
else if input->includes(feature) then
    FeatureDirectionKind::_in'
else if output->includes(feature) then
    FeatureDirectionKind::out
else
    null
endif endif endif
```

inheritedMemberships(excluded : Type [0..\*]) : Membership [0..\*]

Return the inherited Memberships of this Type, excluding those supertypes in the `excluded` set.

visibleMemberships(excluded : Namespace [0..\*],isRecursive : Boolean,includeAll : Boolean) : Membership [0..\*]

The visible Memberships of a Type include `inheritedMemberships`.

```
body: let visibleInheritedMemberships : Sequence(Membership) =
    inheritedMemberships(excluded)->
        select(includeAll or visibility = VisibilityKind::public) in
self.oclAsType(Namespace).visibleMemberships(excluded, isRecursive, includeAll)->
    union(visibleInheritedMemberships)
```

## Constraints

typeOwnedConjugator

[no documentation]

```
let ownedConjugators: Sequence(Conjugator) =
    ownedRelationship->selectByKind(Conjugation) in
    ownedConjugators->size() = 1 and
    ownedConjugator = ownedConjugators->at(1)
```

typeSpecializesAnything

[no documentation]

```
allSupertypes()->includes(Kernel Library::Anything)
```

typeOwnedGeneralizations

[no documentation]

```
ownedGeneralization = ownedRelationship->selectByKind(Generalization)->
    select(g | g.special = self)
```

typeOutput

If this Type is conjugated, then its outputs are the inputs of the originalType. Otherwise, its outputs are all features with FeatureMembership direction of out or inout.

```
output =
  if isConjugated then
    conjugator.originalType.input
  else
    feature->select(direction = out or direction = inout)
  endif
```

#### typeMultiplicity

The multiplicity of this Type is all its features that are Multiplicities. (There must be at most one.)

```
multiplicity = feature->select(oclIsKindOf(Multiplicity))
```

#### typeInheritedMembership

[no documentation]

```
inheritedMembership = inheritedMemberships(Set{})
```

#### typeDirectedFeature

[no documentation]

```
directedFeature = feature->select(direction <> null)
```

#### typeFeatureMembership

The featureMemberships of a Type is the union of the ownedFeatureMemberships and those inheritedMemberships that are FeatureMemberships.

```
featureMembership = ownedMembership->union(
  inheritedMembership->selectByKind(FeatureMembership))
```

#### typeFeature

The features of a Type are the ownedMemberFeatures of its featureMemberships.

```
feature = featureMembership.ownedMemberFeature
```

#### typeOwnedFeature

The ownedFeatures of a Type are the ownedMemberFeatures of its ownedFeatureMemberships.

```
ownedFeature = ownedFeatureMembership.ownedMemberFeature
```

#### typeDisjointType

[no documentation]

```
disjointType = disjoiningTypeDisjoining.disjoiningType
```

#### typeOwnedFeatureMembership

The ownedFeatureMemberships of a Type are its ownedMemberships that are FeatureMemberships.

```
ownedFeatureMembership = ownedRelationship->selectByKind(FeatureMembership)
```

typeInput

If this Type is conjugated, then its inputs are the outputs of the originalType. Otherwise, its inputs are all features with FeatureMembership direction of in or inout.

```
input =
  if isConjugated then
    conjugator.originalType.output
  else
    feature->select(direction = _('in' or direction = inout)
  endif
```

### 8.3.3.2 Classifiers Abstract Syntax

#### 8.3.3.2.1 Overview

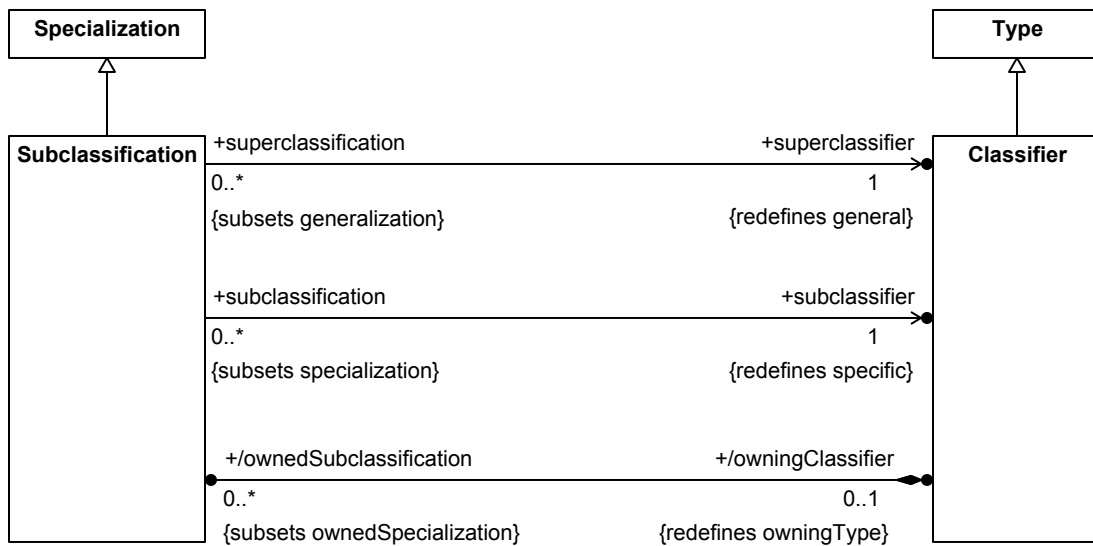


Figure 11. Classifiers

#### 8.3.3.2.2 Classifier

##### Description

A Classifier is a Type for model elements that classify:

- Things (in the universe) regardless of how Features relate them. These are sequences of exactly one thing (sequence of length 1).
- How the above things are related by Features. These are sequences of multiple things (length > 1).

Classifiers that classify relationships (sequence length > 1) must also classify the things at the end of those sequences (sequence length =1). Because of this, Classifiers specializing Features cannot classify anything (any sequences).

##### General Classes

Type

### Attributes

`/ownedSubclassification : Subclassification [0..*] {subsets ownedSpecialization}`

The `ownedSpecializations` of this Classifier that are Subclassifications, for which this Classifier is the subclassifier.

### Operations

No operations.

### Constraints

`classifierOwnedSuperclassings`

[no documentation]

`ownedSuperclassing = ownedGeneralization->intersection(superclassing)`

`classifierMultiplicityDomain`

If a Classifier has a `multiplicity`, then the `multiplicity` shall have no `featuringTypes` (meaning that its domain is implicitly *Base::Anything*).

`multiplicity <> null implies multiplicity.featuringType->isEmpty()`

## 8.3.3.2.3 Subclassification

### Description

Subclassification is Specialization in which both the `specific` and `general` Types are Classifiers. This means all instances of the specific Classifier are also instances of the general Classifier.

### General Classes

Specialization

### Attributes

`/owningClassifier : Classifier [0..1] {redefines owningType}`

The Classifier that owns this Subclassification relationship, which must also be its `subclassifier`.

`subclassifier : Classifier {redefines specific}`

The more specific Classifier in this Subclassification.

`superclassifier : Classifier {redefines general}`

The more general Classifier in this Subclassification.

### Operations

No operations.

## Constraints

None.

### 8.3.3.3 Features Abstract Syntax

#### 8.3.3.3.1 Overview

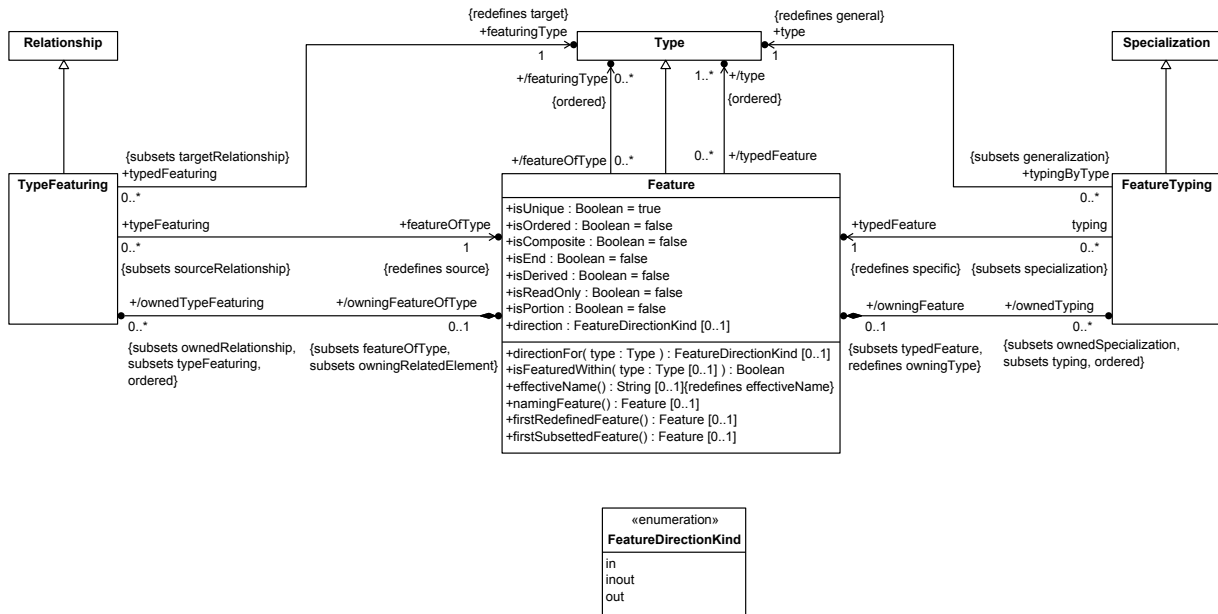


Figure 12. Features

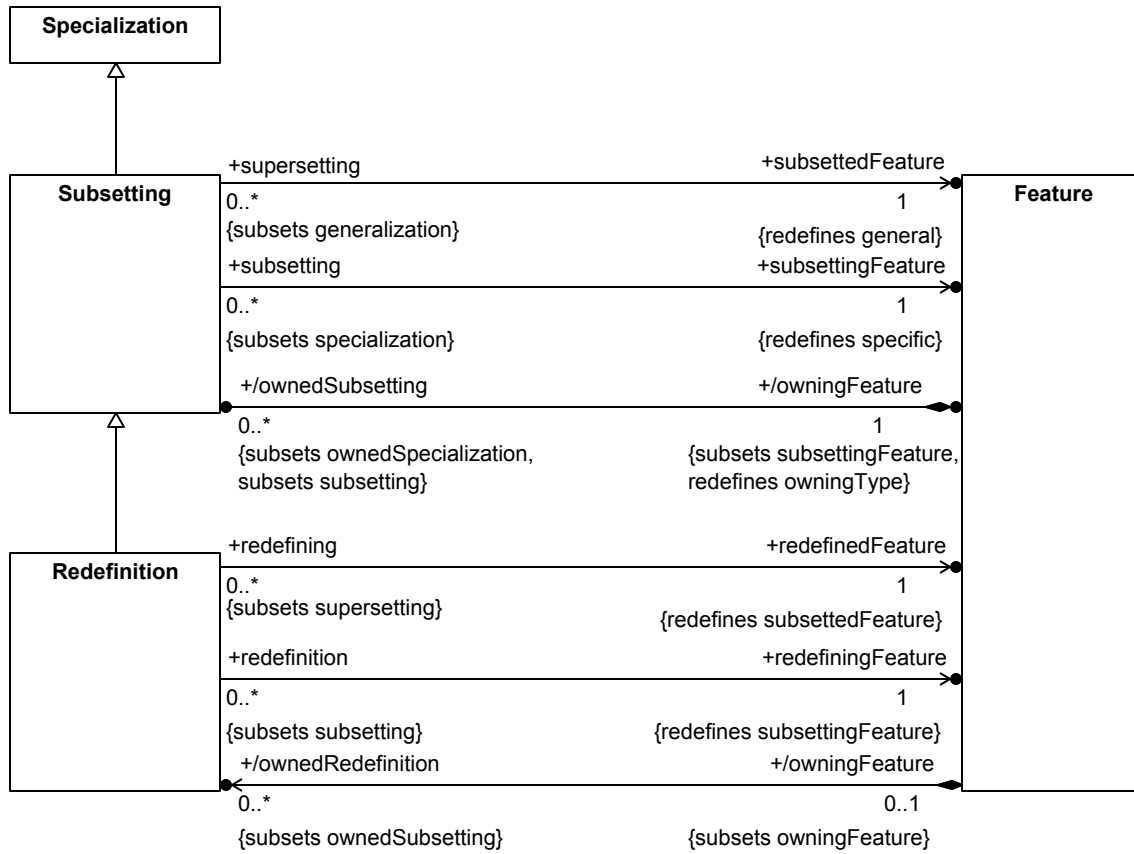


Figure 13. Subsetting

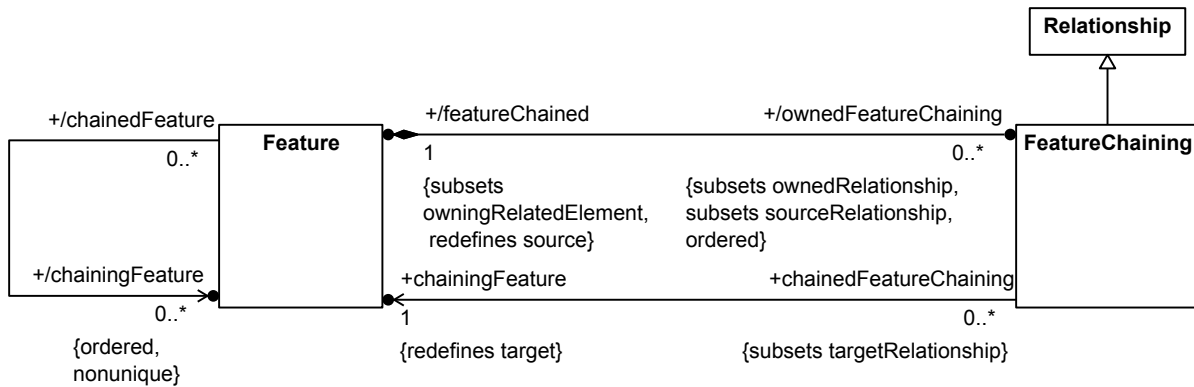
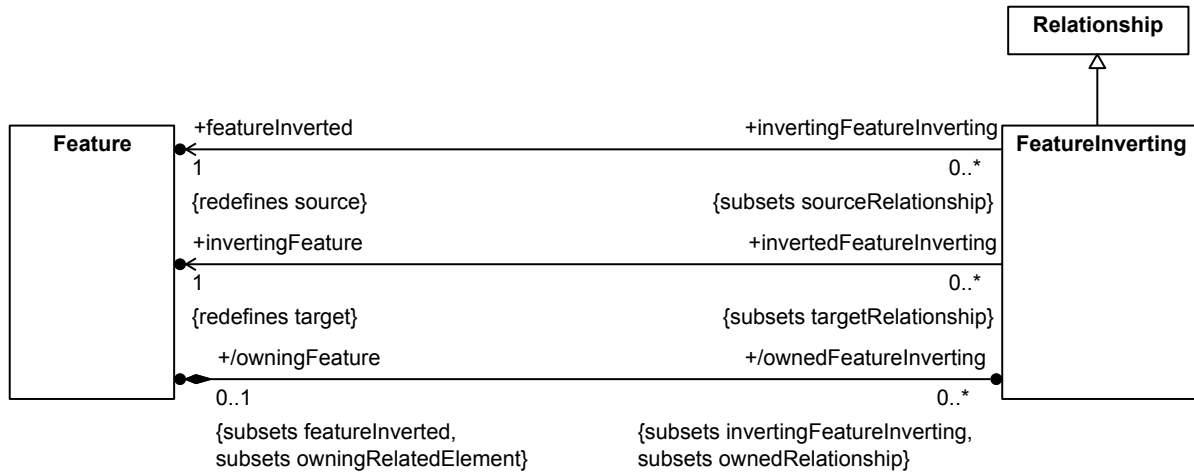
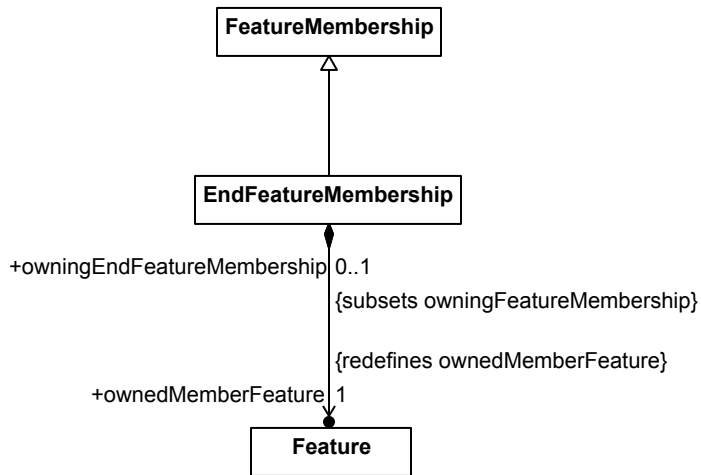


Figure 14. Feature Chaining



**Figure 15. Feature Inverting**



**Figure 16. End Feature Membership**

### 8.3.3.3.2 EndFeatureMembership

#### Description

EndFeatureMembership is a FeatureMembership that requires its `memberFeature` be owned and have `isEnd = true`.

#### General Classes

FeatureMembership

#### Attributes

ownedMemberFeature : Feature {redefines ownedMemberFeature}

#### Operations

No operations.



## Constraints

`endFeatureMembershipIsEnd`

The `ownedMemberFeature` of an `EndFeatureMembership` must be an end Feature.

`ownedMemberFeature.isEnd`

### 8.3.3.3.3 Feature

#### Description

A Feature is a Type that classifies sequences of multiple things (in the universe). These must concatenate a sequence drawn from the intersection of the Feature's `featuringTypes` (*domain*) with a sequence drawn from the intersection of its `types` (*co-domain*), treating (co)domains as sets of sequences. The domain of Features that do not have any `featuringTypes` is the same as if it were the library Type `Anything`. A Feature's `types` include at least `Anything`, which can be narrowed to other Classifiers by Redefinition.

In the simplest cases, a Feature's `featuringTypes` and `types` are Classifiers, its sequences being pairs (length = 2), with the first element drawn from the Feature's domain and the second element from its co-domain (the Feature "value"). Examples include cars paired with wheels, people paired with other people, and cars paired with numbers representing the car length.

Since Features are Types, their `featuringTypes` and `types` can be Features. When both are, Features classify sequences of at least four elements (length > 3), otherwise at least three (length > 2). The `featuringTypes` of *nested* Features are Features.

The values of a Feature with `chainingFeatures` are the same as values of the last Feature in the chain, which can be found by starting with values of the first Feature, then from those values to values of the second feature, and so on, to values of the last feature.

#### General Classes

Type

#### Attributes

`/chainingFeature : Feature [0..*] {ordered, nonunique}`

The Features that are chained together to determine the values of this Feature, derived from the `chainingFeatures` of the `ownedFeatureChainings` of this Feature, in the same order. The values of a Feature with `chainingFeatures` are the same as values of the last Feature in the chain, which can be found by starting with the values of the first Feature (for each instance of the original Feature's domain), then on each of those to the values of the second Feature in `chainingFeatures`, and so on, to values of the last Feature. The Features related to a Feature by a `FeatureChaining` are identified as its `chainingFeatures`.

`direction : FeatureDirectionKind [0..1]`

Determines how values of this Feature are determined or used (see `FeatureDirectionKind`).

`/endOwningType : Type [0..1] {subsets typeWithEndFeature, owningType}`

The Type that is related to this Feature by an `EndFeatureMembership` in which the Feature is an `ownedMemberFeature`.

/featuringType : Type [0..\*] {ordered}

Types that feature this Feature, such that any instance in the domain of the Feature must be classified by all of these Types, including at least all the `featuringTypes` of its `ownedTypeFeaturings`.

isComposite : Boolean

Whether the Feature is a composite feature of its `featuringType`. If so, the values of the Feature cannot exist after the instance of the `featuringType` no longer does.

.

isDerived : Boolean

Whether the values of this Feature can always be computed from the values of other Features.

isEnd : Boolean

Whether or not the this Feature is an end Feature, requiring a different interpretation of the `multiplicity` of the Feature.

An end Feature is always considered to map each domain entity to a single co-domain entity, whether or not a Multiplicity is given for it. If a Multiplicity is given for an end Feature, rather than giving the co-domain cardinality for the Feature as usual, it specifies a cardinality constraint for *navigating* across the `endFeatures` of the `featuringType` of the end Feature. That is, if a Type has  $n$  `endFeatures`, then the Multiplicity of any one of those end Features constrains the cardinality of the set of values of that Feature when the values of the other  $n-1$  end Features are held fixed.

isOrdered : Boolean

Whether an order exists for the values of this Feature or not.

isPortion : Boolean

Whether the values of this Feature are contained in the space and time of instances of the Feature's domain.

isReadOnly : Boolean

Whether the values of this Feature can change over the lifetime of an instance of the domain.

isUnique : Boolean

Whether or not values for this Feature must have no duplicates or not.

/ownedFeatureChaining : FeatureChaining [0..\*] {subsets sourceRelationship, ownedRelationship, ordered}

The `FeatureChainings` that are among the `ownedRelationships` of this Feature (identify their `featureChained` also as an `owningRelatedElement`).

/ownedFeatureInverting : FeatureInverting [0..\*] {subsets ownedRelationship, invertingFeatureInverting}

The `ownedRelationships` of this Feature that are `FeatureInvertings`, for which the Feature is the `featureInverted`.

/ownedRedefinition : Redefinition [0..\*] {subsets ownedSubsetting}

The `ownedSubsettings` of this Feature that are Redefinitions, for which the Feature is the `redefiningFeature`.

`/ownedSubsetting : Subsetting [0..*] {subsets ownedSpecialization, subsetting}`

The `ownedGeneralizations` of this Feature that are Subsettings, for which the Feature is the `subsettingFeature`.

`/ownedTypeFeaturing : TypeFeaturing [0..*] {subsets ownedRelationship, typeFeaturing, ordered}`

The `ownedRelationships` of this Feature that are TypeFeaturings, for which the Feature is the `featureOfType`.

`/ownedTyping : FeatureTyping [0..*] {subsets ownedSpecialization, typing, ordered}`

The `ownedGeneralizations` of this Feature that are FeatureTypings, for which the Feature is the `typedFeature`.

`owningFeatureMembership : FeatureMembership [0..1] {subsets owningMembership, typeFeaturing}`

The `FeatureMembership` that owns this Feature as an `ownedMemberFeature`, determining its `owningType`.

`/owningType : Type [0..1] {subsets typeWithFeature, owningNamespace, featuringType}`

The `Type` that is the `owningType` of the `owningFeatureMembership` of this `Type`.

`/type : Type [1..*] {ordered}`

Types that restrict the values of this Feature, such that the values must be instances of all the types. The types of a Feature are derived from its `ownedFeatureTypings` and the types of its `ownedSubsettings`.

## Operations

`directionFor(type : Type) : FeatureDirectionKind [0..1]`

Return the `directionOf` this Feature relative to the given `type`.

**body:** `type.directionOf(self)`

`effectiveName() : String [0..1]`

If a Feature has no `name`, then its effective name is given by the effective name of the Feature returned by `namingFeature`, if any.

**body:** `if name <> null then  
 name  
else  
 let namingFeature : Feature = namingFeature() in  
 if namingFeature = null then  
 null  
 else  
 namingFeature.effectiveName()  
 endif  
endif`

`firstRedefinedFeature() : Feature [0..1]`

Return the first Feature that is redefined by this Feature, if any.

```

body: let redefinitions : Sequence(Redefinition) = ownedRedefinition in
if redefinitions->isEmpty() then
    null
else
    redefinitions->at(1).redefinedFeature
endif

```

**firstSubsettedFeature() : Feature [0..1]**

Get the first Feature that is subsetted by this Feature but not redefined, if any.

```

body: let subsettings : Sequence(Subsetting) =
    ownedSubsetting->reject(oclIsKindOf(Redefinition)) in
if subsettings->isEmpty() then
    null
else
    subsettings->at(1).subsettedFeature
endif

```

**isFeaturedWithin(type : Type [0..1]) : Boolean**

Return whether this Feature has the given type as a direct or indirect featuringType. If type is null, then check if this Feature is implicitly directly or indirectly featured in *Base::Anything*.

```

body: type = null and feature.featuringType->isEmpty() or
    type <> null and feature.featuringType->includes(type) or
    feature.featuringType->exists(t |
        t.oclIsKindOf(Feature) and
        t.oclAsType(Feature).isFeaturedWithin(type))

```

**namingFeature() : Feature [0..1]**

By default, the naming feature of a Feature is given by its first redefinedFeature, if any.

```

body: firstRedefinedFeature()

```

## Constraints

**featureChainingFeatureNot1**

[no documentation]

```

chainingFeatures->size() <> 1

```

**featureType**

If a Feature has chainingFeatures, then its types are the same as the last chainingFeature. Otherwise its types are the union of the types of its ownedTypings and the types of the subsettedFeatures of its ownedSubsettings, with all redundant supertypes removed.

.

**featureOwnedSubsettings**

[no documentation]

```

ownedSubsetting = ownedGeneralization->selectByKind(Subsetting)

```

## featureOwnedRedefinitions

[no documentation]

```
ownedRedefinition = ownedSubsetting->selectByKind(Redefinition)
```

## featureMultiplicityDomain

If a Feature has a multiplicity, then the featuringTypes of the multiplicity must be the same as those of the Feature itself.

```
multiplicity <> null implies multiplicity.featureingType = featuringType
```

## featureInvertedFeature

[no documentation]

```
invertedFeature = invertedFeatureInverting.featureInverted
```

## featureRequiredSpecialization

A Feature must directly or indirectly specialize Base::things from the Kernel Library.

```
allSupertypes()->includes(KernelLibrary::things)
```

## featureOwnedTypeFeaturing

[no documentation]

```
ownedTypeFeaturing = ownedRelationship->selectByKind(TypeFeaturing)->  
  select(tf | tf.featureOfType = self)
```

## featureOwnedTyping

[no documentation]

```
ownedTyping = ownedGeneralization->selectByKind(FeatureTyping)
```

## featureInverseFeatures

[no documentation]

```
inverseFeature = invertingFeatureInverting.featureInverse
```

## featureChainingFeaturesNotSelf

A Feature cannot be one of its own chainingFeatures.

```
chainingFeatures->excludes(self)
```

## featureIsEnd

[no documentation]

```
isEnd = owningFeatureMembership <> null and owningFeatureMembership.ocIsKindOf(EndFeatureMembership)
```

featureIsDerived

[no documentation]

chainingFeatureChainings->notEmpty() implies (owningFeatureMembership <> null implies owningFeatureM

featureIsComposite

[no documentation]

isComposite = owningFeatureMembership <> null and owningFeatureMembership.isComposite

featureOwnedFeatureChaining

The ownedFeatureChainings of this Feature are the ownedRelationships that are FeatureChainings.

ownedFeatureChaining = ownedRelationship->selectByKind(FeatureChaining)

featureChainingFeature

The chainingFeatures of a Feature are the chainingFeatures of its ownedFeatureChainings.

chainingFeature = ownedFeatureChaining.chainingFeature

#### 8.3.3.3.4 FeatureChaining

##### Description

FeatureChaining is a Relationship that makes its target Feature one of the chainingFeatures of its owning Feature.

##### General Classes

Relationship

##### Attributes

chainingFeature : Feature {redefines target}

The Feature whose values partly determine values of featureChained, as described in  
Feature::chainingFeature.

/featureChained : Feature {subsets owningRelatedElement, redefines source}

The Feature whose values are partly determined by values of the chainingFeature, as described in  
Feature::chainingFeature.

##### Operations

No operations.

##### Constraints

None.

### 8.3.3.3.5 FeatureInverting

#### Description

A FeatureInverting is a Relationship between Features asserting that their interpretations (sequences) are the reverse of each other, identified as `featureInverted` and `invertingFeature`. For example, a Feature identifying each person's parents is the inverse of a Feature identifying each person's children. A person identified as a parent of another will identify that other as one of their children.

#### General Classes

Relationship

#### Attributes

`featureInverted` : Feature {redefines source}

Feature that is an the inverse of `invertingFeature`.

`invertingFeature` : Feature {redefines target}

Feature that is an inverse of `invertedFeature`.

`/owningFeature` : Feature [0..1] {subsets `owningRelatedElement`, `featureInverted`}

A `featureInverted` that is also an `owningRelatedElement`.

#### Operations

No operations.

#### Constraints

None.

### 8.3.3.3.6 FeatureTyping

#### Description

FeatureTyping is Specialization in which the `specific` Type is a Feature. This means the set of instances of the (specific) `typedFeature` is a subset of the set of instances of the (general) `type`. In the simplest case, the `type` is a Classifier, whereupon the `typedFeature` subset has instances interpreted as sequences ending in things (in the modeled universe) that are instances of the Classifier.

#### General Classes

Specialization

#### Attributes

`/owningFeature` : Feature [0..1] {subsets `typedFeature`, redefines `owningType`}

The Feature that owns this FeatureTyping (which must also be the `typedFeature`).

`type` : Type {redefines general}

The Type that is being applied by this FeatureTyping.

typedFeature : Feature {redefines specific}

The Feature that has its Type determined by this FeatureTyping.

### Operations

No operations.

### Constraints

None.

## 8.3.3.3.7 Redefinition

### Description

Redefinition specializes Subsetting to require the `redefinedFeature` and the `redefiningFeature` to have the same values (on each instance of the domain of the `redefiningFeature`). This means any restrictions on the `redefiningFeature`, such as type or multiplicity, also apply to the `redefinedFeature` (on each instance of the `owningType` of the `redefiningFeature`), and vice versa. The `redefinedFeature` might have values for instances of the `owningType` of the `redefiningFeature`, but only as instances of the `owningType` of the `redefinedFeature` that happen to also be instances of the `owningType` of the `redefiningFeature`. This is supported by the constraints inherited from Subsetting on the domains of the `redefiningFeature` and `redefinedFeature`. However, these constraints are narrowed for Redefinition to require the `owningTypes` of the `redefiningFeature` and `redefinedFeature` to be different and the `redefinedFeature` to not be imported into the `owningNamespace` of the `redefiningFeature`. This enables the `redefiningFeature` to have the same name as the `redefinedFeature` if desired.

### General Classes

Subsetting

### Attributes

redefinedFeature : Feature {redefines subsettingFeature}

The Feature that is redefined by the `redefiningFeature` of this Redefinition.

redefiningFeature : Feature {redefines subsettingFeature}

The Feature that is redefining the `redefinedFeature` of this Redefinition.

### Operations

No operations.

### Constraints

None.

## 8.3.3.3.8 Subsetting

### Description



Subsetting is Generalization in which the `specific` and `general` Types that are Features. This means all values of the `subsettingFeature` (on instances of its domain, i.e., the intersection of its `featuringTypes`) are values of the `subsettingFeature` on instances of its domain. To support this, the domain of the `subsettingFeature` must be the same or specialize (at least indirectly) the domain of the `subsettingFeature` (via Generalization), and the range (intersection of a Feature's types) of the `subsettingFeature` must specialize the range of the `subsettingFeature`. The `subsettingFeature` is imported into the `owningNamespace` of the `subsettingFeature` (if it is not already in that namespace), requiring the names of the `subsettingFeature` and `subsettingFeature` to be different.

### General Classes

Specialization

### Attributes

`/owningFeature : Feature {subsets subsettingFeature, redefines owningType}`

The Feature that owns this Subsetting relationship, which must also be its `subsettingFeature`.

`subsettingFeature : Feature {redefines general}`

The Feature that is subsetting by the `subsettingFeature` of this Subsetting.

`subsettingFeature : Feature {redefines specific}`

The Feature that is a subset of the `subsettingFeature` of this Subsetting.

### Operations

No operations.

### Constraints

None.

## 8.3.3.3.9 TypeFeaturing

### Description

A TypeFeaturing is a Relationship between a Type and a Feature that is featured by that Type. Every instance in the domain of the `featureOfType` must be classified by the `featuringType`. This means that sequences that are classified by the `featureOfType` must have a prefix subsequence that is classified by the `featuringType`.

### General Classes

Relationship

### Attributes

`featureOfType : Feature {redefines source}`

The Feature that is featured by the `featuringType`.

`featuringType : Type {redefines target}`

The Type that features the `featureOfType`.

`/owningFeatureOfType : Feature [0..1] {subsets featureOfType, owningRelatedElement}`

The Feature that owns this TypeFeaturing and is also the `featureOfType`.

### Operations

No operations.

### Constraints

None.

## 8.3.4 Kernel Abstract Syntax

### 8.3.4.1 Data Types Abstract Syntax

#### 8.3.4.1.1 Overview

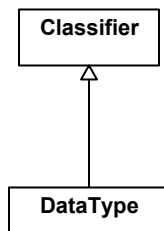


Figure 17. Data Types

#### 8.3.4.1.2 DataType

##### Description

A `DataType` is a `Classifier` of things (in the universe) that can only be distinguished by how they are related to other things (via `Features`). This means multiple things classified by the same `DataType`

- Cannot be distinguished when they are related to other things in exactly the same way, even when they are intended to be about different things.
- Can be distinguished when they are related to other things in different ways, even when they are intended to be about the same thing.

##### General Classes

`Classifier`

##### Attributes

None.

##### Operations

No operations.

##### Constraints

`datatypeClassifiesDataValue`

[no documentation]

```
allSupertypes() -> includes (Kernel Library::DataValue)
```

`datatypeValidSpecialization`

A `DataType` must not specialize a `Class` or an `Association`.

### 8.3.4.2 Classes Abstract Syntax

#### 8.3.4.2.1 Overview

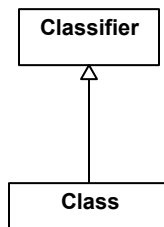


Figure 18. Classes

#### 8.3.4.2.2 Class

##### Description

A `Class` is a `Classifier` of things (in the universe) that can be distinguished without regard to how they are related to other things (via `Features`). This means multiple things classified by the same `Class` can be distinguished, even when they are related other things in exactly the same way.

##### General Classes

`Classifier`

##### Attributes

None.

##### Operations

No operations.

##### Constraints

`classValidSpecialization`

A `Class` must not specialize a `DataType`.

`classClassifiesOccurrence`

[no documentation]

```
allSupertypes() -> includes (Kernel Library::Occurrence)
```

### 8.3.4.3 Structures Abstract Syntax

#### 8.3.4.3.1 Overview

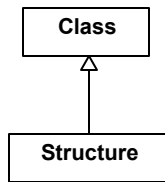


Figure 19. Structures

#### 8.3.4.3.2 Structure

##### Description

A Structure is a Class of objects in the modeled universe that are primarily structural in nature. While an Object is not itself behavioral, it may be involved in and acted on by Behaviors, and it may be the performer of some of them.

##### General Classes

Class

##### Attributes

None.

##### Operations

No operations.

##### Constraints

structureClassifiesObject

[no documentation]

