



Date: May 2021



OMG Systems Modeling Language™ (SysML®)

Version 2.0

Release 2021-04

**Submitted in response to Systems Modeling Language (SysML®) v2 RFP (ad/
2017-11-04) by:**

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

Copyright © 2019-2021, 88Solutions Corporation
Copyright © 2019-2021, Airbus
Copyright © 2019-2021, Aras Corporation
Copyright © 2019-2021, Association of Universities for Research in Astronomy (AURA)
Copyright © 2019-2021, BigLever Software
Copyright © 2019-2021, Boeing
Copyright © 2019-2021, Contact Software GmbH
Copyright © 2019-2021, Dassault Systèmes (No Magic)
Copyright © 2019-2021, DSC Corporation
Copyright © 2020-2021, DEKonsult
Copyright © 2020-2021, Delligatti Associates, LLC
Copyright © 2019-2021, The Charles Stark Draper Laboratory, Inc.
Copyright © 2020-2021, ESTACA
Copyright © 2019-2021, GfSE e.V.
Copyright © 2019-2021, George Mason University
Copyright © 2019-2021, IBM
Copyright © 2019-2021, Idaho National Laboratory
Copyright © 2019-2021, InterCax LLC
Copyright © 2019-2021, Jet Propulsion Laboratory (California Institute of Technology)
Copyright © 2019-2021, Kenntnis LLC
Copyright © 2020-2021, Kungliga Tekniska högskolan (KTH)
Copyright © 2019-2021, LightStreet Consulting LLC
Copyright © 2019-2021, Lockheed Martin Corporation
Copyright © 2019-2021, Maplesoft
Copyright © 2021, MID GmbH
Copyright © 2020-2021, MITRE
Copyright © 2019-2021, Model Alchemy Consulting
Copyright © 2019-2021, Model Driven Solutions, Inc.
Copyright © 2019-2021, Model Foundry Pty. Ltd.
Copyright © 2019-2021, On-Line Application Research Corporation (OAC)
Copyright © 2019-2021, oose Innovative Informatik eG
Copyright © 2019-2021, Østfold University College
Copyright © 2019-2021, PTC
Copyright © 2020-2021, Qualtech Systems, Inc.
Copyright © 2019-2021, SAF Consulting
Copyright © 2019-2021, Simula Research Laboratory AS
Copyright © 2019-2021, System Strategy, Inc.
Copyright © 2019-2021, Thematix
Copyright © 2019-2021, Tom Sawyer
Copyright © 2019-2021, Universidad de Cantabria
Copyright © 2019-2021, University of Alabama in Huntsville
Copyright © 2019-2021, University of Detroit Mercy
Copyright © 2019-2021, University of Kaiserslautern
Copyright © 2020-2021, 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 Requirements Tables	2
0.4.1 Mandatory Language Requirements Table	2
0.4.2 Non-Mandatory Language Requirements Table	31
0.4.3 Mandatory Language Requirements - Satisfied-by Table	40
0.4.4 Non-Mandatory Language Requirements - Satisfied-by Table	51
0.4.5 Changed Language Requirements Table	54
1 Scope	59
2 Conformance	61
3 Normative References	63
4 Terms and Definitions	65
5 Symbols	67
6 Introduction	69
6.1 Language Overview	69
6.2 Document Conventions	69
6.3 Document Organization	70
6.4 Acknowledgements	71
7 Metamodel	73
7.1 Overview	73
7.2 Kernel	74
7.2.1 Basic Elements	74
7.2.1.1 Basic Elements Overview	74
7.2.1.2 Basic Elements Concrete Syntax	75
7.2.1.2.1 Basic Elements Textual Notation	75
7.2.1.2.2 Basic Elements Graphical Notation	75
7.2.1.3 Basic Elements Abstract Syntax	76
7.2.2 Annotations	76
7.2.2.1 Annotations Overview	76
7.2.2.2 Annotations Concrete Syntax	77
7.2.2.2.1 Annotations Textual Notation	77
7.2.2.2.1.1 Comments	77
7.2.2.2.1.2 Documentation	78
7.2.2.2.1.3 Textual Representation	78
7.2.2.2.1.4 Annotating Features	79
7.2.2.2.2 Annotations Graphical Notation	80
7.2.2.3 Annotations Abstract Syntax	82
7.2.3 Packages	83
7.2.3.1 Packages Overview	83
7.2.3.2 Packages Concrete Syntax	84
7.2.3.2.1 Packages Textual Notation	84
7.2.3.2.1.1 Packages	85
7.2.3.2.1.2 Package Elements	88
7.2.3.2.2 Packages Graphical Notation	90
7.2.3.2.2.1 Packages	91
7.2.3.2.2.2 Memberships	93
7.2.3.3 Packages Abstract Syntax	94
7.2.4 Types	94
7.2.4.1 Types Overview	95

7.2.4.2 Types Concrete Syntax.....	95
7.2.4.2.1 Types Textual Notation	95
7.2.4.2.1.1 Types	96
7.2.4.2.1.2 Feature Membership.....	97
7.2.4.2.2 Types Graphical Notation	97
7.2.4.2.2.1 Types	97
7.2.4.2.2.2 Feature Membership.....	98
7.2.4.3 Types Abstract Syntax	99
7.2.5 Classifiers.....	100
7.2.5.1 Classifiers Overview	100
7.2.5.2 Classifiers Concrete Syntax	101
7.2.5.2.1 Classifiers Textual Notation.....	101
7.2.5.2.1.1 Classifiers	101
7.2.5.2.1.2 Superclassing.....	101
7.2.5.2.2 Classifiers Graphical Notation	101
7.2.5.2.2.1 Classifiers	101
7.2.5.2.2.2 Superclassing.....	102
7.2.5.3 Classifiers Abstract Syntax	102
7.2.6 Features	103
7.2.6.1 Features Overview.....	103
7.2.6.2 Features Concrete Syntax.....	104
7.2.6.2.1 Features Textual Notation	104
7.2.6.2.1.1 Features.....	104
7.2.6.2.1.2 Feature Specialization	105
7.2.6.2.2 Features Graphical Notation.....	105
7.2.6.2.2.1 Features.....	106
7.2.6.2.2.2 Feature Specialization	107
7.2.6.3 Features Abstract Syntax.....	107
7.2.7 Associations	109
7.2.7.1 Associations Overview.....	109
7.2.7.2 Associations Concrete Syntax.....	109
7.2.7.3 Associations Abstract Syntax	110
7.2.8 Connectors.....	110
7.2.8.1 Connectors Overview.....	110
7.2.8.2 Connectors Concrete Syntax	111
7.2.8.2.1 Connectors Textual Notation.....	111
7.2.8.2.1.1 Connectors	111
7.2.8.2.1.2 Binding Connectors	112
7.2.8.2.1.3 Successions.....	112
7.2.8.2.2 Connectors Graphical Notation	112
7.2.8.2.2.1 Binding Connectors	112
7.2.8.2.2.2 Successions.....	113
7.2.8.3 Connectors Abstract Syntax	114
7.2.9 Behaviors.....	115
7.2.9.1 Behaviors Overview	115
7.2.9.2 Behaviors Concrete Syntax	115
7.2.9.3 Behaviors Abstract Syntax	116
7.2.10 Interactions.....	116
7.2.10.1 Interactions Overview	116
7.2.10.2 Interactions Concrete Syntax	116
7.2.10.2.1 Interactions Textual Notation	116
7.2.10.2.1.1 Interactions	117
7.2.10.2.1.2 Item Flows.....	117

7.2.10.2.2 Interactions Graphical Notation	118
7.2.10.2.2.1 Interactions	118
7.2.10.2.2.2 Item Flows	118
7.2.10.3 Interactions Abstract Syntax	118
7.2.11 Functions	119
7.2.11.1 Functions Overview	119
7.2.11.2 Functions Concrete Syntax.....	120
7.2.11.3 Functions Abstract Syntax	120
7.2.12 Expressions	121
7.2.12.1 Expressions Overview	121
7.2.12.2 Expressions Concrete Syntax	122
7.2.12.3 Expressions Abstract Syntax	124
7.3 Dependencies	125
7.3.1 Dependencies Overview.....	125
7.3.2 Dependencies Concrete Syntax	125
7.3.2.1 Dependencies Textual Notation	125
7.3.2.2 Dependencies Graphical Notation.....	126
7.3.3 Dependencies Abstract Syntax.....	126
7.3.3.1 Overview	127
7.3.3.2 Dependency	127
7.4 Definition and Usage	127
7.4.1 Definition and Usage Overview.....	128
7.4.2 Definition and Usage Concrete Syntax	130
7.4.2.1 Definition and Usage Textual Notation	130
7.4.2.1.1 Definitions	131
7.4.2.1.2 Usages	134
7.4.2.1.3 Reference Usages	136
7.4.2.1.4 Body Elements	137
7.4.2.2 Definition and Usage Graphical Notation	139
7.4.2.2.1 Definitions	139
7.4.2.2.2 Usages	140
7.4.2.2.3 Compartments	141
7.4.2.2.4 Boundary Features.....	142
7.4.2.2.5 Variant Memberships	143
7.4.3 Definition and Usage Abstract Syntax	143
7.4.3.1 Overview	143
7.4.3.2 Definition	144
7.4.3.3 ReferenceUsage.....	147
7.4.3.4 Usage.....	148
7.4.3.5 VariantMembership.....	151
7.4.4 Definition and Usage Semantics	152
7.5 Attributes.....	152
7.5.1 Attributes Overview	152
7.5.2 Attributes Concrete Syntax	152
7.5.2.1 Attributes Textual Notation.....	152
7.5.2.2 Attributes Graphical Notation	153
7.5.3 Attributes Abstract Syntax	153
7.5.3.1 Overview	153
7.5.3.2 AttributeUsage	153
7.5.3.3 AttributeDefinition	154
7.5.4 Attributes Semantics	154
7.6 Enumerations	154
7.6.1 Enumerations Overview.....	154

7.6.2 Enumerations Concrete Syntax	155
7.6.2.1 Enumerations Textual Notation	155
7.6.2.2 Enumerations Graphical Notation	156
7.6.3 Enumerations Abstract Syntax	156
7.6.3.1 Overview	157
7.6.3.2 EnumerationDefinition	157
7.6.3.3 EnumerationUsage	158
7.6.4 Enumerations Semantics	158
7.7 Items	158
7.7.1 Items Overview	158
7.7.2 Items Concrete Syntax	158
7.7.2.1 Items Textual Notation	159
7.7.2.2 Items Graphical Notation	159
7.7.3 Items Abstract Syntax	159
7.7.3.1 Overview	160
7.7.3.2 ItemDefinition	160
7.7.3.3 ItemUsage	160
7.7.4 Items Semantics	161
7.8 Parts	161
7.8.1 Parts Overview	161
7.8.2 Parts Concrete Syntax	161
7.8.2.1 Parts Textual Notation	162
7.8.2.2 Parts Graphical Notation	162
7.8.3 Parts Abstract Syntax	162
7.8.3.1 Overview	163
7.8.3.2 PartDefinition	163
7.8.3.3 PartUsage	163
7.8.4 Parts Semantics	164
7.9 Ports	164
7.9.1 Ports Overview	164
7.9.2 Ports Concrete Syntax	164
7.9.2.1 Ports Textual Notation	164
7.9.2.1.1 Port Definitions	165
7.9.2.1.2 Port Usages	165
7.9.2.2 Ports Graphical Notation	166
7.9.3 Ports Abstract Syntax	166
7.9.3.1 Overview	167
7.9.3.2 ConjugatedPortDefinition	167
7.9.3.3 ConjugatedPortTyping	168
7.9.3.4 PortConjugation	169
7.9.3.5 PortDefinition	169
7.9.3.6 PortUsage	170
7.9.4 Ports Semantics	170
7.10 Connections	170
7.10.1 Connections Overview	170
7.10.2 Connections Concrete Syntax	171
7.10.2.1 Connections Textual Notation	171
7.10.2.1.1 Connection Definitions	172
7.10.2.1.2 Connection Usages	172
7.10.2.2 Connections Graphical Notation	173
7.10.3 Connections Abstract Syntax	173
7.10.3.1 Overview	174
7.10.3.2 ConnectionDefinition	174
7.10.3.3 ConnectionUsage	175

7.10.4 Connections Semantics	175
7.11 Interfaces	175
7.11.1 Interfaces Overview	175
7.11.2 Interfaces Concrete Syntax	176
7.11.2.1 Interfaces Textual Syntax	176
7.11.2.1.1 Interface Definitions	176
7.11.2.1.2 Interface Usages	176
7.11.2.2 Interfaces Graphical Syntax	177
7.11.3 Interfaces Abstract Syntax	177
7.11.3.1 Overview	177
7.11.3.2 InterfaceDefinition	178
7.11.3.3 InterfaceUsage	178
7.11.4 Interfaces Semantics	179
7.12 Allocations	179
7.12.1 Allocations Overview	179
7.12.2 Allocations Concrete Syntax	179
7.12.2.1 Allocations Textual Notation	179
7.12.2.2 Allocations Graphical Notation	179
7.12.3 Allocations Abstract Syntax	179
7.12.3.1 Overview	180
7.12.3.2 AllocationDefinition	180
7.12.3.3 AllocationUsage	181
7.12.4 Allocations Semantics	181
7.13 Individuals	181
7.13.1 Individuals Overview	181
7.13.2 Individuals Concrete Syntax	182
7.13.2.1 Individuals Textual Notation	182
7.13.2.1.1 Individual Definitions	183
7.13.2.1.2 Individual Usages	184
7.13.2.1.3 Individual Successions	185
7.13.2.2 Individuals Graphical Notation	186
7.13.3 Individuals Abstract Syntax	186
7.13.3.1 Overview	187
7.13.3.2 IndividualDefinition	187
7.13.3.3 IndividualUsage	188
7.13.4 Individuals Semantics	189
7.14 Actions	189
7.14.1 Actions Overview	189
7.14.2 Actions Concrete Syntax	190
7.14.2.1 Actions Textual Notation	190
7.14.2.1.1 Action Definitions	191
7.14.2.1.2 Action Usages	194
7.14.2.1.3 Action Nodes	195
7.14.2.1.4 Action Successions	196
7.14.2.2 Actions Graphical Notation	196
7.14.3 Actions Abstract Syntax	196
7.14.3.1 Overview	196
7.14.3.2 AcceptActionUsage	197
7.14.3.3 ActionDefinition	198
7.14.3.4 ActionUsage	198
7.14.3.5 ControlNode	199
7.14.3.6 DecisionNode	199
7.14.3.7 ForkNode	200
7.14.3.8 JoinNode	200

7.14.3.9 MergeNode	201
7.14.3.10 PerformActionUsage	201
7.14.3.11 SendActionUsage	202
7.14.3.12 TransferActionUsage	202
7.14.4 Actions Semantics	203
7.15 States	203
7.15.1 States Overview	203
7.15.2 States Concrete Syntax	203
7.15.2.1 States Textual Notation	203
7.15.2.1.1 State Definitions	204
7.15.2.1.2 State Usages	206
7.15.2.1.3 Transition Usages	207
7.15.2.2 States Graphical Notation	207
7.15.3 States Abstract Syntax	208
7.15.3.1 Overview	208
7.15.3.2 ExhibitStateUsage	209
7.15.3.3 StateSubactionKind	210
7.15.3.4 StateSubactionMembership	210
7.15.3.5 StateDefinition	211
7.15.3.6 StateUsage	212
7.15.3.7 TransitionFeatureKind	213
7.15.3.8 TransitionFeatureMembership	213
7.15.3.9 TransitionUsage	214
7.15.4 States Semantics	215
7.16 Calculations	215
7.16.1 Calculations Overview	215
7.16.2 Calculations Concrete Syntax	215
7.16.2.1 Calculations Textual Notation	215
7.16.2.1.1 Calculation Definitions	216
7.16.2.1.2 Calculation Usages	217
7.16.2.2 Calculations Graphical Notation	217
7.16.3 Calculations Abstract Syntax	217
7.16.3.1 Overview	217
7.16.3.2 CalculationDefinition	218
7.16.3.3 CalculationUsage	218
7.16.4 Calculations Semantics	219
7.17 Constraints	219
7.17.1 Constraints Overview	219
7.17.2 Constraints Concrete Syntax	219
7.17.2.1 Constraints Textual Notation	220
7.17.2.2 Constraints Graphical Notation	220
7.17.3 Constraints Abstract Syntax	220
7.17.3.1 Overview	220
7.17.3.2 AssertConstraintUsage	221
7.17.3.3 ConstraintDefinition	221
7.17.3.4 ConstraintUsage	222
7.17.4 Constraints Semantics	222
7.18 Requirements	223
7.18.1 Requirements Overview	223
7.18.2 Requirements Concrete Syntax	224
7.18.2.1 Requirements Textual Notation	224
7.18.2.1.1 Requirement Definitions	225
7.18.2.1.2 Requirement Usages	226
7.18.2.1.3 Concern Definitions	227

7.18.2.1.4 Concern Usages	227
7.18.2.1.5 Stakeholders	228
7.18.2.1.6 Stakeholder Usage	228
7.18.2.2 Requirements Graphical Notation	228
7.18.3 Requirements Abstract Syntax	228
7.18.3.1 Overview	228
7.18.3.2 AddressConcernMembership	230
7.18.3.3 ConcernDefinition	231
7.18.3.4 ConcernUsage	231
7.18.3.5 RequirementConstraintKind	232
7.18.3.6 RequirementConstraintMembership	232
7.18.3.7 RequirementDefinition	233
7.18.3.8 RequirementUsage	234
7.18.3.9 SatisfyRequirementUsage	235
7.18.3.10 StakeholderDefinition	236
7.18.3.11 StakeholderUsage	236
7.18.3.12 SubjectMembership	237
7.18.4 Requirements Semantics	237
7.19 Cases	237
7.19.1 Cases Overview	237
7.19.2 Cases Concrete Syntax	237
7.19.2.1 Cases Textual Notation	237
7.19.2.1.1 Case Definitions	238
7.19.2.1.2 Case Usages	238
7.19.2.2 Cases Graphical Notation	238
7.19.3 Cases Abstract Syntax	238
7.19.3.1 Overview	239
7.19.3.2 CaseDefinition	239
7.19.3.3 CaseUsage	240
7.19.3.4 ObjectiveMembership	241
7.19.4 Cases Semantics	241
7.20 Analysis Cases	241
7.20.1 Analysis Cases Overview	242
7.20.2 Analysis Cases Concrete Syntax	242
7.20.2.1 Analysis Cases Textual Notation	243
7.20.2.2 Analysis Cases Graphical Notation	243
7.20.3 Analysis Cases Abstract Syntax	243
7.20.3.1 Overview	243
7.20.3.2 AnalysisCaseDefinition	244
7.20.3.3 AnalysisCaseUsage	244
7.20.4 Analysis Cases Semantics	245
7.21 Verification Cases	245
7.21.1 Verification Cases Overview	245
7.21.2 Verification Cases Concrete Syntax	246
7.21.2.1 Verification Cases Textual Notation	246
7.21.2.2 Verification Cases Graphical Notation	246
7.21.3 Verification Cases Abstract Syntax	246
7.21.3.1 Overview	247
7.21.3.2 RequirementVerificationMembership	247
7.21.3.3 VerificationCaseDefinition	248
7.21.3.4 VerificationCaseUsage	249
7.21.4 Verification Cases Semantics	249
7.22 Views	249
7.22.1 Views Overview	249

7.22.2 Views Concrete Syntax	250
7.22.2.1 Views Textual Notation	250
7.22.2.1.1 View Definitions	250
7.22.2.1.2 View Usages	251
7.22.2.1.3 Viewpoints	251
7.22.2.1.4 Renderings	252
7.22.2.2 Views Graphical Notation	252
7.22.3 Views Abstract Syntax	252
7.22.3.1 Overview	252
7.22.3.2 Expose	254
7.22.3.3 RenderingDefinition	255
7.22.3.4 RenderingUsage	255
7.22.3.5 ViewDefinition	256
7.22.3.6 ViewpointDefinition	256
7.22.3.7 ViewpointUsage	257
7.22.3.8 ViewRenderingMembership	257
7.22.3.9 ViewUsage	258
7.22.4 Views Semantics	259
7.23 Language Extension	259
8 Model Libraries	261
8.1 Systems Model Library	261
8.1.1 Overview	261
8.1.2 Attributes	261
8.1.2.1 Attributes Overview	261
8.1.2.2 Elements	261
8.1.2.2.1 attributeValues	261
8.1.2.2.2 AttributeValue	261
8.1.3 Items	262
8.1.3.1 Items Overview	262
8.1.3.2 Elements	262
8.1.3.2.1 Item	262
8.1.3.2.2 items	263
8.1.4 Parts	263
8.1.4.1 Parts Overview	263
8.1.4.2 Elements	263
8.1.4.2.1 Part	263
8.1.4.2.2 parts	264
8.1.5 Ports	264
8.1.5.1 Ports Overview	264
8.1.5.2 Elements	265
8.1.5.2.1 Port	265
8.1.5.2.2 ports	265
8.1.6 Connections	265
8.1.6.1 Connections Overview	266
8.1.6.2 Elements	266
8.1.6.2.1 Connection	266
8.1.6.2.2 connections	267
8.1.7 Interfaces	267
8.1.7.1 Interfaces Overview	267
8.1.7.2 Elements	268
8.1.7.2.1 Interface	268
8.1.7.2.2 interfaces	268
8.1.8 Allocations	268
8.1.8.1 Allocations Overview	269