```
allSupertypes() -> includes(Kernel Library::Object)
```

#### 8.3.4.4 Associations Abstract Syntax

### 8.3.4.4.1 Overview

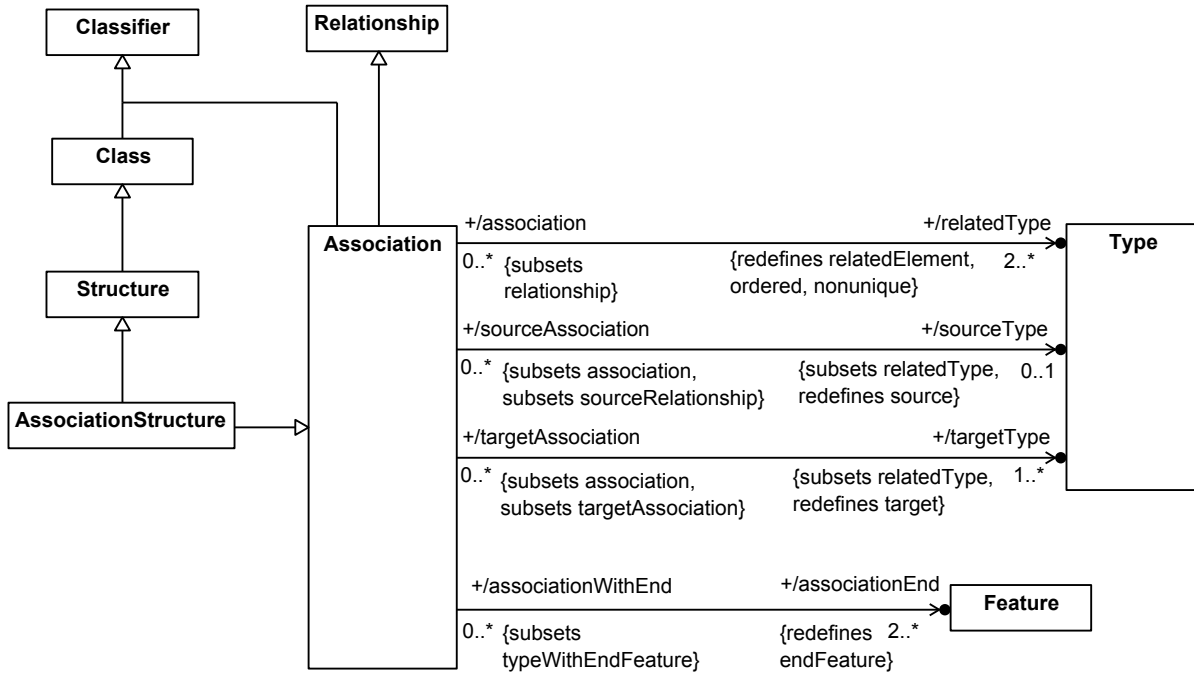


Figure 20. Associations

### 8.3.4.4.2 Association

#### Description

An Association is a Relationship and a Classifier to enable classification of links between things (in the universe). The co-domains (types) of the associationEnd Features are the relatedTypes, as co-domain and participants (linked things) of an Association identify each other.

#### General Classes

Relationship  
Classifier

#### Attributes

/associationEnd : Feature [2..\*] {redefines endFeature}

The features of the Association that identify the things that can be related by it. An Association must have at least two associationEnds. When it has exactly two, the Association is called a *binary* Association.

/relatedType : Type [2..\*] {redefines relatedElement, ordered, nonunique}

The types of the endFeatures of the Association, which are the relatedElements of the Association considered as a Relationship.

/sourceType : Type [0..1] {subsets relatedType, redefines source}

The source `relatedType` for this Association. If this is a binary Association, then the `sourceType` is the first `relatedType`, and the first `associationEnd` of the Association must redefine the `source` Feature of the Association *BinaryLink* from the Kernel Library. If this Association is not binary, then it has no `sourceType`.

`/targetType : Type [1..*] {subsets relatedType, redefines target}`

The target `relatedTypes` for this Association. This includes all the `relatedTypes` other than the `sourceType`. If this is a binary Association, then the `associationEnds` corresponding to the `relatedTypes` must all redefine the `target` Feature of the Association *BinaryLink* from the Kernel Library.

## Operations

No operations.

## Constraints

AssociationLink

[no documentation]

```
let numend : Natural = associationEnd->size() in
  allSupertypes()->includes(
    if numend = 2 then Kernel Library::BinaryLink
    else Kernel Library::Link)
```

associationClassifiesLink

[no documentation]

```
allSupertypes()->includes(Kernel Library::Link)
```

AssociationStructureIntersection

[no documentation]

```
oclIsKindOf(Structure) = oclIsKindOf(AssociationStructure)
```

associationRelatedTypes

[no documentation]

```
relatedTypes = associationEnd.type
```

### 8.3.4.4.3 AssociationStructure

#### Description

#### General Classes

Structure  
Association

#### Attributes

None.

## Operations

No operations.

## Constraints

associationStructureClassifiesLinkObject

[no documentation]

```
allSupertypes() -> includes (Kernel Library::LinkObject)
```

### 8.3.4.5 Connectors Abstract Syntax

#### 8.3.4.5.1 Overview

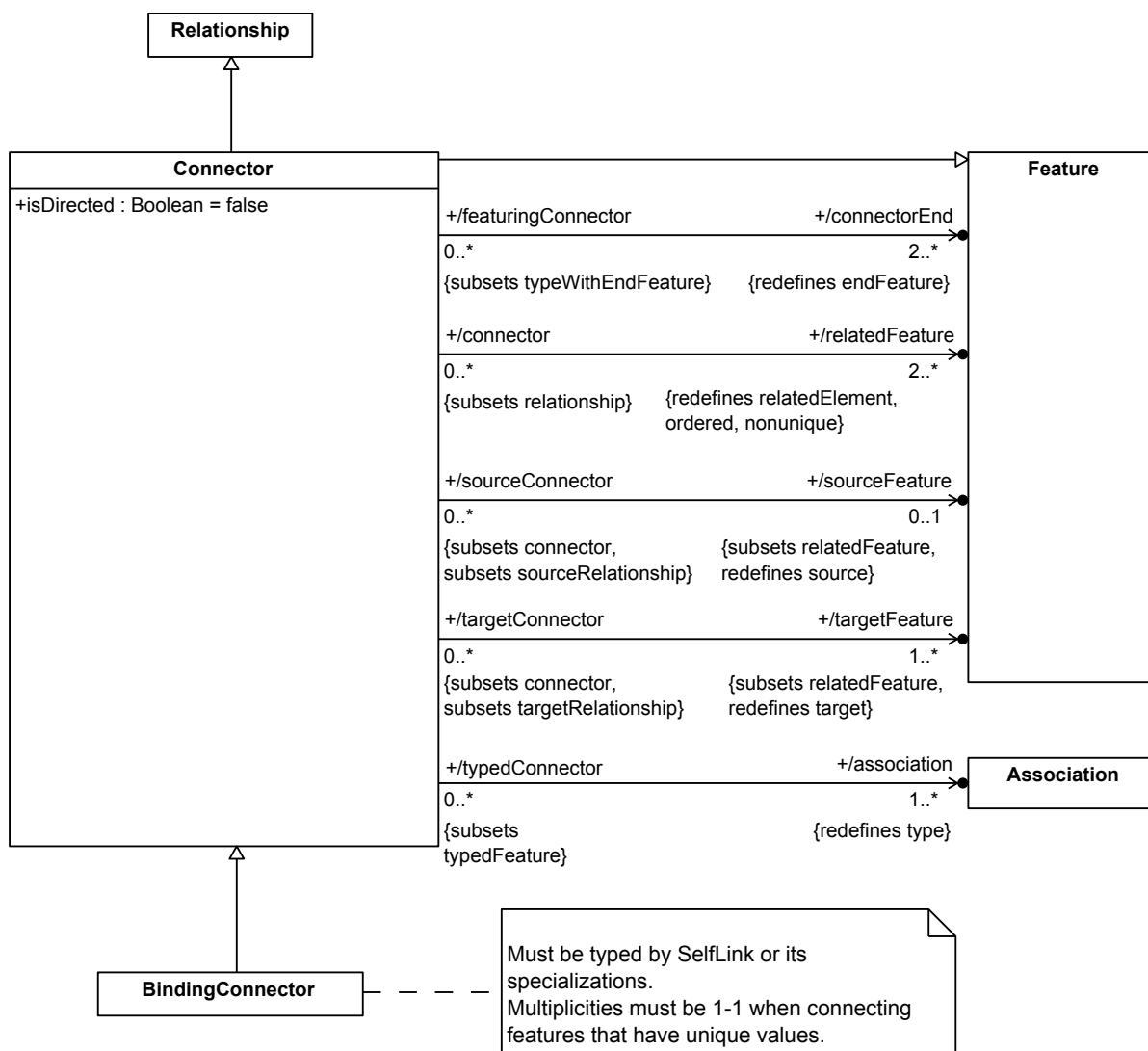
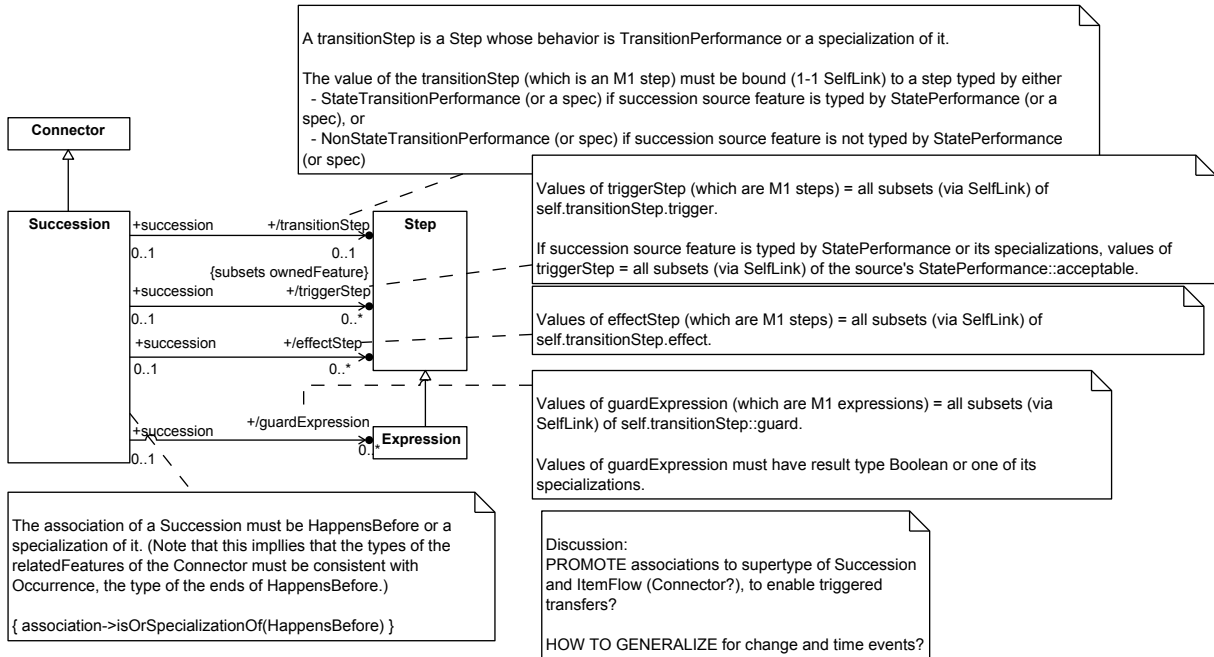


Figure 21. Connectors



**Figure 22. Successions**

### 8.3.4.5.2 Binding Connector

#### Description

A Binding Connector is a binary Connector that requires its `relatedFeatures` to identify the same things (have the same values).

#### General Classes

Connector

#### Attributes

None.

#### Operations

No operations.

#### Constraints

None.

### 8.3.4.5.3 Connector

#### Description

A Connector is a usage of Associations, with links restricted to instances of the Type in which it is used (domain of the Connector). Associations restrict what kinds of things might be linked. The Connector further restricts these links to between values of two Features on instances of its domain.



## General Classes

Relationship  
Feature

## Attributes

/association : Association [1..\*] {redefines type}

The Associations that type the Connector.

/connectorEnd : Feature [2..\*] {redefines endFeature}

These are the ends of the Connector, which show what Features it relates. The connectorEnds of a Connector are the features of the Connector that redefine the end Features of the Connector association.

isDirected : Boolean

Whether or not the Connector should be considered to have a direction from source to target.

/relatedFeature : Feature [2..\*] {redefines relatedElement, ordered, nonunique}

The Features that are related by this Connector considered as a Relationship, derived as the subsetting Features of the connectorEnds of the Connector.

/sourceFeature : Feature [0..1] {subsets relatedFeature, redefines source}

The source relatedFeature for this Connector. If this is a binary Connector, then the sourceFeature is the first relatedFeature, and the first end Feature of the Connector must redefine the source Feature of the Connector binaryLinks from the Kernel Library. If this Connector is not binary, then it has no sourceFeature.

/targetFeature : Feature [1..\*] {subsets relatedFeature, redefines target}

The target relatedFeatures for this Connector. This includes all the relatedFeatures other than the sourceFeature. If this is a binary Connector, then the end Feature corresponding to the targetFeature must redefine the target Feature of the Connector binaryLinks from the Kernel Library.

## Operations

No operations.

## Constraints

connectorEndRedefinition

For each association of a Connector, each associationEnd must be redefined by a different connectorEnd of the Connector.

```
association->forall(a |
  a.associationEnd->forall(ae |
    connectorEnd->one(ce |
      ce.ownedRedefinition.redefinedFeature->includes(ae))))
```

connectorTargetFeature

The `targetFeatures` of a `Connector` are the `relatedFeatures` other than the `sourceFeature`.

```
targetFeature =  
  if sourceFeature = null then relatedFeature  
  else relatedFeature->excluding(sourceFeature)  
endif
```

`connectorConnectorEnd`

The `connectorEnds` of a `Connector` are its `endFeatures`.

```
connectorEnd = feature->select(isEnd)
```

`connectorRelatedFeatures`

The `relatedFeatures` of a `Connector` are the subsetted `Features` of its `connectorEnds`.

```
relatedFeature = connectorEnd.ownedSubsetting.subsettedFeature
```

`connectorFeaturingType`

Each `relatedFeature` of a `Connector` must have some `featuringType` of the `Connector` as a direct or indirect `featuringType` (where a `Feature` with no `featuringType` is treated as if the Classifier *Base::Anything* was its `featuringType`).

```
relatedFeature->forall(f |  
  if featuringType->isEmpty() then f.isFeaturedWithin(null)  
  else featuringType->exists(t | f.isFeaturedWithin(t))  
endif)
```

`connectorSourceFeature`

If this is a binary `Connector`, then the `sourceFeature` is the first `relatedFeature`. If this `Connector` is not binary, then it has no `sourceFeature`.

```
sourceFeature =  
  if relatedFeature->size() = 2 then relatedFeature->at(1)  
  else null  
endif
```

#### 8.3.4.5.4 Succession

##### Description

A `Succession` is a binary `Connector` that requires its `relatedFeatures` to happen separately in time. A `Succession` must be typed by the Association *HappensBefore* from the Kernel Model Library (or a specialization of it).

##### General Classes

`Connector`

##### Attributes

`/effectStep : Step [0..*]`

Steps that represent occurrences that are side effects of the `transitionStep` occurring.

/guardExpression : Expression [0..\*]

Expressions that must evaluate to true before the transitionStep can occur.

/transitionStep : Step [0..1] {subsets ownedFeature}

A Step that is typed by the Behavior *TransitionPerformance* (from the Model Library) that has this Succession as its *transitionLink*.

/triggerStep : Step [0..\*]

Steps that map incoming events to the timing of occurrences of the transitionStep. The values of triggerStep subset the list of acceptable events to be received by a Behavior or the object that performs it.

## Operations

No operations.

## Constraints

None.

### 8.3.4.6 Behaviors Abstract Syntax

#### 8.3.4.6.1 Overview

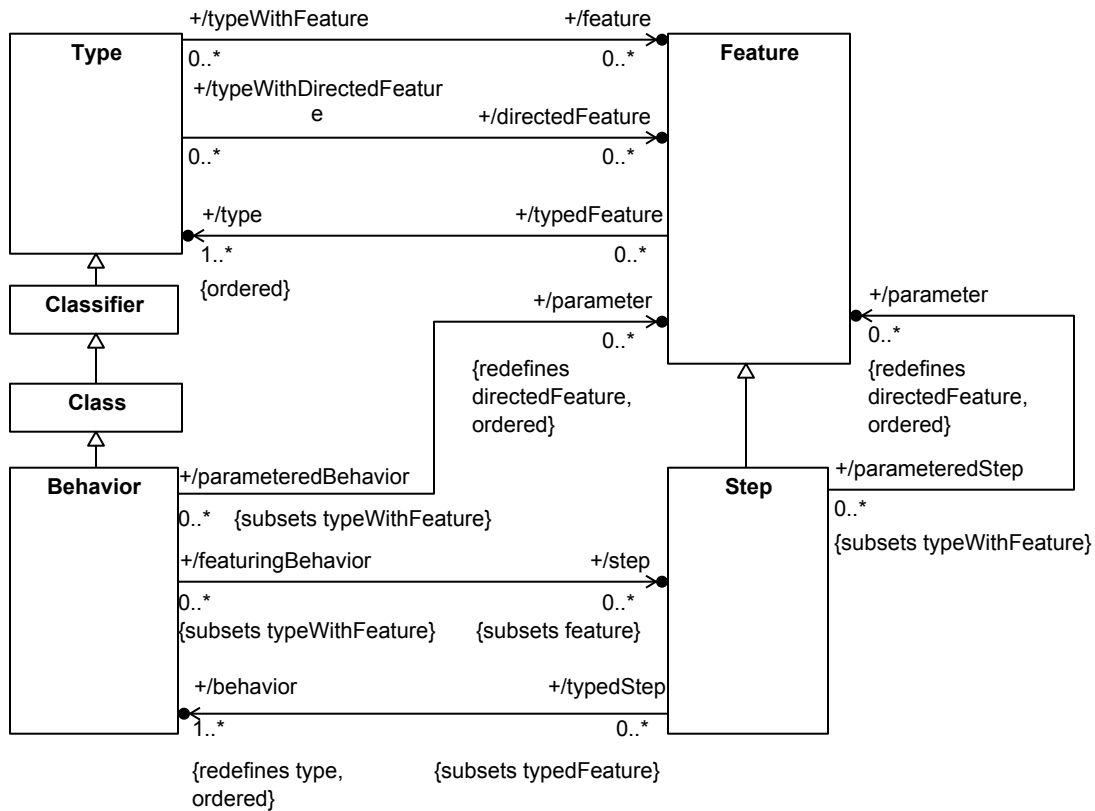
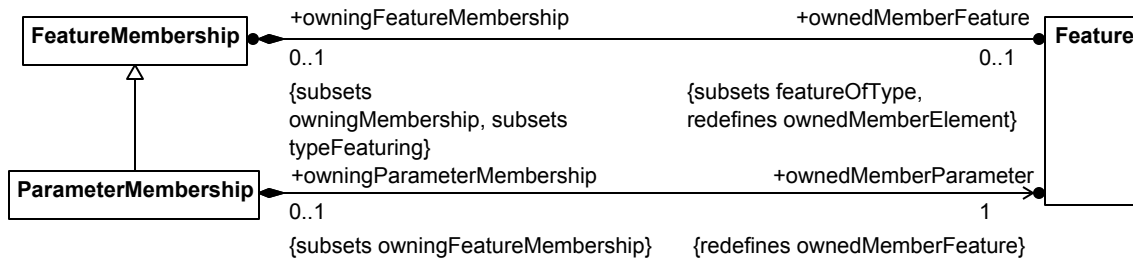


Figure 23. Behaviors



**Figure 24. Parameter Memberships**

#### 8.3.4.6.2 Behavior

##### Description

A Behavior coordinates occurrences of other Behaviors, as well as changes in objects. Behaviors can be decomposed into Steps and be characterized by parameters.

##### General Classes

Class

##### Attributes

/parameter : Feature [0..\*] {redefines directedFeature, ordered}

The parameters of this Behavior, which are all its directedFeatures, whose values are passed into and/or out of a performance of the Behavior.

/step : Step [0..\*] {subsets feature}

The Steps that make up this Behavior.

##### Operations

No operations.

##### Constraints

behaviorClassifiesPerformance

[no documentation]

allSupertypes() -> includes (Kernel Library::Performance)

#### 8.3.4.6.3 Step

##### Description

A Step is a Feature that is typed by one or more Behaviors. Steps may be used by one Behavior to coordinate the performance of other Behaviors, supporting the steady refinement of behavioral descriptions. Steps can be ordered in time and can be connected using ItemFlows to specify things flowing between their parameters.

##### General Classes

Feature

#### Attributes

/behavior : Behavior [1..\*] {redefines type, ordered}

The Behaviors that type this Step.

/parameter : Feature [0..\*] {redefines directedFeature, ordered}

The parameters of this Expression, which are all its `directedFeatures`, whose values are passed into and/or out of a performance of the Behavior.

#### Operations

No operations.

#### Constraints

None.

### 8.3.4.6.4 ParameterMembership

#### Description

A `ParameterMembership` is a `FeatureMembership` that identifies its `memberFeature` as a parameter, which is always owned, and must have a `direction`. A `ParameterMembership` must be owned by a `Behavior` or a `Step`.

#### General Classes

`FeatureMembership`

#### Attributes

ownedMemberParameter : Feature {redefines ownedMemberFeature}

The Feature that is identified as a parameter by this `ParameterMembership`, which is always owned by the `ParameterMembership`.

#### Operations

No operations.

#### Constraints

None.

### 8.3.4.7 Functions Abstract Syntax

### 8.3.4.7.1 Overview

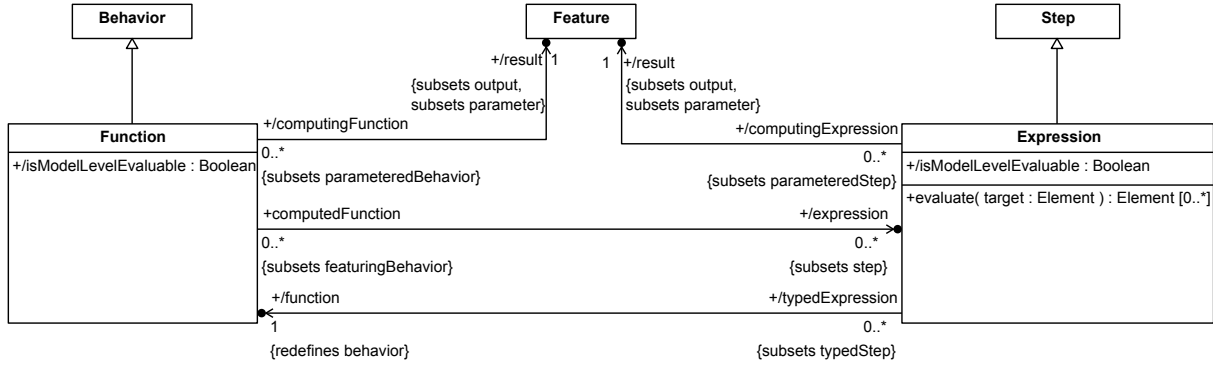


Figure 25. Functions

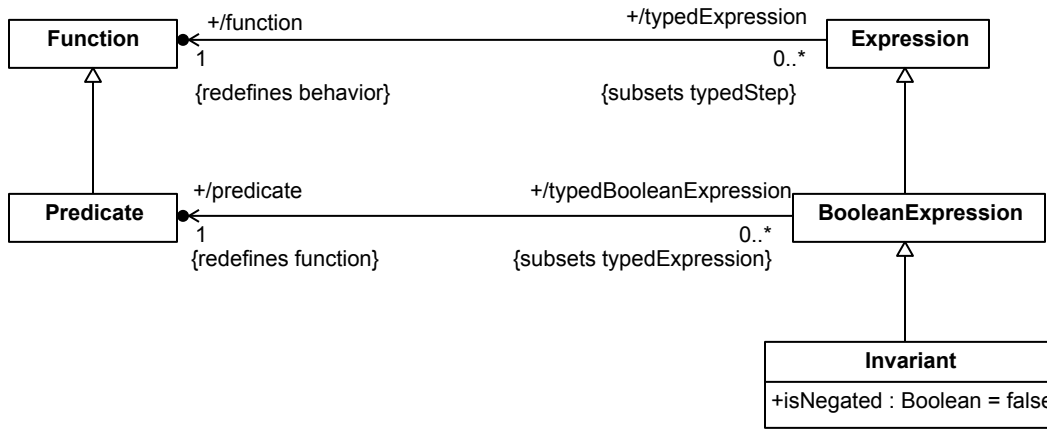


Figure 26. Predicates

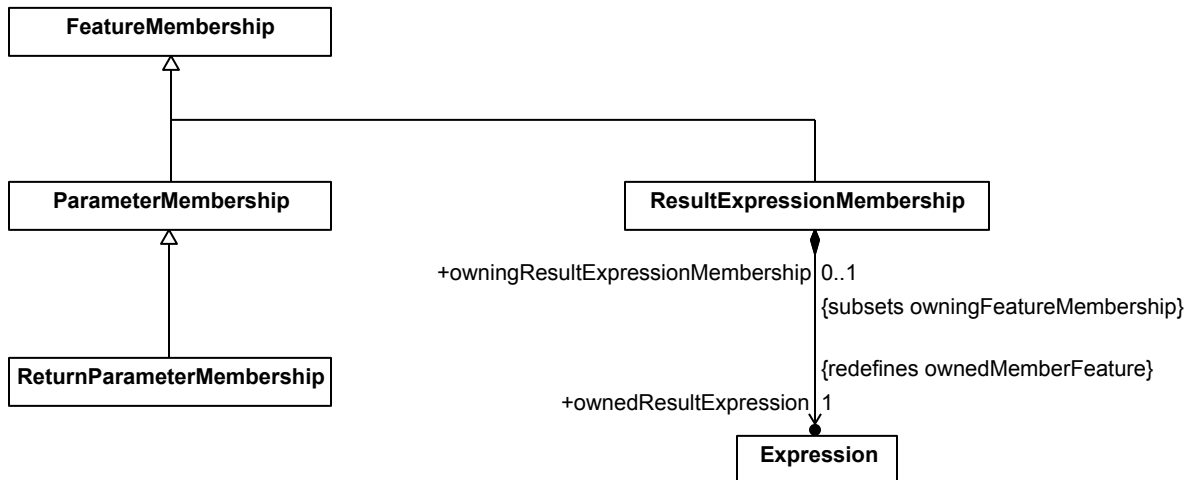


Figure 27. Function Memberships

### 8.3.4.7.2 BooleanExpression

#### Description

A BooleanExpression is a Boolean-valued Expression whose type is a Predicate. It represents a logical condition resulting from the evaluation of the Predicate.

A BooleanExpression must subset, directly or indirectly, the Expression *booleanEvaluations* from the Base model library, which is typed by the base Predicate *BooleanEvaluation*. As a result, a BooleanExpression must always be typed by BooleanEvaluation or a subclass of BooleanEvaluation.

### General Classes

Expression

### Attributes

/predicate : Predicate {redefines function}

The Predicate that types the Expression.

### Operations

No operations.

### Constraints

None.

## 8.3.4.7.3 Expression

### Description

An Expression is a Step that is typed by a Function. An Expression that also has a Function as its *featuringType* is a computational step within that Function. An Expression always has a single *result* parameter, which redefines the *result* parameter of its defining *function*. This allows Expressions to be interconnected in tree structures, in which inputs to each Expression in the tree are determined as the results of other Expressions in the tree.

### General Classes

Step

### Attributes

/function : Function {redefines behavior}

The Function that types this Expression.

/isModelLevelEvaluable : Boolean

Whether this Expression meets the constraints necessary to be evaluated at *model level*, that is, using metadata within the model.

/result : Feature {subsets parameter, output}

The *result* parameter of the Expression, derived as the single *parameter* of the Expression with direction *out*. The result of an Expression must either be inherited from its *function* or (directly or indirectly) redefine the *result* parameter of its *function*.

## Operations

`evaluate(target : Element) : Element [0..*]`

If this Expression `isModelLevelEvaluable`, then evaluate it using the `target` as the context Element for resolving Feature names and testing classification. The result is a collection of Elements, each of which must be a `LiteralExpression` or a Feature that is not an Expression.

**pre:** `isModelLevelEvaluable`

## Constraints

None.

### 8.3.4.7.4 Function

#### Description

A Function is a Behavior that has a single `out` parameter that is identified as its `result`. Any other parameters of a Function than the `result` must have direction `in`. A Function represents the performance of a calculation that produces the values of its `result` parameter. This calculation may be decomposed into Expressions that are `steps` of the Function.

#### General Classes

Behavior

#### Attributes

`/expression : Expression [0..*] {subsets step}`

The Expressions that are steps in the calculation of the `result` of this Function.

`/isModelLevelEvaluable : Boolean`

Whether this Function can be used as the `function` of a model-level evaluable `InvocationExpression`.

`/result : Feature {subsets parameter, output}`

The `result` parameter of the Function, derived as the single `parameter` of the Function with direction `out`.

## Operations

No operations.

## Constraints

None.

### 8.3.4.7.5 Invariant

#### Description

An Invariant is a `BooleanExpression` that is asserted to have a specific Boolean result value. If `isNegated = false`, then the Invariant must subset, directly or indirectly, the `BooleanExpression trueEvaluations` from the



Kernel library, meaning that the result is asserted to be true. If `isNegated = true`, then the Invariant must subset, directly or indirectly, the BooleanExpression *falseEvaluations* from the Kernel library, meaning that the result is asserted to be false.

### General Classes

BooleanExpression

### Attributes

`isNegated` : Boolean

Whether this Invariant is asserted to be false rather than true.

### Operations

No operations.

### Constraints

None.

## 8.3.4.7.6 Predicate

### Description

A Predicate is a Function whose `result` Parameter has type *Boolean* and multiplicity 1..1.

### General Classes

Function

### Attributes

None.

### Operations

No operations.

### Constraints

None.

## 8.3.4.7.7 ResultExpressionMembership

### Description

A ResultExpressionMembership is a FeatureMembership that indicates that the `ownedResultExpression` provides the result values for the Function or Expression that owns it. The owning Function or Expression must contain a BindingConnector between the `result` parameter of the `ownedResultExpression` and the `result` parameter of the Function or Expression.

### General Classes

FeatureMembership

#### **Attributes**

ownedResultExpression : Expression {redefines ownedMemberFeature}

The Expression that provides the result for the owner of the ResultExpressionMembership.

#### **Operations**

No operations.

#### **Constraints**

None.

### **8.3.4.7.8 ReturnParameterMembership**

#### **Description**

A ReturnParameterMembership is a ParameterMembership that indicates that the memberParameter is the result parameter of a Function or Expression. The direction of the memberParameter must be out.

#### **General Classes**

ParameterMembership

#### **Attributes**

None.

#### **Operations**

No operations.

#### **Constraints**

None.

### **8.3.4.8 Expressions Abstract Syntax**

### 8.3.4.8.1 Overview

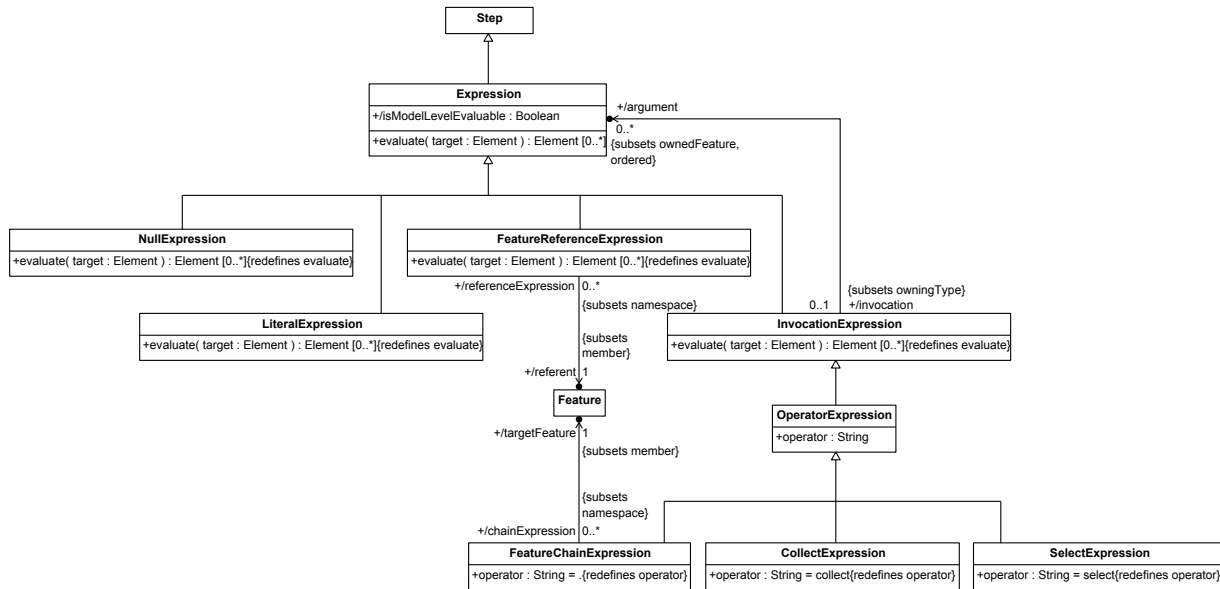


Figure 28. Expressions

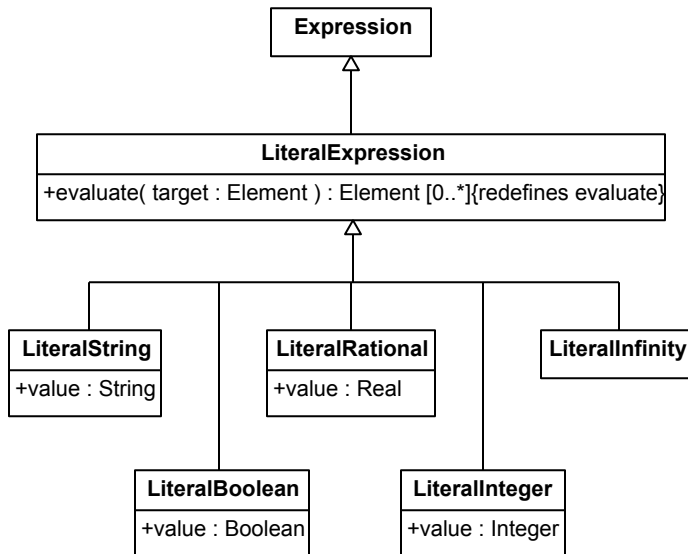


Figure 29. Literal Expressions

### 8.3.4.8.2 CollectExpression

#### Description

A CollectExpression is an OperatorExpression whose operator is "collect", which resolves to the library Function *ControlFunctions::collect*.

#### General Classes

OperatorExpression

### Attributes

operator : String {redefines operator}

### Operations

No operations.

### Constraints

None.

## 8.3.4.8.3 FeatureChainExpression

### Description

A FeatureChainExpression is an OperatorExpression whose operator is ".", which resolves to the library Function *ControlFunctions::'. '*. It evaluates to the result of chaining the *result* Feature of its single argument Expression with its *targetFeature*.

The first two *members* of a FeatureChainExpression must be its single argument Expression and its *targetFeature*. Its only other *members* shall be those necessary to complete it as an InvocationExpression.

### General Classes

OperatorExpression

### Attributes

operator : String {redefines operator}

/targetFeature : Feature {subsets member}

The Feature that is accessed by this FeatureChainExpression, derived as its second *member* Feature (the first being its one *argument* Expression). This Feature must redefine the *target Feature of the Function* *ControlFunctions::'. '*.

### Operations

No operations.

### Constraints

None.

## 8.3.4.8.4 FeatureReferenceExpression

### Description

A FeatureReferenceExpression is an Expression whose *result* is bound a *referent* Feature. The only *members* allowed for a FeatureReferenceExpression are the *referent*, the *result* and the BindingConnector between them.

### General Classes

Expression

## Attributes

/referent : Feature {subsets member}

The Feature that is referenced by this FeatureReferenceExpression, derived as its first member Feature.

## Operations

evaluate(target : Element) : Element [0..\*]

If the target Element is a Type that has a feature that redefines the referent, then return the result of evaluating the Expression given by the FeatureValue of that feature. Otherwise, if the referent has no featuringTypes, return the referent. Otherwise return an empty sequence.

```
body: if not target.ocIsKindOf(Type) then Sequence{}  
else  
  let feature: Sequence(Feature) =  
    target.ocIsType(Type).feature->select(f |  
      f.ownedRedefinition.redefinedFeature->  
        includes(referent)) in  
    if feature->notEmpty() then  
      feature.valuation.value.evaluate(target)  
    else if referent.featuringType->isEmpty()  
      then referent  
    else Sequence{}  
    endif endif  
endif
```

## Constraints

featureReferenceExpressionIsModelLevelEvaluable

A FeatureReferenceExpression is always model-level evaluable (though it may produce no value on some targets).

### 8.3.4.8.5 InvocationExpression

#### Description

An InvocationExpression is an Expression each of whose input parameters are bound to the result of an owned argument Expression. Each input parameter may be bound to the result of at most one argument.

#### General Classes

Expression

#### Attributes

/argument : Expression [0..\*] {subsets ownedFeature, ordered}

The value Expressions of the FeatureValues of the input parameters of the InvocationExpression.

#### Operations

evaluate(target : Element) : Element [0..\*]

Apply the Function that is the `type` of this `InvocationExpression` to the argument values resulting from evaluating each of the `argument` Expressions on the given `target`. If the application is not possible, then return an empty sequence.

### Constraints

`invocationExpressionIsModelLevelEvaluable`

An `InvocationExpression` is model-level evaluable if all its `argument` Expressions are model-level evaluable and its `function` is model-level evaluable.

```
isModelLevelEvaluable =  
    argument->forAll(isModelLevelEvaluable) and  
    function.isModelLevelEvaluable
```

#### 8.3.4.8.6 LiteralBoolean

##### Description

`LiteralBoolean` is a `LiteralExpression` that provides a *Boolean* value as a result. It must have an owned `result` parameter whose type is *Boolean*.

##### General Classes

`LiteralExpression`

##### Attributes

`value` : *Boolean*

The *Boolean* value that is the result of evaluating this Expression.

##### Operations

No operations.

##### Constraints

None.

#### 8.3.4.8.7 LiteralExpression

##### Description

A `LiteralExpression` is an Expression that provides a basic value as a result. It must directly or indirectly specialize the Function *LiteralEvaluation* from the *Base* model library, which has no parameters other than its result, which is a single *DataValue*.

##### General Classes

`Expression`

##### Attributes

None.

## Operations

`evaluate(target : Element) : Element [0..*]`

The model-level value of a `LiteralExpression` is itself.

`body : Sequence{self}`

## Constraints

`literalExpressionIsModelLevelEvaluable`

A `LiteralExpression` is always model-level evaluable.

`isModelLevelEvaluable = true`

### 8.3.4.8.8 LiteralInteger

#### Description

A `LiteralInteger` is a `LiteralExpression` that provides an `Integer` value as a result. It must have an owned `result` parameter whose type is *Integer*.

#### General Classes

`LiteralExpression`

#### Attributes

`value : Integer`

The `Integer` value that is the result of evaluating this `Expression`.

#### Operations

No operations.

#### Constraints

None.

### 8.3.4.8.9 LiteralReal

#### Description

A `LiteralRational` is a `LiteralExpression` that provides a `Rational` value as a result. It must have an owned `result` parameter whose type is *Rational*.

#### General Classes

`LiteralExpression`

#### Attributes

`value : Real`

The value whose rational approximation is the result of evaluating this Expression.

### **Operations**

No operations.

### **Constraints**

None.

#### **8.3.4.8.10 LiteralString**

### **Description**

A `LiteralString` is a `LiteralExpression` that provides a `String` value as a result. It must have an owned `result` parameter whose type is *String*.

### **General Classes**

`LiteralExpression`

### **Attributes**

value : `String`

The `String` value that is the result of evaluating this Expression.

### **Operations**

No operations.

### **Constraints**

None.

#### **8.3.4.8.11 LiteralInfinity**

### **Description**

A `LiteralInfinity` is a `LiteralExpression` that provides the positive infinity value ("\*"). It must have an owned `result` parameter whose type is *Positive*.

### **General Classes**

`LiteralExpression`

### **Attributes**

None.

### **Operations**

No operations.

### **Constraints**



None.

#### 8.3.4.8.12 NullExpression

##### Description

A NullExpression is an Expression that results in a null value. It must be typed by a *NullEvaluation* that results in an empty value.

##### General Classes

Expression

##### Attributes

None.

##### Operations

evaluate(target : Element) : Element [0..\*]

The model-level value of a NullExpression is an empty sequence.

**body:** Sequence{ }

##### Constraints

nullExpressionIsModelLevelEvaluable

A NullExpression is always model-level evaluable.

isModelLevelEvaluable = true

#### 8.3.4.8.13 OperatorExpression

##### Description

An OperatorExpression is an InvocationExpression whose `function` is determined by resolving its `operator` in the context of one of the standard Function packages from the Kernel Model Library.

##### General Classes

InvocationExpression

##### Attributes

/operand : Expression [0..\*] {ordered}

**Implementation note.** This property is currently just an implementation workaround and is not part of the normative abstract syntax.

operator : String

An operator symbol that names a corresponding Function from one of the standard Function packages from the Kernel Model Library .

## Operations

No operations.

## Constraints

None.

### 8.3.4.8.14 SelectExpression

#### Description

A SelectExpression is an OperatorExpression whose operator is "select", which resolves to the library Function *ControlFunctions::select*.

#### General Classes

OperatorExpression

#### Attributes

operator : String {redefines operator}

## Operations

No operations.

## Constraints

None.

### 8.3.4.9 Interactions Abstract Syntax

#### 8.3.4.9.1 Overview

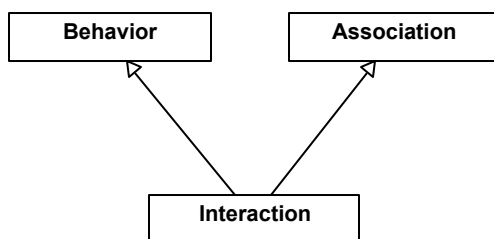
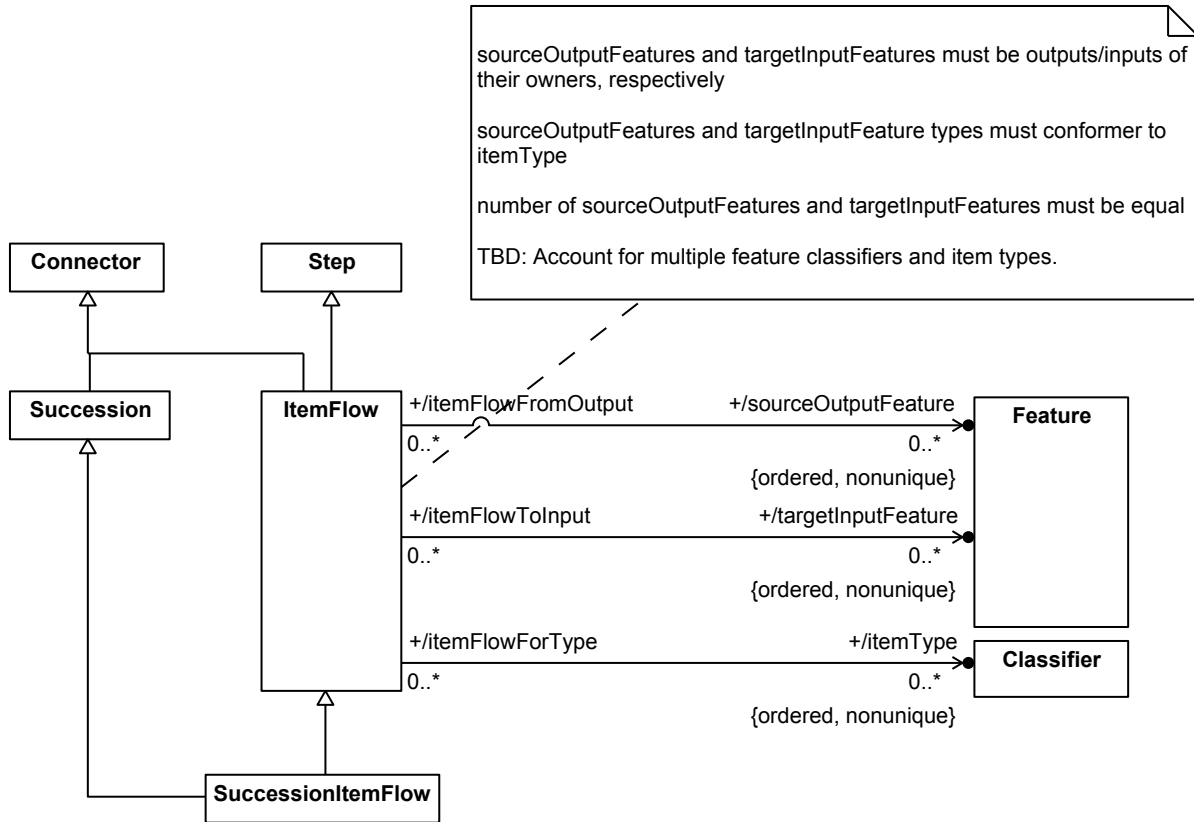


Figure 30. Interactions



**Figure 31. Item Flows**

The Features that originate the ItemFlow. They must be owned outputs of the source participant of the ItemFlow. If there are no such Features, then the ItemFlow must be abstract.

#### 8.3.4.9.2 ItemFlow

##### Description

An ItemFlow is a Step that represents the transfer of objects or values from one Feature to another. ItemFlows can take non-zero time to complete.

An ItemFlow must be typed by the Interaction *Transfer* from the Kernel library, or a specialization of it.

##### General Classes

Connector  
Step

##### Attributes

/itemFeature : ItemFeature [1..\*] {subsets ownedFeature}

The Feature representing the Item in transit between the source and the target during the transfer. (IMPL)

/itemFlowEnd : ItemFlowEnd [2..\*] {redefines connectorEnd}

A `connectorEnd` of this `ItemFlow`. (IMPL)

`/itemFlowFeature` : `ItemFlowFeature` [2..\*]

The `sourceOutputFeatures` and `targetInputFeatures` of this `ItemFlow`. (IMPL).

`/itemType` : `Classifier` [0..\*] {ordered, nonunique}

The type of the item transferred, derived as the `type` of the `feature` of the `ItemFlow` that directly or indirectly redefines `Transfer::item`.

`/sourceOutputFeature` : `Feature` [0..\*] {ordered, nonunique}

The `Feature` that originates the `ItemFlow`.

`/targetInputFeature` : `Feature` [0..\*] {ordered, nonunique}

The `Features` that receive the `ItemFlow`. They must be owned `outputs` of the target participant of the `ItemFlow`. If there are no such `Features`, then the `ItemFlow` must be abstract.

### Operations

No operations.

### Constraints

None.

#### 8.3.4.9.3 Interaction

##### Description

An `Interaction` is a `Behavior` that is also an `Association`, providing a context for multiple objects that have behaviors that impact one another.

##### General Classes

`Behavior`  
`Association`

##### Attributes

None.

##### Operations

No operations.

##### Constraints

None.

#### 8.3.4.9.4 SuccessionItemFlow

##### Description

A SuccessionItemFlow is an ItemFlow that also provides temporal ordering. It classifies *Transfers* that cannot start until the source *Occurrence* has completed and that must complete before the target *Occurrence* can start.

A SuccessionItemFlow must be typed by the Interaction *TransferBefore* from the Kernel Library, or a specialization of it.

## General Classes

Succession  
ItemFlow

## Attributes

None.

## Operations

No operations.

## Constraints

None.

### 8.3.4.10 Feature Values Abstract Syntax

#### 8.3.4.10.1 Overview

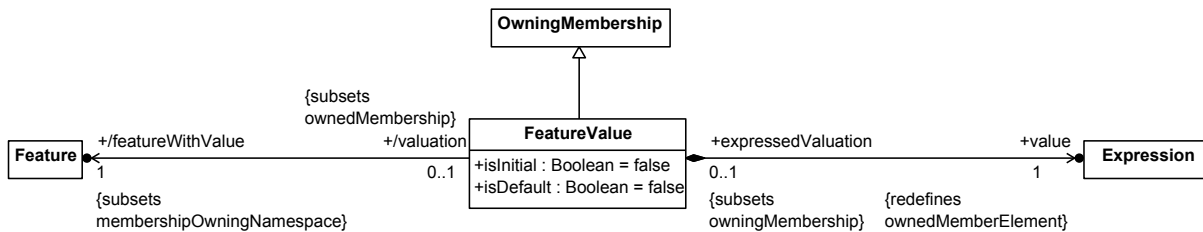


Figure 32. Feature Values

#### 8.3.4.10.2 FeatureValue

### Description

A FeatureValue is a Membership that identifies a particular member Expression that provides the value of the Feature that owns the FeatureValue. The value is specified as either a bound value or an initial value, and as either a concrete or default value. A Feature can have at most one FeatureValue.

If `isInitial = false`, then the result of the value expression is bound to the `featureWithValue` using a BindingConnector. Otherwise, the `featureWithValue` is initialized using a FeatureWritePerformance.

If `isDefault = false`, then the above semantics of the FeatureValue are realized for the given `featureWithValue`. Otherwise, the semantics are realized for any individual of the `featuringType` of the `featureWithValue`, unless another value is explicitly given for the `featureWithValue` for that individual.

## General Classes

OwningMembership

## Attributes

`/featureWithValue : Feature {subsets membershipOwningNamespace}`

The Feature to be provided a value.

`isDefault : Boolean`

Whether this FeatureValue is a concrete specification of the bound of initial value of the `featureWithValue`, or just a default value that may be overridden.

`isInitial : Boolean`

Whether this FeatureValue specifies a bound value or an initial value for the `featureWithValue`.

`value : Expression {redefines ownedMemberElement}`

The Expression that provides the value of the `featureWithValue` as its result.

## Operations

No operations.

## Constraints

`featureValueBindingConnector`

The `valueConnector` must be an `ownedMember` of the `featureWithValue` whose `relatedElements` are the `featureWithValue` and the result of the `value` Expression and whose `featuringTypes` are the same as those of the `featureWithValue`.

```
valueConnector.owningNamespace = featureWithValue and  
valueConnector.relatedFeature->includes(featureWithValue) and  
valueConnector.relatedFeature->includes(value.result) and  
valueConnector.featuringType = featureWithValue.featuringType
```

`featureValueExpressionDomain`

The `value` Expression must have the same `featuringTypes` as the `featureWithValue`.

```
value.featuringType = featureWithValue.featuringType
```

### 8.3.4.11 Multiplicities Abstract Syntax

### 8.3.4.11.1 Overview

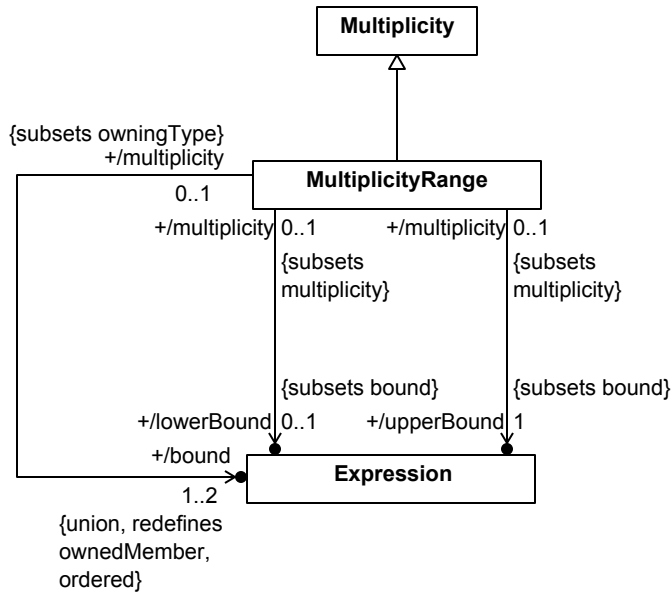


Figure 33. Multiplicities

### 8.3.4.11.2 MultiplicityRange

#### Description

A **MultiplicityRange** is a **Multiplicity** whose value is defined to be the (inclusive) range of natural numbers given by the result of a `lowerBound` Expression and the result of an `upperBound` Expression. The result of the `lowerBound` Expression shall be of type *Natural*, while the result of the `upperBound` Expression shall be of type *UnlimitedNatural*. If the result of the `upperBound` Expression is the unbounded value `*`, then the specified range includes all natural numbers greater than or equal to the `lowerBound` value.

#### General Classes

**Multiplicity**

#### Attributes

`/bound` : Expression [1..2] {redefines ownedMember, ordered, union}

The bound Expressions of the **MultiplicityRange**. These shall be the only `ownedMembers` of the **MultiplicityRange**.

`/lowerBound` : Expression [0..1] {subsets bound}

The Expression whose result provides the lower bound of **MultiplicityRange**. If no `lowerBound` Expression is given, then the lower bound shall have the same value as the upper bound, unless the upper bound is unbounded (`*`), in which case the lower bound shall be 0.

`/upperBound` : Expression {subsets bound}

The Expression whose result is the upper bound of the **MultiplicityRange**.

#### Operations

No operations.

### Constraints

multiplicityRangeExpressionDomain

The bounds of a MultiplicityRange shall have the same `featuringTypes` as the MultiplicityRange.

```
bound->forAll(b | b.featuringType = self.featuringType)
```

## 8.3.4.12 Metadata Abstract Syntax

### 8.3.4.12.1 Overview

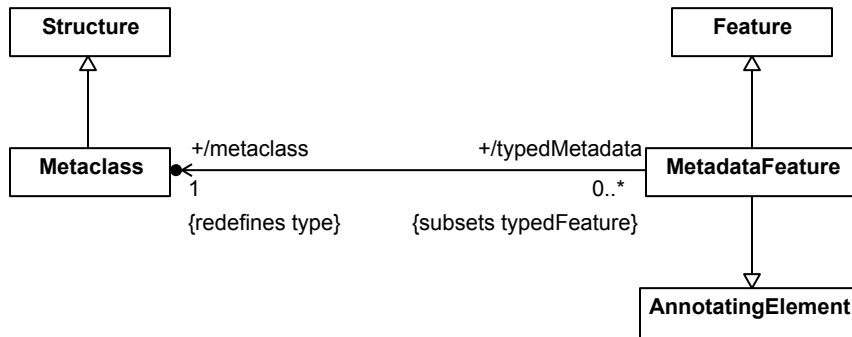


Figure 34. Metadata Annotation

### 8.3.4.12.2 Metaclass

#### Description

A Metaclass is a Structure used to type MetadataFeatures. It must subclassify, directly or indirectly, the base type *Metadata* from the Kernel Library.

#### General Classes

Structure

#### Attributes

None.

#### Operations

No operations.

### Constraints

None.

### 8.3.4.12.3 MetadataFeature

#### Description



A `MetadataFeature` is a `Feature` that is an `AnnotatingElement` used to annotate another `Element` with metadata. It is typed by a `Metaclass`. All its `ownedFeatures` must `redefine` features of its `metaclass` and any feature bindings must be model-level evaluable.

A `MetadataFeature` must subset, directly or indirectly, the base `MetadataFeature` *metadata* from the Kernel Library.

### General Classes

`AnnotatingElement`  
`Feature`

### Attributes

`/metaclass : Metaclass {redefines type}`

The `type` of this `AnnotatingFeature`, which must be a `DataType`.

### Operations

No operations.

### Constraints

None.

## 8.3.4.13 Packages Abstract Syntax

### 8.3.4.13.1 Overview

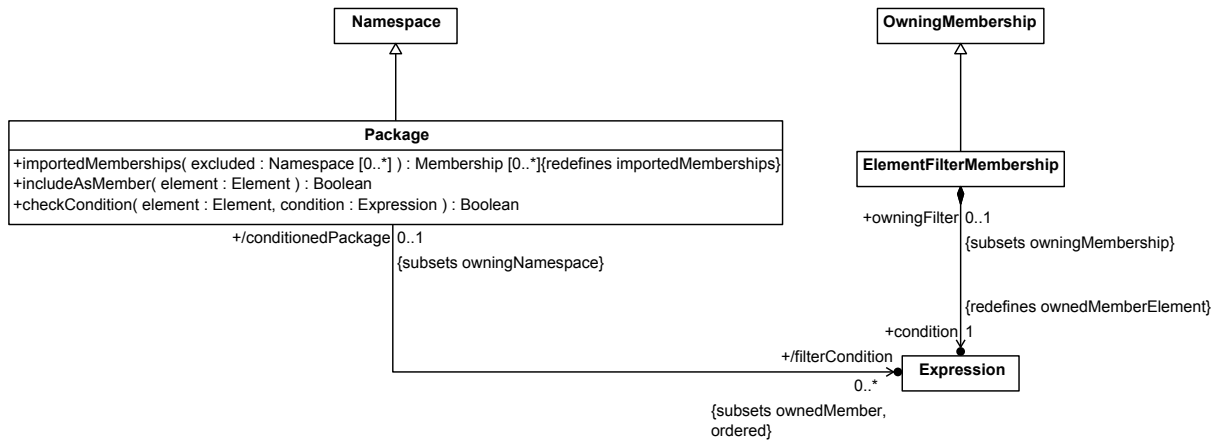


Figure 35. Packages

### 8.3.4.13.2 ElementFilterMembership

#### Description

`ElementFilterMembership` is a Membership between a `Namespace` and a model-level evaluable Boolean `Expression`, asserting that imported members of the `Namespace` should be filtered using the `condition` `Expression`. A general `Namespace` does not define any specific filtering behavior, but such behavior may be defined for various specialized kinds of `Namespaces`.

### General Classes

OwningMembership

### Attributes

`condition` : Expression {redefines ownedMemberElement}

The model-level evaluable Boolean Expression used to filter the members of the membershipOwningNamespace of this ElementFilterMembership.

### Operations

No operations.

### Constraints

`elementFilterIsBoolean`

The result Feature of the `condition` Expression must have *ScalarValues::Boolean* as a type.

`elementFilterIsModelLevelEvaluable`

The `condition` Expression must be model-level evaluable.

`condition.isModelLevelEvaluable`

## 8.3.4.13.3 Package

### Description

A Package is a Namespace used to group Elements, without any instance-level semantics. It may have one or more model-level evaluable `filterCondition` Expressions used to filter its `importedMemberships`. Any imported member must meet all of the `filterConditions`.

### General Classes

Namespace

### Attributes

`/filterCondition` : Expression [0..\*] {subsets ownedMember, ordered}

The model-level evaluable Boolean Expressions used to filter the members of this Package, derived as those ownedMembers of the Package that are owned via ElementFilterMembership.

### Operations

`checkCondition(element : Element, condition : Expression) : Boolean`

Model-level evaluate the given `condition` Expression with the given `element` as its target. If the result is a LiteralBoolean, return its value. Otherwise return false.

```
body: let results: Sequence(Element) = condition.evaluate(element) in
    result->size() = 1 and
    results->at(1).oclIsKindOf(LiteralBoolean) and
    results->at(1).oclAsType(LiteralBoolean).value
```

importedMemberships(excluded : Namespace [0..\*]) : Membership [0..\*]

Exclude Elements that do not meet all the filterConditions.

```
body: self.oclAsType(Namespace).importedMemberships(excluded)->
  select(m | self.includeAsMember(m.memberElement))
```

includeAsMember(element : Element) : Boolean

Determine whether the given element meets all the filterConditions.

```
body: let metadataAnnotations: Sequence(AnnotatingElement) =
  element.ownedAnnotation.annotatingElement->
    select(oclIsKindOf(AnnotatingFeature)) in
  self.filterCondition->forall(cond |
    metadataAnnotations->exists(elem |
      self.checkCondition(elem, cond)))
```

## Constraints

packageImportVisibility

The ownedImports of a Package must not have a visibility of protected.

```
ownedImport->forall(visibility <> VisibilityKind::protected)
```

packageOwnedMembershipVisibility

The ownedMemberships of a Package must not have a visibility of protected.

```
ownedMembership->forall(visibility <> VisibilityKind::protected)
```

## 8.4 Semantics

### 8.4.1 Semantics Overview

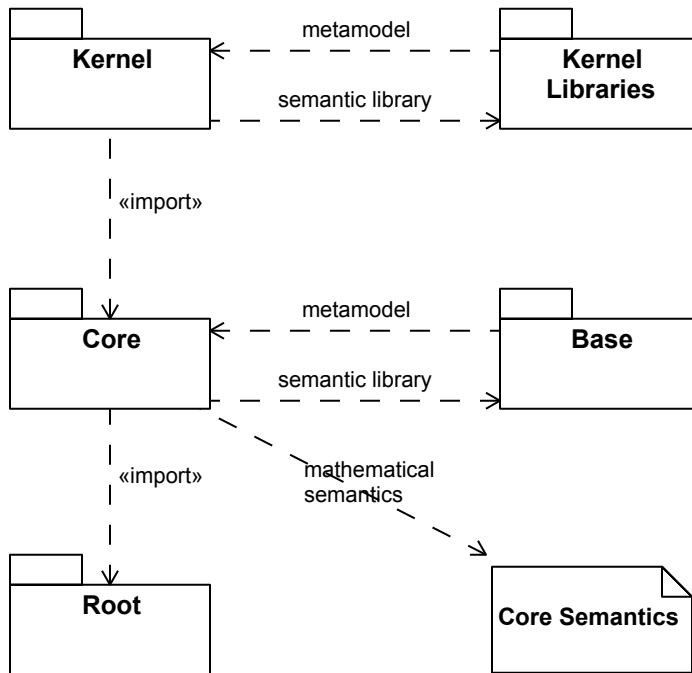
A KerML model is intended to *represent* a system being modeled. The model is *interpreted* to make statements about the modeled system. The model may describe an existing system, in which case, if the model is correct, the statements it is interpreted to make about the system should all be true. A model may also be used to specify a imagined or planned system, in which case the statements the model is interpreted to make should be true for any system that is properly constructed and operated according to the model.

The *semantics* of KerML specify how a KerML model is to be interpreted. The semantics are defined in terms of the abstract syntax representation of the model, and only for models which are *valid* relative to the structure and constraints specified for the KerML abstract syntax (see [8.3](#)). As further specified in this subclause, models expressed in KerML are given semantics by implicitly reusing elements from the semantic models in the Kernel Model Library (see [Clause 9](#)). These library models represent conditions on the structure and behavior of the system being modeled, which are further augmented in a user model as appropriate.

A formal specification of semantics allows models to be interpreted consistently. In particular, all KerML models extend library models expressed in KerML itself, understandable by KerML modelers. These library models can then be ultimately reduced to a small, core subset of KerML, which is grounded in mathematical logic. The goal is to provide uniform model interpretation, which improves communication between everyone involved in modeling, including modelers and tool builders.

KerML semantics are specified by a combination of mathematics and model libraries, as illustrated in [Fig. 36](#). The left side of this diagram shows the abstract syntax packages corresponding to the three layers of KerML (see 6.1). The right side shows the corresponding semantic layering.

1. The Root Layer defines the syntactic foundation KerML and, as such, does not have a semantic interpretation relative to the modeled system.
2. The Core Layer is grounded in mathematical semantics, supported by the `Base` package from the Kernel Model Library (see [9.2.2](#)). Subclause [8.4.2](#) specifies the semantics of the Core layer.
3. The Kernel Layer is given semantics fully through its relationship to the Model Library (see [Clause 9](#)). The semantic specification for each Kernel sub-package summarizes constraints on Kernel abstract syntax elements that specify how the model library is used when models are constructed following the abstract syntax. Subclause [8.4.3](#) specifies the semantics for the Core and Kernel Layers of KerML .



**Figure 36. KerML Semantic Layers**

## 8.4.2 Core Semantics

### 8.4.2.1 Core Semantics Overview

The Core semantics are defined mathematically, using a model-theoretic approach. Core mathematical semantics is expressed in first order logic notation, extended as follows:

1. Quantifiers can specify that variable values must be members of particular sets, rather than leaving this to the body of the statement ( $\forall t_g \in V_T \dots$  is short for  $\forall t_g \ t_g \in V_T \Rightarrow \dots$ ). The same set can be given once for multiple variables ( $\forall t_g, t_s \in V_T \dots$  is short for  $\forall t_g, t_s \ t_g \in V_T \wedge t_s \in V_T \Rightarrow \dots$ ).
2. Dots (.) appearing between metaproperty names have the same meaning as in OCL, including implicit collections [OCL].
3. Sets are identified in the usual set-builder notation, which specifies members of a set between curly braces (" $\{ \}$ "). The notation is extended with "#" before an opening brace to refer to the cardinality of a set.

Element names appearing in the mathematical semantics refer to the element itself, rather than its instances, using the same font conventions as given in [8.1](#).

The mathematical semantics use the following model-theoretic terms, explained in terms of this specification:

- *Vocabulary*: Model elements conforming to the KerML abstract syntax, with additional restrictions given in this subclause.
- *Universe*: All actual or potential things the vocabulary could possibly be about.
- *Interpretation*: The relationship between vocabulary and mathematical structures made of elements of the universe.

The above terms are formally defined below.

- A vocabulary  $V = (V_T, V_C, V_F)$  is a 3-tuple where:
  - $V_T$  is a set of types (model elements classified by Type or its specializations, see [8.3.3.1](#)).
  - $V_C \subseteq V_T$  is a set of classifiers (model elements classified by Classifier or its specializations, see [8.3.3.2](#)), including at least  $\text{Base}::\text{Anything}$  from KerML model library, see [9.2.2](#)).
  - $V_F \subseteq V_T$  is a set of features (model elements classified by Feature or its specializations, see [8.3.3.3](#)), including at least  $\text{Base}::\text{things}$  from the KerML model library (see [9.2.2](#)).
  - $V_T = V_C \cup V_F$
- An interpretation  $I = (\mathcal{A}, \cdot^T)$  for  $V$  is a 2-tuple where:
  - $\mathcal{A}$  is a non-empty set (*universe*), and
  - $\cdot^T$  is an (*interpretation*) function relating elements of the vocabulary to sets of sequences of elements of the universe. It has domain  $V_T$  and co-domain that is the power set of  $S$ , where
 
$$S = \cup_{i \in \mathbb{Z}^+} \mathcal{A}^i$$
 $S$  is the set of all n-ary Cartesian products of  $\mathcal{A}$  with itself, including 1-products, but not 0-products, which are called *sequences*. The Semantics subclauses give other restrictions on the interpretation function.

The semantics of KerML are restrictions on the interpretation relationship, as given mathematically in this and subsequent subclauses on the Core semantics. The phrase *result of interpreting* a model (vocabulary) element refers to sequences paired with the element by  $\cdot^T$ . This specification also refers to this as the *interpretation* of the model element, for short.

The function  $\cdot^{\text{min}T}$  specializes  $\cdot^T$  to the subset of sequences in an interpretation that have no others as tails, except when applied to *Anything*

$$\forall t \in \text{Type}, s_1 \in S \quad s_1 \in (t)^{\text{min}T} \equiv s_1 \in (t)^T \wedge (t \neq \text{Anything} \Rightarrow (\forall s_2 \in S \quad s_2 \in (t)^T \wedge s_2 \neq s_1 \Rightarrow \neg \text{tail}(s_2, s_1)))$$

Functions and predicates for sequences are introduced below. Predicates prefixed with `form:` are defined in [fUML], Clause 10 (Base Semantics).

- *length* is a function version of fUML's *sequence-length*.

$$\forall s, n \quad n = \text{length}(s) \equiv (\text{form:sequence-length } s \ n)$$

- *at* is a function version of fUML's *in-position-count*.

$$\forall x, s, n \quad x = \text{at}(s, n) \equiv (\text{form:in-position-count } s \ n \ x)$$

- *head* is true if the first sequence is the same as the second for some or all of the second starting at the beginning, otherwise is false.

$$\begin{aligned} \forall s_1, s_2 \text{ head}(s_1, s_2) &\Rightarrow \text{form:Sequence}(s_1) \wedge \text{form:Sequence}(s_2) \\ \forall s_1, s_2 \text{ head}(s_1, s_2) &\equiv (\text{length}(s_1) \leq \text{length}(s_2)) \wedge \\ &(\forall p \in \mathbb{Z}^+ \ p \geq 1 \wedge p \leq \text{length}(s_1) \Rightarrow \text{at}(s_1, p) = \text{at}(s_2, p)) \end{aligned}$$

- *tail* is true if the first sequence is the same as the second for some or all of the second finishing at the end, otherwise is false:

$$\begin{aligned} \forall s_1, s_2 \text{ tail}(s_1, s_2) &\Rightarrow \text{form:Sequence}(s_1) \wedge \text{form:Sequence}(s_2) \\ \forall s_1, s_2 \text{ tail}(s_1, s_2) &\equiv (\text{length}(s_1) \leq \text{length}(s_2)) \wedge \\ &(\forall h, p \in \mathbb{Z}^+ \ (h = \text{length}(s_2) - \text{length}(s_1)) \wedge (p > h) \wedge (p \leq \text{length}(s_2) \Rightarrow \text{at}(s_1, p - h) = \text{at}(s_2, p))) \end{aligned}$$

- *concat* is true if the first sequence has the second as head, the third as tail, and its length is the sum of the lengths of the other two, otherwise is false:

$$\begin{aligned} \forall s_0, s_1, s_2 \text{ concat}(s_0, s_1, s_2) &\Rightarrow \text{form:Sequence}(s_0) \wedge \text{form:Sequence}(s_1) \wedge \text{form:Sequence}(s_2) \\ \forall s_0, s_1, s_2 \text{ concat}(s_0, s_1, s_2) &\equiv (\text{length}(s_0) = \text{length}(s_1) + \text{length}(s_2)) \wedge \text{head}(s_1, s_0) \wedge \text{tail}(s_2, s_0) \end{aligned}$$

- *reverse* is true if the sequences have the same elements, but in reverse order, otherwise is false:

$$\begin{aligned} \forall s_1, s_2 \text{ reverse}(s_1, s_2) &\Rightarrow \text{form:Sequence}(s_1) \wedge \text{form:Sequence}(s_2) \\ \forall s_1, s_2 \text{ reverse}(s_1, s_2) &\equiv (\text{length}(s_1) = \text{length}(s_2)) \wedge \\ &(\forall p \in \mathbb{Z}^+ \ p \geq 1 \wedge p \leq \text{length}(s_1) \Rightarrow \text{at}(s_1, (\text{length}(s_1) - p + 1)) = \text{at}(s_2, p)) \end{aligned}$$

### 8.4.2.2 Types Semantics

#### Required Specializations of Model Library

1. All Types shall directly or indirectly specialize *Base::Anything* (see [9.2.2.2.1](#)).

#### Type Semantics

The interpretation of Types in a model shall satisfy the following rules:

1. All sequences in the interpretation of a Type are in the interpretations of the Types it specializes.

$$\forall t_g, t_s \in V_T \ t_g \in t_s.\text{specialization.general} \Rightarrow (t_g)^T \subseteq (t_s)^T$$

2. No sequences in the interpretation of a Type are in the interpretations of its disjoining Types.

$$\forall t, t_d \in V_T \ t_d \in t.\text{disjoiningTypeDisjoining.disjoiningType} \Rightarrow ((t)^T \cap (t_d)^T = \emptyset)$$

### 8.4.2.3 Classifiers Semantics

#### Required Specializations of Model Library

See [8.4.2.2](#).

#### Classifier Semantics

The interpretation of the Classifiers in a model shall satisfy the following rules:

1. If the interpretation of a Classifier includes a sequence, it also includes the 1-tail of that sequence.

$$\forall c \in V_C, s_1 \in S \ s_1 \in (c)^T \Rightarrow (\forall s_2 \in S \ \text{tail}(s_2, s_1) \wedge \text{length}(s_2) = 1 \Rightarrow s_2 \in (c)^T)$$

2. The interpretation of the `Classifier Anything` includes all sequences of all elements of the universe.

$$(\text{Anything})^T = S$$

#### 8.4.2.4 Features Semantics

##### Required Specializations of Model Library

1. All Features shall directly or indirectly specialize `Base::things` (see [9.2.2.2.6](#)) (implied by Rule 1 and 2 below combined with the definition of  $\cdot^T$  in [8.4.2.1](#)).

##### Feature Semantics

The interpretation of the Features in a model shall satisfy the following rules:

1. The interpretations of features must have length greater than one.

$$\forall s \in S, f \in V_F \quad s \in (f)^T \Rightarrow \text{length}(s) > 1$$

2. The interpretation of the Feature `things` is all sequences of length greater than one.

$$(\text{things})^T = \{ s \mid s \in S \wedge \text{length}(s) > 1 \}$$

See other rules below.

Features interpreted as sequences of length two or more can be treated as if they were interpreted as sets of ordered pairs (binary relations), where the first and second elements of each pair are from the domain and co-domain of the Feature, respectively. The predicate `feature-pair` below determines whether two sequences can be treated in this way.

Two sequences are a *feature pair* of a Feature if and only if the interpretation of the Feature includes a sequence  $s_0$  such that following are true:

- $s_0$  is the concatenation of the two sequences, in order.
- The first sequence is in the minimal interpretation of all `featuringTypes` of the Feature.
- The second sequence is in the minimal interpretations of all `types` of the Feature.

$$\begin{aligned} \forall s_1, s_2 \in S, f \in V_F \quad & \text{feature-pair}(s_1, s_2, f) \equiv \\ & \exists s_0 \in S \quad s_0 \in (f)^T \wedge \text{concat}(s_0, s_1, s_2) \wedge \\ & (\forall t_1 \in V_T \quad t_1 \in f.\text{featuringType} \Rightarrow s_1 \in (t_1)^{\text{minT}}) \wedge \\ & (\forall t_2 \in V_T \quad t_2 \in f.\text{type} \Rightarrow s_2 \in (t_2)^{\text{minT}}) \end{aligned}$$

The interpretation of the Features in a model shall satisfy the following rules:

3. All sequences in an interpretation of a Feature have a non-overlapping head and tail that are feature pairs of the Feature.

$$\forall s_0 \in S, f \in V_F \quad s_0 \in (f)^T \Rightarrow \exists s_1, s_2 \in S \quad \text{head}(s_1, s_0) \wedge \text{tail}(s_2, s_0) \wedge (\text{length}(s_0) \geq \text{length}(s_1) + \text{length}(s_2)) \wedge \text{feature-pair}(s_1, s_2, f)$$

4. Values of `redefiningFeatures` are the same as the values of their `redefinedFeatures` restricted to the domain the `redefiningFeature`.

$$\begin{aligned} \forall f_g, f_s \in V_F \quad f_g \in f_s.\text{redefinedFeature} \Rightarrow \\ (\forall s_1 \in S \quad (\forall f_{t_s} \in V_T \quad f_{t_s} \in f_s.\text{featuringType} \Rightarrow s_1 \in (f_{t_s})^{\text{minT}}) \Rightarrow \\ (\forall s_2 \in S \quad (\text{feature-pair}(s_1, s_2, f_s) \equiv \text{feature-pair}(s_1, s_2, f_g)))) \end{aligned}$$

5. The multiplicity of a Feature includes the cardinality of its values.

$$\forall s_1 \in S, f \in V_F \quad \#\{s_2 \mid \text{feature-pair}(s_1, s_2, f)\} \in (f.\text{multiplicity})^T$$

6. Sequences in the interpretation of an inverting feature are the reverse of those in the inverted feature.

$$\begin{aligned} \forall f_1, f_2 \in V_F \quad f_2 \in f_1.\text{invertingFeatureInverting.invertingFeature} \Rightarrow \\ (\forall s_1 \in S \quad s_1 \in (f_1)^T \equiv (\exists s_2 \in S \quad s_2 \in (f_2)^T \wedge \text{reverse}(s_2, s_1))) \end{aligned}$$

7. The interpretation of a Feature with a chain is determined by the interpretations of the subchains, see additional predicates below.

$$\forall f \in V_F, cfl \quad cfl = f.\text{chainingFeature} \wedge \text{form:Sequence}(cfl) \wedge \text{length}(cfl) > 1 \Rightarrow \text{chain-feature-n}(f, cfl)$$

The interpretations of a Feature ( $f$ ) derived from a chain of two others ( $f_1$  and  $f_2$ ) are all the sequences formed from feature pairs of the two others that share the same sequence as second and first in their pairs, respectively.

$$\begin{aligned} \forall f, f_1, f_2 \quad \text{chain-feature-2}(f, f_1, f_2) \Rightarrow f \in V_F \wedge f_1 \in V_F \wedge f_2 \in V_F \\ \forall f, f_1, f_2 \quad \text{chain-feature-2}(f, f_1, f_2) \equiv \\ (\forall s_d, s_{cd} \in S \quad \text{feature-pair}(s_d, s_{cd}, f) \equiv \\ \exists s_m \in S \quad \text{feature-pair}(s_d, s_m, f_1) \wedge \text{feature-pair}(s_m, s_{cd}, f_2)) \end{aligned}$$

The interpretations of a Feature ( $f$ ) derived from a chain of two or more others ( $fl$ , a list of features longer than 1) is the last in a series of features ( $flc$ ) that are features derived from subchains, starting with the first two Features in  $fl$ , (deriving the first Feature in  $flc$ ), then the first three (deriving the second Feature in  $flc$ ), and so on, to all the Features in  $fl$  (deriving the last feature in  $flc$ , which is the original Feature  $f$ ).

$$\begin{aligned} \forall f, fl \quad \text{chain-feature-n}(f, fl) \Rightarrow \\ f \in V_F \wedge fl \subseteq V_F \wedge \text{form:Sequence}(fl) \wedge \text{length}(fl) > 1 \\ \forall f, fl \quad \text{chain-feature-n}(f, fl) \equiv \\ \exists flc \quad flc \subseteq V_F \wedge \text{form:Sequence} \wedge \text{length}(flc) = \text{length}(fl) - 1 \wedge \\ (\forall i \in \mathbb{Z}^+ \quad i > 1 \wedge i \leq \text{length}(fl) \Rightarrow \\ \text{chain-feature-2}(\text{at}(flc, i - 1), \text{at}(fl, i - 1), \text{at}(fl, i))) \wedge \\ f = \text{at}(flc, \text{length}(flc)) \end{aligned}$$

## 8.4.3 Kernel Semantics

### 8.4.3.1 Kernel Semantics Overview

The semantics of constructs in the Kernel Layer are specified in terms of the foundational constructs defined in the Core layer supported by reuse of model elements from the Kernel Semantic Model Library (see 9.2). The most common way in which model elements are used is through specialization, in order to meet subtyping constraints specified in the abstract syntax. For example, `Classes` are required to (directly or indirectly) subclassify `Object` from the `Objects` library model, while `Features` typed by `Classes` must subset `objects`. Similarly, `Behaviors` must subclassify `Performance` from the `Performances` library model, while `Steps` (`Features` typed by `Behaviors`) must subset `performances`.

Sometimes more complicated reuse patterns are needed. For example, binary `Associations` (with exactly two ends) specialize `BinaryLink` from the library, and additionally require the ends of the `Association` to redefine the `source` and `target` ends of `BinaryLink`. In all cases, all Kernel syntactic constructs can be ultimately reduced



to semantically equivalent Core patterns. Various elements of the Kernel abstract syntax essentially act as "markers" for modeling patterns typing the Kernel to the Core.

The following subclauses specify the semantics for each syntactic area of the Kernel Layer in terms of the Core pattern it represents and the model library elements that are reused to support this.

### 8.4.3.2 Data Types Semantics

#### Required Specializations of Model Library

1. `DataTypes` shall (indirectly) specialize `Base::DataValue` (see [9.2.2.2.2](#)).
2. Features typed by `DataTypes` shall (indirectly) subset `Base::dataValues` (see [9.2.2.2.3](#)).

#### Data Type Semantics

For all the things at the end of sequences in the interpretation of a `DataType`, the heads of sequences ending in that thing shall be the same as heads of sequences ending in the other things.

### 8.4.3.3 Classes Semantics

#### Required Specializations of Model Library

1. `Classes` shall directly or indirectly specialize `Occurrences::Occurrence` (see [9.2.3.2.14](#)).
2. Features typed by `Classes` shall directly or indirectly subset `Occurrences::occurrences` (see [9.2.3.2.15](#)).

#### Class Semantics

For all the things at the end of sequences in the interpretation of a `Class`, the heads of sequences ending in that thing shall be different than the heads of sequences ending in the other things.

### 8.4.3.4 Structures Semantics

#### Required Specializations of Model Library

1. Structures shall directly or indirectly specialize `Objects::Object` (see [9.2.5.2.7](#)).
2. Features typed by Structures shall directly or indirectly subset `Objects::objects` (see [9.2.5.2.8](#)).

### 8.4.3.5 Associations Semantics

#### Required Specializations of Model Library

1. Associations shall directly or indirectly specialize `Links::Link` (see [9.2.4.2.3](#)).
2. Every `associationEnd` of an Association shall directly or indirectly subset `Link::participant`.
3. Associations with exactly two `associationEnds` shall directly or indirectly specialize `Links::BinaryLink` (see [9.2.4.2.1](#)).
4. Features typed by Associations shall directly or indirectly specialize `Links::links` (see [9.2.4.2.4](#)).
5. Features typed by Associations with exactly two `associationEnds` shall directly or indirectly specialize `Links::binaryLinks` (see [9.2.4.2.2](#)).
6. AssociationStructures shall directly or indirectly specialize `Objects::LinkObject` (see [9.2.5.2.5](#)).
7. Features typed by AssociationStructures shall directly or indirectly specialize `Objects::linkObjects` (see [9.2.5.2.6](#)).

#### Association Semantics

Association `associationEnds` are given a special semantics compared to other members.

An N-ary Association of the form