8.1.8.2 Elements	269
8.1.8.2.1 Allocation	269
8.1.8.2.2 allocations	270
8.1.9 Actions	270
8.1.9.1 Actions Overview	270
8.1.9.2 Elements	271
8.1.9.2.1 AcceptAction	271
8.1.9.2.2 Action	272
8.1.9.2.3 actions	273
8.1.9.2.4 ControlAction	273
8.1.9.2.5 DecisionAction	274
8.1.9.2.6 ForkAction	274
8.1.9.2.7 JoinAction	274
8.1.9.2.8 MergeAction	275
8.1.9.2.9 SendAction	275
8.1.10 States	276
8.1.10.1 States Overview	276
8.1.10.2 Elements	276
8.1.10.2.1 StateAction	276
8.1.10.2.2 stateActions	277
8.1.10.2.3 TransitionAction	277
8.1.10.2.4 transitionActions	278
8.1.11 Calculations	278
8.1.11.1 Calculations Overview	279
8.1.11.2 Elements	279
8.1.11.2.1 Calculation	279
8.1.11.2.2 calculations	280
8.1.12 Constraints	280
8.1.12.1 Constraints Overview	280
8.1.12.2 Elements	280
8.1.12.2.1 ConstraintCheck	280
8.1.12.2.2 constraintChecks	281
8.1.13 Requirements	281
8.1.13.1 Requirements Overview	282
8.1.13.2 Elements	283
8.1.13.2.1 ConcernCheck	283
8.1.13.2.2 concernChecks	284
8.1.13.2.3 DesignConstraintCheck	284
8.1.13.2.4 FunctionalRequirementCheck	284
8.1.13.2.5 InterfaceRequirementCheck	285
8.1.13.2.6 PerformanceRequirementCheck	285
8.1.13.2.7 PhysicalRequirementCheck	285
8.1.13.2.8 RequirementCheck	286
8.1.13.2.9 requirementChecks	286
8.1.13.2.10 Stakeholder	287
8.1.13.2.11 stakeholders	287
8.1.14 Cases	287
8.1.14.1 Cases Overview	288
8.1.14.2 Elements	288
8.1.14.2.1 Case	288
8.1.14.2.2 cases	288
8.1.15 Analysis Cases	289
8.1.15.1 Analysis Cases Overview	289

8.1.15.2 Elements	289
8.1.15.2.1 AnalysisAction	290
8.1.15.2.2 AnalysisCase	290
8.1.15.2.3 analysisCases	291
8.1.16 Verification Cases	291
8.1.16.1 Verification Cases Overview	291
8.1.16.2 Elements	291
8.1.16.2.1 VerdictKind	292
8.1.16.2.2 VerificationCase	292
8.1.16.2.3 verificationCases	293
8.1.16.2.4 VerificationCheck	293
8.1.17 Views.....	293
8.1.17.1 Views Overview	294
8.1.17.2 Elements	295
8.1.17.2.1 asElementTable	295
8.1.17.2.2 asInterconnectionDiagram	296
8.1.17.2.3 asTextualNotation	296
8.1.17.2.4 asTreeDiagram	296
8.1.17.2.5 GraphicalRendering	297
8.1.17.2.6 Rendering	297
8.1.17.2.7 renderings	297
8.1.17.2.8 TabularRendering	298
8.1.17.2.9 TextualRendering	298
8.1.17.2.10 View	298
8.1.17.2.11 ViewpointCheck	299
8.1.17.2.12 viewpointChecks	299
8.1.17.2.13 viewpointConformance	300
8.1.17.2.14 views	300
8.2 Quantities and Units Domain Library	300
8.2.1 Overview	300
8.2.2 Quantities	301
8.2.2.1 Quantities Overview	301
8.2.2.2 Elements	301
8.2.2.2.1 scalarQuantities	302
8.2.2.2.2 ScalarQuantityValue	302
8.2.2.2.3 tensorQuantities	303
8.2.2.2.4 TensorQuantityValue	303
8.2.2.2.5 vectorQuantities	304
8.2.2.2.6 VectorQuantityValue	304
8.2.3 Units and Scales	305
8.2.3.1 Units and Scales Overview	305
8.2.3.2 Elements	306
8.2.3.2.1 ConversionByConvention	306
8.2.3.2.2 ConversionByPrefix	306
8.2.3.2.3 CoordinateTransformation	307
8.2.3.2.4 CyclicRatioScale	308
8.2.3.2.5 DerivedUnit	308
8.2.3.2.6 IntervalScale	309
8.2.3.2.7 LogarithmBaseKind	309
8.2.3.2.8 LogarithmScale	309
8.2.3.2.9 MeasurementScale	310
8.2.3.2.10 MeasurementUnit	311
8.2.3.2.11 OrdinalScale	311
8.2.3.2.12 ScalarMeasurementReference	312

8.2.3.2.13 ScaleValueDefinition	312
8.2.3.2.14 ScaleValueMapping	313
8.2.3.2.15 SimpleUnit	313
8.2.3.2.16 TensorMeasurementReference	314
8.2.3.2.17 UnitConversion	315
8.2.3.2.18 UnitPowerFactor	315
8.2.3.2.19 UnitPrefix	316
8.2.3.2.20 VectorMeasurementReference	316
8.2.4 ISQ	317
8.2.4.1 ISQ Overview	317
8.2.4.2 Elements	317
8.2.5 SI Prefixes	317
8.2.5.1 SI Prefixes Overview	317
8.2.5.2 Elements	317
8.2.6 SI	317
8.2.6.1 SI Overview	317
8.2.6.2 Elements	317
8.2.7 US Customary Units	317
8.2.7.1 US Customary Units Overview	317
8.2.7.2 Elements	317
8.2.8 Date and Time	317
8.2.8.1 Date and Time Overview	318
8.2.8.2 Elements	318
8.2.8.2.1 DateTime	318
8.2.8.2.2 TimeOfDay	318
8.3 Analysis Domain Library	318
8.3.1 Overview	318
8.3.2 Trade Studies	318
8.3.2.1 Trade Studies Overview	319
8.3.2.2 Elements	319
8.3.2.2.1 MaximizeObjective	319
8.3.2.2.2 MinimizeObjective	320
8.3.2.2.3 ObjectiveFunction	320
8.3.2.2.4 TradeStudy	320
8.3.2.2.5 TradeStudyObjective	321
A Annex: Conformance Test Suite	323
B Annex: Example Model	325
C Annex: SysML v1 to SysML v2 Transformation	341
C.1 General	341
C.1.1 Overview	341
C.1.2 Mapping Approach	341
C.2 Mappings	342
C.2.1 Generic Mappings	342
C.2.1.1 Overview	342
C.2.1.2 Generic Mappings to KerML	342
C.2.1.3 Generic Mappings to Systems	343
C.2.2 UML4SysML	343
C.2.2.1 Overview	343
C.2.2.2 Classification	343
C.2.2.2.1 Overview	343
C.2.2.2.2 Mapping Specifications	344
C.2.2.2.2.1 Classifier_Mapping	344
C.2.2.2.2.2 LowerBoundTyping_Mapping	345
C.2.2.2.2.3 MultiplicityBound_Mapping	345

C.2.2.2.2.4 MultiplicityBoundOwnership_Mapping.....	346
C.2.2.2.2.5 MultiplicityBoundTyping_Mapping.....	347
C.2.2.2.2.6 MultiplicityElement_Mapping.....	348
C.2.2.2.2.7 MultiplicityLowerBound_Mapping.....	349
C.2.2.2.2.8 MultiplicityMembership_Mapping.....	350
C.2.2.2.2.9 MultiplicityLowerBoundOwnership_Mapping.....	351
C.2.2.2.2.10 MultiplicityUpperBound_Mapping.....	352
C.2.2.2.2.11 MultiplicityUpperBoundOwnership_Mapping.....	352
C.2.2.2.2.12 StructuralFeature_Mapping.....	353
C.2.2.2.2.13 TypedElementToFeatureTyping_Mapping.....	354
C.2.2.2.2.14 UpperBoundTyping_Mapping.....	355
C.2.2.3 CommonBehavior.....	356
C.2.2.3.1 Overview.....	356
C.2.2.3.2 Mapping Specifications.....	356
C.2.2.3.2.1 Behavior.....	356
C.2.2.4 CommonStructure.....	357
C.2.2.4.1 Overview.....	357
C.2.2.4.2 Mapping Specifications.....	357
C.2.2.4.2.1 Abstraction Mapping.....	357
C.2.2.4.2.2 Comment_Mapping.....	358
C.2.2.4.2.3 CommentToAnnotation_Mapping.....	359
C.2.2.4.2.4 Dependency_Mapping.....	360
C.2.2.4.2.5 DirectRelationship_Mapping.....	360
C.2.2.4.2.6 ElementMain_Mapping.....	361
C.2.2.4.2.7 ElementOwnership_Mapping.....	362
C.2.2.4.2.8 ElementOwningMembership_Mapping.....	362
C.2.2.4.2.9 FromElement_Mapping.....	363
C.2.2.4.2.10 Namespace_Mapping.....	364
C.2.2.4.2.11 Relationship_Mapping.....	364
C.2.2.5 Packages.....	365
C.2.2.5.1 Overview.....	365
C.2.2.5.2 Mapping Specifications.....	365
C.2.2.5.2.1 ElementImport_Mapping.....	365
C.2.2.5.2.2 Package_Mapping.....	366
C.2.2.5.2.3 PackageImport_Mapping.....	367
C.2.2.6 SimpleClassifiers.....	368
C.2.2.6.1 Overview.....	368
C.2.2.6.2 Mapping Specifications.....	368
C.2.2.6.2.1 BehavoredClassifier_Mapping.....	368
C.2.2.6.2.2 DataType_Mapping.....	369
C.2.2.6.2.3 Enumeration_Mapping.....	370
C.2.2.6.2.4 EnumerationLiteral_Mapping.....	370
C.2.2.6.2.5 Signal.....	371
C.2.2.7 StructuredClassifiers.....	372
C.2.2.7.1 Overview.....	372
C.2.2.7.2 Mapping Specifications.....	372
C.2.2.7.2.1 Association_Mapping.....	372
C.2.2.7.2.2 AssociationClass_Mapping.....	373
C.2.2.7.2.3 AssociationToMetadata_Mapping.....	374
C.2.2.7.2.4 Class_Mapping.....	375
C.2.2.7.2.5 ConnectorMapping.....	375
C.2.2.7.2.6 EncapsulatedClassifier_Mapping.....	376
C.2.2.7.2.7 Port_Mapping.....	377
C.2.2.7.2.8 StructuredClassifier_Mapping.....	378

C.2.3 SysML v1.6	379
C.2.3.1 Overview	379

List of Tables

1. Mandatory Language Requirements Table	2
2. Non-Mandatory Language Requirements Table	31
3. Mandatory Language Requirements - Satisfied-by Table	40
4. Non-Mandatory Language Requirements - Satisfied-by Table	51
5. Changed Language Requirements Table	54
6. List of all Generic Mappings to KerML Mapping Specifications	342
7. List of all Generic Mappings to Systems Mapping Specifications	343
8. List of all Overview Mapping Specifications	343
9. Table Classifier_Mapping Rules	344
10. Table LowerBoundTyping_Mapping Rules	345
11. Table MultiplicityBound_Mapping Rules	346
12. Table MultiplicityBoundOwnership_Mapping Rules	347
13. Table MultiplicityBoundTyping_Mapping Rules	348
14. Table MultiplicityElement_Mapping Rules	348
15. Table MultiplicityLowerBound_Mapping Rules	349
16. Table MultiplicityMembership_Mapping Rules	350
17. Table MultiplicityLowerBoundOwnership_Mapping Rules	351
18. Table MultiplicityUpperBound_Mapping Rules	352
19. Table MultiplicityUpperBoundOwnership_Mapping Rules	353
20. Table StructuralFeature_Mapping Rules	354
21. Table TypedElementToFeatureTyping_Mapping Rules	355
22. Table UpperBoundTyping_Mapping Rules	355
23. List of all Overview Mapping Specifications	356
24. Table Behavior Rules	356
25. List of all Overview Mapping Specifications	357
26. Table Abstraction Mapping Rules	358
27. Table Comment_Mapping Rules	359
28. Table CommentToAnnotation_Mapping Rules	359
29. Table Dependency_Mapping Rules	360
30. Table DirectRelationship_Mapping Rules	361
31. Table ElementMain_Mapping Rules	362
32. Table ElementOwnership_Mapping Rules	362
33. Table ElementOwningMembership_Mapping Rules	363
34. Table FromElement_Mapping Rules	363
35. Table Namespace_Mapping Rules	364
36. Table Relationship_Mapping Rules	365
37. List of all Overview Mapping Specifications	365
38. Table ElementImport_Mapping Rules	366
39. Table Package_Mapping Rules	366
40. Table PackageImport_Mapping Rules	367
41. List of all Overview Mapping Specifications	368
42. Table BehavoredClassifier_Mapping Rules	368
43. Table DataType_Mapping Rules	369
44. Table Enumeration_Mapping Rules	370
45. Table EnumerationLiteral_Mapping Rules	371
46. Table Signal Rules	371
47. List of all Overview Mapping Specifications	372
48. Table Association_Mapping Rules	372
49. Table AssociationClass_Mapping Rules	373
50. Table AssociationToMetadata_Mapping Rules	374
51. Table Class_Mapping Rules	375

52. Table ConnectorMapping Rules	376
53. Table EncapsulatedClassifier_Mapping Rules	377
54. Table Port_Mapping Rules	377
55. Table StructuredClassifier_Mapping Rules	378

List of Figures

1. SysML Language Architecture	69
2. Elements	76
3. Annotation	82
4. Comments	82
5. Textual Representation	83
6. Metadata Annotation	83
7. Namespaces	94
8. Packages	94
9. Types	99
10. Generalization	100
11. Conjugation	100
12. Classifiers	102
13. Classification	102
14. Structures	103
15. Features	107
16. Subsetting	108
17. Multiplicities	108
18. Feature Values	109
19. Associations	110
20. EndFeatureMembership	110
21. Connectors	114
22. Successions	115
23. Behaviors	116
24. Interactions	118
25. Item Flows	119
26. Functions	120
27. Function Memberships	120
28. Predicates	121
29. Expressions	124
30. Literal Expressions	125
31. Dependencies	127
32. Definition and Usage	143
33. Variant Membership	144
34. Attribute Definition and Usage	153
35. Enumeration Definition and Usage	157
36. Item Definition and Usage	160
37. Part Definition and Usage	163
38. Port Definition and Usage	167
39. Port Conjugation	167
40. Connection Definition and Usage	174
41. Interface Definition and Usage	177
42. Allocation Definition and Usage	180
43. Individual Definition and Usage	187
44. Action Definition and Usage	196
45. Control Nodes	197
46. Perform, Send and Accept Actions	197
47. State Definition and Usage	208
48. State Membership	208
49. State Exhibition	209
50. Transition Usage	209
51. Calculation Definition and Usage	217

52. Constraint Definition and Usage.....	220
53. Constraint Assertion.....	221
54. Requirement Definition and Usage.....	228
55. Requirement Satisfaction.....	229
56. Concern Definition and Usage.....	229
57. Stakeholder Definition and Usage.....	230
58. Requirement Membership.....	230
59. Case Definition and Usage.....	239
60. Case Membership.....	239
61. Analysis Case Definition and Usage.....	243
62. Verification Case Definition and Usage.....	247
63. Verification Membership.....	247
64. View Definition and Usage.....	252
65. Viewpoint Definition and Usage.....	253
66. Rendering Definition and Usage.....	253
67. Expose Relationship.....	254
68. View Rendering Membership.....	254
69. Attributes Model Library.....	261
70. Items Model Library.....	262
71. Parts Library Model.....	263
72. Ports Library Model.....	264
73. Connections Library Model.....	266
74. Interfaces Library Model.....	267
75. Allocations Library Model.....	269
76. Actions Library Model.....	270
77. Control Nodes Library Model.....	271
78. Send and Accept Actions.....	271
79. States Library Model.....	276
80. Transitions Library Model.....	276
81. Calculations Library Model.....	279
82. Constraints Library Model.....	280
83. Requirements Library Model.....	282
84. Stakeholders and Concerns Library Model.....	283
85. Cases Library Model.....	288
86. Analysis Case Library Model.....	289
87. Verification Case Library Model.....	291
88. Views Library Model.....	294
89. Viewpoints Library Model.....	294
90. Renderings Library Model.....	295
91. Quantities.....	301
92. Measurement Units and Scales.....	305
93. Coordinate Systems.....	306
94. Calendar Dates and Times.....	318
95. Trade Studies Domain Model.....	319
96. Trade Study Objectives.....	319

0 Submission Introduction

0.1 Submission Overview

This document is the second of two documents submitted in response to the Systems Modeling Language (SysML®) v2 Request for Proposals (RFP) (ad/2017-11-04). The first document defines a Kernel Modeling Language (KerML) that provides a syntactic and semantic foundation for creating more specific modeling languages. This document provides the proposed specification of the actual 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 document for KerML is being proposed as a separate specification from SysML v2. By intent, KerML provides a common kernel for the creation of diverse modeling languages that can be tailored to specific domains while still maintaining fundamental semantic interoperability. SysML v2 is such a modeling language, tailored to the systems modeling domain. It is the combination of the kernel provided by KerML and the systems-domain specific metamodel defined in this document that together satisfy the requirements of the SysML v2 RFP, as documented in subclause 0.4.

Release note. The present document is an update to the initial submission document submitted to OMG in August 2020.

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
- Thematrix

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

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 SysML abstract syntax and textual concrete syntax. There have been four quarterly public releases of this pilot implementation so far, the last being the 2020-06 version released at the beginning of July 2020. However, since the conformance test suite has not been developed as of the time of this initial submission, 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 as of the 2020-06 release is known to not fully conform to the initial submission of the SysML specification, the deviations are identified in "implementation notes" in this document. The SST is currently planning on releasing the 2020-09 version of the

pilot implementation as open source, at which time it is intended that it will be fully conformant with the initial submission of this specification.

The SST has also been prototyping graphical visualization tools using the SysML graphical concrete syntax. However, these implementations are not yet as complete as the implementation of the textual notation.

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](#).

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.

This specification is a replacement for the SysML v1.x series of standards (collectively referred to as "SysML v1"). SysML v1 was defined as a profile of the Unified Modeling Language [UML]. SysML v2, however, is defined with its own metamodel, which is build on the Kernel Metamodel [KerML]. As required in the SysML v2 RFP, the abstract syntax for SysML 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 also [KerML, 0.3] for further discussion of the relationship to MOF.)

The SysML v2 RFP also requires that a UML profile be provided for SysML v2 "that includes, as a minimum, the functional capabilities of the SysML v1.x profile, and a mapping to the SysML v2 metamodel" (RFP requirement LNG 1.1.3). The SysML v2 specification proposed in tihs initial submission includes a model of a transformation from SysML v1.7 (expected to be the last version of SysML v1) to SysML v2 (see [Annex C](#)). This transformation effectively allows the SysML v1.7 profile to also be used as a profile for the subset of SysML v2 functional capabilities that have equivalent capabilities in SysML v1, minimally meeting the RFP requirement.

Consideration is still being given to developing a more extensive SysML v2 profile for the revised submission, depending on the availability of resources and the desires of the SysML community of stakeholders.

0.4 Language Requirements Tables

0.4.1 Mandatory Language Requirements Table

Table 1. Mandatory Language Requirements Table

Req. ID	Req. Name	Text
ANL 1	Analysis Requirements Group	The requirements in this group are used to specify an analysis, along with other requirements such as Properties, Values, and Expressions.
ANL 1.01	Subject of the Analysis	Proposals for SysML v2 shall include the capability to model the relationship between the analysis and the subject of the analysis (system being analyzed).
ANL 1.02	Analysis	Proposals for SysML v2 shall include the capability to specify an Analysis, including the subject of analysis (e.g., system), the analysis case, and the analysis models and related infrastructure to perform the analysis.