```
assoc A {  
    end feature e1;  
    end feature e2;  
    ...  
    end feature eN;  
}
```

is semantically equivalent to the Core model

```
classifier A specializes Links::Link {  
    end feature e1 subsets Links::Link::participant;  
    end feature e2 subsets Links::Link::participant;  
    ...  
    end feature eN subsets Links::Link::participant;  
}
```

The general semantics for the multiplicity of an endFeature is such that, even if a multiplicity other than 1..1 is specified, the Feature is required to effectively have multiplicity 1..1 relative to the *Link*. The *Link* instance for an Association is a tuple of participants, each one of which is a value of an endFeature of the Association. Note that the Feature *Link::participant* is declared **readonly**, meaning that the participants in a link cannot change once the link is created.

If an associationEnd has a multiplicity specified other than 1..1, then this shall be interpreted as follows: For an Association with  $N$  associationEnds, consider the  $i$ -th associationEnd  $e_i$ . The multiplicity, ordering and uniqueness constraints specified for  $e_i$  apply to each set of instances of the Association that have the same (singleton) values for each of the  $N-1$  associationEnds other than  $e_i$ .

For example, each instance of the Association

```
assoc Ternary {  
    end feature a[1];  
    end feature b[0..2];  
    end feature c[*] nonunique ordered;  
}
```

consists of three participants, one value for each of the associationEnds  $a$ ,  $b$  and  $c$ . The multiplicities specified for the associationEnds then assert that:

1. For any specific values of  $b$  and  $c$ , there must be exactly one instance of *Ternary*, with the single value allowed for  $a$ .
2. For any specific values of  $a$  and  $c$ , there may be up to two instances of *Ternary*, all of which must have different values for  $b$  (default uniqueness).
3. For any specific values of  $a$  and  $b$ , there may be any number of instance of *Ternary*, which are ordered and allow repeated values for  $c$ .

**Release Note.** The special semantics for the multiplicity of end Features is still under discussion.

If an Association has an ownedSuperclassing to another Association, then its associationEnds redefine the associationEnds of the superclass Association. In this case, the subclass Association will indirectly specialize Link through a chain of Superclassings, and each of its associationEnds will indirectly subset Links::participant through a chain of redefinitions and a subsetting.

## Binary Association Semantics

Following the usual rules for the `associationEnds` of a specialized Association, the first `associationEnd` of the binary Association will redefine `BinaryLink::source` and the second `associationEnd` of the binary Association will redefine `BinaryLink::target`. The Association `BinaryLink` specializes `Link` and the Features `BinaryLink::source` and `BinaryLink::target` subset `Link::participant`. Therefore, the semantics for binary Associations are consistent with the semantics given above for Associations in general. In addition, the equivalent core model for a binary Association adds implicit nested *navigation* Features to each of the `associationEnds` of the Association, as described below.

A binary Association of the form

```
assoc A {  
    end feature e1;  
    end feature e2;  
}
```

is semantically equivalent to the Core model

```
classifier A specializes Links::BinaryLink {  
    end feature e1 redefines Links::BinaryLink::source {  
        feature e2 = A::e2(e1);  
    }  
    end feature e2 redefines Links::BinaryLink::target {  
        feature e1 = A::e1(e2);  
    }  
}
```

As shown above, the added navigation Feature for each end has the same name as the (effective) name of the *other* end. If the name of a navigation Feature is the same as an inheritable Feature from the `ownedGeneralizations` of the containing `associationEnd`, then the navigation Feature shall redefine that otherwise inherited Feature. The notation `A::e2(e1)` means "all values of the end `e2` of all instances of `A` that have the given value for the end `e1`". Therefore, for each value of `A::e1`, `A::e1::e2` gives the values of `e2` that have `e1` at the other end, that is, it defines a *navigation* across `A` from `e1` to `e2`. The meaning of `A::e2::e1` is similar.

**Release Note.** The model for navigation across binary Associations is still under discussion.

## AssociationStructure Semantics

An AssociationStructure has the same semantics as given above for Associations in general, except that, rather than specializing `Links::Link`, it specializes `Objects::LinkObject`, which in turn specializes `Object`, giving AssociationStructures the semantics of Structures (see [8.4.3.4](#)) as well as Associations.

### 8.4.3.6 Connectors Semantics

#### Required Specializations of Model Library

1. Connectors shall directly or indirectly specialize `Links::links` (see [9.2.4.2.4](#)), which means they shall be typed by Associations ([8.3.4.4.2](#)).
2. Connectors with exactly two `relatedFeatures` shall (indirectly) specialize `Links::binaryLinks` (see [9.2.4.2.2](#)).
3. Connectors with at least one `type` that is an AssociationStructure shall (indirectly) specialize `Objects::linkObjects` (see [9.2.5.2.6](#)).
4. BindingConnectors shall directly or indirectly specialize `Links::selfLink` (see [9.2.4.2.6](#)), which means they shall be typed by (a specialization of) `SelfLink` (see [9.2.4.2.5](#)).

5. Successions shall (indirectly) specialize *Occurrences::happensBeforeLinks* (see [9.2.3.2.4](#)), which means they shall be typed by (a specialization of) *HappensBefore* (see [9.2.3.2.3](#)).

## Connector Semantics

An N-ary Connector of the form

```
connector c : A (f1, f2, ... fN);
```

is semantically equivalent to the Core model

```
feature c : A subsets Links::links {  
  end feature e1 redefines A::e1 subsets f1;  
  end feature e2 redefines A::e2 subsets f2;  
  ...  
  end feature eN redefines A::eN subsets fN;  
}
```

where *e1*, *e2*, ..., *eN* are the names of associationEnds of the Association *A*, in the order they are defined in *A*. If explicit multiplicities are given for the connectorEnds, then these become the multiplicities of the endFeatures in the equivalent core model. (If *A* is an AssociationStructure, then *Links::link* is replaced by *Objects::LinkObjects*, above and in the following.)

If the named notation is used for pairing connectorEnds to associationEnds:

```
connector c : A (e_f1 :> f1, e_f2 :> f2, ... e_fN :> fN);
```

then the model is similar:

```
feature c : A subsets Links::links {  
  end feature e_f1 redefines A::e_f1 subsets f1;  
  end feature e_f2 redefines A::e_f2 subsets f2;  
  ...  
  end feature e_fN redefines A::e_fN subsets fN;  
}
```

where the *e\_f1*, *e\_f2*, ..., *e\_fN* are again names of associationEnds of the Association *A*, but now not necessarily in the order in which they are defined in *A*.

The semantic model of a binary Connector is just that of an N-ary Connector with *N* = 2. In particular, if no type is explicitly declared for a binary Connector, then its connectorEnds simply redefine the *source* and *target* ends of the Association *BinaryLink*, which are inherited by the Feature *binaryLinks*.

A binary Connector of the form

```
connector c : A from f1 to f2;
```

is semantically equivalent to

```
feature c : A subsets Links::binaryLinks {  
  end feature source redefines Objects::binaryLinks::source subsets f1;  
  end feature target redefines Objects::binaryLinks::target subsets f2;  
}
```

If *A* is an AssociationStructure, then the equivalent Feature also subsets *Objects::linkObjects*.

## Binding Connector Semantics

BindingConnectors are typed by *SelfLinks*, which have two `associationEnds` that subset each other, meaning they identify the same things (have the same values, see [9.2.4.2.5](#)). This applies to BindingConnector `connectorEnds` also by redefining the `associationEnds` of *SelfLink*.

A BindingConnector of the form

```
binding f1 = f2;
```

is semantically equivalent to the Core model

```
feature subsets Links::selfLinks {  
    end feature thisThing redefines selfLinks::thisThing subsets f1;  
    end feature thatThing redefines selfLinks::thatThing subsets f2;  
}
```

where *selfLinks* is typed by *SelfLink* and, so, inherits the `endFeatures` *self* and *myself*.

## Succession Semantics

Successions are typed by *HappensBefore*, which require the *Occurrence* identified by (value of) its first `associationEnd` (`earlierOccurrence`) to precede the one identified by its second (`laterOccurrence`, see [9.2.3.2.3](#)). This applies to Succession `connectorEnds` also by redefining the `associationEnds` of *HappensBefore*.

A Succession of the form

```
succession first f1 then f2;
```

is semantically equivalent to the Core model

```
feature subsets Occurrences::successions {  
    end feature earlierOccurrence  
        redefines Occurrences::successions::earlierOccurrence subsets f1;  
    end feature laterOccurrence  
        redefines Occurrences::successions::laterOccurrence subsets f2;  
}
```

where *succession* is typed by *HappensBefore* and, so, inherits the `endFeatures` *earlierOccurrence* and *laterOccurrence*.

### 8.4.3.7 Behavior Semantics

#### Required Specializations of Model Library

1. Behaviors shall directly or indirectly specialize *Performances::Performance* (see [9.2.6.2.11](#)).
2. Steps shall directly or indirectly specialize *Performances::performances* (see [9.2.6.2.12](#)), which means they shall be typed by Behaviors.

## Behavior Semantics

A Behavior of the form

```
behavior B ( in x, out y, inout z);
```

is semantically equivalent to

```
class B specializes Performances::Performance {
  in feature x;
  out feature y;
  inout feature z;
}
```

while a Behavior that explicitly specializes another Behavior:

```
behavior B1 specializes B (in x1, out y1);
```

is semantically equivalent to

```
class B1 specializes B {
  in feature x1 redefines x;
  out feature y1 redefines y;
}
```

## Step Semantics

A Step of the form

```
step s ( in u, out v, inout w);
```

is semantically equivalent to

```
feature s subsets Performances::performances {
  in feature u;
  out feature v;
  inout feature w;
}
```

while a Step that explicitly specializes Behaviors and/or Steps:

```
behavior b : B subsets s (in xx, out yy);
```

is semantically equivalent to

```
feature b : B subsets s {
  in feature xx redefines B::x, s::u;
  out feature yy redefines B::y, s::v;
}
```

**Note.** The behaviors of Steps can have their own Steps, providing for (repeated) refinement of Behaviors by other Behaviors.

### 8.4.3.8 Functions Semantics

#### Required Specializations of Model Library

1. Functions shall directly or indirectly specialize *Performances::Evaluation* (see [9.2.6.2.3](#)).
2. Predicates shall directly or indirectly specialize *Performances::BooleanEvaluation* (see [9.2.6.2.1](#)).
3. Expressions shall directly or indirectly specialize *Performances::evaluations* (see [9.2.6.2.4](#)), which means they shall be directly or indirectly typed by *Performances::Evaluation*.

4. BooleanExpressions (including Invariants) shall directly or indirectly specialize *Performances::booleanEvaluations* (see [9.2.6.2.2](#)), which means they shall be typed by (a specialization of) *Performances::BooleanEvaluation*.

## Function Semantics

A Function of the form

```
function F (a, b) result {  
    resultExpr  
}
```

is semantically equivalent to

```
class F specializes Performances::Evaluation {  
    in a;  
    in b;  
    out result redefines Performances::Evaluation::result  
        = resultExpr;  
}
```

where the binding to *resultExpr* is interpreted as a FeatureValue (see [Clause](#)).

## Expression Semantics

An Expression of the form

```
expr e : F (a, b) result {  
    resultExpr  
}
```

is semantically equivalent to

```
feature e : F subsets Performances::evaluations {  
    in a redefines F::a;  
    in b redefines F::b;  
    out result redefines F::result  
        = resultExpr;  
}
```

## Predicate Semantics

A Predicate is simply a Function with a Boolean result (see [Clause](#)) and, otherwise, has no additional semantics.

## Boolean Expression and Invariant Semantics

An Invariant of the form

```
inv i ( ... ) result {  
    resultExpr  
}
```

is semantically equivalent to

```
feature i subsets Performances::booleanEvaluations {  
    ...  
    out result redefines Performances::booleanEvaluations::result
```

```

    = resultExpr;
    private alwaysTrue = true;
    binding result = alwaysTrue;
}

```

### 8.4.3.9 Expressions Semantics

#### Required Specializations of Model Library

1. LiteralExpressions shall directly or indirectly specialize *Performances::literalEvaluations* (see [9.2.6](#)), which means their function is *Performances::LiteralEvaluations* or a specialization of it.
2. NullExpressions shall directly or indirectly specialize *Performances::nullEvaluations* (see [9.2.6](#)), which means their function is *Performances::NullEvaluations* or a specialization of it.

Also see Required Generalizations for Expressions in [8.4.3.8](#).

#### Null Expression Semantics

Invocations of NullExpressions do not produce any *result* values (see rules above and [Clause](#) ).

#### Literal Expression Semantics

With the exception of LiteralInfinity, each kind of LiteralExpression has a *value* meta-property with its own primitive Type, which is given a required constant value in models to specify the value of the *result* of *LiteralEvaluations* classified by each LiteralExpression (see [9.2.6.1](#)). LiteralInfinity does not have a *value* property, because its *result* parameter value is always a number greater than all the integers ("infinity"), but treated like one, notated by \*, from the standard DataType *Natural*.

LiteralExpressions are Expressions that have the values of their *result* parameter specified as a constant in models by a LiteralExpression's *value* property, ultimately being *DataValues* in the *result* parameter of *LiteralEvaluations* classified by the LiteralExpression (see [9.2.6.1](#)). LiteralInfinities are LiteralExpressions resulting in a number greater than all the integers ("infinity"), but treated like one, notated as \* (see [9.3.2.1](#)).

**Release Note.** The semantics of literals will be more formally addressed in the final submission.

#### Feature Reference Expression Semantics

A FeatureReferenceExpression for a Feature *f* is semantically equivalent the Expression

```

expr () result {
    binding result = f;
}

```

where the types of the *result* parameter are considered to be implicitly the same as those of *f*.

#### Invocation Expression Semantics

Given a function of the form

```
function F(a, b, ...) result;
```

an InvocationExpression of the form

```
F(expr_1, expr_2, ...)
```

is semantically equivalent to *e.result*, where the Expression *e* is



```

    expr e : F (a, b, ...) result {
        expr e_1 ( ) result {
            ...
        }
        expr e_2 ( ) result {
            ...
        }
        ...
        binding a = e_1.result;
        binding b = e_2.result;
        ...
    }

```

and each  $e_n$  is the equivalent of  $expr\_n$  according to this subclause.

With the exception of operators that map to control Functions (see below), the concrete syntax operator Expression notation (see [Clause](#)) is simply special surface syntax for InvocationExpressions of standard library Functions. For example, a unary operator Expression such as

```
! expr
```

is equivalent to the InvocationExpression

```
DataFunctions::'!' (expr)
```

and a binary operator Expression such as

```
expr_1 + expr_2
```

is equivalent to the InvocationExpression

```
DataFunctions::'+' (expr_1, expr_2)
```

where the InvocationExpressions are then semantically interpreted as above.

The + and - operators are the only operators that have both unary and binary usages. However, the corresponding library functions have optional 0..1 multiplicity on their second parameters, so it is acceptable to simply not provide an input for the second argument when mapping the unary usages of these operators.

**Release Note.** Functions in the library Packages *BaseFunctions* and *ScalarFunctions* are extensively specialized in other library Packages to constrain their parameter types (e.g., the Package *RealFunctions* constrains parameter types to be *Real*, etc.). The semantics of Function specialization and dynamic dispatch based on parameter types will be addressed in the final submission.

## Expression Body Semantics

An Expression body used as a base expression (see [Clause](#)) is equivalent to a FeatureReferenceExpression that contains the Expression body as its own referent. That is, a Expression body of the form

```
{ body }
```

is semantically equivalent to

```

expr () result {
    expr e () result { body }
    binding result = e;
}

```

However, when an Expression body is used as the argument to an invocation, this can be more directly realized by directly binding to the Expression body without the intermediate FeatureReferenceExpression. Thus, the invocation

```
F({ body })
```

is semantically equivalent to

```
expr e : F (a) result {
  expr e_1 () result { body }
  binding a = e_1;
}
```

Note that the binding is to `e_1` itself, *not* `e_1.result`.

## Control Function Invocation Semantics

Certain operator expressions (see [Clause](#)) denote invocations of Functions in the *ControlFunctions* library model that have one or more parameters that are Expressions (see [9.4.17](#)). The arguments corresponding to these parameters are handled by special rules that wrap the given argument Expressions in Expression bodies so they can be passed without being immediately evaluated.

The second and third operands of the ternary conditional test operator `?` are for Expression parameters. Therefore, a conditional test Expression of the form

```
expr_1 ? expr_2 : expr_3
```

is semantically equivalent to

```
ControlFunctions::'?'(expr_1, { expr_2 }, { expr_3 })
```

The second operand of the binary conditional logical operators `&&` and `||` is for an Expression parameter. Therefore, a conditional logical Expression of the form

```
expr_1 && expr_2
```

is semantically equivalent to

```
ControlFunctions::'&&'(expr_1, { expr_2 })
```

and similarly for `||`.

## Model-Level Evaluable Expression Semantics

As defined in [Clause](#), a model-level evaluable Expression is an Expression that can be evaluated using metadata available within a model itself. This means that the evaluation rules for such an expression can be defined entirely within the abstract syntax. If an Expression is model-level evaluable, then using `evaluate` operation on it gives the model-level evaluation of the Expression as an ordered list of Elements.

A model-level evaluable Expression is evaluated on a given *target* object (see [8.4.3.13](#) and [8.4.3.14](#) for the targets used in the case of metadata values and filterConditions, respectively), according to the following rules.

1. A NullExpression evaluates to the empty list.
2. A LiteralExpression evaluates to itself.

3. An `FeatureReferenceExpression` evaluates to one of the following.
  - If the target Element has a `MetadataFeature` (see [Clause](#)) with a nested Feature that redefines the `referent`, then the `FeatureReferenceExpression` evaluates to the result of evaluating the corresponding bound `value` expression on the same target Element (if any).
  - Otherwise, if the `referent` is a Feature with no `FeaturingTypes` or with *Anything* as a `FeaturingType`, then the `FeatureReferenceExpression` evaluates to the `referent`.
  - Otherwise, the `FeatureReferenceExpression` evaluates to the empty list.
4. An `InvocationExpression` evaluates to an application of its `function` to argument values corresponding by the results of evaluating each of the `argument` Expressions of the `InvocationExpression`, with the correspondence as given below.

Every Element in the list resulting from a model-level evaluation of an Expression according to the above rules will be either a `LiteralExpression` or a Feature of *Anything*. If each of these Elements is further evaluated according to its regular instance-level semantics, then the resulting list of instances will correspond to the result that would be obtained by evaluating the original Expression using its regular semantics on the referenced metadata of the target Element.

**Release Note.** In the final submission, the semantics of model-level evaluation may be more formally defined as Expression evaluation on a reflective KerML abstract syntax model of the KerML.

### 8.4.3.10 Interactions Semantics

#### Required Specializations of Model Library

1. Interactions shall directly or indirectly specialize `Link::Link` (see [9.2.4.2.3](#)), or `Links::BinaryLink` (see [9.2.4.2.1](#)) for Interactions with exactly two participants.
2. Interactions shall directly or indirectly specialize `Performances::Performance` (see [9.2.6.2.11](#)).
3. ItemFlows shall directly or indirectly specialize `Transfers::transfers` (see [9.2.7.2.3](#)), which means they shall be typed by (a specialization of) `Transfers::Transfer` (see [9.2.7.2.1](#)).
4. The `connectorEnds` of ItemFlows shall
  - a. Redefine `source` and `target` of `Transfers::Transfer` (see [9.2.7.2.1](#)).
  - b. Nest Features that redefine `source::sourceOutput` and `target::targetInput`, and subset the `sourceOutputFeature` and `targetInputFeature` of the ItemFlow.
5. ItemFlows that specify the kind of item flowing (`itemType`) shall add an `ownedFeature` that directly or indirectly redefines `Transfer::item` with that Type.
6. SuccessionItemFlows directly or indirectly specialize `Transfers::flows` (see [9.2.7.2.4](#)), which means they shall be typed by (a specialization of) `Transfers::TransferBefore` (see [9.2.7.2.2](#)).

#### Interaction Semantics

An Interaction of the form

```
interaction I (in x, out y, inout z) {
  end feature e1;
  end feature e2;
}
```

is semantically equivalent to the Core model

```
classifier I specializes Link::BinaryLink, Performances::Performance {
  end feature e1 redefines Link::BinaryLink::source {
    feature e2 = I::e2(e1);
  }
  end feature e2 redefines Link::BinaryLink::target {
    feature e1 = I::e1(e2);
  }
}
```

```

    }
    in feature x;
    out feature y;
    inout feature z;
}

```

## Item Flow Semantics

An ItemFlow of the form

```
flow of item : T from f1.f1_out to f2.f2_in;
```

is semantically equivalent to the core model

```

feature subsets Transfers::transfers {
  end feature redefines source subsets f1 {
    feature redefines sourceOutput subsets f1_out;
  }
  end feature redefines target subsets f2 {
    feature redefines targetInput subsets f2_in;
  }
}

```

A SuccessionItemFlow is semantically the same, except that *Transfers::transfersBefore* is used instead of *transfers*.

### 8.4.3.11 Feature Values Semantics

A Feature of the form

```
feature f = expr;
```

is semantically equivalent to

```

feature f {
  expr e () result { ... }
  binding f = e::result;
}

```

where *e* is the interpretation of *expr* as described in [8.4.3.9](#).

### 8.4.3.12 Multiplicities Semantics

#### Required Specializations of Model Library

1. MultiplicityRanges shall directly subset *Base::naturals* (see [9.2.2.2.4](#)), which means they shall be typed by (a specialization of) *ScalarValues::Natural*.

#### Multiplicity Range Semantics

A MultiplicityRange of the form

```
[expr_1.. expr_2]
```

represents a range of data values of the DataType *Natural* (see [9.3.2.2.4](#)) that are greater than or equal to the result of the Expression *expr\_1* and less than or equal to the result of the Expression *expr\_2*. Essentially, this is

```
all Natural -> select n (expr_1 <= n & n <= expr_2)
```

where, if *expr\_2* evaluates to the unbounded value *\**, all *Natural* data values are less than it.

A *MultiplicityRange* having only a single expression:

```
[expr]
```

is interpreted in one of the following ways:

- If *expr* evaluates to *\**, then the values of the *MultiplicityRange* are the entire extent of *Natural*.
- Otherwise, the values of the *MultiplicityRange* are all *Natural* data values less than or equal to the result of *expr*.

```
all Natural -> select n (n <= expr)
```

**Note.** A conforming tool is not expected to compute the entire set of *Natural* numbers that are values of a *MultiplicityRange*. It is sufficient to check that the values of a Type have a cardinality that is within the range specified by *MultiplicityRange*.

### 8.4.3.13 Metadata Semantics

#### Required Specializations of Model Library

1. Metaclasses shall directly or indirectly specialize *Metaobjects::Metaobject* (see [9.2.16.2.1](#)).
2. MetadataFeatures shall directly or indirectly specialize *Metaobjects::metaobjects* (see [9.2.16.2.2](#)), which means they shall be typed by Metaclasses.
3. If a MetadataFeature directly or indirectly specializes *Metaobjects::SemanticMetadata* (see [9.2.16.2.3](#)), then
  - If one of the following holds:
    - the annotated Type is neither a Classifier nor a Feature
    - the annotated Type is a Classifier and the Type bound to the *baseType* Feature of the MetadataFeature is a Classifier
    - the annotated Type is a Feature and the Type bound to the *baseType* Feature of the MetadataFeature is a Feature
 then the annotated Type shall directly or indirectly specialize the Type referenced by the *baseType*.
  - If the annotated Type is a Classifier and the Type bound to the *baseType* Feature of the MetadataFeature is a Feature, then the annotated classifier shall directly or indirectly specialize each *type* of the Feature referenced by the *baseType*.

#### Metadata

As noted in [Clause](#), while MetadataFeatures are Features, they are defined only within a model and do not have instance-level semantics (i.e., they do not affect instances, as specified in Core – see [Clause](#)). However, at a meta-level, a MetadataFeature can be treated as if the (reflective) Metaclasses of its *annotatedElements* were its *featuringTypes*. In this case, the MetadataFeature defines a map from its *annotatedElements*, as instances of their Metaclasses, to a single instance of its *metaclass*.

Further, a model-level evaluable Expression is simply an Expression that can be evaluated using metadata available within a model itself (see [Clause](#)). If a model-level evaluable Expression is evaluated on such metadata according to the regular semantics of Expressions, then the result will be the same as the static evaluation of the Expression within the model. Therefore, if a MetadataFeature is instantiated as above, the binding of its features to the results of evaluating the model-level evaluable Expressions given as *featureValues* can be interpreted according to the regular semantics of FeatureValues (see [Clause](#)) and BindingConnectors (see [Clause](#)).

When a `value` Expression is model-level evaluated (as described in [8.4.3.9](#)), its target is the `MetadataFeature` that owns the associated `metadataFeature`. This means that the `value` Expression for a nested Feature of a `MetadataFeature` may reference other Features of the `MetadataFeature`, as well as Features with no `featuringTypes` or *Anything* as a `featuringType`.

#### 8.4.3.14 Packages Semantics

Packages do not have semantics (they do not affect instances).

The `filterConditions` of a `Package` are model-level evaluable Expressions that are evaluated as described in [8.4.3.9](#). All `filterConditions` are checked against every `Membership` that would otherwise be imported into the `Package` if it had no `filterCondition`. A `Membership` shall be imported into the `Package` if and only if every `filterCondition` evaluates to true either with no target `Element`, or with any `MetadataFeature` of the `memberElement` of the `Membership` as the target `Element`.

**Implementation Note.** A `MetadataFeature` must be owned by the imported `Element` in order to be accessed when evaluating a `filterCondition`.

# 9 Model Libraries

## 9.1 Model Libraries Overview

A *model library* is a collection of library models that can be reused across many user models. KerML includes three standard model libraries: the Semantic Library (see [9.2](#)), the Data Type Library (see [9.3](#)), and the Function Library (see [9.4](#)). The normative representation of these library models is in the textual concrete syntax, as provided in machine-readable files associated with this specification document. They are described for reference in subclauses of this Clause.

## 9.2 Semantic Library

### 9.2.1 Semantic Library Overview

The Semantic Library is a collection of KerML models that are part of the semantics of the metamodel (see [Clause](#)). They are reused when constructing KerML user models (instantiating the metamodel), as specified by constraints and semantics of metaelements, such as Types being required to specialize *Anything* from the library and Behaviors specializing *Performance* (see [Clause](#) and the Semantics subclauses in [Clause](#)). The library can be specialized for particular applications, such as systems modeling.

The Semantic Library contains a set of packages, one for each library model, as described in a subsequent subclauses. The following are the major areas covered in the Semantic Library.

1. The *Base* library model (see [9.2.2](#)) begins the Specialization hierarchy for all KerML Types, including the most general Classifier *Anything* and the most general Feature *things*. It also contains the most general DataType *DataValue* and its corresponding Feature *dataValues*. The *Links* library model (see [9.2.4](#)) specializes *Base* to provide the semantics for Associations between things.
2. The *Occurrences* library model (see [9.2.3](#)) introduces *Occurrence*, the most general Class of things that exist or happen in time and space, as well as the basic temporal Associations between them. The *Objects* library model (see [9.2.5](#)) specializes *Occurrences* to provide a model of *Objects* and *LinkObjects*, giving semantics to Structures and AssociationStructures, respectively. The *Performances* library model (see [9.2.6](#)) specializes *Occurrences* to provide a model of *Performances* and *Evaluations*, giving semantics to Behaviors and Expressions, respectively. Temporal associations can be used by Successions to specify the order in which *Performances* are carried out during other *Performances*, or when *Objects* exist in relation to each other, or combinations involving *Performances* and *Objects*. The *Transfers* library model (see [9.2.7](#)) models asynchronous flow of items between *Occurrences*, giving semantics to Interactions and ItemFlows. The *FeatureAccessPerformances* library model (see [9.2.8](#)) defines specialized *Performances* for access and modifying the values of features at specific points in time.
3. The *ControlPerformances*, *TransitionPerformances* and *StatePerformances* library models (see [9.2.9](#), [9.2.11](#), and [9.2.10](#)) provide for coordination of multiple *Performances* to carry out some task by using them as types of Steps in an overall containing Behavior. KerML does not provide syntax specific to these library elements (e.g., KerML does not have any "control node" or "state machine" syntax), though it is expected that other languages built on KerML, and using these library models, can add syntax as needed by their applications.

### 9.2.2 Base

### 9.2.2.1 Base Overview

This library model begins the Specialization hierarchy for all KerML Types (see [Clause](#)), starting with the most general Classifier *Anything*, the `type` of the most general Feature *things*, which classify everything in the modeled universe and the relations between them, respectively. Being the most general library elements for their metaclasses means all Classifiers and Features in models, including in libraries, specialize them, respectively. They are specialized into most general DataType *DataValue*, the `type` of *dataValues*, the most general Feature typed by DataTypes, respectively (see [Clause](#)). *DataValues* are *Anything* that can only be distinguished by how they are related to other things (via Features and Associations). These are further specialized into *Natural* and *naturals*, respectively, an extension for mathematical natural numbers (integers zero and greater) extended with a number greater than all the integers ("infinity"), but treated like one, notated as *\** (see [9.3.2.1](#)). The Feature *self* of *Anything* relates each thing in the universe to itself only (see *SelfLinks* in [9.2.4.1](#)).

### 9.2.2.2 Elements

#### 9.2.2.2.1 Anything

##### Element

Classifier

##### Description

Anything is the most general Classifier (M1 instance of M2 Classifier). All other M1 elements (in libraries or user models) specialize it (directly or indirectly). Anything is the `type` for *things*, the most general Feature. Since FeatureTyping is a kind of Generalization, this means that Anything is also a generalization of *things*.

##### General Types

None.

##### Features

`self : Anything {subsets selfSameLife}`

The source of a SelfLink of this thing to itself. `self` is thus a feature that relates everything to itself. It is also the value of the nested `that` feature of all other things featured by this thing.

##### Constraints

None.

#### 9.2.2.2.2 DataValue

##### Element

DataType

##### Description

A DataValue is Anything that can only be distinguished by how it is related to other things (via Features). DataValue is the most general Datatype (M1 instance of M2 Datatype). All other M1 Datatypes (in libraries or user models) specialize it (directly or indirectly).

##### General Types



Anything

### **Features**

None.

### **Constraints**

None.

## **9.2.2.2.3 dataValues**

### **Element**

Feature

### **Description**

`dataValues` is a specialization of `things` restricted to type `DataValue`. All other Features typed by `DataValue` or its specializations (in libraries or user models) specialize it (directly or indirectly).

### **General Types**

`DataValue`

`things`

### **Features**

None.

### **Constraints**

None.

## **9.2.2.2.4 naturals**

### **Element**

Feature

### **Description**

### **General Types**

`Natural`

`dataValues`

### **Features**

None.

### **Constraints**

None.

#### 9.2.2.2.5 SelfSameLifeLink

##### Element

Association

##### Description

SelfLifeLinks are all and only BinaryLinks where the sourceParticipant and targetParticipant are either

- Occurrences (which might be lives) that are portions of the same life, or
- Data values that are equal.

##### General Types

BinaryLink

##### Features

myselfSameLife : Anything [1..\*] {redefines toSources}

The target end of a SelfLifeLink.

selfSameLife : Anything [1..\*] {redefines toTargets}

The source end of a SelfLifeLink.

sourceDataValue : DataValue [0..1] {subsets source}

Same as the sourceParticipant when it is a data value.

sourceOccurrence : Occurrence [0..1] {subsets source}

Same as the sourceParticipant when it is an occurrence.

targetDataValue : DataValue [0..1] {subsets target}

Same as the targetParticipant when it is a data value.

targetOccurrence : Occurrence [0..1] {subsets target}

Same as the targetParticipant when it is an occurrence.

##### Constraints

None.

#### 9.2.2.2.6 things

##### Element

Feature

##### Description

`things` is the most general Feature (M1 instance of M2 Feature). All other Features (in libraries or user models) specialize it (subset or redefine, directly or indirectly). It is typed by `Anything`.

`things` has multiplicity lower bound 1 because, for any featuring instance, it includes at least that instance as the value of `Anything::self`.

## General Types

`Anything`

## Features

`that` : `Anything`

For each value of `things`, the "featuring instance" of that value. Formally, for any sequence  $s$  classified by `things`, the `that` includes a sequence whose prefix is  $s$ , followed by the second-to-last element of  $s$ . This is enforced by declaring `Anything::self` to be the chaining of `things.that`, restricting `that` to the single value of `self` for all `things`.

## Constraints

None.

## 9.2.3 Occurrences

### 9.2.3.1 Occurrences Overview

#### Occurrences

This library adds a time and space model, starting with *Occurrence*, the most general Class (see [Clause](#)), which classifies *Anything* that takes up time and space, and *occurrences*, the most general Feature typed by Classes. *Occurrences* can take up the same or overlapping time and space when they represent different things happening or existing in it. For example, the time and space taken by a room might have air moving in it, as well as light, radio waves, and so on.

*Occurrences* divide into *Objects* and *Performances* (see [9.2.5.1](#) and [9.2.6.1](#), respectively), corresponding to Classes dividing into Structures and Behaviors (see [Clause](#) and [Clause](#), respectively). This subclause covers what is in common between *Objects* and *Performances*.

## Temporal and Spatial Associations

*Occurrences* can be completely separated in time or space, or both, as indicated by these specialized *Links*:

- *HappensBefore Links* between *Occurrences* indicate they are completely separate in time, with one happening or existing completely before another. The *predecessors* and *successors* of *Occurrences* are those that *HappenBefore* them and after them (those that they *HappenBefore*), respectively. *HappensJustBefore Links* are *HappensBefore Links* between *Occurrences* where there is no possibility of other *Occurrences* happening or existing in the time between them. The *immediatePredecessors* and *immediateSuccessors* of *Occurrences* are those that *HappenJustBefore* them and just after them (those that they *HappenJustBefore*), respectively. *Occurrences* separated in time are not necessarily separated in space.
- *OutsideOf Links* between *Occurrences* indicate they are completely separate in space, without specifying their relative positions (such as above or to the left). *JustOutsideOf Links* are *OutsideOf Links* between *Occurrences* where there is no possibility of other *Occurrences* happening or existing in the space

between at least some of their *spaceBoundaries*, see space boundaries below. *Occurrences* separated in space are not necessarily separated in time.

*Without Links* between *Occurrences* are provided as a convenience to indicate one *HappenBefore* another or is *OutsideOf* the other or both. This means they do not overlap at all in space-time.

*Occurrences* can completely overlap others in time or space, or both, as indicated by these specialized *Links*:

- *HappensDuring Links* between *Occurrences* indicate one happens or exists completely within the time taken by another, with the *timeEnclosedOccurrences* of an *Occurrence* being the ones that *HappenDuring* it. *Occurrences* overlapping in time do not necessarily overlap in space.
- *InsideOf Links* between *Occurrences* indicate one happens or exists completely within the space taken by another, with the *spaceEnclosedOccurrences* of an *Occurrence* being the ones that *InsideOf* it. *Occurrences* overlapping in space do not necessarily overlap in time.

*Within Links* between *Occurrences* are provided as a convenience to indicate one *HappensDuring* another and is *InsideOf* that other. This means one is completely overlapped by the other in space-time.

*Occurrences* cannot be linked by both *HappensBefore* and *HappensDuring*, *OutsideOf* and *InsideOf*, or *Within* and *Without*. They also cannot *HappenBefore* or be *OutsideOf* or *Without* themselves, but always *HappenDuring* and are *InsideOf* and *Within* themselves. When an *Occurrence* *HappensBefore* another, all *Occurrences* that *HappenDuring* the earlier one (including itself) also *HappenBefore* those that *HappenDuring* the later one (including itself).

*Occurrences* that *HappenDuring* each other both ways (circularly) happen or exist at the same time, which is provided for convenience by *HappensWhile*, a specialization of *HappenDuring*. *Occurrences* that are *InsideOf* each other both ways occupy exactly the same space, even though they might happen or exist at separate times. *Occurrences* that are *Within* each other both ways happen at exactly the same time and occupy exactly the same space, which is provided for convenience by *WithinBoth*, a specialization of *Within*.

The *Links* above do not take up time or space, they are temporal and spatial relations between things that do (they are disjoint with *LinkObject*, see [9.2.5.1](#)).

## Other Time-Space Relations

The time and space taken by an *Occurrence* can be related in three ways to the time and space taken by others, identified by the Features below. An *Occurrence* with values for these Features takes the same time and space as

- *unionOf*: taken by all the other *Occurrences* together.
- *intersectionOf*: is common to all the other *Occurrences*.
- *differencesOf*: the first other *Occurrence* that is not taken by the rest.

The values of the above Features are *Sets* of *Occurrences* to enable the time and space of an *Occurrence* to be specified in multiple ways, with each set taken as a complete specification of the time and space taken by the *Occurrence*.

## Portions

It is useful to consider *Occurrences* during only some of the time and space they take up, which are other *Occurrences* identified as *portions* (the most general portion Feature, see [Clause](#)). These are the same "thing" as their larger *Occurrences*, just considered for a potentially smaller period of time and region in space. They must be classified the same way as the *Occurrences* they are *portionsOf*, or more specialized.

*Occurrences* are always *portionsOf* themselves. *Occurrences* that are only *portionsOf* of themselves are *Lives* (classified by the library Class *Life*). *Lives* take up the entire time and space of a thing that happens or exists. *Occurrences* have the same *Life* as those they are *portionsOf*, identified by *portionOfLife*. This means following *portionsOf* repeatedly will always reach a single *Life*, even though some *Occurrences* along the way might be *portionsOf* of more than one other *Occurrence*.

*SelfSameLifeLinks* include *SelfLinks* (*Links* between each thing and itself, see [9.2.4.1](#)), as well as *Links* between *Occurrences* that are *portionsOf* the same *Life* (have the same *portionOfLife*).

## Time and Space Slices

Time slices are *portions* that include all the space of their larger *Occurrences* within a potentially smaller period of time than the whole *Occurrence*, identified as *timeSlices* of the *Occurrences* they are *portionsOf*. Time slices might have Feature values and *Links* to other things peculiar to their smaller period of time. *Occurrences* are always *timeSlicesOf* themselves. The *snapShots* of *Occurrences* are *timeSlices* that take no time. The earliest *snapShot* of an *Occurrence* is its *startShot*, the latest is its *endShot*. All the others happen during its *middleTimeSlice*. *Occurrences* with a *startShot* the same as their *endShot* take no time, have no *middleTimeSlice*, and vice-versa.

Space slices are *portions* that include all the space of their larger *Occurrences*, but not necessary all their time, identified as *spaceSlices* of the *Occurrences* they are *portionsOf*. Space slices might have Feature values and *Links* to other things peculiar to their smaller region in space. *Occurrences* are always *spaceSlicesOf* themselves. The *spaceShots* of *Occurrences* are *spaceSlices* that have a lower *innerSpaceDimension* than the *Occurrences* they are *spaceSlicesOf*, which is the number of variables needed to identify any space point occupied by an *Occurrence*, without regard to higher dimensional spaces in which it might be embedded. For example, the *innerSpaceDimension* of a *Curve* is 1 (see [9.2.5.1](#)), because points on it can be identified by the distance from one end, even if the curve bends in two or three dimensions. A *Curve* can be a *spaceShot* of a *Surface* or *Body*, which have *innerSpaceDimension* of 2 and 3, respectively. The *spaceSlices* of an *Occurrence* that are not *spaceShots* must have the same *innerSpaceDimension* as the *Occurrence*. How much an *Occurrence* bends in higher dimensions is its *outerSpaceDimension* (see [9.2.5.1](#)). For example, the *outerSpaceDimension* of a planar curve is 2 or 1 (*Line*), while it is 3 for non-planar.

## Space Boundaries and Interiors

The *spaceSlices* of each *Occurrence* are divided into a *spaceBoundary*, which is a *spaceShot*, and a *spaceInterior*, which is a *spaceSlice* that is not a *spaceShot* (has the same *innerSpaceDimension* as the *Occurrence*). They are *JustOutsideOf* each other and union (see below) to the entire *Occurrence*. Space boundaries cannot have a *spaceBoundary*, which means they also cannot have a *spaceInterior*, indicated by *isClosed=true*. For example, a ball has a sphere as its *spaceBoundary*, but the sphere *isClosed*.

A *spaceBoundary* might have *spaceSlices* that are also closed and have the same *innerSpaceDimension* as the *spaceBoundary* (not among its *spaceShots*). In some cases one of these *spaceSlices* surrounds the others, identified as the *outer*, a nested feature of *spaceBoundary*, and the others as the *inner* ones. This means the *outer* one can be taken as the *spaceBoundary* of another *Occurrence* with a *spaceInterior* that completely includes the *innners*. The *inner spaceBoundaries* can also be taken as *spaceBoundaries* of their own *Occurrences*, the *spaceInteriors* of which are identified as the *innerSpaceOccurrences* ("holes") of the *Occurrence* having the *spaceBoundary*. These two cases are covered by *SurroundedBy Links* between *Occurrences*.

*MatesWith Links* are *JustOutsideOf Links* between *Occurrences* indicating that they union (see below) to an *Occurrence* with a *spaceBoundary* but no *spaceInterior*. This means there is no possibility of other *Occurrences* happening or existing in the space between them. *JustOutsideOf Links* additionally include those

between *Occurrences* where only some of their *spaceSlices* (of their *spaceBoundaries*) are linked by *MatesWith*.

### 9.2.3.2 Elements

#### 9.2.3.2.1 Causation

##### Element

ConnectionDefinition

##### Description

A *Causation* is a binary *Multicausation* in which a single cause occurrence causes a single effect occurrence. (However, a single cause can separately have multiple effects, and a single effect can have separate *Causation* connections with multiple causes.)

##### General Types

Multicausation

##### Features

theCause : Occurrence [0..\*] {redefines causes}

The single causing occurrence.

theEffect : Occurrence [0..\*] {redefines effects}

The single effect occurrence resulting from the cause.

##### Constraints

None.

#### 9.2.3.2.2 causations

##### Element

ConnectionUsage

##### Description

*causations* is the base feature for *Causation* ConnectionUsages.

##### General Types

Causation  
multicausations

##### Features

[no name] : Occurrence

[no name] : Occurrence

## Constraints

None.

### 9.2.3.2.3 HappensBefore

#### Element

Association

#### Description

HappensBefore is a Withoutassociation linking an `earlierOccurrence` to a `laterOccurrence`, indicating that the Occurrences do not overlap in time (not necessarily in space, see `OutsideOf`; none of their `snapshots` happen at the same time), and the `earlierOccurrence` happens first. This means noOccurrence HappensBefore itself. Every Occurrence that HappensDuring the `earlierOccurrence` (including itself) also HappensBefore every Occurrence that HappensDuring the `laterOccurrence` (including itself).

#### General Types

HappensLink  
Without

#### Features

`earlierOccurrence` : Occurrence {redefines `separateOccurrenceToo`}

The participant that happens earlier than (before) the other participant.

`laterOccurrence` : Occurrence {redefines `separateOccurrence`}

The participant that happens later than (after) the other participant.

## Constraints

None.

### 9.2.3.2.4 happensBeforeLinks

#### Element

Feature

#### Description

`happensBeforeLinks` is a specialization of `binaryLinks` restricted to type `HappensBefore`. It is the most general Succession (M1 instance of M2 Succession). All other Successions (in libraries or user models) specialize it (directly or indirectly).

#### General Types

HappensBefore  
`binaryLinks`

#### Features

[no name] : Occurrence

[no name] : Occurrence

### Constraints

None.

## 9.2.3.2.5 HappensDuring

### Element

Association

### Description

HappensDuring links its `shorterOccurrence` to its `longerOccurrence`, indicating that the `shorterOccurrence` completely overlaps the `longerOccurrence` in time (not necessarily in space, see `InsideOf`; all snapshots of the `shorterOccurrence` happen at the same time as some snapshot of the `longerOccurrence`). This means every Occurrence HappensDuring itself and that HappensDuring is transitive. Every Occurrence that HappensBefore the `longerOccurrence` also HappensBefore the `shorterOccurrence`. The `shorterOccurrence` also HappensBefore every Occurrence that the `longerOccurrence` does.

### General Types

HappensLink

### Features

`happensDuring` : Occurrence [1..\*] {subsets happensTarget}

Occurrences that completely overlap this one in time (not necessarily in space, see `insideOf`; they start when this one does or earlier and end when this one does or later), including this one.

`longerOccurrence` : Occurrence {redefines targetOccurrence}

The participant in this HappensDuring Link that takes up more (or equal) time than the other.

`shorterOccurrence` : Occurrence {redefines sourceOccurrence}

The participant in this HappensDuring Link that takes up less (or equal) time than the other.

### Constraints

None.

## 9.2.3.2.6 HappensJustBefore

### Element

Association

### Description



HappensJustBefore is HappensBefore asserting that there is no possibility of other Occurrences happening in the time between the `earlierOccurrence` and `laterOccurrence`.

### General Types

HappensBefore

### Features

None.

### Constraints

None.

## 9.2.3.2.7 HappensLink

### Element

Association

### Description

### General Types

BinaryLink

### Features

`happensSource` : Occurrence [0..\*] {subsets toSources}

`happensTarget` : Occurrence [0..\*] {subsets toTargets}

`sourceOccurrence` : Occurrence {redefines source}

`targetOccurrence` : Occurrence {redefines target}

### Constraints

None.

## 9.2.3.2.8 HappensWhile

### Element

Association

### Description

HappensWhile is a HappensDuring and its inverse. This means the linked Occurrences completely overlap each other in time (they happen at the same time) all `snapshots` of each Occurrence happen at the same time as one of the `snapshots` of other. This means every Occurrence HappensWhile itself and that HappensWhile is transitive.

### General Types

HappensDuring

### Features

happensWhile : Occurrence [1..\*] {subsets happensDuring}

Occurrences that start and end at the same time as this one.

happensWhile<sup>1</sup> : Occurrence [1..\*] {subsets timeEnclosedOccurrences}

Occurrences that happenWhile this one does (Occurrences that start and end at the same time as this one).

thatOccurrence : Occurrence {redefines longerOccurrence}

thisOccurrence : Occurrence {redefines shorterOccurrence}

### Constraints

None.

#### 9.2.3.2.9 InnerSpaceOf

##### Element

##### Description

##### General Types

None.

### Features

innerSpace : Occurrence

innerSpaceOccurrenceOf : Occurrence {subsets separateSpaceToo}

outerSpace : Occurrence

### Constraints

None.

#### 9.2.3.2.10 InsideOf

##### Element

Association

##### Description

InsideOf is a BinaryLink between its `smallerSpace` and `largerSpace`, indicating that the `largerSpace` completely overlaps the `smallerSpace` in space (not necessarily in time, see `HappensDuring`; all four dimensional points of the `smallerSpace` are in the spatial extent of the `largerSpace`). This means every Occurrence/ is InsideOf itself and that InsideOf is transitive.

## General Types

BinaryLink

## Features

insideOf : Occurrence [1..\*] {subsets toTargets}

Occurrences that completely overlap this one in space (not necessarily in time, see happensDuring), including this one.

largerSpace : Occurrence {redefines target}

The participant in this InsideOf Link that takes up more (or equal) space than the other.

smallerSpace : Occurrence {redefines source}

The participant in this InsideOf Link that takes up less (or equal) space than the other.

## Constraints

None.

### 9.2.3.2.11 JustOutsideOf

#### Element

#### Description

#### General Types

None.

#### Features

justOutsideOf : Occurrence [0..\*] {subsets outsideOf}

justOutsideOfToo : Occurrence [0..\*] {subsets outsideOfToo}

#### Constraints

None.

### 9.2.3.2.12 Life

#### Element

Class

#### Description

Life is the class of Occurrences that are "maximal portions". That is, they are only portions of themselves.

#### General Types

Occurrence

### **Features**

portion : Occurrence [1..\*]

Occurrences that are portions of this Life, including at least this Life.

### **Constraints**

None.

#### **9.2.3.2.13 MatesWith**

### **Element**

### **Description**

### **General Types**

None.

### **Features**

matesWith : Occurrence [0..\*] {subsets justOutsideOf}

### **Constraints**

None.

#### **9.2.3.2.14 Occurrence**

### **Element**

Class

### **Description**

An Occurrence is Anything that happens over time and space (the four physical dimensions). Occurrences can be portions of another Occurrence within time and space, including slices in time, leading to snapshots that take zero time.

### **General Types**

Anything

### **Features**

difference : Occurrence [0..1]

A (nested) feature of `differencesOf` identifying an Occurrence that is the intersectionsOf of the Occurrences identified by `interdiff` (`minuend` and `interdiff.notSubtrahend`).

differencesOf : OrderedSet [0..\*]

Ordered sets of Occurrences, where the time and space taken by first Occurrence in each set (*minuend*) that is not in the time and space taken by the remaining Occurrences (*subtrahend*, resulting in *difference*) is the same as taken by this Occurrence (all four dimensional points in the *minuend* that are not in any *subtrahend* are at the same time and space as those in this Occurrence).

elements : Occurrence [0..\*]

A nested feature of *unionsOf*, *intersectionsOf*, and *differencesOf* for the elements of each of their (Ordered)Sets separately.

endShot : Occurrence {subsets snapshots}

The snapshot of this Occurrence that happensAfter all its other snapshots.

immediatePredecessors : Occurrence [0..\*] {subsets predecessors}

*Occurrences* that *HappensJustBefore* this one (*Occurrences* that *HappensBefore* this one, with no possibility of other *Occurrences* happening in the time between them).

immediateSuccessors : Occurrence [0..\*] {subsets successors}

*Occurrences* that this one *HappensJustBefore* (*Occurrences* that this one *HappensBefore*, with no possibility of other *Occurrences* happening in the time between them).

incomingTransfer : Transfer [0..\*]

incomingTransferToSelf : Transfer [0..\*] {subsets incomingTransfer}

Transfers for which this Occurrence is the *targetParticipant*.

inner : Occurrence [0..\*]

A *spaceSlice* of *spaceBoundary*, see *spaceBoundary*.

innerSpaceDimension : Natural

The number of variables needed to identify space points in this Occurrence, from 0 to 3, without regard to higher dimensional spaces it might be emedded in. For example, the *innerSpaceDimension* of a curve is 1, even if it twists in three dimensions, see *outerSpaceDimension*.

innerSpaceOccurrences : Occurrence {subsets separateSpace}

Occurrences that completely occupy the space SurroundBy an *inner spaceBoundary* of this Occurrence.

interdiff : Set [0..\*]

A (nested) feature of *differencesOf* identifying a set that includes its *minuend* and all Occurrences that are not in its *subtrahend*.

intersection : Occurrence [0..1]

A (nested) feature of *intersectionsOf* identifying an Occurrence that a) is completely within (the space and time of) all *intersectionsOf* elements, and b) satisfies the conditions of the same element's *nonIntersection*.

`intersectionsOf` : Set [0..\*]

Sets of Occurrences, where the time and space taken in common between the Occurrences in each set (`intersectionsOf::intersection`) is at the same as taken by this Occurrence (all four dimensional points common to the Occurrences in each set are at the same time and space as those in this Occurrence).

`/isClosed` : Boolean

True if this Occurrence has a `spaceBoundary`, false otherwise.

`localClock` : Clock

A local *Clock* to be used as the corresponding time reference for this *Occurrence* and, by default, all *ownedOccurrences*. By default this is the singleton `Clocks::universalClock`.

`matingOccurrences` : Occurrence [0..\*] {subsets justOutsideOfToo}

Occurrences that have no space between them and this one.

`middleTimeSlice` : Occurrence [0..1] {subsets timeSlices}

`timeSlice` of this Occurrence that takes all of the time between its `startShot` and `endShot`. Occurrences do not have `middleTimeSlice` if their `startShot` is the same as their `endShot` (such as being a `snapShot` of another Occurrence), otherwise they do.

`minuend` : Occurrence [0..1] {subsets }

A (nested) feature of `differencesOf` that identifies the first Occurrence in its `elements`.

`nonIntersection` : Occurrence [0..\*] {subsets spaceTimeEnclosedPoints}

A nested feature of `intersectionsOf.elements` identifying all the `spaceTimeEnclosedPoints` of each `element` that are not identified by `intersection`. These must be without (separate in space or time from) at least one other `element`.

`notSubtrahend` : Occurrence [0..\*]

A (nested) feature of `differencesOf.interdiff` identifying all Occurrences that are not identified by the `subtrahend` in each value `differencesOf` separately.

`outer` : Occurrence [0..1]

A `spaceSlice` of `spaceBoundary`, see `spaceBoundary`.

`outerSpaceDimension` : Natural [0..1]

For Occurrences of `innerSpaceDimension` 1 or 2, the number of variables needed to identify their space points in higher dimensional spaces they might be embedded in, from the `innerSpaceDimension` to 3. For example , an `outerSpaceDimension` 3 for a curve indicates it twists in three dimensions. An `outerSpaceDimension` equal to `innerSpaceDimension` indicates the occurrence is spatially straight (`innerSpaceDimension` 1 embedded in 2 or 3 dimensions) or flat (`innerSpaceDimension` 2 embedded in 3 dimensions).

`outgoingTransfer` : Transfer [0..\*]

outgoingTransferFromSelf : Transfer [0..\*] {subsets outgoingTransfer}

Transfers for which this Occurrence is the sourceParticipant.

portion : Occurrence [1..\*] {subsets spaceTimeEnclosedOccurrences}

All occurrences within this one that are considered the same thing occurring (same portionOfLife), including this one.

portionOf : Occurrence [1..\*] {subsets within}

All occurrences that this one is within that are considered the same thing occurring (same portionOfLife), including this one.

portionOfLife : Life

The Life of which this Occurrence is a portion.

predecessors : Occurrence [0..\*] {subsets withoutToo}

Occurrences that are completely separate from this one in time (not necessarily in space, see outsideOfToo) and that happen before this one (end earlier than this one starts).

self : Occurrence {subsets timeSlices, spaceSlices, redefines self}

This Occurrence (related to itself via a SelfLink).

snapshotOf : Occurrence [0..\*] {subsets timeSliceOf}

Occurrences of which this Occurrence is a snapshot.

snapshots : Occurrence [1..\*] {subsets timeSlices}

All timeSlices of this Occurrence that happen at a single instant of time (zero duration).

spaceBoundary : Occurrence [0..1] {subsets spaceShots}

A spaceShot of this Occurrence that is not among those of its spaceInterior, which it must be OutsideOf. It must not have a spaceBoundary (isClosed = true). It can be divided into spaceSlices that also have no spaceBoundary, where the inner ones are SurroundedBy the outer one.

spaceEnclosedOccurrences : Occurrence [1..\*] {subsets toSources}

Occurrences that this one completely overlaps in space (not necessarily in time, see timeEnclosedOccurrences), including this one.

spaceInterior : Occurrence [0..1] {subsets spaceSlices}

A spaceSlice of this Occurrence that includes all its spaceShots except the spaceBoundary, which must exist and be outsideOf it. The spaceInterior must be of the same innerSpaceDimension as this Occurrence, except if it is zero, whereupon there is no spaceInterior.

spaceShots : Occurrence [1..\*] {subsets spaceSlices}

All spaceSlices of this Occurrence that are of a lower innerSpaceDimension than it.

`spaceSliceOf : Occurrence [1..*] {subsets portionOf}`

An Occurrence this one is a `spaceSlices` of.

`spaceSlices : Occurrence [1..*] {subsets portion}`

All `portions` of this Occurrence that extend for exactly the same time and some or all the space, relative to spatial location of this Occurrence. This means every Occurrence is a `spaceSlice` of itself.

`spaceTimeCoincidentOccurrences : Occurrence [1..*] {subsets spaceTimeEnclosedOccurrences}`

Occurrences that this one completely includes in both space and time, including this one.

`spaceTimeEnclosedOccurrences : Occurrence [1..*] {subsets spaceEnclosedOccurrences, timeEnclosedOccurrences}`

All `timeEnclosedOccurrences` of this one that are `insideOf` it, including itself.

`spaceTimeEnclosedPoints : Occurrence [1..*] {subsets spaceTimeEnclosedOccurrences}`

All `spaceTimeEnclosedOccurrences` of this one that take up no time or space (`innerSpaceDimension 0` and `startShot` the same as `endShot`).

`startShot : Occurrence {subsets snapshots}`

The `snapshot` of this Occurrence that `happensBefore` all its other `snapshots`.

`suboccurrences : Occurrence [0..*]`

Composite *suboccurrences* of this Occurrence. The *localClock* of all *suboccurrences* defaults to the *localClock* of its containing Occurrence.

`subtrahend : Occurrence [0..*] {subsets }`

A (nested) feature of `differencesOf` that identifies all the Occurrences in its `elements` except the first one.

`successors : Occurrence [0..*] {subsets without}`

Occurrences that are completely separate from this one in time (not necessarily in space, see `outsideOf`) and that happen after this one (start later than this one ends).

`this : Occurrence`

The "context" Occurrence within which this Occurrence takes place. By default, it is this Occurrence itself. However, this is overridden for `ownedPerformances` of Objects and `subperformances` of Performances.

`timeEnclosedOccurrences : Occurrence [1..*] {subsets happensSource}`

Occurrences that this one completely overlaps in time (not necessarily in space, see `inside`; they start at the same time or later and end at the same time or earlier), including this one.

`timeSliceOf : Occurrence [1..*] {subsets portionOf}`

Occurrences of which this one is a `timeSlice`, including this one.



timeSlices : Occurrence [1..\*] {subsets portion}

portions that extend for some or all the time of this Occurrence, but all its space during that time, including itself.

union : Occurrence [0..1]

A (nested) feature of unionsOf identifying an Occurrence with a) spaceTimeEnclosedOccurrences including all those identified by a unionsOf element, and b) all the Occurrence's spaceTimeEnclosedPoints within (the space and time of) at least one of the elements.

unionsOf : Set [0..\*]

Sets of Occurrences, where the time and space taken by all the Occurrences in each set together (unionsOf::union) is the same as taken by this Occurrence (all four dimensional points in the Occurrences of each set are at the same time and space as those of this Occurrence).

### Constraints

None.

### 9.2.3.2.15 occurrences

#### Element

Feature

#### Description

occurrences is a specialization of things restricted to type Occurrence. It is the most general feature typed by Occurrence. All other Features typed by Occurrence or its specializations (in libraries or user models) specialize it (directly or indirectly).

#### General Types

things  
Occurrence

#### Features

None.

#### Constraints

None.

### 9.2.3.2.16 OutsideOf

#### Element

Association

#### Description

OutsideOf is a Without association linking its `separateSpaceToo` and its `separateOccurrence`, indicating that these Occurrences do not overlap in space (not necessarily in time, see `HappensBefore`; no four dimensional points of the Occurrences are in the spatial extent of both of them). This means no Occurrence is OutsideOf itself.

### General Types

Without

### Features

`outsideOf` : Occurrence [0..\*] {subsets without}

Occurrences that are completely separate from this one in space (not necessarily in time, see `successors`).

`outsideOfToo` : Occurrence [0..\*] {subsets withoutToo}

Occurrences that are completely separate from this one in space (not necessarily in time, see `predecessors`).

`separateSpace` : Occurrence {redefines `separateOccurrence`}

The second participant in this OutsideOf Link.

`separateSpaceToo` : Occurrence {redefines `separateOccurrenceToo`}

The first participant in this OutsideOf Link.

### Constraints

None.

## 9.2.3.2.17 PortionOf

### Element

Association

### Description

PortionOf is a Within that links its `portionOccurrence` to its `portionedOccurrence`, indicating they are considered the same thing occurring (same `portionOfLife`), but with the `portionOccurrence` potentially taking up less time and space than the `portionedOccurrence`. This means every Occurrence/ is a PortionOf itself. The `innerSpaceDimension` of `portionOccurrence` is the same or lower than of the `portionedOccurrence`.

### General Types

Within

### Features

`portionedOccurrence` : Occurrence {redefines `largerOccurrence`}

The participant in this PortionOf Link that is the `largerOccurrence`.

`portionOccurrence` : Occurrence {redefines `smallerOccurrence`}

The participant in this PortionOf Link that is the `smallerOccurrence`.

#### Constraints

None.

#### 9.2.3.2.18 SnapshotOf

##### Element

Association

##### Description

SnapshotOf is a TimeSliceOf that links its `snapshotOccurrence` to its `snapshottedOccurrence`, indicating that `snapshotOccurrence` takes not time (`startShot` and `endShot` are the same).

##### General Types

TimeSliceOf

##### Features

`snapshotOccurrence` : Occurrence {redefines `timeSliceOccurrence`}

The participant in this SnapshotOf Link that is the `timeSliceOccurrence`.

`snapshottedOccurrence` : Occurrence {redefines `timeSlicedOccurrence`}

The participant in this SnapshotOf Link that is the `timeSlicedOccurrence`.

#### Constraints

None.

#### 9.2.3.2.19 SpaceShotOf

##### Element

Association

##### Description

SpaceShotOf is a SpaceSliceOf that links its `spaceShotOccurrence` to its `spaceSnapshottedOccurrence`, indicating the `spaceShotOccurrence` is of a lower `innerSpaceDimension` than the `spaceShottedOccurrence`.

##### General Types

SpaceSliceOf

##### Features

`spaceShotOccurrence` : Occurrence {redefines `spaceSliceOccurrence`}

The participant in this SpaceShotOf Link that is the `spaceSliceOccurrence`.

`spaceShotOf` : Occurrence [1..\*] {subsets `spaceSliceOf`}

All `spaceSlicesOf` this Occurrence that are of a higher `innerSpaceDimension` than this Occurrence.

`spaceShottedOccurrence` : Occurrence {redefines `spaceSlicedOccurrence`}

The participant in this SpaceShotOf Link that is the `spaceSliced` Occurrence.

### Constraints

None.

### 9.2.3.2.20 SpaceSliceOf

#### Element

Association

#### Description

`SpaceSliceOf` is a `PortionOf` that links its `spaceSliceOccurrence` to its `spaceSlicedOccurrence`, indicating the `spaceSliceOccurrence` extends for exactly the same time and some or all the space of the `spaceSlicedOccurrence` and that the `spaceSliceOccurrence` is of the same or lower `innerSpaceDimension` than the `spaceSlicedOccurrence`. This means every Occurrence/ is a `SpaceSliceOf` itself and `SpaceSliceOf` is transitive.

#### General Types

`PortionOf`

#### Features

`spaceSlicedOccurrence` : Occurrence {redefines `portionedOccurrence`}

The participant in this `SpaceSliceOf` Link that is the `portionedOccurrence`.

`spaceSliceOccurrence` : Occurrence {redefines `portionOccurrence`}

The participant in this `SpaceSliceOf` Link that is the `portionOccurrence`.

### Constraints

None.

### 9.2.3.2.21 SurroundedBy

#### Element

#### Description

#### General Types

None.

## Features

surroundedBy : Occurrence [0..\*] {subsets outsideOfToo}

surroundedSpace : Occurrence

surroundingSpace : Occurrence

surrounds : Occurrence [0..\*] {subsets outsideOf}

## Constraints

None.

### 9.2.3.2.22 TimeSliceOf

#### Element

Association

#### Description

TimeSliceOf is a PortionOf that links its `timeSliceOccurrence` to its `timeSlicedOccurrence`, indicating that extend for exactly the same time and some or all the space of this Occurrence, including itself. This means every Occurrence/ is a PortionOf itself.

#### General Types

PortionOf

#### Features

timeSlicedOccurrence : Occurrence {redefines portionedOccurrence}

The participant in this TimeSliceOf Link that is the `portionedOccurrence`.

timeSliceOccurrence : Occurrence {redefines portionOccurrence}

The participant in this TimeSliceOf Link that is the `portionOccurrence`.

#### Constraints

None.

### 9.2.3.2.23 Within

#### Element

Association

#### Description

Within classifies all and only links that are HappensDuring and InsideOf. They link their `largerOccurrence` to their `smallerOccurrence`, indicating the `largerOccurrence` completely overlaps the `smallerOccurrence` in

time and space (all four dimensional points of the `smallerOccurrence` HappensDuring and are InsideOf the `largerOccurrence`). This means every Occurrence is Within itself and Within is transitive.

### General Types

HappensDuring  
InsideOf

### Features

`largerOccurrence` : Occurrence {redefines `largerSpace`, `longerOccurrence`}

The participant in this Within Link that is the `longerOccurrence` and `largerSpace`.

`smallerOccurrence` : Occurrence {redefines `shorterOccurrence`, `smallerSpace`}

The participant in this Within Link that is the `shorterOccurrence` and `smallerSpace`.

`within` : Occurrence [1..\*] {subsets `insideOf`, `happensDuring`}

All Occurrences that this one happensDuring and is insideOf, including this one.

### Constraints

None.

#### 9.2.3.2.24 WithinBoth

### Element

Association

### Description

WithinEachOther is a Within and its inverse. This means the linked Occurrences completely overlap each other in space and time ( they occupy the same four dimensional region). This means every Occurrence is WithinEachOther with itself and WithinEachOther is transitive.

### General Types

Within

### Features

`thatOccurrence` : Occurrence {redefines `largerOccurrence`}

`thisOccurrence` : Occurrence {redefines `smallerOccurrence`}

`withinBoth` : Occurrence [1..\*] {subsets `within`}

### Constraints

None.

#### 9.2.3.2.25 Without

##### Element

Association

##### Description

Without classifies all links that are HappensDuring or InsideOf, or both. They link their `separateOccurrenceToo` to their `separateOccurrence`, indicating that the Occurrences do not overlap in time or space (no four dimensional point is in both Occurrences). This means no Occurrence is Without itself.

##### General Types

BinaryLink

##### Features

`separateOccurrence` : Occurrence {redefines target}

The second participant in this Without Link.

`separateOccurrenceToo` : Occurrence {redefines source}

The first participant in this Without Link.

`without` : Occurrence [0..\*] {subsets toTargets}

All Occurrences that are `successors` of this one and are `outsideOf` it.

`withoutToo` : Occurrence [0..\*] {subsets toSources}

All Occurrences that are `predecessors` of this one and are `outsideOfToo` it.

##### Constraints

None.

## 9.2.4 Links

### 9.2.4.1 Links Overview

This library model introduces the most general Association *Link*, the type of *links*, the most general Feature typed by Associations (see [Clause](#)). The *participant* Feature of *Link* is the most general *associationEnd*, identifying the things being linked by (at the "ends" of) each *Link* (exactly one thing per end, which might be the same things). *Link* is specialized into *BinaryLink*, the most general Association with exactly two *associationEnds*, *source* and *target*, which subset *participant* and identify the two things linked, which might be the same thing. *BinaryLink* is the type of *binaryLinks*, the most general Feature typed by binary Associations. They are specialized into *SelfLink* and *selfLinks*, respectively, for links that have the same thing on both ends, identified by *thisThing* and *thatThing*, redefining *source* and *target*, respectively. These are used by BindingConnectors to specify that Features have the same values (see [Clause](#)). *SelfLinks* are not in time or space (they are not Occurrences, see [9.2.5.1](#)).

## 9.2.4.2 Elements

### 9.2.4.2.1 BinaryLink

#### Element

Association

#### Description

BinaryLink is a Link with exactly two participant Features ("binary" Association). All other binary associations (in libraries or user models) specialize it (directly or indirectly).

#### General Types

Link

#### Features

participant : Anything {redefines participant, ordered, nonunique}

The participants of this BinaryLink, which are restricted to be exactly two.

source : Anything {subsets participant}

The participant that is the source of this BinaryLink.

target : Anything {subsets participant}

The participant that is the target of this BinaryLink.

toSources : Anything [0..\*]

The end Feature of this BinaryLink corresponding to the sourceParticipant.

toTargets : Anything [0..\*]

The end Feature of this BinaryLink corresponding to the targetParticipant.

#### Constraints

None.

### 9.2.4.2.2 binaryLinks

#### Element

Feature

#### Description

binaryLinks is a specialization of links restricted to type BinaryLink. All other Features typed by BinaryLink or its specializations (in libraries or user models) specialize it (directly or indirectly).

#### General Types



links  
BinaryLink

### Features

[no name] : Anything

[no name] : Anything

### Constraints

None.

## 9.2.4.2.3 Link

### Element

Association

### Description

Link is the most general Association (M1 instance of M2 Association). All other Associations (in libraries or user models) specialize it (directly or indirectly). Specializations of Link are domains of Features subsetting Link::participants, exactly as many as associationEnds of the Association classifying it, each with multiplicity 1. Values of Link::participants on specialized Links must be a value of at least one of its subsetting Features.

### General Types

Anything

### Features

participant : Anything [2..\*] {ordered, nonunique}

The participants that are associated by this Link.

### Constraints

None.

## 9.2.4.2.4 links

### Element

Feature

### Description

links is a specialization of things restricted to type Link. It is the most general feature typed by Link. All other Features typed by Link or its specializations (in libraries or user models) specialize it (directly or indirectly).

### General Types

Link  
things

## Features

None.

## Constraints

None.

### 9.2.4.2.5 SelfLink

#### Element

Association

#### Description

SelfLink is a BinaryLink where the sourceParticipant and targetParticipant are the same. All other BinaryLinks where this is the case specialize it (directly or indirectly).

#### General Types

BinaryLink  
SelfSameLifeLink

## Features

myself : Anything {subsets myselfSameLife}

The target end of a SelfLink.

sameThing : Anything {subsets thisThing, redefines source}

The source participant of this SelfLink, which must be the same as the target participant.

thisThing : Anything {subsets sameThing, redefines target}

The target participant of this SelfLink, which must be the same as the source participant.

## Constraints

None.

### 9.2.4.2.6 selfLinks

#### Element

Feature

#### Description

selfLinks is a specialization of binaryLinks restricted to type SelfLink. It is the most general BindingConnector. All other BindingConnectors (in libraries or user models) specialize it (directly or indirectly).

#### General Types

SelfLink  
binaryLinks

## Features

[no name] : Anything

[no name] : Anything

## Constraints

None.

## 9.2.5 Objects

### 9.2.5.1 Objects Overview

*Objects* are *Occurrences* that take up a single region of time and space, even though they might be in multiple places over time. *Object* is the most general Structure, while *objects* is the most general Feature typed by Structures (see [Clause](#) and compare to *Performances* in [9.2.6.1](#)). *Objects* and *Performances* do not overlap, but *Performances* can Involve *Objects*, which can *Perform Performances* (see [9.2.6.1](#)).

*LinkObjects* are *Objects* that are also *Links*, and *linkObjects* is the most general Feature typed by *LinkObject*. *LinkObjects* occupy time and space, like other *Objects*, with potentially varying relationships to other things over time, except for which things are its *participants* (the things being linked), identified by its *associationEnd* Features (the "ends" of a link are permanent, though *participants* can be *Occurrences* with changing relationships to other things). The values of *LinkObject* Features that are not *associationEnds* can change over time. *LinkObjects* can exist between the same *Occurrences* for only some of the time those *Occurrences* exist, reflecting changing relationships of those *Occurrences*. *BinaryLinkObjects* are *BinaryLinks* that are also *LinkObjects*, and *binaryLinkObjects* is the most general Feature typed by *BinaryLinkObject*.

*Body(s)*, *Surfaces*, *Curves*, and *Points* are *Objects* with *innerSpaceDimension* of 3, 2, 1, and 0, respectively.

## Structured Space Objects

*StructuredSpaceObjects* are *Objects* with three Features Subsetting *spaceSlices*:

- *faces*, identifying *Surfaces*.
- *edges*, identifying *Curves*.
- *vertices*, identifying *Points*.

The above are collectively *structuredSpaceCells*, which are also *StructuredSpaceObjects*, enabling *faces* to identify *edges* and *vertices* among the *spaceSlices* of their *spaceBoundaries*, if any, and *edges* to identify *vertices* among theirs. Cells of closed *StructuredSpaceObjects* (*isClosed*=true) must be *JustOutside* others along their entire *spaceBoundary* (every cell's *spaceSlices* must *MateWith* some *spaceSlice* of another cell, see Space Boundaries and Interiors in [9.2.3.1](#)), which usually means all the *edges* and *vertices* of cells *MateWith* those of other cells, enabling the *StructuredSpaceObject* to be the *spaceBoundary* for other *Objects*. The *innerSpaceDimension* of a *StructuredSpaceObject* is the highest *innerSpaceDimension* of its *structuredSpaceCells*.

Models can specialize the three Features above for various kinds of *Objects*, for example, one for cylinders would include:

- Three Features Subsetting *faces* for the top, bottom, and middle *Surfaces* of a cylinder. The *edges* of these Features are *Curves* (circles) that are *spaceBoundaries* of the top and bottom *Surfaces* (discs), and *spaceSlices* of the *spaceBoundary* of the middle *Surface* (a rectangle joined at two opposite sides).
- Two Features Subsetting *edges* for the top and bottom of the cylinder. Each Feature identifies two *Curves* that are the *edges* of adjacent *faces*, specified by *BindingConnectors* between the Feature and required *edges*. These two *Curves* must mate, specified by a *MateWith* Connector between the Feature and itself.
- A Feature redefining *vertices* to multiplicity 0.

## 9.2.5.2 Elements

### 9.2.5.2.1 BinaryLinkObject

#### Element

AssociationStructure

#### Description

#### General Types

LinkObject

BinaryLink

#### Features

source : Anything [0..\*]

target : Anything [0..\*]

#### Constraints

None.

### 9.2.5.2.2 binaryLinkObjects

#### Element

Feature

#### Description

#### General Types

linkObjects

BinaryLinkObject

binaryLinks

#### Features

[no name] : Anything

[no name] : Anything

### **Constraints**

None.

### **9.2.5.2.3 Body**

#### **Element**

Structure

#### **Description**

Objects of `innerSpaceDimension 3`.

#### **General Types**

Object

#### **Features**

`innerSpaceDimension : Integer {redefines innerSpaceDimension}`

`volume`

### **Constraints**

None.

### **9.2.5.2.4 Curve**

#### **Element**

Structure

#### **Description**

Objects of `innerSpaceDimension 1`.

#### **General Types**

Object

#### **Features**

`innerSpaceDimension : Integer {redefines innerSpaceDimension}`

### **Constraints**

None.

### **9.2.5.2.5 LinkObject**

#### **Element**

AssociationStructure

### **Description**

LinkObject is the most general AssociationStructure (M1 instance of M2 AssociationStructure). All other AssociationStructures (in libraries or user models) specialize it (directly or indirectly).

### **General Types**

Object  
Link

### **Features**

None.

### **Constraints**

None.

## **9.2.5.2.6 linkObjects**

### **Element**

Feature

### **Description**

linkObjects is a specialization of links and objects restricted to type LinkObject. It is the most general feature typed by LinkObject. All other Features typed by LinkObject or its specializations (in libraries or user models) specialize it (directly or indirectly).

### **General Types**

LinkObject  
links  
objects

### **Features**

None.

### **Constraints**

None.

## **9.2.5.2.7 Object**

### **Element**

Structure

### **Description**

An *Object* is an *Occurrence* that is not a *Performance*. It is the most general *Structure*. All other *Structures* specialize it directly or indirectly.

## General Types

Occurrence

## Features

enactedPerformances : Performance [0..\*] {subsets timeEnclosedOccurrences, involvingPerformances}

Performances that are enacted by this object.

involvingPerformances : Performance [0..\*]

Performances in which this Object is involved.

ownedPerformances : Performance [0..\*] {subsets timeEnclosedOccurrences, involvingPerformances, suboccurrences}

*Performances* that are owned by this *Object*. The owning *Object* is the default *this* reference for all *ownedPerformances*.

structuredSpaceBoundary : StructuredSpaceObject [0..1] {subsets spaceBoundary}

A *spaceBoundary* that is a *StructuredSpaceObject*.

subobjects : Object [0..\*] {subsets suboccurrences}

The *suboccurrences* of this *Object* that are also *Objects*.

## Constraints

None.

### 9.2.5.2.8 objects

## Element

Feature

## Description

*objects* is a specialization of *occurrences* restricted to type *Object*. It is the most general feature typed by *Object*. All other Features typed by *Object* or its specializations (in libraries or user models) specialize it (directly or indirectly).

## General Types

occurrences

Object

## Features

None.

## Constraints

None.

#### 9.2.5.2.9 Point

##### Element

Structure

##### Description

Objects of `innerSpaceDimension 0`.

##### General Types

Object

##### Features

`innerSpaceDimension` : Integer {redefines `innerSpaceDimension`}

##### Constraints

None.

#### 9.2.5.2.10 StructuredSpaceObject

##### Element

Structure

##### Description

Objects that are broken up into smaller `structuredSpaceCells` of the same or lower `innerSpaceDimension`: faces of `innerSpaceDimension 2`, edges of `innerSpaceDimension 1`, and vertices of `innerSpaceDimension 0`, with the highest of these being the `innerSpaceDimension` of the `StructuredSpaceObject`. Boundaries of `structuredSpaceObjectCells` are the union of others of lower `innerSpaceDimension` (edges and vertices on the boundary of faces, and vertices on the boundary of edges), some of which meet when this `StructuredSpaceObject` is `isClosed` (faces meet at their edges and/or vertices, while edges meet at their vertices), as required to be a `spaceBoundary` of an `Object` .

##### General Types

Object

##### Features

`cellOrientation` : Integer [0..1]

A nested feature of `structuredSpaceObjectCell` that gives them a "direction" (1 or -1) or none (0). For example, the `cellOrientation` of a face indicates to which side the "positive" normal vector points, of an edge the positive direction along the edge, and of a vertex the positive direction "in or out" of it. When the `cellOrientation` of all edges and vertices are given, and the `StructuredSpaceObject` is `isClosed`, the `cellOrientations` of the (completely) overlapping ones sum to zero.

`edges` : Curve [0..\*] {subsets `structuredSpaceObjectCells`, ordered}



The `structuredSpaceObjectCells` of `innerSpaceDimension 1` in this `StructuredSpaceObject`.

`faces : Surface [0..*]` {subsets `structuredSpaceObjectCells`, ordered}

The `structuredSpaceObjectCells` of `innerSpaceDimension 2` in this `StructuredSpaceObject`.

`/innerSpaceDimension : Integer` {redefines `innerSpaceDimension`}

Highest `innerSpaceDimension` of the `structuredSpaceObjectCells`.

`structuredSpaceObjectCells : StructuredSpaceObject [1..*]` {subsets `spaceSlices`}

All and only the `spaceSlices` of this `StructuredSpaceObject` that are its `faces`, `edges`, and `vertices`.

`vertices : Point [0..*]` {subsets `structuredSpaceObjectCells`, ordered}

The `structuredSpaceObjectCells` of `innerSpaceDimension 0` in this `StructuredSpaceObject`.

### Constraints

None.

#### 9.2.5.2.11 Surface

##### Element

Structure

##### Description

Objects of `innerSpaceDimension 2`.

##### General Types

Object

##### Features

`genus : Integer [0..1]`

The number of "holes" in this `Surface`, assuming it `isClosed`. For example, it is 0 for spheres and 1 for toruses, including one-handed coffee cups.

`innerSpaceDimension : Integer` {redefines `innerSpaceDimension`}

### Constraints

None.

## 9.2.6 Performances

## 9.2.6.1 Performances Overview

### Performances

*Performances* are *Occurrences* that can be spread out in disconnected portions of space and time. *Performance* is the most general Behavior, while *performances* is the most general Feature typed by Behaviors (see [Clause](#) and compare to *Objects* in [9.2.5.1](#)). *Performances* can coordinate others that *HappenDuring* them, identified as their *subperformances* (see Steps in [Clause](#)). *Performances* also coordinate and potentially affect other things, some of which might come into existence (start, be "created") or cease to exist (end, be "destroyed") during a Performance, and some that might be used without being affected at all ("catalysts"). Some of these other things might be *Objects*, identified as a *Performance's* *involvedObjects*, some of which might be "responsible" for (enact, *Perform*) a *Performance*, identified as its *performers*. *Performances* can also accept things as input or provide them as output (see Parameters paragraph in [Clause](#)).

### Evaluations

*Evaluations* are *Performances* that produce at most one thing (value) identified by their *result* parameter. *Evaluation* is the most general Function, while *evaluations* is the most general Feature identifying them, typed by Functions (see [Clause](#)). In other respects *Evaluations* are like any other *Performance*.

*LiteralEvaluations* are *Evaluations* with exactly one *result*, specified as a constant in a model via classification by *LiteralExpression* (see [Clause](#) for this and the rest of the paragraph). *LiteralEvaluation* is the most general *LiteralExpression*, specialized in the same way, and *literalEvaluations* is the most general feature identifying them, also similarly specialized.

*BooleanEvaluations* are *Evaluations* (but not *LiteralEvaluations*) with exactly one *true* or *false* *result*. *BooleanEvaluation* is the most general Predicate, and *booleanEvaluations* is the most general feature identifying them, specialized (incompletely) into those that always have *true* or always *false* *results*, *trueEvaluations* and *falseEvaluations*, respectively. *LiteralBooleanEvaluations* are *LiteralEvaluations* and *BooleanEvaluations*, with *result* specified in a model, potentially identified by *trueEvaluations* or *falseEvaluations*, or one of their specializations.

*NullEvaluations* are *Evaluations* that produce no values for their *result*. *NullEvaluation* is the most general *NullExpression*, and *nullEvaluations* is the most general Feature typed by *NullExpression* (see [Clause](#)).

## 9.2.6.2 Elements

### 9.2.6.2.1 BooleanEvaluation

#### Element

Predicate

#### Description

*BooleanEvaluation* is a specialization of *Evaluation* that is the most general predicate that may be evaluated to produce a Boolean truth value.

#### General Types

Evaluation

#### Features

result : Boolean {redefines result}

The Boolean result of this BooleanExpression.

#### **Constraints**

None.

### **9.2.6.2.2 booleanEvaluations**

#### **Element**

BooleanExpression

#### **Description**

`booleanEvaluations` is a specialization of `evaluations` restricted to type `BooleanEvaluation`.

#### **General Types**

BooleanEvaluation  
evaluations

#### **Features**

None.

#### **Constraints**

None.

### **9.2.6.2.3 Evaluation**

#### **Element**

Function

#### **Description**

An Evaluation is a Performance that ends with the production of a result.

#### **General Types**

Performance

#### **Features**

`result` : Anything [0..\*] {nonunique}

The `result` is the outcome of the Evaluation.

#### **Constraints**

None.

#### **9.2.6.2.4 evaluations**

##### **Element**

Expression

##### **Description**

`evaluations` is a specialization of `performances` for Evaluations of functions.

##### **General Types**

`performances`  
`Evaluation`

##### **Features**

None.

##### **Constraints**

None.

#### **9.2.6.2.5 falseEvaluations**

##### **Element**

`BooleanExpression`

##### **Description**

`booleanEvaluations` is a specialization of `evaluations` restricted to type `BooleanEvaluation`.

##### **General Types**

`booleanEvaluations`

##### **Features**

[no name] : `LiteralEvaluation`

##### **Constraints**

None.

#### **9.2.6.2.6 Involves**

##### **Element**

Association

##### **Description**

Involves classifies relationships between Performances and Objects.

### **General Types**

None.

### **Features**

None.

### **Constraints**

None.

#### **9.2.6.2.7 LiteralEvaluation**

##### **Element**

Function

##### **Description**

LiteralEvaluation is a specialization of Evaluation for the case of LiteralExpressions.

### **General Types**

Evaluation

### **Features**

result : DataValue {redefines result}

The result of this LiteralEvaluation, which is always a single DataValue.

### **Constraints**

None.

#### **9.2.6.2.8 literalEvaluations**

##### **Element**

Expression

##### **Description**

literalEvaluations is a specialization of evaluations restricted to type LiteralEvaluation.

### **General Types**

LiteralEvaluation  
evaluations

### **Features**

None.

**Constraints**

None.

**9.2.6.2.9 NullEvaluation****Element**

Function

**Description**

NullEvaluation is a specialization of Evaluation for the case of null expressions.

**General Types**

Evaluation

**Features**

result : Anything {redefines result}

The result of this NullEvaluation, which always must be empty (i.e., "null").

**Constraints**

None.

**9.2.6.2.10 nullEvaluations****Element**

Expression

**Description**

evaluations is a specialization of performances for Evaluations of functions.

**General Types**

NullEvaluation  
evaluations

**Features**

None.

**Constraints**

None.

**9.2.6.2.11 Performance****Element**

Behavior

### Description

A Performance is an Occurrence that applies constraints to how Objects interact or change over its life.

### General Types

Occurrence

### Features

enclosedPerformances : Performance [0..\*] {subsets timeEnclosedOccurrences}

timeEnclosedOccurrences of this Performance that are also Performances.

involvedObjects : Object [0..\*]

Objects that are involved in this Performance.

performers : Object [0..\*] {subsets involvedObjects}

Objects that enact this performance.

subperformances : Performance [0..\*] {subsets enclosedPerformances, suboccurrences}

enclosedPerformances that are composite. The default `this` context of a subperformance is by default the same as that of its owning Performance. This means that the context for any Performance that is in a composition tree rooted in a Performance that is not itself owned by an Object is the root Performance. If the root Performance is an `ownedPerformance` of an Object, then that Object is the context.

### Constraints

None.

## 9.2.6.2.12 performances

### Element

Step

### Description

performances is the most general feature for Performances of behaviors.

### General Types

Performance  
things

### Features

None.

### Constraints

None.

#### 9.2.6.2.13 Performs

##### Element

Association

##### Description

Performs is a specialization of Involves that asserts that the `performer` enacts the behavior carried out by the `enactedPerformance`.

##### General Types

Involves

##### Features

None.

##### Constraints

None.

#### 9.2.6.2.14 trueEvaluations

##### Element

BooleanExpression

##### Description

`booleanEvaluations` is a specialization of `evaluations` restricted to type `BooleanEvaluation`.

##### General Types

`booleanEvaluations`

##### Features

[no name] : `LiteralEvaluation`

##### Constraints

None.

### 9.2.7 Transfers

#### 9.2.7.1 Transfers Overview

*Transfers* are *Performances* that are also *BinaryLinks*, defined to ensure the things provided by their *source Occurrence* (via output Features) are accepted by their *target Occurrence* (via input Features, see Feature Direction in [Clause](#)). They do this by specifying the existence of *Links* between their *source / target Occurrence* and values of the output / input Features of those *Occurrences*, as identified by *sourceOutputLink* and



*targetOutputLink*, respectively. These two Connectors are typed by *BinaryLink*, and can be redefined to more specialized associations when *Transfer* is reused in models. The outputs of the *source Occurrence* (the things being "transferred") are identified as the *transferPayload* of *sourceOutputLinks* at the time a *Transfer* starts, also identified as the *sourceOutput* of the *Transfer* *source*, and as the *Transfer items*. The inputs of the *target Occurrence* (the things being "dropped of") are identified as the *transferPayload* of *targetInputLinks* at the time a *Transfer* ends, also identified as the *targetOutput* of the *Transfer* *target*, and as the *Transfer items*. Which things are being transferred does not change during a *Transfer*.

Three Boolean Features of *Transfers* affect their timing and of their *sourceOutputLinks* and *targetOutputLinks*:

- *isMove*: When true, the *sourceOutputLinks* end (cease to exist) when the *Transfer* starts, otherwise the *Transfer* has no effect on the *sourceOutputLinks*.
- *isPush*: When true, the *Transfer* starts when its *sourceOutputLinks* do (begin to exist), otherwise the *Transfer* can start anytime after the *sourceOutputLinks* do.
- *isInstant*: When true, the *Transfer* takes zero time (its *startShot* and *endShot* are the same, see Portions and Time Slices in [9.2.3.1](#)).

*Transfer* and its specializations are binary Interactions, while *transfers* is the most general Feature typed by *Transfer* or its specializations, and the most general ItemFlow (see [Clause](#)). *Transfer* is not the most general binary Interaction, and *transfers* is not the most general feature typed by binary Interactions, because binary Interactions can specify more than one *Transfer*.

ItemFlow *sourceOutputFeatures* and *targetInputFeatures* specify which Features of its connected Feature *Occurrences* identify outputs and inputs, respectively (most generally *sourceOutput* and *targetInput* above, respectively), as well as the kind of outputs and inputs, as its *itemType* (most generally the type of *item*, above).

## 9.2.7.2 Elements

### 9.2.7.2.1 Transfer

#### Element

Interaction

#### Description

#### General Types

Performance

BinaryLink

#### Features

*isInstant* : Boolean

*isMove* : Boolean

*isPush* : Boolean

*item* : Anything [1..\*]

*self* : Transfer {redefines self}

source : Occurrence [0..\*] {subsets toSources}

sourceOutputLink : BinaryLinkObject [1..\*]

sourceParticipant : Occurrence {redefines source}

sourceSendShot : Occurrence

target : Occurrence [0..\*] {subsets toTargets}

Occurrences whose input is the target of a Transfer of items from this Occurrence.

targetInputLink : BinaryLinkObject [1..\*]

targetParticipant : Occurrence {redefines target}

targetReceiveShot : Occurrence

### **Constraints**

None.

### **9.2.7.2.2 TransferBefore**

#### **Element**

Interaction

#### **Description**

#### **General Types**

Transfer

HappensBefore

#### **Features**

source : Occurrence [0..\*] {redefines predecessors, source}

sourceParticipant : Occurrence {redefines earlierOccurrence, sourceParticipant}

target : Occurrence [0..\*] {redefines target, successors}

Occurrences whose input is the target of a TransferBefore of items from this Occurrence.

targetParticipant : Occurrence {redefines laterOccurrence, targetParticipant}

### **Constraints**

None.

### **9.2.7.2.3 transfers**

#### **Element**

Feature

### **Description**

### **General Types**

Transfer

### **Features**

[no name] : Occurrence

[no name] : Occurrence

### **Constraints**

None.

## **9.2.7.2.4 transfersBefore**

### **Element**

Feature

### **Description**

### **General Types**

TransferBefore  
transfers

### **Features**

[no name] : Occurrence

[no name] : Occurrence

### **Constraints**

None.

## **9.2.8 Feature Referencing Performances**

The *FeatureAccessPerformances* package defines Behaviors used to read and write values of a referenced Feature of an Occurrence as of the time the Performance of the Behavior ends.

### **9.2.8.1 Feature Referencing Performances Overview**

### **9.2.8.2 Elements**

#### **9.2.8.2.1 BooleanEvaluationResultMonitorPerformance**

### **Element**

### **Description**

A `BooleanEvaluationResultMonitorPerformance` is a `EvaluationResultMonitorPerformance` that waits for changes in the `result` of a `BooleanEvaluation` identified by `onOccurrence`.

### General Types

`EvaluationResultMonitorPerformance`

### Features

`afterValues` : `Boolean` {redefines `afterValues`}

`beforeValues` : `Boolean` {redefines `beforeValues`}

`monitoredOccurrence` : `BooleanEvaluation` {subsets `timeSlices`, redefines `monitoredOccurrence`}

A `timeSlice` of `onOccurrence` during which its values for `result` change.

`onOccurrence` : `BooleanEvaluation` {redefines `onOccurrence`}

The `BooleanEvaluation` being monitored for changes in its `result` values.

`result` : `Boolean` {redefines `result`, nonunique}

Redefines `BooleanEvaluation::result` and `monitoredFeature`.

### Constraints

None.

## 9.2.8.2.2 BooleanEvaluationResultToMonitorPerformance

### Element

### Description

A `BooleanEvaluationResultToMonitorPerformance` is a `FeatureReferencingPerformance` that waits for the `result` of a `BooleanEvaluation` (identified by `onOccurrence`) to change to either true or false, as indicated by `isToTrue` (defaulting to true). If the `result` is already true (or false), the performance waits for the `result` to become false (or true) before waiting again for it to change back.

### General Types

`FeatureReferencingPerformance`

### Features

`afterValues` : `Boolean` {redefines `values`, nonunique}

The values of `monitoredFeature` for `onOccurrence` immediately after they change. Always the same as `isToTrue`.

`endWhen` : `HappensJustBefore`

See `FeatureMonitorPerformance::endWhen`. It is restricted to `HappensJustBefore` in `monitor1` and `monitor2`.

isToTrue : Boolean

monitor1 : BooleanEvaluationResultMonitorPerformance

Waits for the `result` of `onOccurrence` to change.

monitor2 : BooleanEvaluationResultMonitorPerformance [0..1]

Waits for the `result` of `onOccurrence` to change again, only if the change detected by `monitor1` was not the same as `isToTrue`.

onOccurrence : BooleanEvaluation {redefines onOccurrence}

The BooleanEvaluation being monitored for changes in its `result` values.

### Constraints

bertmpMonitor1ElseMonitor2

[no documentation]