Req. ID	Req. Name	Text
ANL 1.03	Parameters of Interest	Proposals for SysML v2 shall include the capability to identify the key parameters of interest including measures-of-effectiveness (MoE) and other key measures of performance (MoP).
ANL 1.04	Analysis Case	<p>Proposals for SysML v2 shall include the capability to model the analysis case to specify the analysis scenarios and associated analysis methods needed to produce an analysis result that achieves the analysis objectives.</p> <p>Supporting Information: This is intended to be a specialization of Case.</p>
ANL 1.05	Analysis Objectives	Proposals for SysML v2 shall include the capability to model the objective of the analysis being performed in text or as a mathematical formalism, e.g. math expression, so that it can be evaluated.
ANL 1.06	Analysis Scenarios	Proposals for SysML v2 shall include the capability to model the scenarios that identify the analysis models to be executed, the conditions and assumptions, and the configurations of the subject of the analysis and the related infrastructure to perform the analysis.
ANL 1.07	Analysis Assumption	Proposals for SysML v2 shall include the capability to model the assumptions of the analyses in a text or mathematical form, e.g. constraints and boundary conditions.
ANL 1.08	Analysis Decomposition	Proposals for SysML v2 shall include the capability to decompose an analysis into constituent analyses.
ANL 1.09	Analysis Model	<p>Proposals for SysML v2 shall include the capability to specify an analysis model.</p> <p>Supporting Information: Analysis models can be defined natively in SysML (e.g. parametric model or behavior model) or externally (e.g. equation-based math models, finite element analysis models, or computational fluid dynamics models). The level of fidelity of the specification of the analysis model can vary from an abstract specification that defines the intent of the analysis including its input and output parameters, to a detailed specification that a particular solver can execute.</p>
ANL 1.11	Analysis Result	<p>Proposals for SysML v2 shall include the capability to relate the results of executing analysis models to the analysis.</p> <p>Supporting Information: The results may be stored in the SysML v2 model itself or in an external store (e.g. CSV file or database). The results can be used to evaluate how well the analysis objectives are satisfied, and to obtain the supporting rationale for decisions taken based on the analysis.</p>

Req. ID	Req. Name	Text
ANL 1.13	Analysis Metadata	Proposals for SysML v2 shall include the capability to represent the metadata relevant to specifying the analysis, including the specification of dependent and independent parameters.
ANL 1.14	Decision Group	The requirements in this group support trade-off analysis among alternatives. This typically involves making decisions during the design process to evaluate alternative designs based on a set of criteria, and selecting a preferred design.
ANL 1.14.2	Alternative	Proposals for SysML v2 shall include a capability to represent a set of alternatives.
ANL 1.14.4	Decision	<p>Proposals for SysML v2 shall include a capability to represent a decision as one or more selections among alternatives.</p> <p>Supporting Information: This Decision and Rationale can be related through an Explanation relationship. The Rationale can refer to the supporting analysis.</p>
ANL 1.14.5	Criteria	Proposals for SysML v2 shall include a capability to represent criteria that is used as a basis for a decision or evaluation.
ANL 1.14.6	Rationale	Proposals for SysML v2 shall include a capability to represent rationale for a decision or other conclusion.
BHV 1	Behavior Requirements Group	
BHV 1.01	Behavior	Proposals for SysML v2 shall include the capability to model a Behavior that represents the interaction between individual structural elements and their change of state over time.

Req. ID	Req. Name	Text
BHV 1.02	Behavior Decomposition	<p>Proposals for SysML v2 shall include the capability to decompose a behavior to any level of decomposition, and to define localized usages of behavior at nested levels of decomposition.</p> <p>Supporting Information:</p> <p>The decomposition of behavior should conform to a similar pattern as the decomposition of structure, and include capabilities for specialization, redefinition, and sub-setting.</p> <p>The decomposition should also include the equivalent capability to decompose a SysML v1 activity on a BDD, and the ability to decompose actions using a structured activity node.</p>
BHV 1.03	Function-based Behavior Group	
BHV 1.03.1	Function-based Behavior	<p>Proposals for SysML v2 shall include the capability to represent a controlled sequence of actions (or functions) that can transform a set of input items to a set of output items.</p> <p>Supporting Information:</p> <p>SysML v2 should provide an integrated approach to specify behavior that reflects similar capabilities to SysML v1 activities and sequence diagrams, which are expected to be different views of the same underlying model.</p> <p>The input items and output items correspond to item usages and their associated value properties whose values can vary over time. Item flows connect an output item usage to an input item usage.</p> <p>The start and stop events should be represented explicitly (e.g., control pins). Event flows connect a stop event to a start event.</p> <p>The specific features of activities and sequence diagrams to be included in SysML v2 beyond what is specified in this section should be defined in the proposal.</p>
BHV 1.03.3	Function-based Behavior Constraints	<p>Proposals for SysML v2 shall include the capability to model constraints on a function-based behavior that includes the ability to represent a declarative specification in terms of its pre-conditions and post-conditions, and any constraints that apply throughout execution of the behavior.</p>
BHV 1.03.4	Opaque Behavior	<p>Proposals for SysML v2 shall include the capability to represent a behavior that embeds the definition in a language such as a programming language.</p>

Req. ID	Req. Name	Text
BHV 1.03.6	Structure Modification Behavior	<p>Proposals for SysML v2 shall include the capability to represent behaviors that can modify the structure of an element over time, such as the creation and destruction of interconnections and composition.</p> <p>Supporting Information:</p> <p>An example is the behavior associated with the separation of a first stage rocket, or the assembly or disassembly of a product.</p>
BHV 1.04	State-based Behavior Group	
BHV 1.04.1	Regions, States, and Transitions	<p>Proposals for SysML v2 shall include the capability to represent the state behavior of a structural element in terms of its concurrent regions with mutually exclusive finite states, and transitions between finite states.</p> <p>Supporting Information:</p> <p>A state change can result from a change in structure.</p>
BHV 1.04.2	Integration of Function-based Behavior with Finite State Behavior	<p>Proposals for SysML v2 shall include the capability to model function-based behavior both on transitions between finite states, and upon entry, exit, and while in a finite state.</p>
BHV 1.04.3	Integration of Constraints with Finite State Behavior	<p>Proposals for SysML v2 shall include the capability to model constraints both on transitions between finite states, and upon entry, exit, and while in a finite state.</p>
BHV 1.05	Discrete and Continuous Time Behavior	<p>Proposals for SysML v2 shall include the capability to model behaviors whose inputs and outputs vary continuously as a function of time, or discretely as a function of time.</p>
BHV 1.06	Events	<p>Proposals for SysML v2 shall include the capability to model signal events, time events, and change events and their ordering.</p> <p>Supporting Information:</p> <p>The ordering of actions (i.e., functions) is accomplished through ordering of their start and completion events.</p> <p>Events can trigger a change from one finite-state to another.</p> <p>Events should be able to be explicitly represented in both function-based behavior and finite-state behavior.</p> <p>Events can be defined and used in different contexts.</p>

Req. ID	Req. Name	Text
BHV 1.07	Control Nodes	<p>Proposals for SysML v2 shall include the capability to model control nodes that specify a logical expression of conditions and events to enable a flow.</p> <p>Supporting Information: For Example: {Inputs A < a1 AND B>=b2 OR C AND NOT D} must be true).</p>
BHV 1.08	Time Constraints	<p>Proposals for SysML v2 shall include the capability to specify the absolute or relative time associated with an event that includes start events, stop events, and duration constraints between events to represent the time-line associated with a behavior.</p> <p>Supporting Information: Time is a property typed by a Value Type whose quantity kind and units are specified as part of QUDV.</p>
BHV 1.10	Behavior Execution	<p>Proposals for SysML v2 shall include the capability to execute function-based and state-based behavior to specify the state history of individual elements and their interactions with other individual elements.</p> <p>Supporting Information: The behavior of a Definition Element or Configuration Element represent the default behavior of the conforming Individual Elements.</p>
BHV 1.11	Integration between Structure and Behavior	
BHV 1.11.1	Allocation of Behavior to Structure	<p>Proposals for SysML v2 shall include the capability to represent the behavior of one or more structural elements.</p> <p>Supporting Information:</p> <p>This should support the ability to define a state machine of a structural element, with finite states that enable actions (i.e., functions) and constraints. In addition, this should support the ability to specify the functions performed by a component, and the applicable constraints, without specifying the finite state that enables them. The representation should allow more than one structural element to perform a single function, such as when two people carry a load. This is analogous to a reference interaction in a SysML v1 sequence diagram that spans multiple lifelines and displays the participating lifelines. The reference interaction refers to another sequence diagram.</p>

Req. ID	Req. Name	Text
BHV 1.11.2	Integration of Control Flow and Input/Output Flow	<p>Proposals for SysML v2 shall ensure that inputs, outputs, and events can be represented consistently across behavior and structure.</p> <p>Supporting Information:</p> <p>In SysML v1, it is often difficult to ensure consistent representation of control flow and input/output flow. Examples include potential inconsistencies between:</p> <ul style="list-style-type: none"> • Flows on activity diagrams and messages on sequence diagrams. • Flows on activity diagrams and item flows on ibd • Inputs and outputs on activity diagram and corresponding inputs and outputs on activity decomposition on a bdd • Inability to represent input/output of activities on do behaviors of state machines
BHV 1.11.3	Storing Items in Storage Elements Requirements Group	
BHV 1.11.3.1	Storage Element and Stored Item Usages	<p>Proposals for SysML v2 shall include the capability to model a storage element that can store items declared by stored item usages. The stored items shall be identified as conserved (e.g., a physical element) or copied (e.g., data from memory). Conservation constraints shall apply to conserved item usages (e.g., amount in - amount out=amount stored).</p> <p>Supporting Information: Examples include:</p> <p>A storage element called tank that stores a stored item usage called fluid. (example of a conserved stored item usage)</p> <p>A storage element called common value table that stores a stored item usage called system mode. (example of a copied stored item usage)</p>

Req. ID	Req. Name	Text
BHV 1.11.3.2	Create, Modify, and Consume Stored Items	<p>Proposals for SysML v2 shall include the capability to model outputs and inputs of a behavior that create, modify, or consume stored items of a storage element. An input to or output from a storage element that results in the creation, modification, or consumption of stored items can be assigned to one or more ports of the storage element.</p> <p>Supporting Information: Examples include:</p> <p>A pump fluid action produces an output called fluid that is stored in a tank, and another action consumes the fluid from the tank. (example of a conserved stored item usage)</p> <p>An update mode variable action produces a logical data item that is stored in common value table, and another action called verify mode consumes the logical data item from the common value table. (example of a copied stored item usage)</p>
BHV 1.12	Case	<p>Proposals for SysML v2 shall include the capability to represent a case that can be specialized into a use case, verification case, analysis case, and domain specific cases, such as safety case and assurance case.</p> <p>Supporting Information: A case is a series of steps with an associated objective that produce a result or conclusion. An analysis case and assurance case correspond to a set of steps to implement a study or investigation. Refer to the Structured Assurance Case Metamodel (SACM).</p>
CNF 1	Conformance Requirements Group	<p>These requirements specify that the proposals provide a suite of test cases that a conformant SysML v2 implementation must satisfy. The test cases can more generally be verification cases.</p> <p>The SysML v2 specification will specify the conformance levels for each conformance area below. Vendors are expected to identify specific levels of conformance within each of the sub-section of groupings in this document so that a cross functional compliance matrix can be developed for each tool implementation. This enables the ecosystem of potential SysML tool vendors who only wish to partially implement the SysML specification to expand, (i.e. only the requirements or test aspects for example).</p>
CNF 1.1	Metamodel Conformance	Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 metamodel specification (abstract syntax, concrete syntax, and semantics).
CNF 1.2	Profile Conformance	Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 profile specification.

Req. ID	Req. Name	Text
CNF 1.3	Model Interoperability Conformance	Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 model interoperability specification.
CNF 1.4	Traceability Matrix	Proposals for SysML v2 shall include a traceability matrix (include reference) that demonstrates how each language feature is verified by the conformance test suite.
CRC 1	Cross-cutting Requirements Group	The following specify the requirements that apply to all model elements.
CRC 1.1	Model and Model Library Group	
CRC 1.1.1	Model	<p>Proposals for SysML v2 shall include a capability to represent a Model (aka system model) that contains a set of uniquely identifiable model elements.</p> <p>Supporting Information: This is intended to be a kind of Container or Namespace.</p>
CRC 1.1.2	Model Library	<p>Proposals for SysML v2 shall include a capability to represent a Model Library that contains a set of model elements that are intended to support reuse.</p> <p>Supporting Information: This is intended to be a kind of Container or Namespace.</p>
CRC 1.1.3	Container	<p>Proposals for SysML v2 shall include the capability to represent a Container that is a model element that contains other model elements. Model elements within a container shall be distinguishable from one another.</p> <p>Supporting Information: This provides a way to organize the model. Containers can contain other containers.</p>
CRC 1.2	Model Element Group	
CRC 1.2.2	Unique Identifier	<p>Proposals for SysML v2 shall include a capability to represent a single universally unique identifier for each model element that cannot be changed.</p> <p>Supporting Information: The unique identifier should enable assignment of URIs.</p>

Req. ID	Req. Name	Text
CRC 1.2.3	Name and Aliases	<p>Proposals for SysML v2 shall include a capability to represent a name and one or more aliases for any named model element.</p> <p>Supporting Information:</p> <p>Selected kinds of model elements may not require a name (e.g. dependency), or the name may be optional, but still should be distinguishable within a namespace.</p> <p>Aliases enable users to assign more than one name for the same element, such as a shortened name. A common use of aliases is the use of an abbreviated or shortened name.</p>
CRC 1.2.4	Definition / Description	<p>Proposals for SysML v2 shall include a capability to represent one or more definitions and/or descriptions for each model element.</p>
CRC 1.2.5	Annotation	<p>Proposals for SysML v2 shall include a capability to represent an annotation of one or more model elements that includes a text string. The text string can include a link that refers to a Navigation relationship (refer to CRC 1.3.10), and a classification field to identify the kind of annotation.</p> <p>Supporting Information: Annotations should be able to be related to other elements.</p>
CRC 1.2.6	Element Group	<p>Proposals for SysML v2 shall include a capability to represent a group of model elements that can satisfy user-defined criteria for membership in the group.</p> <p>Supporting Information:</p> <ol style="list-style-type: none"> 1. A member of an element group is not intended to impose ownership constraints on the members. 2. Element group can be specialized for different kinds of members, such as groups that contain requirements, functions, and structural elements, which may impose additional constraints on its members. 3. It shall be possible to define a relationship with an element group that is equivalent to defining the relationship with each member of the group.
CRC 1.2.7	Additional Cross-Cutting Concepts Group	<p>The requirements in this group include additional concepts that can be associated with any model element.</p>

Req. ID	Req. Name	Text
CRC 1.2.7.1	Problem	<p>Proposals for SysML v2 shall include a capability to represent a problem that causes an undesired affect.</p> <p>Supporting Information: A problem is often represented as a cause in a cause-effect relationship.</p>
CRC 1.2.7.2	Risk	<p>Proposals for SysML v2 shall include a capability to represent a Risk that identifies the kind of risk (e.g., cost, schedule, technical), and the likelihood of occurrence, and the potential impact.</p>
CRC 1.3	Model Element Relationships Requirements Group	
CRC 1.3.01	Relationship	<p>Proposals for SysML v2 shall include a capability to represent a Relationship between any two model elements, which may have a name and direction.</p>
CRC 1.3.02	Derived Relationship	<p>Proposals for SysML v2 shall include a capability to represent a relationship that is derived from other relationships.</p> <p>Supporting Information:</p> <p>An example is a derived relationship from a transitive relationship where B relates to A and C relates to B, then C relates to A.</p> <p>Another example is a connector between two composite parts that is derived from a connector between their nested parts.</p>
CRC 1.3.03	Dependency Relationship	<p>Proposals for SysML v2 shall include a capability to represent a Dependency Relationship where one side of the relationship refers to the independent element and the other side of the relationship refers to the dependent element.</p>
CRC 1.3.04	Cause-Effect Relationship	<p>Proposals for SysML v2 shall include a capability to represent a Cause-Effect Relationship where one side of the relationship refers to the cause and the other side of the relationship refers to the effect.</p>
CRC 1.3.05	Explanation Relationship	<p>Proposals for SysML v2 shall include a capability to represent an Explanation Relationship where one side of the relationship refers to the rationale and the other side of the relationship refers to the element being explained.</p>

Req. ID	Req. Name	Text
CRC 1.3.06	Conform Relationship	Proposals for SysML v2 shall include a capability to represent a Conform Relationship where the conforming element is constrained by the element on the other side of the relationship.
CRC 1.3.07	Refine Relationship	Proposals for SysML v2 shall include a capability to represent a Refine Relationship where the refined side of the relationships refers to the more precisely specified element.
CRC 1.3.08	Allocation Relationship	Proposals for SysML v2 shall include a capability to represent an Allocation Relationship where one side of the relationship refers to the allocated from, and the other side of the relationship refers to the allocated to.
CRC 1.3.09	Element Group Relationship	Proposals for SysML v2 shall include a capability to represent an Element Group Relationship where one side of the relationship refers to the member, and the other side of the relationship refers to the Element Group.
CRC 1.3.10	Navigation Relationship	<p>Proposals for SysML v2 shall include a capability to represent a Navigation Relationship between a model element and another model element or an external element, similar to a hyperlink, where one side of the relationship refers to the linked to, and the other side of the relationship refers to the linked from. The external element can be a data element, a file, and/or an element of an external model.</p> <p>Supporting information:</p> <p>This is a navigation aid that standardizes what many tools already do.</p> <p>The navigation can specify the ability to navigate from either end of the relationship.</p>
CRC 1.4	Variability Modeling Group	<p>The requirements in this group should accommodate approaches to model variants as choices among design options. The modeling approaches may include a separate variability model to identify the design choices. Additional variability modeling concepts may be included.</p> <p>Supporting Information: refer to ISO/IEC 26550:2015</p>
CRC 1.4.1	Variation Point	Proposals for SysML v2 shall include a capability to model variation points that identify features that can vary across a set of variants (e.g., vehicles with manual or automatic transmission, variable number of axles, or variable wheel size). A variation point may be dependent on another variant selection. (e.g., number of axles and wheel size is dependent on selection of load size).
CRC 1.4.2	Variant	Proposals for SysML v2 shall include a capability to model variants that correspond to particular selections that are associated with a variation point.