```
isEmpty(monitor2) == (monitor1.afterValues == isToTrue)
```

## 9.2.8.2.3 EvaluationResultMonitorPerformance

### Element

Behavior

### Description

An EvaluationResultMonitorPerformance is a FeatureMonitorPerformance that waits for changes in `result` of an Evaluation identified by `onOccurrence`. The Predicate being evaluated must be able to produce multiple `results` over time, for example by only using Binding (SelfLink) Connectors between Steps, rather than Successions or ItemFlows, including in its Step `behaviors`.

### General Types

FeatureMonitorPerformance

### Features

monitoredOccurrence : Evaluation {subsets timeSlices, redefines monitoredOccurrence}

A `timeSlice` of `onOccurrence` during which its values for `result` change.

onOccurrence : Evaluation {redefines onOccurrence}

The Evaluation being monitored for changes in its `result` values.

result : Anything [0..\*] {redefines monitoredFeature, nonunique}

Redefines Evaluation::`result` and `monitoredFeature`.

### Constraints

None.

#### 9.2.8.2.4 FeatureAccessPerformance

##### Element

Behavior

##### Description

A FeatureAccessPerformance is a FeatureReferencingPerformance where `values` are all the values of `accessedFeature` for `onOccurrence` at the time the Performance ends. Specializations or usages of this narrow `accessedFeature` to particular features.

##### General Types

FeatureReferencingPerformance

##### Features

`accessedFeature` : Anything [0..\*] {nonunique}

Feature of `onOccurrence` that has `values` at the time this FeatureAccessPerformance ends.

`startingAt` : Occurrence {subsets `timeSlices`}

A `timeslice` of `onOccurrence` that starts when this FeatureAccessPerformance ends.

##### Constraints

None.

#### 9.2.8.2.5 FeatureMonitorPerformance

##### Element

Behavior

##### Description

A FeatureMonitorPerformance is a FeatureReferencingPerformance that waits for values of `monitoredFeature` to change on `onOccurrence` from what they were when the performance started. The values before and after the change are given by `beforeValues` and `afterValues`

##### General Types

FeatureReferencingPerformance

##### Features

`afterSnapshot` : Occurrence {subsets `snapshots`}

A `snapshot` of `monitoredOccurrence` just after its values for `monitoredFeature` change.

`afterValues` : Anything [0..\*] {redefines `values`}

The values of `monitoredFeature` for `monitoredOccurrence` immediately after they change

`beforeTimeSlice` : `Occurrence` {subsets `timeSlices`}

A `timeSlice` of `monitoredOccurrence`, starting at the same time, and ending just before its values for `monitoredFeature` change.

`beforeValues` : `Anything` [0..\*]

The values of `monitoredFeature` for `monitoredOccurrence` before any change

`endWhen` : `HappensBefore`

Succession (Connector typed by `HappensBefore`) from `afterSnapshot` to the `endShot` of this `FeatureMonitorPerformance`. Can be specialized to specify how soon the performance should end after the change in `monitoredFeature`.

`monitoredFeature` : `Anything` [0..\*] {nonunique}

The Feature being monitored for changes in values on `monitoredOccurrence`.

`monitoredOccurrence` : `Occurrence` {subsets `timeSlices`}

A `timeSlice` of `onOccurrence`, starting when this `FeatureMonitorPerformance` starts, during which the values of `monitoredFeature` change.

### Constraints

`fmpBeforeAfterValuesNotSame`

[no documentation]

`not beforeValues == afterValues`

#### 9.2.8.2.6 FeatureReadEvaluation

##### Element

Function

##### Description

A `FeatureReadEvaluation` is a `FeatureAccessPerformance` that is a Function providing as its result the values of `accessedFeature` of `onOccurrence` at the time the evaluation ends.

##### General Types

Evaluation  
`FeatureAccessPerformance`

##### Features

`result` : `Anything` [0..\*] {redefines `result`, `values`, `nonunique`}

Values of the Feature being accessed, as an `out` parameter.

## Constraints

None.

### 9.2.8.2.7 FeatureReferencingPerformance

#### Element

Behavior

#### Description

A FeatureReferencingPerformance is a Performance generalizing other Behaviors relating to `values` of a Feature of `onOccurrence`, as specified in the specialized Behaviors.

#### General Types

Performance

#### Features

`onOccurrence` : Occurrence

An Occurrence that has `values` for a Feature determined in specializations of this behavior.

`values` : Anything [0..\*] {nonunique}

Values of a Feature of `onOccurrence`, determined in specializations of this Behavior.

## Constraints

None.

### 9.2.8.2.8 FeatureWritePerformance

#### Element

Behavior

#### Description

A FeatureWritePerformance is a FeatureAccessPerformance that ensures the values of `onOccurrence` are exactly the `replacementValues` at the time the performance ends.

#### General Types

FeatureAccessPerformance

#### Features

`replacementValues` : Anything [0..\*] {redefines values, nonunique}

Values of the Feature being accessed, as an `inout` parameter to replace all the values.

## Constraints



None.

## 9.2.9 Control Performances

### 9.2.9.1 Control Performances Overview

The *ControlPerformances* package defines Behaviors to be used to type Steps that control the sequencing of performance of other Steps, including the following.

*DecisionPerformances* are *Performances* used by ("decision") Steps to ensure that each *DecisionPerformance* (value) of the Step is the *earlierOccurrence* of exactly one *HappensBefore* link of the Successions going out of the Step. Successions going out of steps typed by *DecisionPerformance* or its specializations must:

- have connector end multiplicities of 1 towards the Step, and 0..1 away from it.
- subset a Feature of its *featuringBehavior* derived as a chain of the Step and *DecisionPerformance::outgoingHBLink* (see Feature Chaining in [Clause](#)).

*MergePerformances* are *Performances* used by ("merge") Steps to ensure that each *MergePerformance* (value) of the Step is the *laterOccurrence* of exactly one *HappensBefore* link of the Successions coming into the step. Successions coming into steps typed by *MergePerformance* or its specializations must:

- have connector end multiplicities of 1 towards the Step, and 0..1 away from it.
- subset a Feature of its *featuringBehavior* derived as a chain of the Step and *MergePerformance::incomingHBLink*.

*IfPerformances* are *Performances* that determine whether one or more clauses occur based on the value of a Boolean argument. The concrete specializations of *IfPerformance* are *IfThenPerformance*, *IfElsePerformance* and *IfThenElsePerformance*.

*LoopPerformances* are *Performances* that whose body occurs iteratively as determined by Boolean "while" and "until" conditions.

### 9.2.9.2 Elements

#### 9.2.9.2.1 DecisionPerformance

##### Element

Behavior

##### Description

A *DecisionPerformance* is a *Performance* that represents the selection of one of the Successions that have the *DecisionPerformance* behavior as their source. All such Successions must subset the *outgoingHBLink* feature of the source *DecisionPerformance*. For each instance of *DecisionPerformance*, the *outgoingHBLink* is an instance of exactly one of the Successions, ordering the *DecisionPerformance* as happening before an instance of the target of that Succession.

##### General Types

Performance

##### Features

*outgoingHBLink* : *HappensBefore*

## Constraints

None.

### 9.2.9.2.2 IfElsePerformance

#### Element

Behavior

#### Description

An IfElsePerformance is an IfPerformance where `else` occurs after and only after the `ifTest` Evaluation `result` is not true.

#### General Types

IfPerformance

#### Features

`elseClause` : Occurrence [0..1]

#### Constraints

None.

### 9.2.9.2.3 IfPerformance

#### Element

Behavior

#### Description

An IfPerformance is a Performance that determines whether the `if` Evaluation `result` is true (by whether the `ifTrue` connector has a value).

#### General Types

Performance

#### Features

`ifTest` : BooleanEvaluation

`trueLiteral` : LiteralEvaluation

#### Constraints

None.

### 9.2.9.2.4 IfThenElsePerformance

#### Element

Behavior

### Description

An IfThenElsePerformance is an IfThenPerformance and an IfElsePerformance.

### General Types

IfElsePerformance  
IfThenPerformance

### Features

None.

### Constraints

None.

## 9.2.9.2.5 IfThenPerformance

### Element

Behavior

### Description

An IfThenPerformance is an IfPerformance where `then` occurs after and only after the `if` Evaluation result is true.

### General Types

IfPerformance

### Features

`thenClause` : Occurrence [0..1]

### Constraints

None.

## 9.2.9.2.6 LoopPerformance

### Element

Behavior

### Description

A LoopPerformance is a Performance where `body` occurs repeatedly in sequence (iterates) as long as the `while` evaluation result is true before each iteration (and after the previous one, except the first time) and the `until` evaluation result is not true after each iteration and before the next one (except the last one).

### General Types

Performance

### Features

body : Occurrence [0..\*]

untilDecision : IfElsePerformance [0..\*]

untilTest : BooleanEvaluation [0..\*]

whileDecision : IfThenPerformance [1..\*]

whileTest : BooleanEvaluation [1..\*]

### Constraints

None.

## 9.2.9.2.7 MergePerformance

### Element

Behavior

### Description

A MergePerformance is a Performance that represents the merging of all Successions that target the MergePerformance behavior. All such Successions must subset the `incomingHBLink` feature of the target MergePerformance. For each instance of MergePerformance, the `incomingHBLink` is an instance of exactly one of the Successions, ordering the MergePerformance as happening after an instance of the source of that Succession.

### General Types

Performance

### Features

incomingHBLink : HappensBefore

### Constraints

None.

## 9.2.10 State Performances

### 9.2.10.1 State Performances Overview

The *StatePerformance* package contains a library model of the semantics of state-based behavior, including *StatePerformances* and *StateTransitionPerformances* .

*StatePerformances* are *DecisionPerformances* (see [9.2.9.1](#)) that

- only have Steps defined in this library, or specialized from them.
- can identify *Transfers* that happen before the last *Performance* of the above Steps (see *exit* below).

Usages of *StatePerformance* can specialize the library Steps to specify how they are carried out, as well as specify how the *Transfer* above is identified. Additional modeler-defined Steps must subset the *middle* of the library Steps:

- *entry* [1]: happens before all *Performances* of *middle*.
- *middle* [1..\*]: happen during the *Performance* of *do*.
- *do* [1]: one of the *Performances* of *middle* that starts before the others and ends after them.
- *exit* [1]: happens after all *Performances* of *middle* and the end of the *Transfers* identified by the *StatePerformance* (see *accepted* below).

*StatePerformances* identify *Transfers* that happen before the *Performance* of *exit*, as specified by usages of *StatePerformance* by redefining these library Steps:

- *acceptable* [\*]: candidates for being identified as *accepted*.
- *accepted* [0..1]: one of the *acceptable* transfers. This must have a value if *acceptable* does.

If *isTriggerDuring* is *true* then *accepted* must end during the *StatePerformance*. If *isAcceptFirst* is *true*, *accepted* must end before the other *acceptable* ones.

Steps typed by *StatePerformances* can

- be *steps* of any Behavior (not only "state machines")
- employ any method of identifying *Transfers* needed to start ("trigger") their *exit*, including none at all, as well as requiring their *targets* to be, for example, the *StatePerformance* itself, or a *Performance* it is a *subperformance* of, or an *Object* enacts that *Performance*.
- have outgoing Successions constrained to have values (*HappensBefore* links) or not based on their *earlierOccurrences* (a *StatePerformance* of the step) and other conditions.
- be used in conjunction with other Steps typed by *TransitionPerformances* (see [9.2.11.1](#)) to determine which Succession going out of a Step "is chosen" (has a *HappenBeforeLink* value with a *StatePerformance* of that Step as its *earlierOccurrence*).

*StateTransitionPerformances* are *TransitionPerformances* (see [9.2.11.1](#)) that have a *StatePerformance* as their *transitionLinkSource*. *StateTransitionPerformance*

- *triggers* subset the *accepted Transfers* of its *transitionLinkSource*. If its *isTriggerDuring* is *true*, *triggers* must end during its *transitionLinkSource*.
- *guards* happen after the *middle* Step of its *transitionLinkSource* and before the *exit* Step.

## 9.2.10.2 Elements

### 9.2.10.2.1 StatePerformance

**Element**

Behavior

**Description**

**General Types**

DecisionPerformance

**Features**

/acceptable : Transfer [0..\*] {union}

accepted : Transfer [0..1] {subsets acceptable}  
 do : Performance {subsets middle}  
 entry : Performance {subsets timeEnclosedOccurrences}  
 exit : Performance {subsets timeEnclosedOccurrences}  
 isAcceptFirst : Boolean  
 isTriggerDuring : Boolean  
 /middle : Performance [1..\*] {subsets timeEnclosedOccurrences, union}  
 /nonDoMiddle : Performance [0..\*] {subsets middle}

### Constraints

None.

## 9.2.10.2.2 StateTransitionPerformance

### Element

Behavior

### Description

### General Types

TransitionPerformance

### Features

acceptable : Transfer [0..\*]  
 isTriggerDuring : Boolean  
 transitionLinkSource : StatePerformance {redefines transitionLinkSource}

### Constraints

None.

## 9.2.11 Transition Performances

### 9.2.11.1 Transition Performances Overview

The *TransitionPerformances* package contains a library model of the semantics of conditional transitions between *Occurrences*, including the performance of specified Behaviors when the transition occurs.

*TransitionPerformances* are *Performances* used to

- determine whether a Succession going out of an *Occurrence* Feature (`Succession::sourceFeature`) has values (*HappensBefore* links), based on values of `sourceFeature` (*Occurrences*) and other conditions, including ending of *Transfers*.
- perform specified Behaviors for each value of the Succession above.

The Succession constrained by a *TransitionPerformance* is specified by a Connector between the Succession and its `transitionStep` (see Successions in [Clause](#)), a unique Step typed by *TransitionPerformance* or a specialization of it, of the same Behavior as the Succession. This connector is

- typed by an Association defined to give a value to the *transitionLink* of *TransitionPerformances*,
- has connector end multiplicity 0.1 on the Succession end and 1 on the *TransitionPerformance* Step end.

The connector end multiplicities above ensure every *HappensBefore* link of the Succession is paired with a unique *TransitionPerformance* that has its conditions satisfied for that *Link*, while all the other *TransitionPerformances* of `transitionStep` fail their conditions and have no values for *transitionLink*.

The `transitionStep` above is also connected to the Succession's `sourceFeature`, because conditions on the Succession depend on each *Occurrence* of its `sourceFeature` separately, which *TransitionPerformances* identify as their *transitionLinkSource*. This connector is

- typed by an Association defined to give a value to the *transitionLinkSource* of *TransitionPerformances*.
- with connector end multiplicity 1 on both ends.

The connector end multiplicities above ensure every *Occurrence* of the Succession's `sourceFeature` is paired with a unique *TransitionPerformance*, and vice-versa, that determines whether the Succession has a value (*HappensBefore* link) for that *Occurrence*.

*TransitionPerformances* with a *transitionLink* must satisfy these conditions:

- identify at least one *Transfer* as *trigger*.
- all *Transfers* identified by *trigger* must happen before all *Evaluations* identified by *guard*.
- all *Evaluations* identified by *guard* must have *result* value *true*.

The *effect* of a *TransitionPerformance* can have values (*Performances*) only if the above conditions hold. The *effect Performances* must happen after the *guards* and before the *laterOccurrence* of *transitionLink*.

Usages of (Steps typed by) *TransitionPerformance* or its specializations can redefine or subset *guard* and *effect* to specify how they are carried out, as well as specify how *triggers* are identified. These usages can

- be steps of any Behavior (not only "state machines"), as well as constrain Successions going out of any kind of Step (not only those identifying *StatePerformances*, see [9.2.10.1](#)).
- employ any method of identifying *triggers*, including requiring none at all, as well as constraining *Transfer targets* to be, for example, the *StatePerformance* itself, or a *Performance* it is a *subperformance* of, or an *Object* enacting that *Performance*.

*TransitionPerformances* are either *StateTransitionPerformances* or *NonStateTransitionPerformances*, depending on whether the *transitionLinkSource* is a *StatePerformance* or not. Both ensure *guards* happen before the *laterOccurrence* of *transitionLink*, in case there are no *effects*, but do this in different ways (see [9.2.10.1](#) about *StateTransitionPerformances*).

### **9.2.11.2 Elements**

#### **9.2.11.2.1 NonStateTransitionPerformance**

##### **Element**

Behavior

##### **Description**

##### **General Types**

TransitionPerformance

##### **Features**

isTriggerAfter : Boolean

##### **Constraints**

None.

#### **9.2.11.2.2 TPCGuardConstraint**

##### **Element**

Association

##### **Description**

##### **General Types**

BinaryLink

##### **Features**

constrainedGuard : Evaluation {redefines target}

constrainedHBLink : HappensBefore {redefines source}

guardedBy : Evaluation [0..\*] {redefines toTargets}

guards : HappensBefore [0..1] {redefines toSources}

true : Boolean

##### **Constraints**

None.

#### **9.2.11.2.3 TransitionPerformance**

##### **Element**

Behavior



## Description

## General Types

Performance

## Features

effect : Performance [0..\*] {subsets enclosedPerformances}

guard : Evaluation [0..\*] {subsets enclosedPerformances}

guardConstraint : TPCGuardConstraint [0..\*]

transitionLink : HappensBefore [0..1]

transitionLinkSource : Performance

trigger : Transfer [0..\*]

## Constraints

None.

## 9.2.12 Clocks

### 9.2.12.1 Clocks Overview

This package models *Clocks* that provide an advancing numerical reference usable for quantifying the time of an *Occurrence*.

### 9.2.12.2 Elements

#### 9.2.12.2.1 BasicClock

## Element

Structure

## Description

A *BasicClock* is a *Clock* whose *currentTime* is a *Real* number.

## General Types

Clock

## Features

currentTime : Real {redefines currentTime}

## Constraints

None.

#### 9.2.12.2.2 BasicDurationOf

##### Element

Function

##### Description

*BasicDurationOf* returns the *DurationOf* an *Occurrence* as a *Real* number relative to a *BasicClock*.

##### General Types

DurationOf

##### Features

clock : BasicClock {redefines clock}

Default is inherited *Occurrence::localClock*.

duration : Real {redefines duration}

o : Occurrence {redefines o}

##### Constraints

None.

#### 9.2.12.2.3 BasicTimeOf

##### Element

Function

##### Description

*BasicTimeOf* returns the *TimeOf* an *Occurrence* as a *Real* number relative to a *BasicClock*.

##### General Types

TimeOf

##### Features

clock : BasicClock {redefines clock}

Default is inherited *Occurrence::localClock*.

o : Occurrence {redefines o}

timeValue : Real {redefines timeInstant}

##### Constraints

None.

#### 9.2.12.2.4 Clock

##### Element

Structure

##### Description

A *Clock* provides a scalar *currentTime* that advances monotonically over its lifetime. *Clock* is an abstract base Structure that can be specialized for different kinds of time quantification (e.g., discrete time, continuous time, time with units, etc.).

##### General Types

Object

##### Features

*currentTime* : NumericalValue

A numerical time reference that advances over the lifetime of the *Clock*.

##### Constraints

*timeFlowConstraint*

The *currentTime* of a snapshot of a *Clock* is equal to the *TimeOf* the snapshot relative to that *Clock*.

#### 9.2.12.2.5 DurationOf

##### Element

Function

##### Description

*DurationOf* returns the duration of a given *Occurrence* relative to a given *Clock*, which is equal to the *TimeOf* the end snapshot of the *Occurrence* minus the *TimeOf* its start snapshot.

##### General Types

Evaluation

##### Features

*clock* : Clock

Default is inherited *Occurrence::localClock*.

*duration* : NumericalValue

*o* : Occurrence

##### Constraints

None.

#### 9.2.12.2.6 TimeOf

##### Element

Function

##### Description

*TimeOf* returns a scalar *timeValue* for a given *Occurrence* relative to a given *Clock*. The *timeValue* is the time of the start of the *Occurrence*, which is considered to be synchronized with the snapshot of the *Clock* with a *currentTimeValue*.

##### General Types

Evaluation

##### Features

clock : Clock

Default is inherited *Occurrence::localClock*.

o : Occurrence

timeInstant : NumericalValue

##### Constraints

timeContinuityConstraint

If one *Occurrence* happens immediately before another, then the *TimeOf* the end snapshot of the first *Occurrence* equals the *TimeOf* the second *Occurrence*.

startTimeConstraint

The *TimeOf* an *Occurrence* is equal to the time of its start snapshot.

timeOrderingConstraint

If one *Occurrence* happens before another, then the *TimeOf* the end snapshot of the first *Occurrence* is no greater than the *TimeOf* the second *Occurrence*.

#### 9.2.12.2.7 universalClock

##### Element

Feature

##### Description

*universalClock* is a single *Clock* that can be used as a default universal time reference.

##### General Types

Clock  
objects

### Features

None.

### Constraints

None.

## 9.2.13 Observation

### 9.2.13.1 Observation Overview

This package models a framework for monitoring *Boolean* conditions and notifying registered observers when they change from false to true.

### 9.2.13.2 Elements

#### 9.2.13.2.1 CancelObservation

##### Element

Behavior

##### Description

Cancel all observations of a given *ChangeSignal* for a given *Occurrence*.

##### General Types

Performance

##### Features

observer : Occurrence

signal : ChangeSignal

##### Constraints

None.

#### 9.2.13.2.2 changeCondition

##### Element

Expression

##### Description

##### General Types

None.

### Features

None.

### Constraints

None.

#### 9.2.13.2.3 ChangeMonitor

### Element

Structure

### Description

A *ChangeMonitor* is a collection of ongoing *ChangeSignal* observations for various observer *Occurrences*. It provides convenient operations for starting and canceling the observations it manages.

### General Types

Object

### Features

cancelObservation : CancelObservation [0..\*]

Cancel all observations of a given *ChangeSignal* for a given *Occurrence*.

observations : ObserveChange [0..\*]

startObservation : StartObservation [0..\*]

Start an observation of a given *ChangeSignal* for a given *Occurrence*.

### Constraints

None.

#### 9.2.13.2.4 ChangeSignal

### Element

Structure

### Description

A *ChangeSignal* is a signal to be sent when the *Boolean* result of its *changeCondition* Expression changes from false to true.

### General Types

Object

### Features

changeMonitor : ChangeMonitor

The *ChangeMonitor* responsible for monitoring the *signalCondition*.

signalCondition : BooleanEvaluation

A BooleanExpression whose result is being monitored.

#### Constraints

None.

#### 9.2.13.2.5 defaultMonitor

##### Element

Feature

##### Description

*defaultMonitor* is a single *ChangeMonitor* that can be used as a default.

##### General Types

ChangeMonitor  
objects

##### Features

None.

##### Constraints

None.

#### 9.2.13.2.6 ObserveChange

##### Element

Behavior

##### Description

Each *Performance* of *ObserveChange* waits for the result of the *Boolean changeCondition* of a given *ChangeSignal* to change from false to true, and, when it does, sends the *ChangeSignal* to a given observer *Occurrence*.

##### General Types

Performance

##### Features

changeObserver : Occurrence

changeSignal : ChangeSignal

transfer : TransferBefore [0..1]

After waiting for the condition change (if necessary), then send *changeSignal* to *changeObserver*.

wait : IfThenPerformance

If the result of the *changeSignal.signalCondition* is false, then wait for it to become true:

```
in ifTest { not changeSignal.signalCondition() }
in thenClause : BooleanEvaluationResultToMonitorPerformance {
    in onOccurrence = changeSignal.signalCondition;
}
```

### Constraints

None.

#### 9.2.13.2.7 StartObservation

##### Element

Behavior

##### Description

Start an observation of a given *ChangeSignal* for a given *Occurrence*.

##### General Types

Performance

##### Features

observer : Occurrence

signal : ChangeSignal

##### Constraints

None.

#### 9.2.14 Triggers

##### 9.2.14.1 Triggers Overview

This package contains functions that return *ChangeSignals* for triggering when a *Boolean* condition changes from false to true, at a specific time or after a specific time delay.

##### 9.2.14.2 Elements

###### 9.2.14.2.1 TimeSignal

##### Element



Structure

### Description

A *TimeSignal* is a *ChangeSignal* whose condition is the *currentTime* of a given *Clock* reaching a specific *signalTime*.

### General Types

ChangeSignal

### Features

signalClock : Clock

The *Clock* whose *currentTime* is being monitored.

signalCondition : BooleanEvaluation {redefines signalCondition}

The *Boolean* condition of the *currentTime* of the *signalClock* being equal to the *signalTime*.

signalTime : NumericalValue

The time at which the *TimeSignal* should be sent.

### Constraints

None.

## 9.2.14.2.2 TriggerAfter

### Element

Function

### Description

*TriggerAfter* returns a monitored *TimeSignal* to be sent to a *receiver* after a certain time *delay* relative to a given *Clock*.

### General Types

Evaluation

### Features

clock : Clock

The *Clock* to be used as the reference for the time *delay*. The default is the *localClock*, which will be bound when the function is invoked.

delay : NumericalValue

The time duration, relative to the *clock*, after which the *TimeSignal* is sent.

monitor : ChangeMonitor

The *ChangeMonitor* to be used to monitor the *TimeSignal* condition. The default is the *Observation::defaultMonitor*.

receiver : Occurrence

The *Occurrence* to which the *TimeSignal* is to be sent.

signal : TimeSignal

### Constraints

None.

## 9.2.14.2.3 TriggerAt

### Element

Function

### Description

*TriggerAt* returns a monitored *TimeSignal* to be sent to a *receiver* when the *currentTime* of a given *Clock* reaches a specific *time*.

### General Types

Evaluation

### Features

clock : Clock

The *Clock* to be used as the reference for the *timeInstant*. The default is the *localClock*, which will be bound when the function is invoked.

monitor : ChangeMonitor

The *ChangeMonitor* to be used to monitor the *TimeSignal* condition. The default is the *Observation::defaultMonitor*.

receiver : Occurrence

The *Occurrence* to which the *TimeSignal* is to be sent.

signal : TimeSignal

timeInstant : NumericalValue

The time instant, relative to the *clock*, at which the *TimeSignal* should be sent.

### Constraints

None.

#### 9.2.14.2.4 TriggerWhen

##### Element

Function

##### Description

*TriggerWhen* returns a monitored *ChangeSignal* for a given *condition*, to be sent to a given *receiver* when the *condition* occurs.

##### General Types

Evaluation

##### Features

condition : BooleanEvaluation

The BooleanExpression to be monitored for changing from false to true.

monitor : ChangeMonitor

The *ChangeMonitor* to be used to monitor the *ChangeSignal* condition. The default is the *Observation::defaultMonitor*.

receiver : Occurrence

The *Occurrence* to which the *ChangeSignal* is to be sent.

signal : ChangeSignal

##### Constraints

None.

### 9.2.15 SpatialFrames

#### 9.2.15.1 SpatialFrames Overview

This package models spatial frames of reference for quantifying the position of points in a three-dimensional space.

#### 9.2.15.2 Elements

##### 9.2.15.2.1 CartesianCurrentDisplacementOf

##### Element

Function

##### Description

The *CurrentDisplacementOf* two Points relative to a *CartesianSpatialFrame* is a *CartesianThreeVectorValue*.

## General Types

CurrentDisplacementOf

## Features

clock : Clock {redefines clock}

displacementVector : CartesianThreeVectorValue {redefines displacementVector}

frame : CartesianSpatialFrame {redefines frame}

point1 : Point {redefines point1}

point2 : Point {redefines point2}

## Constraints

None.

### 9.2.15.2.2 CartesianCurrentPositionOf

## Element

Function

## Description

The *CurrentPositionOf* a Point relative to a *CartesianSpatialFrame* is a *CartesianThreeVectorValue*.

## General Types

CurrentPositionOf

## Features

clock : Clock {redefines clock}

Defaults to the *localClock* of the *frame*.

frame : CartesianSpatialFrame {redefines frame}

point : Point {redefines point}

positionVector : CartesianThreeVectorValue {redefines positionVector}

## Constraints

None.

### 9.2.15.2.3 CartesianDisplacementOf

## Element

Function

## Description

The *DisplacementOf* two Points relative to a *CartesianSpatialFrame* is a *CartesianThreeVectorValue*.

## General Types

DisplacementOf

## Features

clock : Clock {redefines clock}

Defaults to the *localClock* of the *frame*.

displacementVector : CartesianThreeVectorValue {redefines displacementVector}

frame : CartesianSpatialFrame {redefines frame}

point1 : Point {redefines point1}

point2 : Point {redefines point2}

time : NumericalValue {redefines time}

## Constraints

None.

### 9.2.15.2.4 CartesianPositionOf

## Element

Function

## Description

The *PositionOf* a Point relative to a *CartesianSpatialFrame* is a *CartesianThreeVectorValue*.

## General Types

PositionOf

## Features

clock : Clock {redefines clock}

Defaults to the *localClock* of the *frame*.

frame : CartesianSpatialFrame {redefines frame}

point : Point {redefines point}

positionVector : CartesianThreeVectorValue {redefines positionVector}

time : NumericalValue {redefines time}

## Constraints

None.

### 9.2.15.2.5 CartesianSpatialFrame

#### Element

Structure

#### Description

A *CartesianSpatialFrame* is a *SpatialFrame* relative to which all position and displacement vectors can be represented as *CartesianThreeVectorValues*.

#### General Types

SpatialFrame

#### Features

None.

#### Constraints

None.

### 9.2.15.2.6 CurrentDisplacementOf

#### Element

Function

#### Description

The *CurrentDisplacementOf* two *Points* relative to a *SpatialFrame* and *Clock* is the *DisplacementOf* the *Points* relative to the *SpacialFrame* at the *currentTime* of the *Clock*.

#### General Types

Evaluation

#### Features

clock : Clock

Defaults to the *localClock* of the *frame*.

displacementVector : ThreeVectorValue

frame : SpatialFrame

point1 : Point

point2 : Point

### Constraints

None.

### 9.2.15.2.7 CurrentPositionOf

#### Element

Function

#### Description

The *CurrentPositionOf* a *Point* relative to a *SpatialFrame* and a *Clock* is the *PositionOf* the *Point* relative to the *SpatialFrame* at the *currentTime* of the *Clock*.

#### General Types

Evaluation

#### Features

clock : Clock

Defaults to the *localClock* of the *frame*.

frame : SpatialFrame

point : Point

positionVector : ThreeVectorValue

### Constraints

None.

### 9.2.15.2.8 defaultFrame

#### Element

Feature

#### Description

*defaultFrame* is a fixed *SpatialFrame* used as a universal default.

#### General Types

SpatialFrame

#### Features

None.

### Constraints

None.

### 9.2.15.2.9 DisplacementOf

#### Element

Function

#### Description

The *DisplacementOf* two *Points* relative to a *SpatialFrame*, at a specific *time* relative to a given *Clock*, is the *displacementVector* computed as the difference between the *PositionOf* the first *Point* and *PositionOf* the second *Point*, relative to that *SpatialFrame*, at that *time*.

#### General Types

Evaluation

#### Features

clock : Clock

Defaults to the *localClock* of the *frame*.

displacementVector : ThreeVectorValue

frame : SpatialFrame

point1 : Point

point2 : Point

time : NumericalValue

#### Constraints

zeroDisplacementConstraint

If either *point1* or *point2* occurs within the other, then the *displacementVector* is the zero vector.

```
(point1.spaceTimeEnclosedOccurrences->includes(point2) or  
point2.spaceTimeEnclosedOccurrences->includes(point1)) implies  
  isZeroVector(displacementVector)
```

### 9.2.15.2.10 PositionOf

#### Element

Function

#### Description

The *PositionOf* a *Point* relative to a *SpatialFrame*, at a specific *time* relative to a given *Clock*, as a *positionVector* that is a *ThreeVectorValue*.



## General Types

Evaluation

## Features

clock : Clock

Defaults to the *localClock* of the *frame*.

frame : SpatialFrame

point : Point

positionVector : ThreeVectorValue

time : NumericalValue

## Constraints

positionTimePrecondition

The given *point* must exist at the given *time*.

`TimeOf(point.startShot) <= time and  
time <= TimeOf(point.endShot)`

spacePositionConstraint

The result *positionVector* is equal to the *PositionOf* the *Point* *spaceShot* of the frame that encloses the given *point*, at the given *time*.

```
(frame.spaceShots as Point)->forall{in p : Point;  
  p.spaceTimeEnclosedOccurrences->includes(point) implies  
  positionVector == PositionOf(p, time, frame)  
}
```

### 9.2.15.2.11 SpatialFrame

## Element

Structure

## Description

## General Types

Body

## Features

None.

## Constraints

None.

## 9.2.16 Metaobjects

### 9.2.16.1 Metaobjects Overview

This package defines Metaclasses and Features that are related to the typing of syntactic and semantic metadata.

### 9.2.16.2 Elements

#### 9.2.16.2.1 Metaobject

##### Element

Metaclass

##### Description

A *Metaobject* contains syntactic or semantic information about one or more *annotatedElements*. It is the most general Metaclass. All other Metaclasses must subclassify it directly or indirectly.

##### General Types

Object

##### Features

annotatedElement : Element [1..\*]

The Elements annotated by this *Metaobject*. This is set automatically when a *Metaobject* is instantiated as the value of a MetadataFeature.

##### Constraints

None.

#### 9.2.16.2.2 metaobjects

##### Element

Feature

##### Description

*metaobjects* is a specialization of *objects* restricted to type *Metaobject*. It is the most general MetadataFeature. All other MetadataFeatures must subset it directly or indirectly.

##### General Types

objects

Metaobject

##### Features

None.

##### Constraints

None.

### 9.2.16.2.3 SemanticMetadata

#### Element

Metaclass

#### Description

*SemanticMetadata* is *Metadata* that requires its single *annotatedType* to directly or indirectly specialize a *baseType* that models the semantics for the *annotatedType*.

#### General Types

Metaobject

#### Features

`annotatedElement : Type {redefines annotatedElement}`

The single *annotatedElement* of this *SemanticMetadata*, which must be a *Type*.

`baseType : Type`

The required base *Type* for the *annotatedType*.

#### Constraints

None.

## 9.2.17 KerML

### 9.2.17.1 KerML Overview

This package contains a reflective KerML model of the KerML abstract syntax.

**Release Note:** This model is currently incomplete. It includes all KerML abstract syntax metaclasses, but none of their properties.

### 9.2.17.2 Elements

```
metaclass AnnotatingElement :> Element;
metaclass Annotation :> Relationship;
metaclass Comment :> AnnotatingElement;
metaclass Documentation :> Annotation;
metaclass TextualRepresentation :> AnnotatingElement;

metaclass Import :> Relationship;
metaclass Membership :> Relationship;
metaclass Namespace :> Element;

metaclass Type :> Namespace;
metaclass Multiplicity :> Feature;
metaclass FeatureMembership :> Membership, TypeFeaturing;
metaclass Specialization :> Relationship;
```

```

metaclass Conjugation :> Relationship;
metaclass Disjoining :> Relationship;

metaclass Classifier :> Type;
metaclass Subclassification :> Specialization;

metaclass Feature :> Type;
metaclass Subsetting :> Specialization;
metaclass Redefinition :> Subsetting;
metaclass FeatureTyping :> Specialization;
metaclass TypeFeaturing :> Relationship;
metaclass FeatureChaining :> Relationship;
metaclass EndFeatureMembersip :> FeatureMembership;

metaclass Class :> Classifier;
metaclass DataType :> Classifier;

metaclass Structure :> Class;

metaclass Association :> Classifier, Relationship;
metaclass AssociationStructure :> Association, Structure;

metaclass Connector :> Feature, Relationship;
metaclass BindingConnector :> Connector;
metaclass Succession :> Connector;

metaclass Behavior :> Class;
metaclass Step :> Feature;
metaclass ParameterMembership :> FeatureMembership;

metaclass Function :> Behavior;
metaclass Predicate :> Function;
metaclass Expression :> Step;
metaclass BooleanExpression :> Expression;
metaclass Invariant :> BooleanExpression;
metaclass ReturnParameterMembership :> ParameterMembership;
metaclass ResultExpressionMembership :> FeatureMembership;

metaclass FeatureReferenceExpression :> Expression;
metaclass InvocationExpression :> Expression;
metaclass LiteralExpression :> Expression;
metaclass LiteralInteger :> LiteralExpression;
metaclass LiteralRational :> LiteralExpression;
metaclass LiteralBoolean :> LiteralExpression;
metaclass LiteralString :> LiteralExpression;
metaclass LiteralInfinity :> LiteralExpression;
metaclass NullExpression :> Expression;
metaclass OperatorExpression :> InvocationExpression;
metaclass FeatureChainExpression :> OperatorExpression;
metaclass CollectExpression :> OperatorExpression;
metaclass SelectExpression :> OperatorExpression;

metaclass Interaction :> Behavior, Association;
metaclass ItemFlow :> Step, Connector;
metaclass SuccessionItemFlow :> ItemFlow, Succession;

metaclass FeatureValue :> Membership;

metaclass MultiplicityRange :> Multiplicity;

```

```
metaclass Metaclass :> Structure;
metaclass MetadataFeature :> AnnotatingElement, Feature;

metaclass Package :> Namespace;
metaclass ElementFilterMembership :> Membership;
```

## 9.3 Data Type Library

### 9.3.1 Data Types Library Overview

The Data Types Library provides a standard set of commonly used `DataType`s for scalar, vector and collection values.

### 9.3.2 Scalar Values

#### 9.3.2.1 Scalar Values Overview

This package contains a basic set of primitive scalar (non-collection) data types. These include *Boolean* and *String* types and a hierarchy of concrete *Number* types, from the most general type of *Complex* numbers to the most specific type of *Positive* integers.

#### 9.3.2.2 Elements

##### 9.3.2.2.1 Boolean

###### Element

`DataType`

###### Description

`Boolean` is a `ScalarValue` type whose instances are `true` and `false`.

###### General Types

`ScalarValue`

###### Features

None.

###### Constraints

None.

##### 9.3.2.2.2 Complex

###### Element

`DataType`

###### Description

`Complex` is the type of complex numbers.

###### General Types

Number

**Features**

None.

**Constraints**

None.

**9.3.2.2.3 Integer**

**Element**

DataType

**Description**

Integer is the type of mathematical integers, extended with values for positive and negative infinity.

**General Types**

Rational

**Features**

None.

**Constraints**

None.

**9.3.2.2.4 Natural**

**Element**

DataType

**Description**

Natural is the type of non-negative integers, extended with a value for positive infinity.

**General Types**

DataValue

Integer

**Features**

None.

**Constraints**

None.

#### **9.3.2.2.5 Number**

##### **Element**

DataType

##### **Description**

Number is the base type for all NumericalValue types that represent numbers.

##### **General Types**

NumericalValue

##### **Features**

None.

##### **Constraints**

None.

#### **9.3.2.2.6 NumericalValue**

##### **Element**

DataType

##### **Description**

NumericalValue is the base type for all ScalarValue types that represent numerical values.

##### **General Types**

ScalarValue

##### **Features**

None.

##### **Constraints**

None.

#### **9.3.2.2.7 Positive**

##### **Element**

DataType

##### **Description**

Positive is the type of positive integers (not including zero), extended with a value for positive infinity.

##### **General Types**

Natural

**Features**

None.

**Constraints**

None.

**9.3.2.2.8 Rational**

**Element**

DataType

**Description**

Rational is the type of rational numbers, extended with values for positive and negative infinity.

**General Types**

Real

**Features**

None.

**Constraints**

None.

**9.3.2.2.9 Real**

**Element**

DataType

**Description**

Real is the type of mathematical (extended) real numbers. This includes both rational and irrational numbers, and values for positive and negative infinity.

**General Types**

Complex

**Features**

None.

**Constraints**

None.



#### 9.3.2.2.10 ScalarValue

##### Element

DataType

##### Description

A ScalarValue is a DataValue whose instances are considered to be primitive, not collections or structures of other values.

##### General Types

DataValue

##### Features

None.

##### Constraints

None.

#### 9.3.2.2.11 String

##### Element

DataType

##### Description

String is a ScalarValue type whose instances are strings of characters.

##### General Types

ScalarValue

##### Features

None.

##### Constraints

None.

### 9.3.3 Collections

#### 9.3.3.1 Collections Overview

This package defines a standard set of *Collection* data types. Unlike sequences of values defined directly using multiplicity, these data types allow for the possibility of collections as elements of collections.

### 9.3.3.2 Elements

#### 9.3.3.2.1 Array

##### Element

DataType

##### Description

An Array is a fixed size, multi-dimensional Collection of which the `elements` are nonunique and ordered. Its `dimensions` specify how many dimensions the array has, and how many elements there are in each dimension. The `rank` is equal to the number of `dimensions`. The `flattenedSize` is equal to the total number of `elements` in the array.

Feature `elements` is a flattened sequence of all elements of an Array and can be accessed by a tuple of indices. The number of indices is equal to `rank`. The `elements` are packed according to row-major convention, as in the C programming language.

**Note 1.** Feature `dimensions` may be empty, which denotes a zero dimensional array, allowing an Array to collapse to a single element. This is useful to allow for specialization of an Array into a type restricted to represent a scalar. The `flattenedSize`,/code> of a zero dimensional array is 1.

**Note 2.** An Array can also represent the generalized concept of a mathematical matrix of any rank, i.e. not limited to rank two.

##### General Types

OrderedCollection

##### Features

`dimensions` : Positive [0..\*] {ordered, nonunique}

`flattenedSize` : Positive

`rank` : Natural

##### Constraints

`sizeConstraint`

[no documentation]

`flattenedSize == size(elements)`

#### 9.3.3.2.2 Bag

##### Element

DataType

##### Description

A Bag is a variable size Collection of which the `elements` are unordered and nonunique.

## General Types

Collection

## Features

None.

## Constraints

None.

### 9.3.3.2.3 Collection

## Element

DataType

## Description

A Collection is an abstract DataType that represents a collection of elements of a given type.

A Collection is either mutable or immutable, or mutability is unspecified.

TODO: Decide on whether to add Mutability, and if so, how.

## General Types

Anything

## Features

elements : Anything [0..\*] {nonunique}

## Constraints

None.

### 9.3.3.2.4 KeyValuePair

## Element

DataType

## Description

A KeyValuePair is an abstract DataType that represents a tuple of a `key` and an associated value `val`.

## General Types

DataValue

## Features

key : Anything

val : Anything

#### **Constraints**

None.

#### **9.3.3.2.5 List**

##### **Element**

DataType

##### **Description**

A Sequence is a variable size Collection of which the `elements` are nonunique and ordered.

##### **General Types**

OrderedCollection

##### **Features**

None.

#### **Constraints**

None.

#### **9.3.3.2.6 Map**

##### **Element**

DataType

##### **Description**

A Map is a variable size Collection of which the `elements` are `KeyValuePairs`. The keys must be unique within in the Map. The values need not be unique.

##### **General Types**

UniqueCollection

##### **Features**

`elements : KeyValuePair [0..*] {redefines elements}`

#### **Constraints**

None.

#### **9.3.3.2.7 OrderedCollection**

##### **Element**

DataType

### **Description**

An `OrderedCollection` is a `Collection` of which the `elements` are ordered, and not necessarily unique).

### **General Types**

`Collection`

### **Features**

`elements` : `Anything [0..*]` {redefines `elements`, `ordered`, `nonunique`}

### **Constraints**

None.

## **9.3.3.2.8 OrderedMap**

### **Element**

DataType

### **Description**

An `OrderedMap` is a variable size `Map` that maintains ordering of its `elements`.

The ordering may be by key of the `KeyValuePair` `elements`, or by order of construction, or any other method. The essential aspect is that ordering is maintained and guaranteed across accesses to the `OrderedMap`.

### **General Types**

`Map`

`OrderedCollection`

### **Features**

`elements` : `KeyValuePair [0..*]` {redefines `elements`, `ordered`}

### **Constraints**

None.

## **9.3.3.2.9 OrderedSet**

### **Element**

DataType

### **Description**

An `OrderedSet` is a variable size `Collection` of which the `elements` are unique and ordered.

### **General Types**

OrderedCollection  
UniqueCollection

### **Features**

elements : Anything [0..\*] {redefines elements, ordered}

### **Constraints**

None.

### **9.3.3.2.10 Set**

#### **Element**

DataType

#### **Description**

A Set is a variable size Collection of which the `elements` are unique and unordered.

#### **General Types**

UniqueCollection

### **Features**

None.

### **Constraints**

None.

### **9.3.3.2.11 UniqueCollection**

#### **Element**

DataType

#### **Description**

A UniqueCollection is a Collection of which the `elements` are unique, and not necessarily ordered).

#### **General Types**

Collection

### **Features**

elements : Anything [0..\*] {redefines elements}

### **Constraints**

None.

## 9.3.4 Vector Values

### 9.3.4.1 Vector Values Overview

### 9.3.4.2 Elements

#### 9.3.4.2.1 CartesianThreeVectorValue

##### Element

DataType

##### Description

A *CartesianThreeVectorValue* is a *NumericalVectorValue* that is both Cartesian and has dimension 3.

##### General Types

ThreeVectorValue  
CartesianVectorValue

##### Features

None.

##### Constraints

None.

#### 9.3.4.2.2 CartesianVectorValue

##### Element

DataType

##### Description

A *CartesianVectorValue* is a *NumericalVectorValue* for which there are specific implementations in *VectorFunctions* of the abstract vector-space functions.

**Note:** The restriction of the element type to *Real* is to facilitate the complete definition of these functions.

##### General Types

NumericalVectorValue

##### Features

elements : Real [0..\*] {redefines elements}

##### Constraints

None.

#### 9.3.4.2.3 NumericalVectorValue

##### Element

DataType

##### Description

A *NumericalVectorValue* is a kind of *VectorValue* that is specifically represented as a one-dimensional *Array* of *NumericalValues*. The dimension is allowed to be empty, permitting a *NumericalVectorValue* of rank 0, which is essentially isomorphic to a scalar *NumericalValue*.

##### General Types

Array

VectorValue

##### Features

dimension : Positive [0..1] {redefines dimensions}

elements : NumericalValue [0..\*] {redefines elements}

##### Constraints

None.

#### 9.3.4.2.4 ThreeVectorValue

##### Element

DataType

##### Description

A *ThreeVectorValue* is a *NumericalVectorValue* that has dimension 3.

##### General Types

NumericalVectorValue

##### Features

dimension : Positive [0..\*] {redefines elements}

##### Constraints

None.

#### 9.3.4.2.5 VectorValue

##### Element

DataType



## Description

A *VectorValue* is an abstract data type whose values may be operated on using *VectorFunctions*.

## General Types

None.

## Features

None.

## Constraints

None.

## 9.4 Function Library

The Function Library includes library models of basic *Functions* that operate on *DataTypes* from the Data Type Library (see [9.3](#)). The KerML operator expression notation translates to invocations of some of these library *Functions*. It is expected that other languages built on KerML will provide additional domain models as needed by their applications, which can include specializations of the library *Functions* for domain-specific *DataTypes*. The same KerML concrete syntax for *Expressions* can be used with these specialized *Functions* and *DataTypes*, extended with domain-specific semantics.

### 9.4.1 Function Library Overview

#### 9.4.2 Base Functions

##### 9.4.2.1 Base Functions Overview

This package defines a basic set of *Functions* defined on all kinds of values. Most correspond to similarly named operators in the KerML expression notation.

##### 9.4.2.2 Elements