Req. ID	Req. Name	Text
CRC 1.4.3	Variability Expression and Constraints	Proposals for SysML v2 shall include a capability to model variability expressions that constrain possible variant choices (e.g., 3 axles plus large wheel size or 2 axles plus small wheel size).
CRC 1.4.4	Variant Binding	<p>Proposals for SysML v2 shall include a capability to model the binding between a variant and the model elements that vary.</p> <p>Supporting Information: The binding is intended to enable the use of a separate variability model that defines variation that may span multiple kinds of models such as a SysML model, simulation model, and a CAD model.</p>
CRC 1.5	View and Viewpoint Group	The following specify the requirements associated with View and Viewpoint.
CRC 1.5.1	View Definition	<p>Proposals for SysML v2 shall include a capability to define a class of artifacts that can be presented to a stakeholder.</p> <p>Supporting Information: The View Definition for a document can be thought of as its table of contents along with the list of figures and tables. The View Definition can be specialized, and decomposed into sub-views that can be ordered. An individual View is intended to be a specific artifact, such as a document, diagram, or table that is presented to a stakeholder. The individual View conforms to a View Definition that defines construction methods to create an individual View. The execution of the construction methods involves querying a particular model (or more generally one or more data sources) to select the kinds of model elements, and then presenting the information in a specified format.</p>
CRC 1.5.2	Viewpoint	<p>Proposals for SysML v2 shall include a capability to represent a Viewpoint that frames a set of stakeholders and their concerns. It specifies the requirements a View must satisfy.</p> <p>Supporting Information:</p> <p>The stakeholder and their concerns should be represented in the model.</p> <p>The concern represents aspects of the domain of interest that the stakeholder has an interest in.</p> <p>The intent is to align the view and viewpoint concepts with the update to ISO 42010.</p>

Req. ID	Req. Name	Text
CRC 1.6	Metadata Group	The requirements in this group identify metadata is a kind of model element that can apply to other model elements or to other elements external to the model that refer to a model element (e.g., a model configuration item). Also, refer to the requirement for Analysis Metadata in the Analysis requirements section.
CRC 1.6.1	Version	Proposals for SysML v2 shall include a capability to represent the version of one or more model elements, or of an element external to the model that refers to one or more model elements.
CRC 1.6.2	Time Stamp	Proposals for SysML v2 shall include a capability to represent a model management time stamp for one or more elements, or of another element that refers to one or more model elements.
CRC 1.6.3	Data Protection Controls	<p>Proposals for SysML v2 shall include a capability to represent Data Protection Controls for one or more model elements, or of another element that refers to one or more elements.</p> <p>Supporting Information: This can include markings such as ITAR, proprietary or security classifications</p>
INF 1	Interface Requirements Group	<p>SysML v2 is intended to provide a robust capability to model interfaces that constrain the physical and functional interaction between structural elements. An interface in SysML v2 includes two (2) interface ends, the connection between them, and any constraints on the interaction.</p> <p>Supporting Information:</p> <p>An interface should support the following:</p> <ol style="list-style-type: none"> 1. Different levels of abstraction that include logical, functional, and physical interfaces, nested interfaces, and interface layers; 2. Diverse domains that include a combination of electrical, mechanical, software, and user interfaces; 3. Reuse of interfaces in different contexts; 4. Generation of interface control documents and interface specifications <p>A Port is also used to refer to an Interface End.</p>

Req. ID	Req. Name	Text
INF 1.01	Interface Definition and Reuse	<p>Proposals for SysML v2 shall provide the capability to define an interface that can be used in different contexts that includes the definition of the interface ends, the interface connections, and the constraints on the interaction.</p> <p>Supporting Information:</p> <p>Interfaces must conform to the structural concepts of definition and usage. The constraints can constraint properties, such as conservation laws that can apply to a physical interface, and/or constraints on exchanged items such as protocol constraints that can apply to message exchange, and/or geometric constraints that can apply to a physical interface such as between a plug and socket.</p>
INF 1.02	Interface Usage	<p>Proposals for SysML v2 shall provide the capability to represent a usage of an interface that constrains the interaction between any two (2) structural elements.</p>
INF 1.03	Interface Decomposition	<p>Proposals for SysML v2 shall provide the capability to represent nested interfaces, such as when modeling two electrical connectors with pin to pin connections.</p>
INF 1.04	Interface End Definitions	<p>Proposals for SysML v2 shall provide the capability to represent the definition of an Interface End whose features constrain the interaction of the end, including items that can be exchanged and their direction, behavioral features, and constraints on properties.</p> <p>Supporting Information:</p> <p>Interface End Definitions are also referred to as Port Definitions and Interface End Usages are referred to as Port Usages or Ports for short.</p>
INF 1.05	Conjugate Interface Ends	<p>Proposals for SysML v2 shall provide the capability to reverse the direction of the items that are exchanged in an Interface End.</p>

Req. ID	Req. Name	Text
INF 1.06	Item Definition	<p>Proposals for SysML v2 shall provide the capability to represent the kind of items that can be exchanged between Interface Ends.</p> <p>Supporting Information: The items represent the type of things that are exchanged, such as water or electrical signals. The items may have physical characteristics such as mass, energy, charge, and force, and logical characteristics such as information that is encoded in the physical exchange. In addition to being exchanged, these items may also be stored.</p> <p>An item that is input to a component should become a stored item usage that can be transformed by function usages. An item, such as an engine that is an input and output of an assembly process, may also have the role as a component, when it is assembled into a vehicle. Item Definitions must conform to the structural concepts of definition and usage. The rate at which a usage of an Item Definition is updated may be marked with an update rate that is continuous or discrete valued. (Refer to Behavior Requirement called "Discrete and Continuous Time Behavior")</p>
INF 1.07	Interface Agreement Group	
INF 1.07.1	Item Exchange Constraints	<p>Proposals for SysML v2 shall provide the capability to constrain the interaction between the interface ends that includes constraints on the items to be exchanged, the allowable sequences and directions of those items, timing of the exchange and other characteristics. The items exchanged shall be consistent with the type and direction of the items specified in the connected Interface Ends.</p>
INF 1.07.2	Property Constraints	<p>Proposals for SysML v2 shall provide the capability to constrain the interaction between the interface ends that include mathematical constraints on the properties exposed by the Interface Ends.</p> <p>Supporting Information: The value properties may further be identified as Across or Through variables consistent with standard usage of the terms (e.g. specify properties that are constrained by conservation laws).</p>

Req. ID	Req. Name	Text
INF 1.08	Interface Medium	<p>Proposals for SysML v2 shall include a capability to represent an Interface Medium that enable 2 or more components to interact.</p> <p>Supporting Information: The Interface Medium may represent either an abstract or physical element that connects elements to enable interactions. Examples of an interface medium included an electrical harness, a communications network, a fluid pipe, the atmosphere, or even empty space. The interface medium may connect one to many components, which include support for peer-to-peer, multi-cast, and broadcast communications.</p> <p>Consider replacing the term Interface Medium with Transport Medium.</p>
INF 1.09	Interface Layers	<p>Proposals for SysML v2 shall provide the capability to represent interfaces between layers of an interface stack.</p> <p>Supporting Information:</p> <p>A layer of a stack can be represented as a component. A layer in a stack transforms the data to match the input to the adjacent layer. For example, an application layer may correspond to a component that transforms packets to match the TCP layer, and the TCP layer may correspond to a component that transforms the data to match the IP layer.</p>
INF 1.10	Allocating Functional Exchange to Interfaces	<p>Proposals for SysML v2 shall provide the capability to allocate or bind the outputs and inputs of a function to interface ends (or nested interface ends).</p> <p>Supporting Information:</p> <p>It is expected that there are validation rules to ensure consistency between the inputs and outputs of a function and the interface ends.</p> <p>This allocate or binding should be inherited by the Component subclasses.</p>
LNG 1	Language Architecture and Formalism Requirements Group	<p>This group specifies how the language is structured and defined.</p> <p>Supporting Information: Some concepts may be implemented as user-level model libraries.</p>
LNG 1.1	Metamodel and Profile Group	

Req. ID	Req. Name	Text
LNG 1.1.1	SysML Profile	<p>Proposals for SysML v2 shall be specified as a SysML v2 profile of UML that includes, as a minimum, the functional capabilities of the SysML v1.x profile, and a mapping to the SysML v2 metamodel.</p> <p>Supporting Information: Equivalent functional capability can be demonstrated by mapping the UML metaclasses and SysML stereotypes between SysML v2 and SysML v1.</p>
LNG 1.1.2	SysML Metamodel	<p>Proposals for SysML v2 shall be specified using a metamodel that includes abstract syntax, concrete syntax, semantics, and the relationships between them.</p>
LNG 1.1.3	Metamodel Specification	<p>Proposals for the SysML v2 metamodel shall be specified in MOF or SMOF.</p> <p>Supporting Information: MOF is a subset of SMOF. SMOF provides support for the Metamodel Extension Facility (MEF).</p>
LNG 1.3	Abstract Syntax Group	
LNG 1.3.1	Syntax Specification	<p>Proposals for SysML v2 abstract and concrete syntax shall be specified using MOF or SMOF (including constraints on syntactic structure).</p> <p>Supporting Information:</p> <p>Expressing the syntax formally using a single consistent language which is more understandable to the user.</p>
LNG 1.3.2	View Independent Abstract Syntax	<p>Proposals for the SysML v2 abstract syntax representation of SysML v2 models shall be independent of all views of the models.</p> <p>Supporting Information: Rationale</p> <p>This is intended to define the concept independent of how it is presented. This enables a consistent representation of concepts with common semantics across a diverse range of views, including graphical, tabular, and other textual representations.</p>
LNG 1.4	Concrete Syntax Group	

Req. ID	Req. Name	Text
LNG 1.4.1	Concrete Syntax to Abstract Syntax Mapping	<p>Proposals for the SysML v2 concrete syntax representation of all views of a SysML model shall be separate from, and mapped to the abstract syntax representation of that model. The concrete syntax representation can include one or more images or snippets of images.</p> <p>Supporting Information:</p> <p>Enables views to provide unambiguous concrete representation of the abstract syntax of the model.</p> <p>Enables views to be rendered in a consistent way across tools.</p>
LNG 1.4.2	Graphical Concrete Syntax	Proposals for SysML v2 shall provide a standard graphical concrete syntax.
LNG 1.4.3	Syntax Examples	<p>All examples of model views in the proposals for the SysML v2 specification shall include the concrete syntax of the view, and the mapping to the abstract syntax representation of the parts of the models being viewed.</p> <p>Supporting Information:</p> <p>Experience has shown that the mapping of examples to the concrete and abstract syntax is not always obvious. Making these mappings explicit helps clarify their formal specification.</p>
LNG 1.5	Extensibility Group	
LNG 1.5.1	Extension Mechanisms	<p>Proposals for SysML v2 syntax and semantics shall include mechanisms to subset and extend the language.</p> <p>Supporting Information: This is essential to enable further customization of the language. SysML v1 includes a stereotype and profile mechanism to extend the language.</p>
LNG 1.6	Model Interchange, Mapping, and Transformations Group	
LNG 1.6.3	UML Interoperability	Proposals for SysML v2 shall provide the capability to map shared concepts between SysML and UML.

Req. ID	Req. Name	Text
OTR 1	Interoperability Requirements Group	Other requirements from other topic areas that also relate to interoperability include API 01, LNG 1.6, and the Interoperability Services Group, SVC 6.
OTR 2	Usability Group	<p>An objective for SysML v2 is to address SysML v1 usability issues, and enable systems engineers and others to perform MBSE more effectively. The following usability goals apply to a diverse class of SysML v2 users:</p> <ol style="list-style-type: none"> 1. User understanding when creating or interpreting models 2. User engagement when creating or interpreting models (this particularly applies to consumers of the model data) 3. User productivity when creating models across the lifecycle
OTR 2.1	Usability Evaluation	<p>The SysML v2 submission shall demonstrate how the SysML v2 specification satisfies the following usability criteria to meet the usability goals for the different classes of users and goals.</p> <p>To be provided</p>
PRP 1	Properties, Values and Expressions Requirements Group	The requirements in this group provide a unified representation of the type of properties, variables, constants, operation parameters and return types as well as literal values and value expressions. This includes types to represent variable size collections, compound value types, and measurement units and scales.
PRP 1.01	Unified Representation of Values	<p>Proposals for SysML v2 shall include a capability to represent any value-based characteristic in a unified way, called a value property, which shall include representation of a constant, a variable in an expression or a constraint, state variable, as well as any formal parameter and the return type of an operation.</p> <p>Supporting Information:</p> <p>A classification of "invariant" can be attached to a value property to assert that is does not vary over time. A constant is an invariant value property of some higher-level context (ultimately the "universe" in case of fundamental physics constants).</p> <p>Provisions should be made to distinguish between a fundamental physical or mathematical constant (i.e., Pi) from a constant value within the context of a particular model or model execution (i.e., amplifier gain).</p>
PRP 1.02	Value Type	Proposals for SysML v2 shall include a capability to represent a Value Type as a named definition of the essential semantics and structure of the set of allowable values of a value-based characteristic.

Req. ID	Req. Name	Text
PRP 1.03	Value Expression	Proposals for SysML v2 shall include a capability to represent a value as a literal or through a reusable Value Expression that is stated in an expression language. A Value Expression shall include the capability to represent opaque expressions.
PRP 1.05	Unification of Expression and Constraint Definition	Proposals for SysML v2 shall include a capability to represent a reusable constraint definition in the form of an equality or inequality of value expressions which can be evaluated to true or false.
PRP 1.06	System of Quantities	<p>Proposals for SysML v2 shall include a capability to represent a named system of quantities that support definition of numerical Value Types in accordance with the ISO/IEC 80000 standard.</p> <p>Supporting Information: The typical Systems of Quantities is the ISO/IEC 80000 International System of Quantities (ISQ) with seven base quantities: length, mass, time, electric current, thermodynamic temperature, amount of substance and luminous intensity.</p>
PRP 1.07	System of Units and Scales	<p>Proposals for SysML v2 shall include a capability to represent a named system of measurement units and scales to define the precise semantics of numerical Value Types in accordance with the [ISO/IEC 80000] standard.</p> <p>Supporting Information: Similar to SysML v1 QUDV, SysML v2 should include model libraries representing the [ISO/IEC 80000] units, as well as the conversion to US Customary Units defined in [NIST SP 811] Appendix B.</p>
PRP 1.08	Range Restriction for Numerical Values	<p>Proposals for SysML v2 shall include a capability to represent a value range restriction for any numerical Value Type.</p> <p>Supporting Information: This requirement allows further restriction of the range of values beyond what is specified by its type. A simple example is a planar angle typed by a real number Value Type and a degree measurement scale. However, the value range may be further restricted from 0 to 360 degrees for positioning a rotational knob. This can also include the definition of optional lower and upper bounds on an associated measurement scale.</p>
PRP 1.10	Primitive Data Types	<p>Proposals for SysML v2 shall include a capability to represent the following primitive data types as a minimum: signed and unsigned integer, signed and unsigned real, string, Boolean, enumeration type, ISO 8601 date and time, and complex.</p> <p>Supporting Information: These are intended to be represented in a Value Type Library as they are in SysML v1.</p>

Req. ID	Req. Name	Text
PRP 1.11	Variable Length Collection Value Types	Proposals for SysML v2 shall include a capability to represent variable length value collections where each member of the collection is typed by a particular Value Type and is referable by index, and where the collection may be one of the established collection types: sequence (ordered, non-unique), set (unordered, unique), ordered set (ordered, unique) or bag (unordered, non-unique).
PRP 1.12	Compound Value Type	<p>Proposals for SysML v2 shall include a capability to represent both scalar and compound Value Types, where a scalar Value Type represents elements with a single value, and compound Value Type represents elements with a fixed number of component values, where each component value is typed in turn by a scalar Value Type or another compound Value Type.</p> <p>Supporting Information: Such compound Value Types are needed to support the representation of vector, matrix, higher order tensor, complex number, quaternion, and other richer Value Types.</p>
PRP 1.15	Probabilistic Value Distributions	Proposals for SysML v2 shall include a capability to represent the value of a quantity with a probabilistic value distribution, including an extensible mechanism to detail the kind of distribution, i.e. the probability density function for continuous random variables, or the probability mass function for discrete random variables.
PRP 1.19	Materials with Properties	<p><html></p> <p>Proposals for SysML v2 shall include a capability to represent named materials with their material properties in a model library and assignment of such materials to physical elements such as hardware components.</p> <p>Supporting information: This requirement is intended to specify a model library with a generic material kind that has generic material properties that can be further specialized. Examples of generic material properties include density, hardness, and tensile yield strength.</p>
RML 1	Example Model and Model Libraries Group	
RML 1.1	Example Model	Proposals for SysML v2 shall include an example model that demonstrates the application of the SysML v2 language concepts to a commonly understood domain.
RQT 1	Requirement Group	The requirements in this group are used to represent requirements and their relationships.

Req. ID	Req. Name	Text
RQT 1.1	Requirement Definition Group	
RQT 1.1.1	Requirement Definition Name	Proposals for SysML v2 shall include a capability to represent a requirement definition that can be used to constrain a solution.
RQT 1.1.2	Requirement Identifier	Proposals for SysML v2 shall include a capability to represent an identifier for each requirement that is adaptable to a user defined numbering scheme, and can be set to not change.
RQT 1.1.3	Requirement Attributes	<p>Proposals for SysML v2 shall include a capability to represent the following optional requirement attributes for a requirement definition.</p> <ul style="list-style-type: none"> • Requirement Status • Priority • Risk • Originator/Author • Owner • User-defined Attributes (e.g., confidence level, uncertainty status, etc.) <p>Supporting Information: These attributes are derived from commonly used attributes as defined in the INCOSE Handbook and ReqIF, and should be reconciled with other model element metadata and model element attributes that apply more generally.</p>
RQT 1.1.4	Textual Requirement Statement	Proposals for SysML v2 shall include a capability to represent a requirement definition that contains an optional textual requirement statement.
RQT 1.1.5	Restricted Requirement Statement Group	<p><html></p> <p>Supporting Information: Refer to Restricted Use Case Modeling (RUCM) [36] as an example of a restricted requirement statement.</p>
RQT 1.1.5.1	Restricted Requirement Statement	Proposals for SysML v2 shall include a capability to represent a requirement definition that contains an optional restricted requirement statement which may include predefined key words and sentence structures.
RQT 1.1.5.2	Restricted Requirement Statement Extensibility	Proposals for SysML v2 shall include a capability to extend a restricted requirement statement with additional key words and sentence structures.

Req. ID	Req. Name	Text
RQT 1.1.5.3	Restricted Requirement Statement Transformation	SysML v2 will include a capability to maintain traceability between the restricted requirement statement and the textual requirement statement and/or the formal requirement statement.
RQT 1.1.6	Formal Requirement Statement Group	
RQT 1.1.6.1	Formal Requirement Statement	<p>Proposals for SysML v2 shall include a capability to represent a requirement definition that contains an optional formal requirement statement that includes one or more constraints that an acceptable solution must satisfy.</p> <p>Supporting Information: It is desired to also enable the element that is intended to satisfy the requirement to contain the formal requirement statement. This can provide a more lightweight modeling style.</p>
RQT 1.1.6.2	Assumptions	<p>Proposals for SysML v2 shall include a capability to represent a formal requirement statement that includes one or more expressions to specify the assumptions and conditions for acceptable solutions (e.g., the weight of a car includes the fuel weight)</p> <p>Supporting Information: This should be consistent with the concept of Assumption that is applied in other parts of the model.</p>
RQT 1.2	Groups of Requirements	
RQT 1.2.1	Requirement Group	<p>Proposals for SysML v2 shall provide the capability to model a group of requirements that are used to constrain a solution.</p> <p>Supporting Information: This is intended to be a sub-class of Element Group.</p>
RQT 1.2.2	Requirement Usage (localized)	<p>Proposals for SysML v2 shall include a capability to represent localized values of a requirement usage that can over-ride the values of its requirement definition.</p> <p>Supporting Information: The structural concepts of definition, usage, configuration, and individuals are intended to support reuse of requirement definitions, and unambiguously define a tree of requirements that specify a design configuration or an individual element.</p>

Req. ID	Req. Name	Text
RQT 1.2.3	Requirement Usage Identifier	Proposals for SysML v2 shall include a capability to represent each requirement in a requirement group with an identifier that is adaptable to a user defined numbering scheme, and that the user can specify whether the identifier can change or not.
RQT 1.2.4	Requirement Usage Ordering	<p>Proposals for SysML v2 shall include a capability to represent the order of each requirement in a requirement group that is not constrained by its requirement identifier.</p> <p>Supporting Information: This primarily allows the user to further organize the requirements, but it does not impact the meaning of the requirements. For example, there may be a requirement group with one requirement to open a valve and another requirement to close a valve. The user may want to order the open requirement as the first requirement in the group.</p>
RQT 1.3	Requirement Relationships Group	
RQT 1.3.1	Requirement Specialization	Proposals for SysML v2 shall include a capability to represent a generalization relationship that relates a specialized requirement definition to a more general requirement definition.
RQT 1.3.2	Requirement Satisfaction	<p>Proposals for SysML v2 shall include a capability to represent a satisfy relationship that relates a requirement to a model element that is asserted to satisfy it.</p> <p>Supporting Information: This is intended to be a specialization of the Conform Relationship.</p>
RQT 1.3.3	Requirement Verification	Proposals for SysML v2 shall include a capability to represent a verify relationship that relates a verification case to the requirement it is intended to verify.
RQT 1.3.4	Requirement Derivation	Proposals for SysML v2 shall include a capability to represent a derive relationship that relates a derived requirement to a source requirement.
RQT 1.3.5	Requirement Group Relationship	<p>Proposals for SysML v2 shall include a capability to represent a relationship between a requirement group and the members of the group that can include either a requirement or another requirement group.</p> <p>Supporting Information: This relationship groups requirements into a shared context.</p>

Req. ID	Req. Name	Text
RQT 1.3.6	Relationships to a Requirement Group	<p>Proposals for SysML v2 shall specify the meaning of relationships with a requirement group on each member of the requirement group.</p> <p>Supporting Information: This applies more generally to element groups.</p>
RQT 1.4	Requirement Supporting Information	<p>Proposals for SysML v2 shall include a capability to represent supporting information for a requirement, requirement definition, and a requirement group.</p> <p>Supporting Information: This is a kind of annotation that applies more generally to any model element.</p>
RQT 1.5	Goals, Objectives, and Evaluation Criteria	<p>Proposals for SysML v2 shall include a capability to represent goals, objectives, and evaluation criteria.</p> <p>Supporting Information:</p> <p>Criteria can be viewed as a superclass of a requirement that is used as a basis for evaluation, but does not specify specific values. For example, a cost requirement may be to require the cost to be less than a particular value, where-as a cost criterion may be to select a design with the lowest cost. Goals can be a type of criteria. For example, a goal of the system is to minimize the cost. An objective represents a desired end state. For example, the mission objective is to land a person on the moon and safely return them to earth. An objective can be thought of as a kind of requirement.</p> <p>Refer to Business Motivation Metamodel (BMM).</p>

Req. ID	Req. Name	Text
STC 1	Structure Requirements Group	<p>This group of requirements is intended to represent composable, deeply nested, connectible structure that supports definition of a family of configurations, specific configurations, and individual elements that are uniquely identified.</p> <p>Supporting Information:</p> <p>These requirements refer to definition elements and usage elements analogous to structured classifiers and classifier features in UML. A particular specialization of these concepts in SysML v1 is used to represent blocks and parts,</p> <p>The requirements also refer to configuration elements and individual elements. Configuration elements are used to unambiguously represent deeply nested structures as a tree of configuration elements. Individual elements are used to represent a particular element that can be uniquely identified, which is not to be interpreted as a UML or SysML instance. A particular system, such as a system with a serial number on the manufacturing floor, can be represented by an individual element which in turn can be represented as a tree of individual elements.</p> <p>The terms Component Definition and Component Usage refer to a particular kind of Definition Element and Usage Element that are analogous to a Block and Part in SysML v1. The terms Item Definition and Item Usage are also used to refer to a particular kind of Definition Element and Usage Element that correspond to something that flows through a system, such as Water. Component and Item are introduced in the Interface requirements section.</p>
STC 1.01	Modular Unit of Structure	<p>Proposals for SysML v2 shall include a capability to represent a modular unit of structure that defines its characteristics through value properties, interface ends (ports), constraints, and other structural and behavioral features.</p> <p>Supporting Information: The term used in this RFP to refer to a modular unit of structure is Definition Element. Such modular units of structure can be regarded as the fundamental named building blocks from which system representations, i.e. architectures, can be constructed. The capability enables modeling multiple levels of a hierarchy (e.g., system-of systems, system, subsystem and components) that can include logical and physical representations of hardware, software, information, people, facilities, and natural objects.</p> <p>The concept model refers many specializations of Definition Element. One example is the Component Definition which is intended to represent any level of a product structure. The concept model refers to an Item Definition as a specialized Definition Element to represent an element that flows through a system, such as water or a message. As noted above, the decomposition of Definition Elements may include variability that may be represented by multiplicity, subclasses, and/or a range of property values, which is removed when selecting a specific design configuration.</p>

Req. ID	Req. Name	Text
STC 1.02	Usage Element	Proposals for SysML v2 shall include a capability to represent the usage of a Definition Element, called a Usage Element, in order to support reuse in different contexts.
STC 1.03	Generic Hierarchical Structure	Proposals for SysML v2 shall include a capability to represent hierarchical composition structure of Definition and/or Usage Elements.
STC 1.04	Reference Element	Proposals for SysML v2 shall include a capability to represent a reference from one element to any other element within a shared scope.
STC 1.05	Multiplicity of Usage	<p>Proposals for SysML v2 shall include a capability to define the multiplicity of any particular Usage Element or Reference Element as an integer range (i.e., lower bound and upper bound).</p> <p>Supporting Information:</p> <p>Multiplicity refers to the number of Individual Elements.</p>
STC 1.06	Definition Element Specialization	Proposals for SysML v2 shall include a capability to represent a specialization from a more general Definition Element into a more specific Definition Element, where the more specific element inherits all features of the more general element.
STC 1.07	Unambiguous Deeply Nested Structure	Proposals for SysML v2 shall support a capability to unambiguously represent Usage Elements at any level of nesting.
STC 1.08	Structure With Variability	<p>Proposals for SysML v2 shall include a capability to represent multiple possible variant configurations of a system-of-interest with a single collection of Definition Elements and Usage Elements, where at each usage level in the (de)composition, a variant from different possible variant choices can be selected.</p> <p>Supporting Information: A Structure With Variability enables the definition of a product line architecture, see e.g. ISO 26550. Some common variant choices are defined by multiplicity range, sub-classes, and different values of a value property.</p>

Req. ID	Req. Name	Text
STC 1.10	Structure of an Individual	<p>Proposals for SysML v2 shall include a capability to represent a (de)composition of an Individual Element that is uniquely identifiable, and that can conform to an associated Structure resolved to a Single Variant and/or a Structure with Variability.</p> <p>Supporting Information: Such a digital representation of a real-world system is sometimes called a 'digital twin'. The elements in a Structure of an Individual are typically designated by a unique serial number, a batch number or an effectivity code.</p>
STC 1.11	Usage Specific Localized Type	<p>Proposals for SysML v2 shall include a capability to represent local override, redefinition, or addition of features with respect to the features defined by its more general type at any level of nesting.</p> <p>Supporting Information: The more-general to more-specific type chain is: Definition Element - direct Usage Feature - deeply nested Usage Feature - Configuration Element - Individual Element.</p> <p>The localized usage should support capabilities equivalent to redefinition and sub-setting for usage elements at any level of nesting.</p>
VRF 1	Verification and Validation Requirements Group	<p>The requirements in this group represent how to evaluate whether systems satisfy their requirements using verification methods.</p> <p>Supporting Information: The requirements for validation are not called out explicitly, but are intended to be supported in a similar way as the requirements for verification.</p>
VRF 1.1	Verification Context	<p>Proposals for SysML v2 shall include the capability to model a Verification Context that includes the unit-under-verification, the verification case, and the verification system and associated environment that performs the verification.</p>
VRF 1.2	Verification Case Group	
VRF 1.2.1	Verification Case	<p>Proposals for SysML v2 shall include the capability to model a verification case to evaluate whether one or more requirements are satisfied by a unit under verification.</p> <p>Supporting Information: This is intended to be a specialization of Case.</p>
VRF 1.2.2	Verification Objectives	<p>The verification case shall include verification objectives to be implemented by the verification activities.</p>

Req. ID	Req. Name	Text
VRF 1.2.3	Verification Success Criteria	The verification case shall include the criteria used to evaluate whether the verification objectives are met, the requirements are satisfied, and any subset of verification steps in a verification case are successfully performed.
VRF 1.2.4	Verification Methods	<p>The verification case shall include the methods used to verify the requirements. The methods, including inspection, analysis, demonstration, test, external verification, engineering reviews, and similarity, shall be included in a library. More than one method can be applied to verify a requirement.</p> <p>Supporting information:</p> <p>A verification method may include additional classification such as qualification test and acceptance test.</p> <p>An external verification is a method used in some industries, such as an Underwriters Labs.</p>
VRF 1.3	Verification System	Proposals for SysML v2 shall include the capability to model the system and associated environment that is used to verify the unit under verification. (Note: the verification system may include verification elements that are combinations of operational and simulated hardware, software, people, and facilities.)
VRF 1.4	Verification Relationships Group	
VRF 1.4.1	Verification Objectives to Verification Cases	Proposals for SysML v2 shall include the capability to model relationship between the verification cases and their verification objectives.
VRF 1.4.2	Validate Relationship	<p>Proposals for SysML v2 shall include the capability to model the relationship between the validation case and the model element being validated.</p> <p>Supporting Information: An element being validated may represent a requirement, design, as-built system, model, etc.</p> <p>The Verify Relationship is included in the requirements section.</p>

0.4.2 Non-Mandatory Language Requirements Table

Table 2. Non-Mandatory Language Requirements Table

Req. ID	Req. Name	Text
ANL 1	Analysis Requirements Group	

Req. ID	Req. Name	Text
ANL 1.10	Analysis Model - System Model Transformation	<p>Proposals for SysML v2 may include the capability to represent the transformation and the mapping between the analysis model and the system model.</p> <p>Supporting Information:</p> <p>This transformation will represent the algorithm or derivation process, if used, for generating analysis models from system model (or vice versa), and the mapping will provide a mechanism to verify and synchronize analysis models when the system model changes (or vice versa). Refer to the requirement for Model Mappings and Transformations under the Language Architecture and Formalism Requirements.</p>
ANL 1.12	Analysis Infrastructure	Proposals for SysML v2 may include the capability to represent the hardware, software, and the personnel (analysis experts) required for performing the analysis.
ANL 1.14	Decision Group	
ANL 1.14.1	Trade-off	Proposals for SysML v2 may include a capability to represent an evaluation among a set of alternatives that can result in a decision based on a set of criteria. A trade-off may be dependent on other decisions.
ANL 1.14.3	Decision Expression	Proposals for SysML v2 may include a capability to model decision expressions that constrain the possible decisions (e.g., alternative A OR (alternative B and alternative C)).
BHV 1	Behavior Requirements Group	
BHV 1.03	Function-based Behavior Group	
BHV 1.03.2	Composite Input and Output	<p>Proposals for SysML v2 may include the capability to model composite inputs and outputs of function-based behavior with separate flows defined for the constituent inputs and outputs.</p> <p>Supporting Information:</p> <p>Refer to a Simulink Bus Object and a Modelica Expandable Connector</p>

Req. ID	Req. Name	Text
BHV 1.03.5	Behavior Library	Proposals for SysML v2 may include a library that can be populated with commonly used behaviors to support execution that includes functions to store items, such as data and energy.
BHV 1.09	State History	<p>Proposals for SysML v2 may provide the capability to represent a state history of an individual element as a sequence of snapshots to describe how the individual element changes over time. The state history may contain a reference time scale consistent with QUDV, and can include a start time, end time, and time step.</p> <p>Supporting Information:</p> <p>A snapshot represents the state of an individual element at a point in time by capturing the values of each of its value properties. An example is a snapshot of a vehicle that may include the value of its position, velocity, and acceleration at a point in time, and the snapshot of its engine that may include the value of its power-out and temperature at the same point in time. The value properties that vary with time are also called state variables.</p> <p>The state history of a configuration element represents the default state history for each of its conforming individual elements.</p>
CNF 1	Conformance Requirements Group	
CNF 1.3	Model Interoperability Conformance	Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 model interchange, model mappings and transformations requirements in LNG 1.6.
CRC 1	Cross-cutting Requirements Group	
CRC 1.2	Model Element Group	
CRC 1.2.1	Model Element	Proposals for SysML v2 may include a root element that contains features that apply to all other kinds of elements in the model.
CRC 1.3	Model Element Relationships Requirements Group	

Req. ID	Req. Name	Text
CRC 1.3.11	Copy Relationship	<p>Proposals for SysML v2 may include a capability to represent a Copy Relationship where one side of the relationship refers to the element (or elements) being copied and the other side of the relationship refers to the copy (or copies).</p> <p>Supporting Information:</p> <p>The primary goals for this relationship are to establish provenance to support traceability, and to enable reuse of catalog items. This relationship provides the ability to copy elements such as a Container (e.g., package) and its contents, within a model and from one model to another. Additional constraints can be defined to specify the rules for what part of the element being copied can be modified in the copy. It is assumed that updates to the copied element are not propagated, unless there is a rule to support this.</p>
INF 1	Interface Requirements Group	
INF 1.07	Interface Agreement Group	
INF 1.07.3	Geometric Constraints	<p>Proposals for SysML v2 may provide the capability to constrain the interaction between the interface ends that include geometrical constraints on either Interface End.</p> <p>Supporting Information: An example are the geometric constraints associated with connecting a plug and socket.</p>
LNG 1	Language Architecture and Formalism Requirements Group	
LNG 1.2	Semantics Group	
LNG 1.2.1	Semantic Model Libraries	<p>Proposals for SysML v2 semantics may be modeled with SysML v2 model libraries.</p> <p>Supporting Information:</p> <ol style="list-style-type: none"> 1. Simplifies the language when model libraries are used to extend the base declarative semantics without additional abstract syntax. 2. Enables SysML to be improved and extended more easily by changes and additions to model libraries, rather than always through abstract syntax.

Req. ID	Req. Name	Text
LNG 1.2.2	Declarative Semantics	<p>Proposals for SysML v2 models may be grounded in a declarative semantics expressed using mathematical logic.</p> <p>Supporting Information:</p> <p>Semantics are defined formally to reduce ambiguity. Declarative semantics enable reasoning with mathematical proofs. This contrasts with operational semantics that requires execution in order to determine correctness.</p> <p>The semantics provide the meaning to the concepts defined in the language, and enable the ability to reason about the entity being represented by the models.</p>
LNG 1.2.3	Reasoning Capability	<p>Proposals for SysML v2 may provide a subset of its semantics that is complete and decidable.</p> <p>Supporting Information: This enables the ability to reason about the entity being modeled by querying the model, and returning results that satisfy the specified set of constraints.</p> <p>As an example, a query could return valid vehicle configurations that have a vehicle mass<2000kg AND vehicles that have a sunroof.</p>
LNG 1.4	Concrete Syntax Group	
LNG 1.4.4	Textual Concrete Syntax	<p>Proposals for SysML v2 may provide a standard human readable textual concrete syntax.</p> <p>Supporting information: Graphical and textual concrete syntax representations can be used in combination to more efficiently and effectively present the model. Refer to Alf as an example of a textual notation.</p>
LNG 1.5	Extensibility Group	
LNG 1.5.2	Extensibility Consistency	<p>Proposals for all SysML v2 extension mechanisms may be applicable to SysML v2 syntax (concrete and abstract) and semantics, and be consistent with how these are specified in SysML v2.</p> <p>Supporting Information:</p> <p>The SysML v2 Specification includes syntax, semantics, and vocabulary, so extending the language requires all of these to be extensible.</p>

Req. ID	Req. Name	Text
LNG 1.6	Model Interchange, Model Mapping, and Transformations Group	
LNG 1.6.1	Model Interchange	<p>Proposals for SysML v2 may provide a format for unambiguously interchanging the abstract syntax representation of a model and the concrete syntax representation of views of the model, which supports exchange of models that are created using either the metamodel or the profile.</p> <p>Supporting Information: The interchange should facilitate long term retention, file exchange, and version upgrades.</p> <p>Consider consistency with related interchange standards, such as AP233. For the concrete syntax, consider consistency with Diagram Definition and Diagram Interchange.</p>
LNG 1.6.2	Model Mappings and Transformations	<p>Proposals for SysML v2 may provide a capability to specify model mappings and transformations.</p> <p>Supporting Information: SysML may be used to represent the metamodel of other languages and data sources to enable transformation between SysML models, other data sources, and models in other languages. These languages include languages for queries, validation rules, expressions, viewpoint methods, and transformations.</p> <p>A common need is to map elements between SysML and Excel that supports import of Excel data into a SysML model, and export of SysML model elements to Excel. Another example is a mapping between SysML models and Simulink models.</p>
PRP 1	Properties, Values and Expressions Requirements Group	<html>
PRP 1.04	Logical Expressions	<p>Proposals for SysML v2 may include a capability to represent, as part of the Expression language, logical expressions that support as a minimum the standard boolean operators AND, OR, XOR, NOT, and conditional expressions like IF-THEN-ELSE and IF-AND-ONLY-IF, in which symbols bound to any characteristics (e.g. value properties or usage features) may be used.</p>

Req. ID	Req. Name	Text
PRP 1.09	Automated Quantity Value Conversion	<p>Proposals for SysML v2 may include a capability to represent all information necessary to perform automated conversion of the value of a quantity (typed by a numerical Value Type) expressed in one measurement scale to the value expressed in another compatible measurement scale with the same quantity kind.</p> <p>Supporting Information: This capability is needed to rebase a set of (smaller) system models coming from various contributors on a single coherent set of measurement scales, so that an integrated (larger) system model can be consistently constructed and analyzed.</p>
PRP 1.13	Discretely Sampled Function Value Type	<p>Proposals for SysML v2 may include a capability to represent variable length sets of values that constitute discrete time series data, frequency spectra, temperature dependent material properties, and any other datasets that can be represented through a discretely sampled mathematical function.</p> <p>Supporting Information: Such a discretely sampled function can be defined by a tuple of one or more Value Types that prescribe the type of the domain (independent) variables, and a tuple of one or more Value Types that prescribe the range (dependent) variables, as well as a variable length sequence of tuples that represent the actual set of sampled values.</p>
PRP 1.14	Discretely Sampled Function Interpolation	<p>Proposals for SysML v2 may include a capability to represent an interpolation scheme for a Discretely Sampled Function Value Type for derivation of the function's range values for domain values that are in-between sampled values.</p>
PRP 1.16	System Simulation Models	<p>Proposals for SysML v2 may include a capability to represent signal flow graph models and lumped parameter models as well as combinations thereof.</p> <p>Supporting Information: See [SysPISF] for details.</p> <p>This requirement is augmented by the analysis requirements.</p>
PRP 1.17	Across and Through Value Properties	<p><html></p> <p>Proposals for SysML v2 may include a capability to define across and through properties of flows on Interface Ends that participate in representing physical interactions in lumped parameter models.</p> <p>Supporting Information: Typically, the across and through properties are defined together as a pair, where the across property does not conserve energy and the through property does. For example, in a lumped parameter model of an electric circuit, the across and through properties are voltage and current respectively. See [SysPISF] for details.</p>

Req. ID	Req. Name	Text
PRP 1.18	Basic Geometry	<p>Proposals for SysML v2 may include a capability to represent basic two- and three-dimensional geometry of a structural element, including a base coordinate frame as well as relative orientation and placement of shapes through nested coordinate frame transformations, where the basic shape definitions are provided in a model library.</p> <p>Supporting Information: These capabilities are intended to provide basic geometry and coordinate frame representations to support specification of physical envelopes. The intent is that each block or equivalent will have its own reference coordinate system, and transformations can be applied between coordinate systems of different blocks. The shape of a block is defined as a property (e.g., 3-dimensional rectangular shape with length, height, and depth) whose values can be defined in its reference coordinate system.</p> <p>Consider references to standard formats (e.g., ISO 10303 (STEP), IGES)</p>
PRP 1.20	Equivalent Element	<p>Proposals for SysML v2 shall include a capability to represent an element that can reference another model element or an external element, indicating that the reference element is semantically equivalent to the referenced element.</p> <p>Supporting Information:</p> <p>This requirement is intended to be supported by transclusion, which enables the content that is referenced to be displayed in place of the reference element.</p> <p>A URI or URL can be used to refer to an external element.</p> <p>Example: Define a reference element 'x' that has a real value of 100.0. Transclude this reference element in "The top speed of this car shall be greater than <x> mph.", such that it is rendered textually as "The top speed of this car shall be greater than 100.0 mph."</p> <p>Example: A reference element called 'part A' refers to 'part A1' in a bill of materials in a PLM application</p> <p>SysML v1.X Constructs: Adjunct property (partial satisfaction)</p>
RML 1	Example Model and Model Libraries Group	

Req. ID	Req. Name	Text
RML 1.2	Model Libraries	<p>Proposals for SysML v2 may include Model Libraries that contain generic elements that can be further specialized to define domain specific libraries in the following domain areas:</p> <ul style="list-style-type: none"> • Primitive Value Types • Units and Quantity Kinds • Components • Natural environments • Interfaces • Behaviors • Requirements • Verification methods • Analyses • Basic geometric shapes • Basic material kinds • Viewpoint methods • View definitions (i.e. different kinds of documents and other artifacts) • Domain-specific symbols <p>Supporting information: The generic elements provide a common starting point for development of domain specific model libraries that can be elicited in future RFPs and/or the open source community.</p>
RQT 1	Requirement Group	
RQT 1.3	Requirement Relationships Group	
RQT 1.3.7	Relationship Logical Constraint	<p>Proposals for SysML v2 may include a capability to represent a logical expression (e.g. AND, OR, XOR, NOT, and conditional expressions like IF-THEN-ELSE and IF-AND-ONLY-IF) to one or more requirement relationships of the same kind, with an associated completeness property (e.g., complete satisfaction or partial satisfaction) and with a default expression of "And" for the logical expression.</p> <p>Supporting Information: As an example, two blocks that have a satisfy relationship with the same requirement are asserted to completely satisfy the requirement by default</p>
STC 1	Structure Requirements Group	

Req. ID	Req. Name	Text
STC 1.09	Structure Resolved to a Single Variant	<p>Proposals for SysML v2 may include a capability to represent a single variant of a system-of-interest as a tree of Configuration Elements that establishes a fully expanded hierarchical (de)composition that can conform to an associated Structure with Variability where a single selection is made for each variability choice (aka variation point).</p> <p>Supporting Information: A SysML v2 implementation should support auto-generation of a tree of configuration elements from a decomposition of definition elements with variability based on a set of rules. A SysML v2 implementation should ideally also provide a capability to semi-automatically generate the reverse transformation from a tree of configuration elements to a decomposition of definition elements.</p>
VRF 1	Verification and Validation Requirements Group	
VRF 1.2	Verification Case Group	
VRF 1.2.5	Verification Activity	<p>Proposals for SysML v2 may include a verification method that includes activities to collect the verification data, and include the ability to reference this data.</p> <p>Supporting Information: The data may be extensive and not captured directly in the model.</p>
VRF 1.2.6	Verification Evaluation Activity	<p>Proposals for SysML v2 may include a verification method that includes activities to evaluate the verification data and the verification success criteria and generate a verification result of how well the requirements are satisfied (e.g., pass/fail/unverified).</p>