```
abstract function '='{
  in x: Anything[0..1];
  in y: Anything[0..1];
  return : Boolean[1];
}
function '!='{
  in x: Anything[0..1];
  in y: Anything[0..1];
  return : Boolean[1] = not (x == y);
}

function ToString{
  in x: Anything[0..1];
  return : String;
}

function '['{
  in seq: Anything[0..*] ordered nonunique;
  in index: Anything[0..*] ordered nonunique;
  return : Anything[0..1];
}
```

```

function ','{
  in seq1: Anything[0..*] ordered nonunique;
  seq2: Anything[0..*] ordered nonunique;
  return : Anything[0..*] ordered nonunique;
}

abstract function 'all'{
  abstract feature all 'type': Object;
  return : Object[0..*];
}

abstract function 'istype'{
  in x: Anything[1];
  abstract feature 'type': Anything;
  return : Boolean[1];
}

abstract function '@'{
  in x: Anything[1];
  abstract feature 'type': Anything;
  return : Boolean[1];
}

abstract function 'hastype'{
  in x: Anything[1];
  abstract feature 'type': Anything[1];
  return : Boolean;
}

abstract function 'as'{
  in seq: Anything[0..*] ordered nonunique;
  abstract feature 'type': Anything[1];
  return : Anything[0..*] ordered nonunique;
}

```

## 9.4.3 Data Functions

### 9.4.3.1 Data Functions Overview

This package defines the abstract base Functions corresponding to all the unary and binary operators in the KerML expression notation that might be defined on various kinds of DataValues.

### 9.4.3.2 Elements

```

abstract function '+'
  { in x: DataValue[1]; in y: DataValue[0..1]; return : DataValue[1]; }
abstract function '-'
  { in x: DataValue[1]; in y: DataValue[0..1]; return : DataValue[1]; }
abstract function '*'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '/'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '**'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '^'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '%'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

```

```

abstract function 'not'
  { in x: DataValue[1]; return : DataValue[1]; }
abstract function 'xor'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

abstract function '~'
  { in x: DataValue[1]; return : DataValue[1]; }
abstract function '|'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function '&'
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

abstract function '<'
  { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }
abstract function '>'
  { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }
abstract function '<='
  { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }
abstract function '>='
  { in x: DataValue[1]; in y: DataValue[1]; return : Boolean[1]; }

abstract function Max
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }
abstract function Min
  { in x: DataValue[1]; in y: DataValue[1]; return : DataValue[1]; }

abstract function '==' specializes BaseFunctions::'=='
  { in x: DataValue[0..1]; in y: DataValue[0..1]; return : Boolean[1]; }

abstract function '...'
  { in lower: DataValue[1]; in upper: DataValue[1]; return : DataValue[0..*] ordered; }

```

## 9.4.4 Scalar Functions

### 9.4.4.1 Scalar Functions Overview

This package defines abstract functions that specialize the DataFunctions for use with ScalarValues.

#### 9.4.4.2 Elements

```

abstract function '+' specializes DataFunctions::'+'
  { in x: ScalarValue[1]; in y: ScalarValue[0..1]; return : ScalarValue[1]; }
abstract function '-' specializes DataFunctions::-'-'
  { in x: ScalarValue[1]; in y: ScalarValue[0..1]; return : ScalarValue[1]; }
abstract function '*' specializes DataFunctions::'*'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '/' specializes DataFunctions::'/'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '**' specializes DataFunctions::'**'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '^' specializes DataFunctions::'^'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '%' specializes DataFunctions::'%'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }

abstract function 'not' specializes DataFunctions::'not'
  { in x: ScalarValue[1]; return : ScalarValue[1]; }
abstract function 'xor' specializes DataFunctions::'xor'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }

```

```

abstract function '~' specializes DataFunctions::~ '~'
  { in x: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '|' specializes DataFunctions::~ '|'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }
abstract function '&' specializes DataFunctions::~ '&'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : ScalarValue[1]; }

abstract function '<' specializes DataFunctions::~ '<'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }
abstract function '>' specializes DataFunctions::~ '>'
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }
abstract function '<=' specializes DataFunctions::~ '<='
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }
abstract function '>=' specializes DataFunctions::~ '>='
  { in x: ScalarValue[1]; in y: ScalarValue[1]; return : Boolean[1]; }

abstract function max specializes DataFunctions::~ Max { in x: ScalarValue[1]; in y: ScalarValue[1]; r
abstract function min specializes DataFunctions::~ Min { in x: ScalarValue[1]; in y: ScalarValue[1]; r

abstract function '..' specializes DataFunctions::~ '..' { in lower: ScalarValue[1]; in upper: ScalarV

```

## 9.4.5 Boolean Functions

### 9.4.5.1 Boolean Functions Overview

This package defines functions on Boolean values, including those corresponding to (non-conditional) logical operators in the KerML expression notation.

#### 9.4.5.2 Elements

```

function 'not' specializes ScalarFunctions::~ 'not'
  { in x: Boolean[1]; return : Boolean[1]; }
function 'xor' specializes ScalarFunctions::~ 'xor'
  { in x: Boolean[1]; in y: Boolean[1]; return : Boolean[1]; }

function '|' specializes ScalarFunctions::~ '|'
  { in x: Boolean[1]; in y: Boolean[1]; return : Boolean[1]; }
function '&' specializes ScalarFunctions::~ '&'
  { in x: Boolean[1]; in y: Boolean[1]; return : Boolean[1]; }

function '==' specializes DataFunctions::~ '=='
  { in x: Boolean[0..1]; in y: Boolean[0..1]; return : Boolean[1]; }

function ToString specializes BaseFunctions::~ ToString
  { in x: Boolean[1]; return : String[1]; }
function ToBoolean
  { in x: String[1]; return : Boolean[1]; }

```

## 9.4.6 String Functions

### 9.4.6.1 String Functions Overview

This package defines functions on String values, including those corresponding to string concatenation and comparison operators in the KerML expression notation.

#### 9.4.6.2 Elements

```

function '+' specializes ScalarFunctions::~ '+'
  { in x: String[1]; in y: String[1]; return : String[1]; }

```

```

function Length
  { in x: String[1]; return : Natural[1]; }
function Substring
  { in x: String[1]; in lower: Integer[1]; in upper: Integer[1];
    return : String[1]; }

function '<' specializes ScalarFunctions::'<'
  { in x: String[1]; in y: String[1]; return : Boolean[1]; }
function '>' specializes ScalarFunctions::'>'
  { in x: String[1]; in y: String[1]; return : Boolean[1]; }
function '<=' specializes ScalarFunctions::'<='
  { in x: String[1]; in y: String[1]; return : Boolean[1]; }
function '>=' specializes ScalarFunctions::'>='
  { in x: String[1]; in y: String[1]; return : Boolean[1]; }

function '==' specializes DataFunctions::'=='
  { in x: String[0..1]; in y: String[0..1]; return : Boolean[1]; }

function ToString specializes BaseFunctions::ToString
  { in x: String[1]; }

```

## 9.4.7 Numerical Functions

### 9.4.7.1 Numerical Functions Overview

This package defines abstract Functions on Numerical values for general arithmetic and comparison operations.

#### 9.4.7.2 Elements

```

abstract function isZero
  { in x: NumericalValue[1]; return : Boolean; }
abstract function isUnit
  { in x : NumericalValue[1]; return : Boolean; }

abstract function abs
  { in x: NumericalValue[1]; return : NumericalValue[1]; }

abstract function '+' specializes ScalarFunctions::'+'
  { in x: NumericalValue[1]; in y: NumericalValue[0..1];
    return : NumericalValue[1]; }
abstract function '-' specializes ScalarFunctions::'-'
  { in x: NumericalValue[1]; in y: NumericalValue[0..1];
    return : NumericalValue[1]; }
abstract function '*' specializes ScalarFunctions::'*'
  { in x: NumericalValue[1]; in y: NumericalValue[1];
    return : NumericalValue[1]; }
abstract function '/' specializes ScalarFunctions::'/'
  { in x: NumericalValue[1]; in y: NumericalValue[1];
    return : NumericalValue[1]; }
abstract function '**' specializes ScalarFunctions::'**'
  { in x: NumericalValue[1]; in y: NumericalValue[1];
    return : NumericalValue[1]; }
abstract function '^' specializes ScalarFunctions::'^'
  { in x: NumericalValue[1]; in y: NumericalValue[1];
    return : NumericalValue[1]; }
abstract function '%' specializes ScalarFunctions::'%'
  { in x: NumericalValue[1]; in y: NumericalValue[1];
    return : NumericalValue[1]; }

```

```

abstract function '<' specializes ScalarFunctions::'<'
  { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }
abstract function '>' specializes ScalarFunctions::'>'
  { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }
abstract function '<=' specializes ScalarFunctions::'<='
  { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }
abstract function '>=' specializes ScalarFunctions::'>='
  { in x: NumericalValue[1]; in y: NumericalValue[1]; return : Boolean[1]; }

abstract function max specializes ScalarFunctions::max
  { in x: NumericalValue[1]; in y: NumericalValue[1];
    return : NumericalValue[1]; }
abstract function min specializes ScalarFunctions::min
  { in x: NumericalValue[1]; in y: NumericalValue[1];
    return : NumericalValue[1]; }

abstract function sum
  { in collection: ScalarValue[0..*]; return : ScalarValue[1]; }
abstract function product
  { in collection: ScalarValue[0..*]; return : ScalarValue[1]; }

```

## 9.4.8 Complex Functions

### 9.4.8.1 Complex Functions Overview

This package defines Functions on Complex values, including concrete specializations of the general arithmetic and comparison operations.

#### 9.4.8.2 Elements

```

feature i: Complex[1] = rect(0.0, 1.0);

function rect
  { in re: Real[1]; in im: Real[1]; return : Complex[1]; }
function polar
  { in abs: Real[1]; in arg: Real[1]; return : Complex[1]; }

function re
  { in x: Complex[1]; return : Real[1]; }
function im
  { in x: Complex[1]; return : Real[1]; }

function isZero specializes NumericalFunctions::isZero
  { in x: Complex[1]; return : Boolean[1]; }
function isUnit specializes NumericalFunctions::isUnit
  { in x: Complex[1]; return : Boolean[1]; }

function abs specializes NumericalFunctions::abs
  { in x: Complex[1]; return : Real[1]; }
function arg
  { in x: Complex[1]; return : Real[1]; }

function '+' specializes NumericalFunctions::'+'
  { in x: Complex[1]; in y: Complex[0..1]; return : Complex[1]; }
function '-' specializes NumericalFunctions::'-'
  { in x: Complex[1]; in y: Complex[0..1]; return : Complex[1]; }
function '*' specializes NumericalFunctions::'*'
  { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }
function '/' specializes NumericalFunctions::'/'
  { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }

```

```

function '**' specializes NumericalFunctions::'**'
  { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }
function '^' specializes NumericalFunctions::'^'
  { in x: Complex[1]; in y: Complex[1]; return : Complex[1]; }

function '==' specializes DataFunctions::'=='
  { in x: Complex[0..1]; in y: Complex[0..1]; return : Boolean[1]; }

function ToString specializes BaseFunctions::ToString
  { in x: Complex[1]; return : String[1]; }
function ToComplex
  { in x: String[1]; return : Complex[1]; }

function sum specializes NumericalFunctions::sum
  { in collection: Complex[0..*]; return : Complex[1]; }

function product specializes NumericalFunctions::product
  { in collection: Complex[0..*]; return : Complex[1]; }

```

## 9.4.9 Real Functions

### 9.4.9.1 Real Functions Overview

This package defines Functions on Real values, including concrete specializations of the general arithmetic and comparison operations.

### 9.4.9.2 Elements

```

function re :> ComplexFunctions::re
  { in x: Real[1]; return : Real[1] = x; }
function im :> ComplexFunctions::im
  { in x: Real[1]; return : Real[1] = 0.0; }

function abs specializes ComplexFunctions::abs
  { in x: Real[1]; return : Real[1]; }
function arg specializes ComplexFunctions::arg
  { in x: Real[1]; return : Real[1] = 0.0; }

function '+' specializes ComplexFunctions::'+'
  { in x: Real[1]; in y: Real[0..1]; return : Real[1]; }
function '-' specializes ComplexFunctions::'-'
  { in x: Real[1]; in y: Real[0..1]; return : Real[1]; }
function '*' specializes ComplexFunctions::'*'
  { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function '/' specializes ComplexFunctions::'/'
  { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function '**' specializes ComplexFunctions::'**'
  { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function '^' specializes ComplexFunctions::'^'
  { in x: Real[1]; in y: Real[1]; return : Real[1]; }

function '<' specializes NumericalFunctions::'<'
  { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }
function '>' specializes NumericalFunctions::'>'
  { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }
function '<=' specializes NumericalFunctions::'<='
  { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }
function '>=' specializes NumericalFunctions::'>='
  { in x: Real[1]; in y: Real[1]; return : Boolean[1]; }

```

```

function max specializes NumericalFunctions::max
  { in x: Real[1]; in y: Real[1]; return : Real[1]; }
function min specializes NumericalFunctions::min
  { in x: Real[1]; in y: Real[1]; return : Real[1]; }

function '=' specializes ComplexFunctions::'=='
  { in x: Real[0..1]; in y: Real[0..1]; return : Boolean[1]; }

function sqrt
  { in x: Real[1]; return : Real[1]; }

function floor
  { in x: Real[1]; return : Integer[1]; }
function round
  { in x: Real[1]; return : Integer[1]; }

function ToString specializes ComplexFunctions::ToString
  { in x: Real[1]; return : String[1]; }
function ToInteger
  { in x: Real[1]; return : Integer[1]; }
function ToRational
  { in x: Real[1]; return : Rational[1]; }
function ToReal
  { in x: String[1]; return : Real[1]; }

function sum specializes ComplexFunctions::sum
  { in collection: Real[0..*]; return : Real; }

function product specializes ComplexFunctions::product
  { in collection: Real[0..*]; return : Real; }

```

## 9.4.10 Rational Functions

### 9.4.10.1 Rational Functions Overview

This package defines Functions on Rational values, including concrete specializations of the general arithmetic and comparison operations.

### 9.4.10.2 Elements

```

function rat
  { in numer: Integer[1]; in denum: Integer[1]; return : Rational[1]; }
function numer
  { in rat: Rational[1]; return : Integer[1]; }
function denom
  { in rat: Rational[1]; return : Integer[1]; }

function abs specializes RealFunctions::abs
  { in x: Rational[1]; return : Rational[1]; }

function '+' specializes RealFunctions::'+'
  { in x: Rational[1]; in y: Rational[0..1]; return : Rational[1]; }
function '-' specializes RealFunctions::'-'
  { in x: Rational[1]; in y: Rational[0..1]; return : Rational[1]; }
function '*' specializes RealFunctions::'*'
  { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }
function '/' specializes RealFunctions::'/'
  { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }
function '**' specializes RealFunctions::'**'
  { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }

```



```

function '^' specializes RealFunctions::'^'
  { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }

function '<' specializes RealFunctions::'<'
  { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }
function '>' specializes RealFunctions::'>'
  { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }
function '<=' specializes RealFunctions::'<='
  { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }
function '>=' specializes RealFunctions::'>='
  { in x: Rational[1]; in y: Rational[1]; return : Boolean[1]; }

function max specializes RealFunctions::max
  { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }
function min specializes RealFunctions::min
  { in x: Rational[1]; in y: Rational[1]; return : Rational[1]; }

function '==' specializes RealFunctions::'=='
  { in x: Rational[0..1]; in y: Rational[0..1]; return : Boolean[1]; }

function gcd
  { in x: Rational[1]; in y: Rational[1]; return : Integer[1]; }

function floor specializes RealFunctions::floor
  { in x: Rational[1]; return : Integer[1]; }
function round specializes RealFunctions::round
  { in x: Rational[1]; return : Integer[1]; }

function ToString specializes RealFunctions::ToString
  { in x: Rational[1]; return : String[1]; }
function ToInteger
  { in x: Rational[1]; return : Integer[1]; }
function ToRational
  { in x: String[1]; return : Rational[1]; }

function sum specializes RealFunctions::sum
  { in collection: Rational[0..*]; return : Rational[1]; }

function product specializes RealFunctions::product
  { in collection: Rational[0..*]; return : Rational[1]; }

```

## 9.4.11 Integer Functions

### 9.4.11.1 Integer Functions Overview

This package defines Functions on Integer values, including concrete specializations of the general arithmetic and comparison operations.

### 9.4.11.2 Elements

```

function abs specializes RationalFunctions::abs
  { in x: Integer[1]; return : Natural[1]; }

function '+' specializes RationalFunctions::'+'
  { in x: Integer[1]; in y: Integer[0..1]; return : Integer[1]; }
function '-' specializes RationalFunctions::'-'
  { in x: Integer[1]; in y: Integer[0..1]; return : Integer[1]; }
function '*' specializes RationalFunctions::'*'
  { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }
function '/' specializes RationalFunctions::'/'

```

```

    { in x: Integer[1]; in y: Integer[1]; return : Rational[1]; }
function '*' specializes RationalFunctions::'*'
    { in x: Integer[1]; in y: Natural[1]; return : Integer[1]; }
function '^' specializes RationalFunctions::'^'
    { in x: Integer[1]; in y: Natural[1]; return : Integer[1]; }
function '%' specializes NumericalFunctions::'%'
    { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }

function '<' specializes RationalFunctions::'<'
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }
function '>' specializes RationalFunctions::'>'
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }
function '<=' specializes RationalFunctions::'<='
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }
function '>=' specializes RationalFunctions::'>='
    { in x: Integer[1]; in y: Integer[1]; return : Boolean[1]; }

function max specializes RationalFunctions::max
    { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }
function min specializes RationalFunctions::min
    { in x: Integer[1]; in y: Integer[1]; return : Integer[1]; }

function '==' specializes DataFunctions::'=='
    { in x: Integer[0..1]; in y: Integer[0..1]; return : Boolean[1]; }

function '..' specializes ScalarFunctions::'..'
    { in lower: Integer[1]; in upper: Integer[1]; return : Integer[0..*]; }

function ToString specializes RationalFunctions::ToString
    { in x: Integer[1]; return : String[1]; }
function ToNatural
    { in x: Integer[1]; return : Natural[1]; }
function ToInteger
    { in x: String[1]; return : Integer[1]; }

function sum specializes RationalFunctions::sum
    { in collection: Integer[0..*]; return : Integer[1]; }

function product specializes RationalFunctions::product
    { in collection: Integer[0..*]; return : Integer[1]; }

```

## 9.4.12 Natural Functions

### 9.4.12.1 Natural Functions Overview

This package defines Functions on Natural values, including concrete specializations of the general arithmetic and comparison operations.

### 9.4.12.2 Elements

```

function '+' specializes IntegerFunctions::'+'
    { in x: Natural[1]; in y: Natural[0..1]; return : Natural[1]; }
function '*' specializes IntegerFunctions::'*'
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }
function '/' specializes IntegerFunctions::'/'
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }
function '%' specializes IntegerFunctions::'%'
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }

function '<' specializes IntegerFunctions::'<'

```

```

    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }
function '>' specializes IntegerFunctions::'>'
    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }
function '<=' specializes IntegerFunctions::'<='
    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }
function '>=' specializes IntegerFunctions::'>='
    { in x: Natural[1]; in y: Natural[1]; return : Boolean[1]; }

function max specializes IntegerFunctions::max
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }
function min specializes IntegerFunctions::min
    { in x: Natural[1]; in y: Natural[1]; return : Natural[1]; }

function '==' specializes IntegerFunctions::'=='
    { in x: Natural[0..1]; in y: Natural[0..1]; return : Boolean[1]; }

function ToString specializes IntegerFunctions::ToString
    { in x: Natural[1]; return : String[1]; }
function ToNatural
    { in x: String[1]; return : Natural[1]; }

```

## 9.4.13 Trig Functions

### 9.4.13.1 Trig Functions Overview

This package defines basic trigonometric functions on real numbers.

### 9.4.13.2 Elements

```

feature pi : Real;
inv piPrecision { RealFunctions::round(pi * 1E20) == 314159265358979323846.0 }

function deg {
    in theta_rad : Real[1];
    return : Real[1] = theta_rad * 180 / pi;
}
function rad {
    in theta_deg : Real;
    return : Real[1] = theta_deg * pi / 180;
}

datatype UnitBoundedReal :> Real {
    inv unitBound { -1.0 <= that & that <= 1.0 }
}

function sin {
    in theta : Real[1];
    return : UnitBoundedReal[1];
}
function cos {
    in theta : Real[1];
    return : UnitBoundedReal[1];
}
function tan {
    in theta : Real[1];
    return : Real = sin(theta) / cos(theta);
}
function cot {
    in theta : Real;

```

```

    return : Real = cos(theta) / sin(theta);
}

function arcsin {
    in x : UnitBoundedReal[1];
    return : Real[1];
}
function arccos {
    in x : UnitBoundedReal[1];
    return : Real[1];
}
function arctan {
    in x : Real[1];
    return : Real[1];
}

```

## 9.4.14 Sequence Functions

### 9.4.14.1 Sequence Functions Overview

This package defines Functions that operate on general sequences of values. (For Functions that operate on Collection values, see CollectionFunctions.)

### 9.4.14.2 Elements

```

function equals{
    in x: Anything[0..*] ordered nonunique;
    in y: Anything[0..*] ordered nonunique;
    return : Boolean[1];
}

function size{
    in seq: Anything[0..*] nonunique;
    return : Natural[1];
}
function isEmpty{
    in seq: Anything[0..*] nonunique;
    return : Boolean[1];
}
function notEmpty{
    in seq: Anything[0..*] nonunique;
    return : Boolean[1];
}
function includes{
    in seq1: Anything[0..*] nonunique;
    in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
}
function includesOnly{
    in seq1: Anything[0..*] nonunique;
    in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
}
function excludes{
    in seq1: Anything[0..*] nonunique;
    in seq2: Anything[0..*] nonunique;
    return : Boolean[1];
}

function union{

```

```

    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}
function intersection{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}
function including{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}
function excluding{
    in seq1: Anything[0..*] ordered nonunique;
    in seq2: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}

function subsequence{
    in seq: Anything[0..*] ordered nonunique;
    in startIndex: Positive[1];
    in endIndex: Positive[1];
    return : Anything[0..*];
}
function head{
    in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..1] = seq[1];
}
function tail{
    in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..*] ordered nonunique;
}
function last{
    in seq: Anything[0..*] ordered nonunique;
    return : Anything[0..1];
}

function '[' specializes BaseFunctions::'[' {
    in seq: Anything[0..*] ordered nonunique;
    in index: Positive[1];
    return : Anything[0..1];
}

```

## 9.4.15 Collection Functions

### 9.4.15.1 Collection Functions Overview

This package defines Functions on Collections (as defined in the Collections package). For Functions on general sequences of values, see the SequenceFunctions package.

### 9.4.15.2 Elements

```

function '==' specializes BaseFunctions::'==' {
    in col1: Collection[0..1];
    in col2: Collection[0..1];
    return : Boolean[1];
}

```

```

function size {
  in col: Collection[1];
  return : Natural[1];
}

function isEmpty {
  in col: Collection[1];
  return : Boolean[1];
}

function notEmpty {
  in col: Collection[1];
  return : Boolean[1];
}

function contains {
  in col: Collection[1];
  in values: Anything[*];
  return : Boolean[1];
}

function containsAll {
  in col1: Collection[1];
  in col2: Collection[2];
  return : Boolean[1];
}

function head {
  in col: OrderedCollection[1];
  return : Anything[0..1];
}

function tail {
  in col: OrderedCollection[1];
  return : Anything[0..*] ordered nonunique;
}

function last {
  in col: OrderedCollection[1];
  return : Anything[0..1];
}

function '[' specializes BaseFunctions:: '[' {
  in col: OrderedCollection[1];
  in index: Positive[1];
  return : Anything[0..1];
}

function 'array[' specializes BaseFunctions:: '[' {
  in arr: Array[1];
  in indexes: Positive[n] ordered nonunique;
  return : Anything[0..1];
}

```

## 9.4.16 Vector Functions

### 9.4.16.1 Vector Functions Overview

This package defines abstract functions on *VectorValues* corresponding to the algebraic operations provided by a vector space with inner product. It also includes concrete implementations of these functions specifically for *CartesianVectorValues*.

### 9.4.16.2 Elements

```
abstract function isZeroVector {
  doc
  /*
   * Return whether a VectorValue is a zero vector.
   */

  in v: VectorValue[1];
  return : Boolean[1];
}

abstract function '+' specializes DataFunctions::'+' {
  doc
  /*
   * With two arguments, returns the sum of two VectorValues.
   * With one argument, returns that VectorValue.
   */

  in v: VectorValue[1];
  in w: VectorValue[0..1];
  return u: VectorValue[1];
  inv zeroAddition { w == null or isZeroVector(w) implies u == w }
  inv commutivity { w != null implies u == w + v }
}

abstract function '-' specializes DataFunctions::-'-' {
  doc
  /*
   * With two arguments, returns the difference of two VectorValues.
   * With one arguments, returns the inverse
   * of the given VectorValue, that is, the VectorValue that,
   * when added to the original VectorValue, results in
   * the zeroVector.
   */

  in v: VectorValue[1];
  in w: VectorValue[0..1];
  return u: VectorValue[1];
  inv negation { w == null implies isZeroVector(v + u) }
  inv difference { w != null implies v + u == w }
}

abstract function sum0 {
  doc
  /*
   * Return the sum of a collection of VectorValues.
   * If the collection is empty, return a given zero vector.
   */

  in coll: VectorValue[*] nonunique;
  in zero: VectorValue[1];
  inv precondition { isZeroVector(zero) }
```

```

    return s: VectorValue[1] = coll->reduce '+' ?? zero;
}

/* Functions specific to NumericalVectorValues. */

function VectorOf {
  doc
  /*
   * Construct a NumericalVectorValue whose elements are a
   * non-empty list of component NumericalValues.
   * The dimension of the NumericalVectorValue is equal to
   * the number of components.
   */

  in components: NumericalValue[1..*] ordered nonunique;
  return : NumericalVectorValue[1] {
    :>> dimension = size(components);
    :>> elements = components;
  }
}

abstract function scalarVectorMult specializes DataFunctions::'*' {
  doc
  /*
   * Scalar product of a NumericalValue and a NumericalVectorValue.
   */

  in x: NumericalValue[1];
  in v: NumericalVectorValue[1];
  return w: NumericalVectorValue[1];
  inv scaling { norm(w) == x * norm(v) }
  inv zeroLength { isZeroVector(w) implies isZero(norm(w)) }
}
alias '*' for scalarVectorMult;

abstract function vectorScalarMult specializes DataFunctions::'*' {
  doc
  /*
   * Scalar product of a NumericalVectorValue and a NumericalValue,
   * which has the same value as the scalar product of the
   * NumericalValue and the NumericalVectorValue.
   */

  in v: NumericalVectorValue[1];
  in x: NumericalValue[1];
  return w: NumericalVectorValue[1] = scalarVectorMult(x, v);
}

abstract function vectorScalarDiv specializes DataFunctions:: '/' {
  doc
  /*
   * Scalar quotient of a NumericalVectorValue and a NumericalValue,
   * defined as the scalar product of the inverse of the
   * NumericalValue and the NumericalVectorValue.
   */

  in v: NumericalVectorValue[1];
  in x: NumericalValue[1];
  return w: NumericalVectorValue[1] = scalarVectorMult(1.0 / x, v);
}

```



```

abstract function inner specializes DataFunctions::'*' {
  doc
  /*
   * Inner product of two NumericalVectorValues.
   */

  in v: NumericalVectorValue[1];
  in w: NumericalVectorValue[1];
  return x: NumericalValue[1];
  inv commutivity { x == inner(w, v) }
  inv zeroInner { isZeroVector(v) or isZeroVector(w) implies isZero(x) }
}

abstract function norm {
  doc
  /*
   * The norm (magnitude) of a NumericalVectorValue, as a NumericalValue.
   */

  in v: NumericalVectorValue[1];
  return l : NumericalValue[1];
  inv squareNorm { l * l == inner(v,v) }
  inv lengthZero { isZero(l) == isZeroVector(v) }
}

abstract function angle {
  doc
  /*
   * The angle between two NumericalVectorValues, as a NumericalValue.
   */

  in v: NumericalVectorValue[1];
  in w: NumericalVectorValue[1];
  return theta: NumericalValue[1];
  inv commutivity { theta == angle(w, v) }
  inv lengthInsensitive { theta == angle(w / norm(w), v / norm(v)) }
}

/* Specialized functions with concrete definitions for CartesianVectorValues. */

function CartesianVectorOf {
  doc
  /*
   * Construct a CartesianVectorValue whose elements are
   * a non-empty list of Real components.
   * The dimension of the NumericalVectorValue is equal
   * to the number of components.
   */

  in components: Real[*] ordered nonunique;
  return : CartesianVectorValue[1] {
    :>> dimension = size(components);
    :>> elements = components;
  }
}

function CartesianThreeVectorOf specializes CartesianVectorOf {
  in components: Real[3] ordered nonunique;
  return : CartesianThreeVectorValue[1];
}

```

```

feature cartesianZeroVector: CartesianVectorValue[3] =
  (
    CartesianVectorOf(0.0),
    CartesianVectorOf((0.0, 0.0)),
    CartesianThreeVectorOf((0.0, 0.0, 0.0))
  ) {
  doc
  /*
  * Cartesian zero vectors of 1, 2 and 3 dimensions.
  */
}
feature cartesian3DZeroVector: CartesianThreeVectorValue[1] =
  cartesianZeroVector[3];

function isCartesianZeroVector specializes isZeroVector {
  doc
  /*
  * A CartesianVectorValue is a zero vector if all its elements are zero.
  */

  in v: CartesianVectorValue[1];
  return : Boolean[1] = v.elements->forall{in x; x == 0.0};
}

function 'cartesian+' specializes '+' {
  in v: CartesianVectorValue[1];
  in w: CartesianVectorValue[0..1];
  inv precondition { w != null implies v.dimension == w.dimension }
  return u: CartesianVectorValue[1] =
    if w == null? v
    else CartesianVectorOf(
      (1..w.dimension)->collect{in i : Positive; v[i] + w[i]}
    );
}

function 'cartesian-' specializes '-' {
  in v: CartesianVectorValue[1];
  in w: CartesianVectorValue[0..1];
  inv precondition { w != null implies v.dimension == w.dimension }
  return u: CartesianVectorValue[1] =
    CartesianVectorOf(
      if w == null?
        CartesianVectorOf(v.elements->collect{in x : Real; -x})
      else CartesianVectorOf(
        (1..v.dimension)->collect{in i : Positive; v[i] - w[i]}
      )
    );
}

function cartesianScalarVectorMult specializes scalarVectorMult {
  in x: Real[1];
  in v: CartesianVectorValue[1];
  return w: CartesianVectorValue[1] =
    CartesianVectorOf(
      v.elements->collect{in y : Real; x * y}
    );
}

function cartesianVectorScalarMult specializes vectorScalarMult {
  in v: CartesianVectorValue[1];

```

```

    in x: Real[1];
    return w: CartesianVectorValue[1] = cartesianScalarVectorMult(x, v);
}

function cartesianInner specializes inner {
    in v: CartesianVectorValue[1];
    in w : CartesianVectorValue[1];
    inv precondition { v.dimension == w.dimension }
    return x: Real[1] =
        (1..v.dimension)->collect{in i : Positive; v[i] * w[i]}->reduce RealFunctions::'+';
}

function cartesianNorm specializes norm {
    in v: CartesianVectorValue[1];
    return l : NumericalValue[1] = sqrt(cartesianInner(v, v));
}

function cartesianAngle specializes angle {
    in v: CartesianVectorValue[1]; in w: CartesianVectorValue[1];
    inv precondition { v.dimension == w.dimension }
    return theta: Real[1] = arccos(cartesianInner(v, w) / (norm(v) * norm(w)));
}

function sum {
    in coll: CartesianThreeVectorValue[*];
    return : CartesianThreeVectorValue[1] = sum0(coll, cartesian3DZeroVector);
}

```

## 9.4.17 Control Functions

### 9.4.17.1 Control Functions Overview

This package defines Functions that correspond to operators in the KerML expression notation for which one or more operands are Expressions whose evaluation is determined by another operand.

### 9.4.17.2 Elements

```

abstract function '.' {
    in feature source : Anything[0..*] nonunique {
        abstract feature target : Anything[0..*] nonunique;
    }
    private feature chain chains source.target;
    chain
}

abstract function 'if' {
    in test: Boolean[1];
    in expr thenValue[0..1] { return : Anything[0..*] ordered nonunique; }
    in expr elseValue[0..1] { return : Anything[0..*] ordered nonunique; }
    return : Anything[0..*] ordered nonunique;
}

abstract function '??' {
    in firstValue: Anything[0..*] ordered nonunique;
    in expr secondValue[0..1] { return : Anything[0..*] ordered nonunique; }
    return : Anything[0..*] ordered nonunique;
}

function 'and' {
    in firstValue: Boolean[1];

```

```

    in expr secondValue[0..1] { return : Boolean[1]; }
    return : Boolean[1];
}

function 'or'{
    in firstValue: Boolean[1];
    in expr secondValue[0..1] { return : Boolean[1]; }
    return : Boolean[1];
}

function 'implies'{
    in firstValue: Boolean[1];
    in expr secondValue[0..1] { return : Boolean[1]; }
    return : Boolean[1];
}

abstract function collect {
    in collection: Anything[0..*] ordered nonunique;
    in expr mapper[0..*] {
        in argument: Anything[1];
        return : Anything[0..*] ordered nonunique;
    }
    return : Anything[0..*] ordered nonunique;
}

abstract function select {
    in collection: Anything[0..*] ordered nonunique;
    in expr selector[0..*] {
        in argument: Anything[1];
        return : Boolean[1];
    }
    return : Anything[0..*] ordered nonunique;
}

function selectOne {
    in collection: Anything[0..*] ordered nonunique;
    in expr selector1[0..*] {
        in argument: Anything[1];
        return : Boolean[1]; }
    return : Anything[0..1] =
        collection->select {in x; selector1(x)}[1];
}

abstract function reject{
    in collection: Anything[0..*] ordered nonunique;
    in expr rejector[0..*] {
        in argument: Anything[1];
        return : Boolean[1];
    }
    return : Anything[0..*] ordered nonunique;
}

abstract function reduce {
    in collection: Anything[0..*] ordered nonunique;
    in expr reducer[0..*] {
        in firstArg: Anything[1];
        in secondArg: Anything[1];
        return : Anything[1];
    }
    return : Anything[0..*] ordered nonunique;
}

```

```

}

abstract function forAll {
  in collection: Anything[0..*] ordered nonunique;
  in expr test[0..*] {
    in argument: Anything[1];
    return : Boolean[1];
  }
  return : Boolean[1];
}

abstract function exists {
  in collection: Anything[0..*] ordered nonunique;
  in expr test[0..*] {
    in argument: Anything[1];
    return : Boolean[1];
  }
  return : Boolean[1];
}

function allTrue {
  in collection: Boolean[0..*];
  return : Boolean[1] = collection->forAll {in x; x};
}

function anyTrue {
  in collection: Boolean[0..*];
  return : Boolean[1] = collection->exists {in x; x};
}

function minimize {
  in collection: ScalarValue[1..*];
  in expr fn[0..*] {
    in argument: ScalarValue[1];
    return : ScalarValue[1];
  }
  return : ScalarValue[1] =
    collection->collect {in x; fn(x)}->reduce min;
}

function maximize {
  in collection: ScalarValue[1..*];
  in expr fn[0..*] {
    in argument: ScalarValue[1];
    return : ScalarValue[1];
  }
  return : ScalarValue =
    collection->collect {in x; fn(x)}->reduce max;
}

```



# 10 Model Interchange

KerML models may be interchanged between conformant KerML modeling tools (see [Clause 2](#)) using text files in any of the following formats:

1. Textual notation, using the textual concrete syntax defined in this specification. Note that in certain limited cases, models conformant with the KerML syntax, but prepared by a means other than using the KerML textual concrete syntax, may not be fully serializable into the standard textual notation. In this case, a tool may either not export such model at all using the textual notation, or export the model as closely as possible, informing the user of any changes from the original model.
2. JSON, using a format consistent with the JSON schema based on the KerML abstract syntax, consistent with the REST/HTTP platform-specific binding of the Element Navigation Service of the Systems Modeling API and Services specification [SysAPI].
3. XML, using the XML Metadata Interchange [XMI] format based on the MOF-conformant abstract syntax metamodel for KerML.

Every conformant KerML modeling tool shall provide the ability to import and/or export (as appropriate) models in at least one of the first two formats.

**Release Note.** Model interchange will be addressed more fully in the final submission. Issues to be addressed include interchanging tool-generated metadata (such as Element identifiers) in the textual notation and full documentation of the JSON format.





# A Annex: Conformance Test Suite

**Release Note.** The conformance test suite will be provided in the final submission.