0.4.3 Mandatory Language Requirements - Satisfied-by Table

Table 3. Mandatory Language Requirements - Satisfied-by Table

ID	Name	Satisfied?	Satisfied-by	Comment
ANL 1	Analysis Requirements Group			
ANL 1.01	Subject of the Analysis	Yes	AnalysisCases	
ANL 1.02	Analysis	Partial	AnalysisCases	need support to specify analysis model
ANL 1.03	Parameters of Interest	Partial	AnalysisCases	need to identify moe, mop

ID	Name	Satisfied?	Satisfied-by	Comment
ANL 1.04	Analysis Case	Yes	AnalysisCases	
ANL 1.05	Analysis Objectives	Yes	AnalysisCases	
ANL 1.06	Analysis Scenarios	Yes	AnalysisCases	
ANL 1.07	Analysis Assumption	Yes	AnalysisCases	
ANL 1.08	Analysis Decomposition	Yes	AnalysisCases	
ANL 1.09	Analysis Model	No		
ANL 1.11	Analysis Result	Partial	AnalysisCases	
ANL 1.13	Analysis Metadata	No		
ANL 1.14	Decision Group			
ANL 1.14.2	Alternative	No		
ANL 1.14.4	Decision	No		
ANL 1.14.5	Criteria	No		
ANL 1.14.6	Rationale	No		
BHV 1	Behavior Requirements Group			
BHV 1.01	Behavior	Yes	Behaviors	
BHV 1.02	Behavior Decomposition	Yes	Behaviors	
BHV 1.03	Function-based Behavior Group			
BHV 1.03.1	Function-based Behavior	Partial	Actions	integrate seq diag, events
BHV 1.03.3	Function-based Behavior Constraints	Partial	Actions Constraints	constraint on start/edn snapshot

ID	Name	Satisfied?	Satisfied-by	Comment
BHV 1.03.4	Opaque Behavior	Partial	Actions Annotations TextualRepresentation	textual representation
BHV 1.03.6	Structure Modification Behavior	No		
BHV 1.04	State-based Behavior Group			
BHV 1.04.1	Regions, States, and Transitions	Yes	States	confirm state change can result from change in structure
BHV 1.04.2	Integration of Function-based Behavior with Finite State Behavior	Yes	States	
BHV 1.04.3	Integration of Constraints with Finite State Behavior	Yes	States Constraints	
BHV 1.05	Discrete and Continuous Time Behavior	Yes	Behaviors	
BHV 1.06	Events	Partial	Actions AcceptActionUsage States TransitionUsage	Signal events are supported, but not time or change events.
BHV 1.07	Control Nodes	Partial	Actions Control Nodes	Control nodes are supported on control flows, but not on item flows.
BHV 1.08	Time Constraints	No		
BHV 1.10	Behavior Execution	No		
BHV 1.11	Integration between Structure and Behavior			
BHV 1.11.1	Allocation of Behavior to Structure	Partial	Actions PerformActionUsage	need support for multiple parts participating in a single action (val case 4b)

ID	Name	Satisfied?	Satisfied-by	Comment
BHV 1.11.2	Integration of Control Flow and Input/Output Flow	Yes	Parts Actions States Connectors Succession Interactions ItemFlow	
BHV 1.11.3	Storing Items in Storage Elements Requirements Group			
BHV 1.11.3.1	Storage Element and Stored Item Usages	No		
BHV 1.11.3.2	Create, Modify, and Consume Stored Items	No		
BHV 1.12	Case	Yes	Cases	
CNF 1	Conformance Requirements Group			
CNF 1.1	Metamodel Conformance	No		
CNF 1.2	Profile Conformance	No		
CNF 1.3	Model Interoperability Conformance	No		
CNF 1.4	Traceability Matrix	Yes	1 SysML v2 Specific Requirements	
CRC 1	Cross-cutting Requirements Group			
CRC 1.1	Model and Model Library Group			
CRC 1.1.1	Model	Partial	Namespaces	Can use package, but no explicitly identified model construct
CRC 1.1.2	Model Library	Partial	Namespaces	Can use package, but no explicitly identified model library construct

ID	Name	Satisfied?	Satisfied-by	Comment
CRC 1.1.3	Container	Yes	Namespaces	
CRC 1.2	Model Element Group			
CRC 1.2.2	Unique Identifier	Yes	Elements	
CRC 1.2.3	Name and Aliases	Yes	Elements	
CRC 1.2.4	Definition / Description	Yes	Annotations	
CRC 1.2.5	Annotation	Partial	Annotations	Need link
CRC 1.2.6	Element Group	No		
CRC 1.2.7	Additional Cross-Cutting Concepts Group			
CRC 1.2.7.1	Problem	No		
CRC 1.2.7.2	Risk	No		
CRC 1.3	Model Element Relationships Requirements Group			
CRC 1.3.01	Relationship	Yes		
CRC 1.3.02	Derived Relationship	Not Determined		
CRC 1.3.03	Dependency Relationship	Yes	Dependencies	
CRC 1.3.04	Cause-Effect Relationship	No		
CRC 1.3.05	Explanation Relationship	No		
CRC 1.3.06	Conform Relationship	No		
CRC 1.3.07	Refine Relationship	No		
CRC 1.3.08	Allocation Relationship	No		

ID	Name	Satisfied?	Satisfied-by	Comment
CRC 1.3.09	Element Group Relationship	No		
CRC 1.3.10	Navigation Relationship	Not Planned		This is addressed by API xternal relationship service
CRC 1.4	Variability Modeling Group			
CRC 1.4.1	Variation Point	Yes	General	
CRC 1.4.2	Variant	Yes	General	
CRC 1.4.3	Variability Expression and Constraints	Yes	Constraints	
CRC 1.4.4	Variant Binding	No		
CRC 1.5	View and Viewpoint Group			
CRC 1.5.1	View Definition	Yes	Views	
CRC 1.5.2	Viewpoint	Yes	Views	
CRC 1.6	Metadata Group	No		
CRC 1.6.1	Version	Not Determined		Refer to API versioning service
CRC 1.6.2	Time Stamp	Not Determined		Refer to API versioning service
CRC 1.6.3	Data Protection Controls	Not Determined		Consider language extension that includes access control markings
INF 1	Interface Requirements Group			
INF 1.01	Interface Definition and Reuse	Yes	Interfaces	
INF 1.02	Interface Usage	Yes	Interfaces	
INF 1.03	Interface Decomposition	Yes	Interfaces	

ID	Name	Satisfied?	Satisfied-by	Comment
INF 1.04	Interface End Definitions	Yes	Ports	
INF 1.05	Conjugate Interface Ends	Yes	Ports	
INF 1.06	Item Definition	Partial	Items	need support for update rate
INF 1.07	Interface Agreement Group			
INF 1.07.1	Item Exchange Constraints	No		This will be addressed by conartaining transfers.
INF 1.07.2	Property Constraints	No		
INF 1.08	Interface Medium	No		
INF 1.09	Interface Layers	Not Determined		A layer is a kind of part with ports. Consider adding to model library.
INF 1.10	Allocating Functional Exchange to Interfaces	Partial		An interface can reference an item flow.
LNG 1	Language Architecture and Formalism Requirements Group			
LNG 1.1	Metamodel and Profile Group			
LNG 1.1.1	SysML Profile	Partial		The minimum requirements for a SysML v2 profile will be satisfied by the SysML v1 to SysML v2 transformation specification, which provides a way to represent SysML v1 concepts in SysML v2.
LNG 1.1.2	SysML Metamodel	Yes	1. Abstract Syntax	
LNG 1.1.3	Metamodel Specification	Partial	1. Abstract Syntax	
LNG 1.3	Abstract Syntax Group			
LNG 1.3.1	Syntax Specification	Partial	1. Abstract Syntax	concrete syntax in BNF

ID	Name	Satisfied?	Satisfied-by	Comment
LNG 1.3.2	View Independent Abstract Syntax	Yes	1. Abstract Syntax	
LNG 1.4	Concrete Syntax Group			
LNG 1.4.1	Concrete Syntax to Abstract Syntax Mapping	Partial		The textual syntax is informally mapped to the abstract syntax using Xtext.
LNG 1.4.2	Graphical Concrete Syntax	Yes		The standard graphical syntax is the textual syntax, which will be further extended to establish a standard graphical syntax.
LNG 1.4.3	Syntax Examples	No		
LNG 1.5	Extensibility Group			
LNG 1.5.1	Extension Mechanisms	No		
LNG 1.6	Model Interchange, Mapping, and Transformations Group			
LNG 1.6.3	UML Interoperability	Partial		This is being accomplished by the SysML v1 to SysML v2 transformation specification.
OTR 1	Interoperability Requirements Group			
OTR 2	Usability Group			
OTR 2.1	Usability Evaluation	Not Determined		
PRP 1	Properties, Values and Expressions Requirements Group			
PRP 1.01	Unified Representation of Values	Partial	Attributes Expressions	need to idettify constant
PRP 1.02	Value Type	Yes	Attributes	
PRP 1.03	Value Expression	Yes	Expressions TextualRepresentation	

ID	Name	Satisfied?	Satisfied-by	Comment
PRP 1.05	Unification of Expression and Constraint Definition	Yes	Constraints	
PRP 1.06	System of Quantities	Yes	Quantities and Units	
PRP 1.07	System of Units and Scales	Yes	Quantities and Units	
PRP 1.08	Range Restriction for Numerical Values	Yes	Constraints	
PRP 1.10	Primitive Data Types	Partial	ScalarValues	Missing enumerations
PRP 1.11	Variable Length Collection Value Types	Yes	Collections	
PRP 1.12	Compound Value Type	Yes	ScalarValues	
PRP 1.15	Probabilistic Value Distributions	No		
PRP 1.19	Materials with Properties	No		
RML 1	Example Model and Model Libraries Group			
RML 1.1	Example Model	No		
RQT 1	Requirement Group			
RQT 1.1	Requirement Definition Group			
RQT 1.1.1	Requirement Definition Name	Yes	Requirements	
RQT 1.1.2	Requirement Identifier	Partial	Requirements	not adaptable
RQT 1.1.3	Requirement Attributes	No		

ID	Name	Satisfied?	Satisfied-by	Comment
RQT 1.1.4	Textual Requirement Statement	Yes	Requirements	
RQT 1.1.5	Restricted Requirement Statement Group			
RQT 1.1.5.1	Restricted Requirement Statement	Not Planned		This can be addressed as a separate RFP.
RQT 1.1.5.2	Restricted Requirement Statement Extensibility	Not Planned		This can be addressed as a separate RFP.
RQT 1.1.5.3	Restricted Requirement Statement Transformation	Not Planned		This can be addressed as a separate RFP.
RQT 1.1.6	Formal Requirement Statement Group			
RQT 1.1.6.1	Formal Requirement Statement	Yes	Requirements	
RQT 1.1.6.2	Assumptions	Yes	Requirements	
RQT 1.2	Groups of Requirements			
RQT 1.2.1	Requirement Group	Partial	Requirements	
RQT 1.2.2	Requirement Usage (localized)	Yes	Requirements	
RQT 1.2.3	Requirement Usage Identifier	No		
RQT 1.2.4	Requirement Usage Ordering	Not Determined		Supported by feature ordering.
RQT 1.3	Requirement Relationships Group			
RQT 1.3.1	Requirement Specialization	Yes	Requirements	

ID	Name	Satisfied?	Satisfied-by	Comment
RQT 1.3.2	Requirement Satisfaction	Yes	Requirements	
RQT 1.3.3	Requirement Verification	No		
RQT 1.3.4	Requirement Derivation	No		
RQT 1.3.5	Requirement Group Relationship	No		
RQT 1.3.6	Relationships to a Requirement Group	No		
RQT 1.4	Requirement Supporting Information	Partial	Annotations	May require an explicit annotation kind for requirements.
RQT 1.5	Goals, Objectives, and Evaluation Criteria	Partial	Cases	Only objective is supported as part of Case.
STC 1	Structure Requirements Group			
STC 1.01	Modular Unit of Structure	Yes	Items Parts	
STC 1.02	Usage Element	Yes	General	
STC 1.03	Generic Hierarchical Structure	Yes	General	
STC 1.04	Reference Element	Yes	General	
STC 1.05	Multiplicity of Usage	Yes	Features	
STC 1.06	Definition Element Specialization	Yes	Types	
STC 1.07	Unambiguous Deeply Nested Structure	Yes	General	
STC 1.08	Structure With Variability	Yes	General	

ID	Name	Satisfied?	Satisfied-by	Comment
STC 1.10	Structure of an Individual	Yes	Individuals	
STC 1.11	Usage Specific Localized Type	Yes	General	
VRF 1	Verification and Validation Requirements Group			
VRF 1.1	Verification Context	No		
VRF 1.2	Verification Case Group			
VRF 1.2.1	Verification Case	Yes	VerificationCases	
VRF 1.2.2	Verification Objectives	Yes	VerificationCases	
VRF 1.2.3	Verification Success Criteria	No		
VRF 1.2.4	Verification Methods	No		
VRF 1.3	Verification System	No		
VRF 1.4	Verification Relationships Group			
VRF 1.4.1	Verification Objectives to Verification Cases	No		
VRF 1.4.2	Validate Relationship	No		

0.4.4 Non-Mandatory Language Requirements - Satisfied-by Table

Table 4. Non-Mandatory Language Requirements - Satisfied-by Table

ID	Name	Satisfied?	Satisfied-by	Comment
ANL 1	Analysis Requirements Group			
ANL 1.10	Analysis Model - System Model Transformation	No		external relationship service
ANL 1.12	Analysis Infrastructure	Not Planned		

ID	Name	Satisfied?	Satisfied-by	Comment
ANL 1.14	Decision Group			
ANL 1.14.1	Trade-off	No		
ANL 1.14.3	Decision Expression	No		
BHV 1	Behavior Requirements Group			
BHV 1.03	Function-based Behavior Group			
BHV 1.03.2	Composite Input and Output	Not Determined		
BHV 1.03.5	Behavior Library	Not Determined		
BHV 1.09	State History	Yes	Individuals	
CNF 1	Conformance Requirements Group			
CNF 1.3	Model Interoperability Conformance	No		
CRC 1	Cross-cutting Requirements Group			
CRC 1.2	Model Element Group			
CRC 1.2.1	Model Element	Not Determined		
CRC 1.3	Model Element Relationships Requirements Group			
CRC 1.3.11	Copy Relationship	Not Planned		
INF 1	Interface Requirements Group			
INF 1.07	Interface Agreement Group			
INF 1.07.3	Geometric Constraints	Not Determined		
LNG 1	Language Architecture and Formalism Requirements Group			
LNG 1.2	Semantics Group			
LNG 1.2.1	Semantic Model Libraries	Yes	Kernel Library	
LNG 1.2.2	Declarative Semantics	Yes	Kernel Library	
LNG 1.2.3	Reasoning Capability	Partial		

ID	Name	Satisfied?	Satisfied-by	Comment
LNG 1.4	Concrete Syntax Group			
LNG 1.4.4	Textual Concrete Syntax	Yes		
LNG 1.5	Extensibility Group			
LNG 1.5.2	Extensibility Consistency	No		
LNG 1.6	Model Interchange, Model Mapping, and Transformations Group			
LNG 1.6.1	Model Interchange	No		
LNG 1.6.2	Model Mappings and Transformations	Not Determined		
PRP 1	Properties, Values and Expressions Requirements Group			
PRP 1.04	Logical Expressions	Yes	Control Functions	
PRP 1.09	Automated Quantity Value Conversion	Yes	Quantities and Units	
PRP 1.13	Discretely Sampled Function Value Type	No		
PRP 1.14	Discretely Sampled Function Interpolation	Not Planned		
PRP 1.16	System Simulation Models	Not Determined		
PRP 1.17	Across and Through Value Properties	No		
PRP 1.18	Basic Geometry	No		
PRP 1.20	Equivalent Element	No		
RML 1	Example Model and Model Libraries Group			
RML 1.2	Model Libraries	No		
RQT 1	Requirement Group			
RQT 1.3	Requirement Relationships Group			
RQT 1.3.7	Relationship Logical Constraint	No		
STC 1	Structure Requirements Group			

ID	Name	Satisfied?	Satisfied-by	Comment
STC 1.09	Structure Resolved to a Single Variant	Not Determined		
VRF 1	Verification and Validation Requirements Group			
VRF 1.2	Verification Case Group			
VRF 1.2.5	Verification Activity	No		
VRF 1.2.6	Verification Evaluation Activity	No		

0.4.5 Changed Language Requirements Table

Table 5. Changed Language Requirements Table

ID	Name	Requirement Text	Change Status	Change Description
ANL 1.13	Analysis Metadata	Proposals for SysML v2 shall include the capability to represent the metadata relevant to specifying the analysis, including the specification of dependent and independent parameters.	Modified	June 14, 2018 - Modified by adding text starting at "including the specification...". Drivers based on SysML v1.6 RTF Out of Scope Issues, SYSML16-38: Inability to represent dependent, independent parameters on constraint properties
BHV 1.11.3	Storing Items in Storage Elements Requirements Group		Added	16 June 2018 - Added this requirement group to mandatory requirements to provide a requirement group for BHV 1.11.3.1 and BHV 1.11.3.2

ID	Name	Requirement Text	Change Status	Change Description
BHV 1.11.3.1	Storage Element and Stored Item Usages	<p>Proposals for SysML v2 shall include the capability to model a storage element that can store items declared by stored item usages. The stored items shall be identified as conserved (e.g., a physical element) or copied (e.g., data from memory). Conservation constraints shall apply to conserved item usages (e.g., amount in - amount out=amount stored).</p> <p>Supporting Information: Examples include:</p> <p>A storage element called tank that stores a stored item usage called fluid. (example of a conserved stored item usage)</p> <p>A storage element called common value table that stores a stored item usage called system mode. (example of a copied stored item usage)</p>	Added	16 June 2018 - Added to mandatory requirements based on Creating and Accessing Stored Items Use Case
BHV 1.11.3.2	Create, Modify, and Consume Stored Items	<p>Proposals for SysML v2 shall include the capability to model outputs and inputs of a behavior that create, modify, or consume stored items of a storage element. An input to or output from a storage element that results in the creation, modification, or consumption of stored items can be assigned to one or more ports of the storage element.</p> <p>Supporting Information: Examples include:</p> <p>A pump fluid action produces an output called fluid that is stored in a tank, and another action consumes the fluid from the tank. (example of a conserved stored item usage)</p> <p>An update mode variable action produces a logical data item that is stored in common value table, and another action called verify mode consumes the logical data item from the common value table. (example of a copied stored item usage)</p>	Added	16 June 2018 - Added to mandatory requirements based on Creating and Accessing Stored Items Use Case

ID	Name	Requirement Text	Change Status	Change Description
CNF 1	Conformance Requirements Group		Added	26 April 2018 - Added this requirement group to non-mandatory to provide a group for CNF 1.3
CNF 1.3	Model Interoperability Conformance	Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 model interchange, model mappings and transformations requirements in LNG 1.6.	Added Modified Moved	26 April 2018 - Moved from Mandatory requirements and replaced text "interoperability specification" with "interchange, model mappings and transformations requirements in LNG 1.6"
CNF 1.3	Model Interoperability Conformance	Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 model interoperability specification.	Deleted Moved	26 April 2018 - Moved to Non-mandatory

ID	Name	Requirement Text	Change Status	Change Description
PRP 1.20	Equivalent Element	<p>Proposals for SysML v2 shall include a capability to represent an element that can reference another model element or an external element, indicating that the reference element is semantically equivalent to the referenced element.</p> <p>Supporting Information:</p> <p>This requirement is intended to be supported by transclusion, which enables the content that is referenced to be displayed in place of the reference element.</p> <p>A URI or URL can be used to refer to an external element.</p> <p>Example: Define a reference element 'x' that has a real value of 100.0. Transclude this reference element in "The top speed of this car shall be greater than <x> mph.", such that it is rendered textually as "The top speed of this car shall be greater than 100.0 mph."</p> <p>Example: A reference element called 'part A' refers to 'part A1' in a bill of materials in a PLM application</p> <p>SysML v1.X Constructs: Adjunct property (partial satisfaction)</p>	Added	14 June 2018 - Added this new requirement based on the use case for semantic reference to an internal or external model element (or set of model elements).
STC 1.03	Generic Hierarchical Structure	Proposals for SysML v2 shall include a capability to represent hierarchical composition structure of Definition and/or Usage Elements.	Modified	25 Oct 2018 - Changed the last few words of the requirement text from "structure of Definition Elements" to "structure of Definition and/or Usage Elements".
VRF 1.2.3	Verification Success Criteria	The verification case shall include the criteria used to evaluate whether the verification objectives are met, the requirements are satisfied, and any subset of verification steps in a verification case are successfully performed.	Modified	3 Dec 2018 Added text to end of requirement statement "and any subset of verification steps..."

1 Scope

The purpose of this standard is to specify the Systems Modeling Language™ (SysML), to guide the implementation of conformant modeling tools, and to provide the basis for the development of material and other resources to train users in the application of SysML. It also serves as the baseline for future revisions of SysML.

SysML is a general-purpose modeling language for modeling systems that is intended to facilitate a model-based systems engineering (MBSE) approach to engineer systems. It provides the capability to create and visualize models that represent many different aspects of a system including its requirements, structure, behavior, and the constraints on its system properties to support engineering analysis.

The language is intended to support multiple systems engineering methods and practices. The specific methods and practices may impose additional constraints on how the language is used. SysML is defined as an extension of the Kernel Modeling Language (KerML), which provides a common, domain-independent language for building semantically rich and interoperable modeling languages. SysML also provides a capability to provide further language extensions. It is anticipated that SysML will be customized using this language extension mechanism to model more specialized domain-specific applications, such as automotive, aerospace, healthcare, and information systems, as well as discipline specific extensions such as safety and reliability.

Note. Definitions of *system* and *systems engineering* can be found in ISO/IEC 15288.

SysML Version 2 is intended to enhance the precision, expressiveness, interoperability, and the consistency and integration of the language relative to SysML Version 1. SysML also includes a textual notation in addition to a graphical notation that was not provided with SysML v1. SysML v2 is specified as a metamodel that extends the kernel metamodel provided by KerML, which is greatly simplified compared to the UML metamodel that the SysML v1 profile was based. In order to facilitate the transition from SysML v1 to SysML v2, the standard also specifies a formal transformation from UML models using the SysML v1 profile to models using the SysML v2 metamodel.

2 Conformance

This specification defines the Systems Modeling Language (SysML), a language used to construct *models* of systems (whether they are real, planned or imagined). The specification comprises this document together with 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 *SysML model* shall conform to this specification only if it can be represented according to the syntactic requirements specified in [Clause 7](#). The model may be represented in a form consistent with the requirements for the SysML concrete syntax, in which case it can be parsed (as specified in [Clause 7](#)) into an abstract syntax form, or it may be represented directly in an abstract syntax form.

A *SysML modeling tool* is a software application that creates, manages, analyzes, visualizes, executes or other performs other services on SysML 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 SysML 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 SysML metamodel. A well-formed model represented according to the abstract syntax is syntactically conformant to SysML as defined above. (See [Clause 7](#).)
2. *Concrete Syntax Conformance.* A tool demonstrating Concrete Syntax Conformance provides a user interface and/or API that enables instances of SysML 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 SysML. 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 7](#).)
3. *Semantic Conformance.* A tool demonstrating Semantic Conformance provides a demonstrable way to interpret a syntactically conformant model (as defined above) according to the SysML semantics, e.g., via semantic model analysis or model execution. Semantic Conformance implies Abstract Syntax Conformance, in that the semantics for SysML are only defined on well-formed models represented in the abstract syntax. (See [Clause 7](#) and [8.1](#). See also [KerML, 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 SysML models (as defined above) in one or more of the formats specified in [KerML, Clause 9].

Every conformant SysML 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. The tool may also provide one or more of the following additional capabilities, conformant with this specification:

1. *Domain Library Support.* In addition to the Systems Model Library, a conformant tool may provide one or more domain model libraries as specified in [Clause 8](#).
2. *SysML v1 Transformation Support.* A conformant tool may provide the capability to import a model conformant with SysML v1 and, at least, export the model into one of the valid model interchange formats for SysML v2, as specified in [Annex C](#). For the purposes of this conformance point, "SysML v1"

shall mean at least SysML v1.7, and optionally earlier versions, and "SysML v2" shall mean the latest version of SysML as of v2.0 or later.

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.

[KerML] *Kernel Modeling Language (KerML)*, Version 1.0
(as submitted with this proposed specification)

[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>

[SysMLv1] *OMG Systems Modeling Language (SysML)*, Version 1.7
(currently in preparation)

[UML] *Unified Modeling Language (UML)*, Version 2.5.1
<https://www.omg.org/spec/UML/2.5.1>

The following references were used in the definition of the Quantities and Units model library (see [8.2](#)):

[GUM] JCGM 100:2008 and ISO/IEC Guide 98-3, Evaluation of measurement data - Guide to the expression of uncertainty in measurement
<https://www.bipm.org/en/publications/guides/#gum>

[ISO 80000-1] ISO 80000-1:2009, Quantities and units - Part 1: General
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-1:ed-1:v1:en>

[ISO 80000-2] ISO 80000-2:2019, Quantities and units - Part 2: Mathematical signs and symbols to be used in the natural sciences and technology
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-2:ed-2:v1:en>

[ISO 80000-3] ISO 80000-3:2019, Quantities and units - Part 3: Space and Time
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-3:ed-2:v1:en>

[ISO 80000-4] ISO 80000-4:2019, Quantities and units - Part 4: Mechanics
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-4:ed-2:v1:en>

[ISO 80000-5] ISO 80000-5:2019, Quantities and units - Part 5: Thermodynamics
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-5:ed-2:v1:en>

[IEC 80000-6] IEC 80000-6:2008, Quantities and units - Part 6: Electromagnetism
<https://www.iso.org/obp/ui/#iso:std:iec:80000:-6:ed-1:v1:en,fr>

[ISO 80000-7] ISO 80000-7:2019, Quantities and units - Part 7: Light
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-7:ed-2:v1:en>

[ISO 80000-8] ISO 80000-8:2020, Quantities and units - Part 8: Acoustics
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-8:ed-2:v1:en>

[ISO 80000-9] ISO 80000-9:2019, Quantities and units - Part 9: Physical chemistry and molecular physics
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-9:ed-2:v1:en>

[ISO 80000-10] ISO 80000-10:2019, Quantities and units - Part 10: Atomic and nuclear physics
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-10:ed-2:v1:en>

[ISO 80000-11] ISO 80000-11:2019, Quantities and units - Part 11: Characteristic numbers
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-11:ed-2:v1:en>

[ISO 80000-12] ISO 80000-12:2019, Quantities and units - Part 12: Solid state physics
<https://www.iso.org/obp/ui/#iso:std:iso:80000:-12:ed-2:v1:en>

[IEC 80000-13] IEC 80000-13:2008, Quantities and units - Part 13: Information science and technology
<https://www.iso.org/obp/ui/#iso:std:iec:80000:-13:ed-1:v1:en>

[IEC 80000-14] IEC 80000-14:2008, Quantities and units - Part 14: Telebiometrics related to human physiology
<https://www.iso.org/obp/ui/#iso:std:iec:80000:-14:ed-1:v1:en>

[NIST SP-811] NIST Special Publication 811, The NIST Guide for the use of the International System of Units
(In particular its Appendix B "Conversion Factors")
<https://www.nist.gov/pml/special-publication-811>

[VIM] JCGM 200:2012 and ISO/IEC Guide 99, International vocabulary of metrology - Basic and general concepts and associated terms (VIM)
<https://www.bipm.org/en/publications/guides/#vim>

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 Overview

As shown in [Fig. 1](#), SysML is built as an extension to the Kernel metamodel from [KerML]. The SysML abstract syntax (see [Clause 7](#)) extends the Kernel abstract syntax, providing specialized constructs for modeling systems. Further, the SysML Systems Model Library (see [8.1](#)) extends the Kernel Model Library to provide the semantic specification for SysML (see also [KerML, 7.1] on the use of model libraries for semantic specification). Finally, SysML provides an additional set of Domain Libraries (see [Clause 8](#)) to provide a rich set of reference models in various domains important to systems modeling (such as Quantities and Units and Basic Geometry).

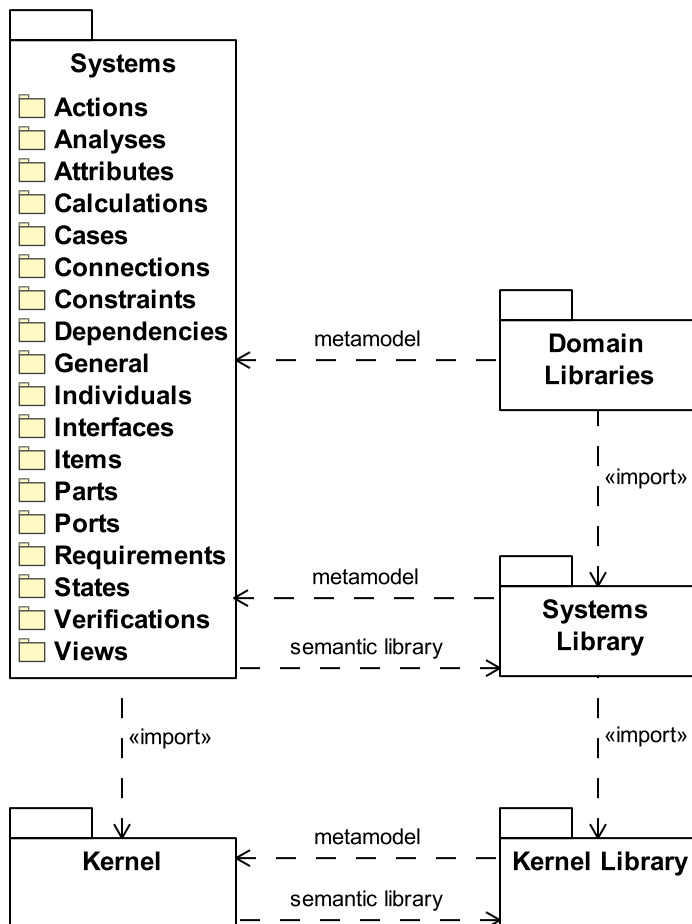


Figure 1. SysML Language Architecture

6.2 Document Conventions

The following stylistic conventions are applied in the abstract syntax, concrete syntax and semantics descriptions of [Clause 7](#) (Metamodel) and in the element descriptions of [Clause 8](#) (Model Library) when model elements are referenced by name in body text paragraphs. However, they are generally *not* used in overview subclauses, which are written in a more descriptive and colloquial style, and should be considered informative rather than normative.

1. Names of metaclasses from the SysML abstract syntax model are capitalized and written exactly as defined in the abstract syntax, but they are otherwise used as if they were nouns in English, e.g.,

"PartDefinition, "ActionUsage". When used in this way, the metaclass name refers to the instances of the metaclass (in models), e.g., "A PartUsage must be defined by a PartDefinition". This can be modified using the "metaclass" as necessary to refer to the metaclass itself instead of its instances, e.g., "The PartDefinition metaclass is contained in the Parts package."

2. Names of properties of metaclasses in the text are styled in a "code" font, and used as if they are English nouns, pluralized where necessary, e.g., "the ownedParts of a Definition".
3. Names of classes and features of elements from a SysML model are styled the same as abstract syntax metaclass and property names, but put in italics. This includes elements from any of the SysML Model Libraries (e.g., "Action" and "quantity") and elements referenced from sample user models (e.g., "Vehicle" and "wheels").

In addition, the following additional conventions used in the Concrete Syntax subclauses within [Clause 7](#) for the SysML textual notation.

1. In all cases, text in the SysML textual notation is styled in a "code" font.
2. When individual keywords are referenced, they are written in boldface, e.g., "PartUsages are declared using the **part** keyword."
3. Symbols (such as ~ and :>>) and short segments of textual notation (but longer than an individual name) may be written in-line in body text (without being italic or bold).
4. Longer samples of textual notation are written in separate paragraphs, indented relative to body paragraphs.

The grammar of the textual Concrete Syntax and its mapping to the Abstract Syntax is specified using a specialized *Extended Backus-Naur Form* (EBNF) notation that is described in [KerML, 7.1.3]. For the graphical Concrete Syntax, this BNF notation is further extended to allow the use of graphical symbols within productions.

Submission Note. A paragraph marked as a "submission 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 the initial submission of the proposed specification, or changes to the specification that are expected before the revised submission of the proposal. Such notes will be removed in future revised submissions 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 the submission.

Release Note. Paragraphs marked like this one provide additional information on the status of updates to this specification document in releases since the initial submission.

6.3 Document Organization

The rest of this document is organized into two major clauses.

- [Clause 7](#) specifies the Metamodel that defines the SysML language. The first subclause of this clause is an overview, followed by a summary of the Kernel metamodel on which the rest of the SysML metamodel is built. The Kernel metamodel has the same abstract syntax and semantics as the metamodel for the Kernel Modeling Language (KerML), as defined in [KerML]. However, the textual concrete syntax for the Kernel is not identical to that for KerML, and SysML also provides a graphical concrete syntax for certain Kernel elements. Each subclause following the Kernel summary then describes each of the packages in the SysML metamodel proper, including the concrete syntax, abstract syntax, and semantics for the metamodel elements in each package.

- [Clause 8](#) specifies a set of model libraries defined in SysML itself. The Systems Model Library extends the Kernel Model Library from [KerML] in order to provide systems-modeling-specific semantics to SysML language constructs. The other model libraries (such as Quantities and Units) provide rich domain-specific models on which users can draw when creating their own models. Each model library is described with a set of subclauses to describe the content of each of the top-level packages in the model library, referred to as its *library models*.

These clauses are followed by three annexes.

- [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](#)).
- [Annex B](#) is an informative annex that presents a realistic extended example model using the SysML language as defined in this specification.
- [Annex C](#) defines the a formal transformation from SysML v1 models to SysML v2 models that, to the greatest extent possible, preserves the semantics of the original models.

In addition, Clause 9 of [KerML] on Model Interchange is included by reference as a normative part of this specification, in order to define allowable methods for interchanging SysML models.

6.4 Acknowledgements

The primary authors of this specification document and the syntactic and semantic models defined in it are:

- Sanford Friedenthal, SAF Consulting
- Ed Seidewitz, Model Driven Solutions
- Yves Bernard, Airbus
- Tim Weilkiens, oose
- Hans Peter de Koning, DEKonsult

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

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), by contributing use cases, providing critical review and comment, and validating the language design. The following individuals had leadership roles in the SST:

- Manas Bajaj, InterCax LLC (API and services development lead)
- Yves Bernard, Airbus (profile development 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 Weilkiens, oose (profile development co-lead)

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

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

The following individuals made significant contributions to the pilot implementation developed by the SST in conjunction with the development of this specification:

- Ivan Gomes, Jet Propulsion Laboratory
- Hisashi Miyashita, Mgnite
- Miyako Wilson, Georgia Institute of Technology
- Santiago Leon, Tom Sawyer
- Zoltán Ujhelyi, IncQuery Labs

7 Metamodel

7.1 Overview

The SysML metamodel defined in this clause contains concepts that are used to model systems, their components, and the external environment in a context.

The SysML metamodel extends the KerML metamodel as specified in the KerML specification [KerML]. The most basic KerML metaclasses are Element and Relationship. All other KerML metaclasses are specializations of these elements. The SysML metamodel imports the KerML metamodel and reuses some KerML metaclasses directly, and further specializes other KerML metaclasses. The subclauses that follow summarize the description of the KerML concepts in [7.2](#), and specify the SysML metaclasses in [7.3](#) through [7.22](#).

SysML directly reuses the Package, Import, and Membership metaclasses from KerML to provide a flexible means to logically organize a model into a containment tree. SysML reuses and extends Classifier, Feature, Association, Connector, Behavior, Functions, Expressions, and other KerML metaclasses. SysML also reuses Generalization that provides the mechanism to support subclassing, subsetting and redefining elements, which are all KerML concepts.

The general pattern of a Definition element and Usage element is applied to many of the SysML language constructs to facilitate reuse. The Definition element is a kind of KerML Classifier and the UsageElement is a kind of KerML Feature. The Definition element and Usage element are metaclasses that are further subclassed in the SysML metamodel to create other metaclasses such as PartDefinition and PartUsage. The SysML Model Library includes instances of these metaclasses. For example, instances of the metaclasses PartDefinition and PartUsage are contained in the SysML Model Library as Part and parts respectively. These are the specific constructs used to develop a model of a system. The Part and parts from the library are further specialized to create particular part definitions and part usages, such as a part called vehicle that is defined by a part definition called Vehicle.

The definition elements and usage elements facilitate model reuse, such that a concept can be defined once and then used in many different contexts. For example, a front wheel and a rear wheel can be represented as two different usages of the same definition of Wheel. A usage element can also be further specialized for its specific context. For example, the front wheels and rear wheels may be usages of the same definition of Wheel, but have different kinds of tires with different tire pressures.

The modeling constructs specific to SysML, as specified in subclauses [7.3](#) through [7.22](#), are built on the KerML foundation, and cover the following areas:

- Fundamental aspects of constructing a model, including:
 - The modeling of *dependencies* between modeling elements (see [7.3](#)).
 - The basic concepts of *definition* and *usage*, as discussed above (see [7.4](#)).
 - The modeling of *variability*, which includes the definition of *variation* points within a model where choices can be made to select a specific *variant*, and the selection of a particular variant may constrain the allowable choices at other variation points. A system can be *configured* by making appropriate choices at each of the variation points of a variability model, consistent with specified constraints. Variation points can be defined in any of the specific modeling areas listed below, so the ability to model variability is built into the base syntax of definitions and usages (see [7.4](#)).
- The modeling of *structure* to represent how parts are decomposed, interconnected and classified, and includes:
 - *Attributes* that specify characteristics of something that can be defined by simple or compound data types, and dimensional quantities such as mass, length, etc. (see [7.5](#)).
 - *Enumerations* that are attributes restricted to a specified set of enumerated values (see [7.6](#)).
 - *Items* that may flow through a process or system or be stored by a system (see [7.7](#)).

- *Parts* are the foundational units of structure that can be composed and interconnected, forming composite parts and entire systems (see [7.8](#)).
- *Ports* that define connection points on parts that enable interactions between parts (see [7.9](#)).
- *Connections* (see [7.10](#)) and *interfaces* (see [7.11](#)) that define how parts are interconnected.
- *Allocations* of some or all of the responsibility to realize the intent of a one element to another element (see [7.12](#)).
- The modeling of *individual* items and parts with identity, which can be represented at specific points in time, over a duration in time, or over an entire lifetime (see [7.13](#)).
- The modeling of *behavior*, which specify how systems and or components interact and include:
 - *Actions* performed by a part, including their temporal ordering, and the flows of items between them (see [7.14](#)).
 - *States* exhibited by a part, the allowable *transitions* between those states, and the actions enabled in a state or during a transition (see [7.15](#)).
- The modeling of *calculations* which are parameterized expressions that can be computed to produce specific results (see [7.16](#)).
- The modeling of *constraints*, which specify conditions that a system or part is expected or required to satisfy, and can be evaluated as true or false, or asserted to be true (see [7.17](#)).
- The modeling of *requirements*, which is a special kind of constraint that a *subject* system or part must satisfy to be a valid solution (see [7.18](#)).
- The modeling of *cases*, which define the steps required to produce a desired result relative to a *subject* (see [7.19](#)), to achieve a specific *objective*, including:
 - *Analysis cases*, whose steps are the *actions* necessary to analyze a subject (see [7.20](#)).
 - *Verification cases*, whose objective is to verify how a requirement is satisfied by the subject (see [7.21](#)).
- The modeling of *viewpoints* that specify information of interest by a set of stakeholders, and *views* that specify a query of the model, and a rendering of the query results, that is intended to satisfy a particular viewpoint (see [7.22](#)).

In a similar way that SysML extends KerML, SysML also provides a language extension capability to allow users to build domain and user-specific extensions of SysML, both syntactically and semantically. This allows SysML to be highly adaptable for specific application domains and user needs, while maintaining a high level of underlying standardization and tool interoperability. (See [7.23](#).)

It should be noted that SysML does not contain specific language constructs called system, subsystem, assembly, component, and many other commonly used terms. An entity with structure and behavior in SysML is represented as a part (see [7.8](#)). The language provides straightforward extension mechanisms to specify terminology that is appropriate for the domain of interest.

7.2 Kernel

7.2.1 Basic Elements

7.2.1.1 Basic Elements Overview

The SysML metamodel reuses basic elements from the KerML Root that includes Element and Relationship. All other KerML and SysML model elements are extensions of these basic elements.

An element has a unique identifier. Elements can have a name and any number of aliases.

A relationship is a kind of element that relates other elements. Some relationships are constrained to have two ends (i.e., binary relationship) while others are not (e.g., Association and Connector in the Kernel, Dependency and Expose in SysML). The ends on relationships are ordered. A directed relationship designates its ends as sources and targets.

7.2.1.2 Basic Elements Concrete Syntax

7.2.1.2.1 Basic Elements Textual Notation

```
Identification (e : Element, m : Membership) =  
    ( 'id' e.humanId = NAME )? ( m.memberName = NAME )?
```

SysML does not provide any concrete syntax for generic Elements or Relationships that are not instances of a more specialized metaclass. While the Element and Relationship metaclasses are not abstract in the KerML abstract syntax (for reasons discussed in [KerML, 7.2.2.1]), they shall not be directly instantiated in a SysML model.

However, the SysML concrete syntax does provide a consistent notation for identifying any kind of element, in one or both of the following ways:

- The declaration of an Element may specify a `humanId` for it, as a lexical name preceded by the keyword **id**. Note that the `humanId` of an Element is separate from the unique `identifier` or other `aliasIds` of an Element that are managed by the underlying modeling tooling. It is the responsibility of the modeler to maintain other structural or uniqueness properties for `humanIds` as appropriate to the model being created.
- If the Element is an `ownedMember` of a Package, then a name may also be given for the Element (after its `humanId`, if any). This name is actually the `memberName` of the Membership by which the Element is owned by the Package (see [7.2.3](#)).

Note that it is not required to specify either a `humanId` 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 concrete syntax.

7.2.1.2.2 Basic Elements Graphical Notation

```
id-text (e : Element) =  
    ( 'uuid' NAME(e.identifier) )? ( 'id' NAME(e.aliasId) ) * NAME(e.name)?  
  
qualified-name-text (e : Element) =  
    ( qualified-name-text (e.owningNamespace) '::' )? element-name-text(e)  
  
element-name-text (e : Element) =  
    NAME(e.name) | NAME(e.humanId)  
  
annotated-element-text (e : Element) =  
    element-text(e) element-graphic(e.ownedAnnotation) *  
  
element-graphic (e : Element) =  
    element-symbol(e) element-graphic(e.ownedRelationship) *  
  
element-text (e : Element) = ...  
  
element-symbol (e : Element) = ...
```

Generic textual elements used within the SysML graphical notation include:

- Identifying an Element by its `identifier` (UUID), `aliasId` (including `humanId`) and/or name.

- Referencing an Element using a qualified name.

The specific textual and symbolic representation for an Element, as used within graphical diagrams, are specified for each kind of Element in following subclauses. In general,

- The *annotated text* for an Element is the basic Element text, optionally with the graphical representation of some or all of its ownedAnnotations (see 7.2.2).
- The full *graphical representation* of an Element is the Element symbol, optionally with the graphical representation of some or all of its ownedRelationships.

7.2.1.3 Basic Elements Abstract Syntax

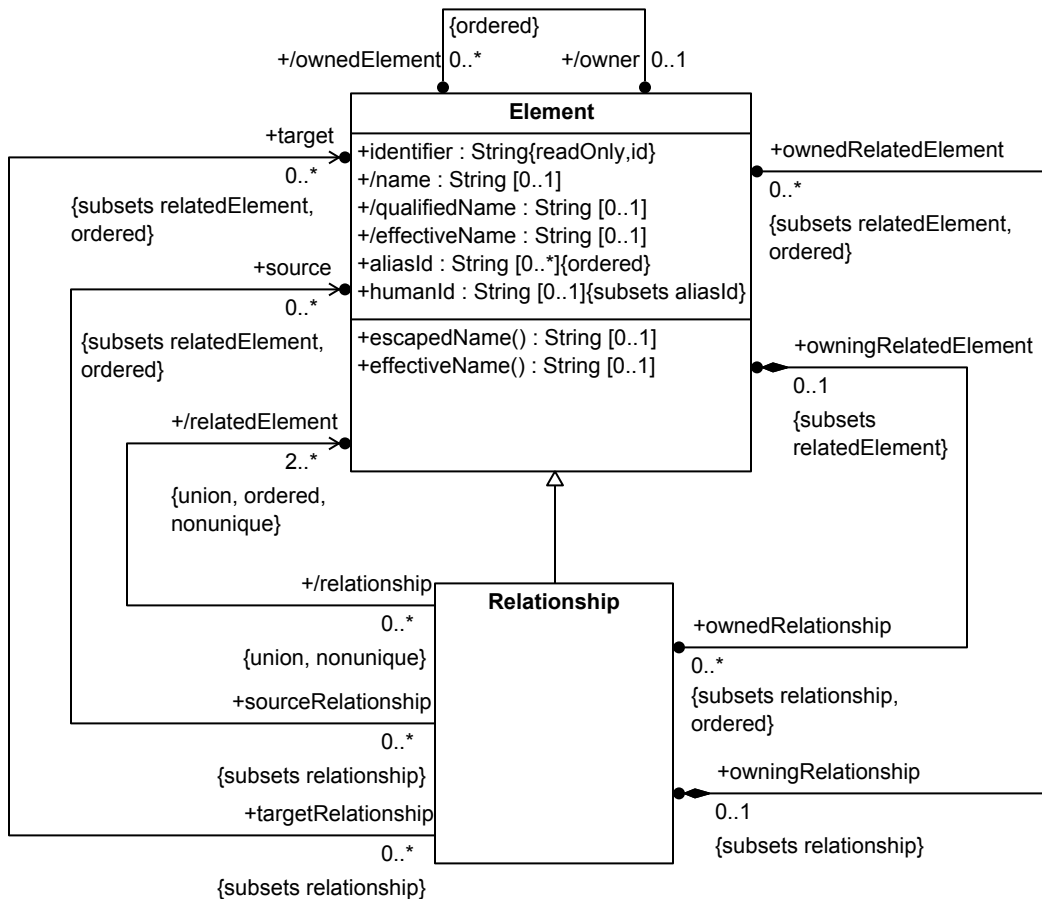


Figure 2. Elements

7.2.2 Annotations

7.2.2.1 Annotations Overview

An **AnnotatingElement** is an **Element** that is used to provide additional information on other elements. An **Annotation** is a **Relationship** between an annotating element and an element that is being described. An annotated element can annotate multiple elements, and each element can have multiple annotations.

A **Comment** is one kind of **AnnotatingElement** that is used to provide textual descriptions about other elements. The comments about an element need not be owned by that element, but an element can contain distinguished documentation comments that are specifically used to document it. Comments can be named.

A TextualRepresentation is an AnnotatingElement whose textual body can be represented in another language. In particular, if the named language is machine-parsable, then the body text should be legal input text as defined for that language. In particular, annotating a SysML model element with a textual annotation in a language other than SysML can be used as a semantically "opaque" element specified in the other language.

An AnnotatingFeature is a kind of AnnotatingElement that allows for the definition of structured metadata with modeler-specified attributes. This may be used, for example, to add tool-specific information to a model that can be relevant to the function various kinds of tooling that may use or process a model, or domain-specific information relevant to a certain project or organization. An annotating feature is syntactically a feature (see [7.2.6](#)) that is typed by a single data type (see [7.2.5](#)) or attribute definition (see [7.5](#)). If the data type has no features itself, then the annotating feature simply acts as a user-defined syntactic tag on annotated element. If the data type does have features, then the annotating feature must provide value bindings for all of them, specifying attributive metadata for the annotated element.

7.2.2.2 Annotations Concrete Syntax

7.2.2.2.1 Annotations Textual Notation

7.2.2.2.1.1 Comments

```
Comment (m : Membership, e : Element) : Comment =
  ( 'comment' Identification(this, m)
    'about' annotation += Annotation
    { ownedRelationship += annotation }
    ( ',' annotation += Annotation
      { ownedRelationship += annotation } ) *
  | ( 'comment' Identification(this, m) )?
    annotation += ElementAnnotation(e)
    { ownedRelationship += annotation } )
  body = REGULAR_COMMENT

Annotation : Annotation =
  annotatedElement = [Element | QualifiedId]

ElementAnnotation ( e : Element ) : Annotation =
  { annotatedElement = e }
```

The SysML textual concrete syntax for Comments is the same as the KerML notation (see [KerML, 7.2.3.2.1]).

7.2.2.2.1.2 Documentation

```
OwnedDocumentation : Documentation =  
    documentingElement = DocumentationComment  
  
DocumentationComment : Comment =  
    'doc' ( 'id' humanId = Name )? body = REGULAR_COMMENT  
  
PrefixDocumentation : Documentation =  
    documentingElement = PrefixDocumentationComment  
  
PrefixDocumentationComment : Comment =  
    ( 'doc' ( 'id' humandId = Name )? )? body = DOCUMENTATION_COMMENT
```

The SysML textual concrete syntax for Documentation is the same as the KerML notation (see [KerML, 7.2.3.2.2]).

7.2.2.2.1.3 Textual Representation

```
TextualRepresentation (m : Membership) : TextualRepresentation =  
    ( 'rep' Identification(this, m)  
      'about' annotation += Annotation  
    | ( 'rep' Identification(this, m) )?  
      ElementAnnotation(e)  
    )  
    'language' language = STRING_VALUE  
    body = ML_COMMENT
```

The SysML textual concrete syntax for TextualRepresentations is the same as that of KerML (see [KerML, 7.2.3.2.3]).

In addition to the standard TextualRepresentaion language names defined for KerML, a conformant SysML modeling tool shall also recognize the language name "sysml" (ignoring case), in which case the body text of the TextualRepresentation shall be a legal representation of the annotatedElement in the SysML textual concrete syntax as defined in this specification.

7.2.2.2.1.4 Annotating Features

```
AnnotatingFeature (m : Membership, e : Element) : AnnotatingFeature =
  ( '@' | 'metadata' ) AnnotatingFeatureDeclaration(this, m)
  ( 'about' annotation += Annotation
    ownedRelationship += Annotation
    ( ',' annotation += Annotation
      { ownedRelationship += Annotation } ) *
  | annotation += ElementAnnotation(e)
    { ownedRelationship += Annotation }
  )

AnnotatingFeatureDeclaration (a : AnnotatingFeature, m : Membership) =
  ( Identification(this, m) ( ':' | 'typed' 'by' ) )?
  a.ownedRelationship += ownedFeatureTyping

AnnotatingFeatureBody (a : AnnotatingFeature) =
  ';' | '{' ( a.ownedRelationship += MetadataFeatureMember ) * '}'

MetadataFeatureMember : FeatureMembership =
  ownedMemberFeature = MetadataFeature

MetadataFeature : MetadataFeature =
  'feature'? ( ';>' | 'redefines'? ownedRelationship += OwnedRedefinition
    '=' metadataFeatureValue = MetadataFeatureValue ';'

MetadataFeatureValue : MetadataFeatureValue =
  metadataValue = MetadataExpression
```

The SysML textual concrete syntax for AnnotatingFeatures is the same as that of KerML (see [KerML, 7.4.12.2]).

7.2.2.2.2 Annotations Graphical Notation

```

element-graphic (a : Annotation) =
    comment-graphic(a)
    | documentation-graphic(a)
    | textual-representation-graphic(a)

comment-graphic (a : Annotation) =
    ( &element(a.annotatedElement) annotation-symbol ) *
    comment-symbol(a.annotatingElement)

documentation-graphic (d : Documentation ) =
    &element(d.documentedElement) annotationSymbol
    documentation-symbol(d.documentingComment)

textual-representation-graphic (a : Annotation) =
    &element(a.annotatedElement) annotation-symbol
    textual-representation-symbol(a.annotatingElement)

annotating-feature-graphic (a : Annotation) =
    &element(a.annotatedElement) annotation-symbol
    annotating-feature-symbol(a.annotatingElement)

element (e : Element) =
    element-text(e) | element-symbol(e)

annotation-symbol =
    -----

comment-symbol (c : Comment) =
    «comment»
    TEXT(c.body)

documentation-symbol (c : Comment) =
    «doc»
    TEXT(c.body)

textual-representation-symbol (t : TextualRepresentation) =
    «rep»
    STRING(t.language)?
    TEXT(t.body)

annotating-feature-symbol (f : AnnotatingFeature) =
    «metadata»
    annotating-feature-decl-text(f)
    metadata-feature-text(f.ownedMetadata)*

annotating-feature-decl-text (f : AnnotatingFeature) =

```

```

    AnnotatingFeatureDeclaration(f, f.owningMembership)

metadata-feature-text (f : MetadataFeature) =
    f.ownedRedefinition.redefinedFeature.name
    '=' Expression(f.metadataFeatureValue.metadataValue)

```

Note. The graphical representation of an Annotation may be used to annotate both textual and symbolic representations of an element within a larger graphical diagram.

7.2.2.3 Annotations Abstract Syntax

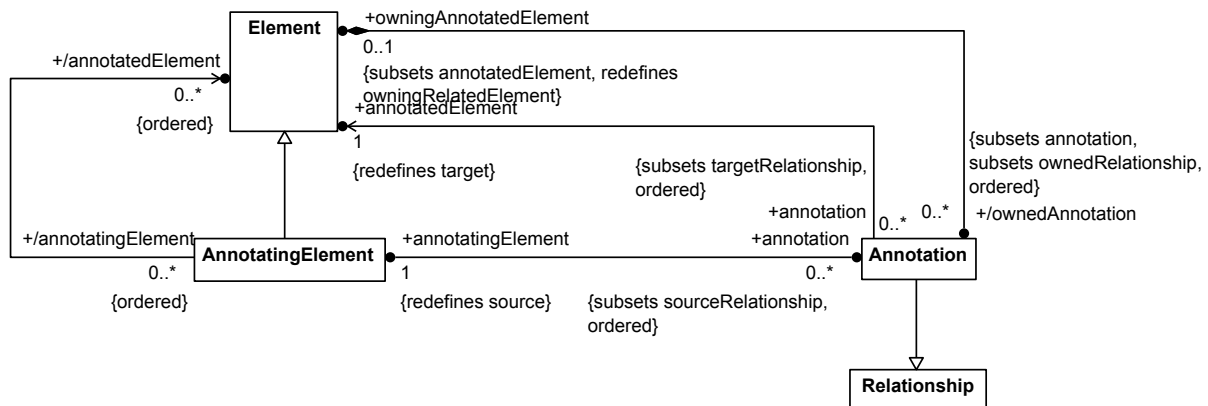


Figure 3. Annotation

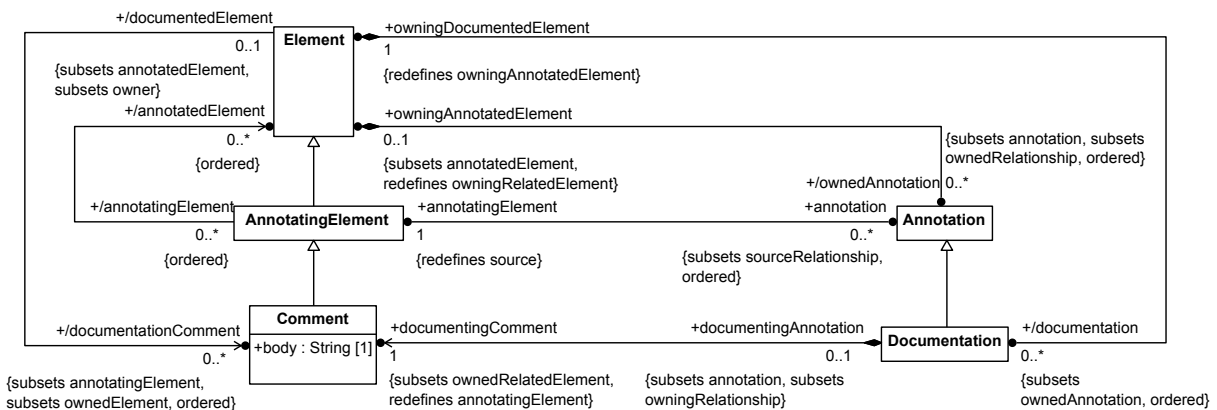


Figure 4. Comments

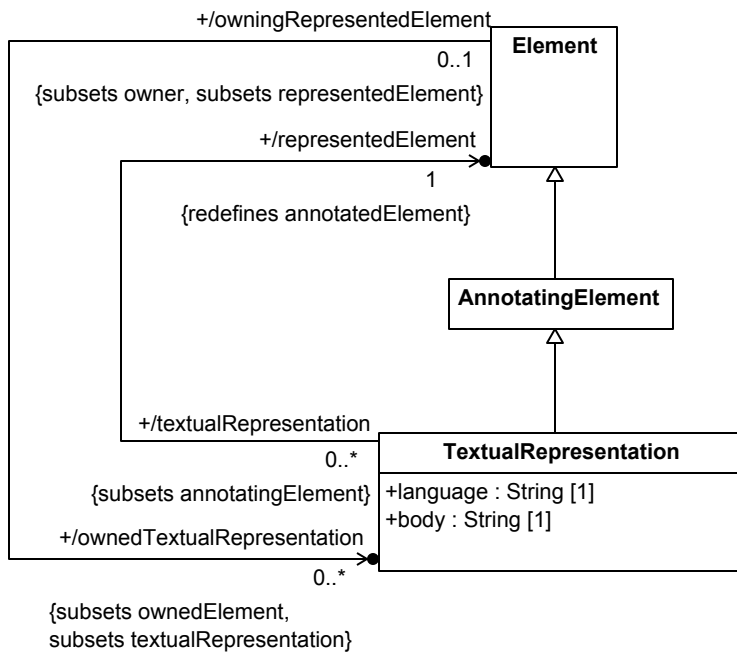


Figure 5. Textual Representation

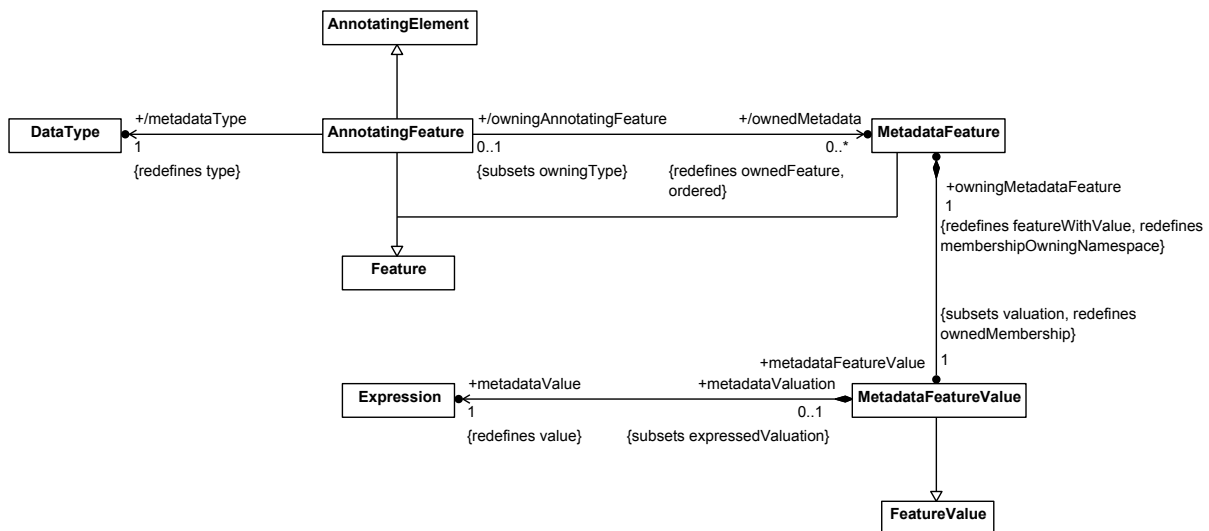


Figure 6. Metadata Annotation

7.2.3 Packages

7.2.3.1 Packages Overview

A Namespace is a kind of Element that can contain other elements and provide names for them. The elements contained in a namespace are referred to as its member elements. Membership is a kind of Relationship that relates a namespace to its members. The members can be either owned members or unowned members. Specific semantics apply to owned members that include deletion semantics. Deletion semantics specify that when a namespace is deleted, all of its owned members are also deleted. A namespace can import another namespace using an import relationship. This enables the importing namespace to refer to the members of the imported namespace directly,

without having to qualify the member names with the name of the imported namespace. The imported members are unowned members of the importing namespace.

A Package is a kind of Namespace that is used solely as a container for other elements to organize the model. A package can also specify that an import of another package is recursive, which means that, in addition to importing members of the referenced package itself, all packages that are owned members of the imported package are also recursively imported. Finally, a package can define filter conditions on the elements that it imports, in terms of the metadata provided by annotating features of those elements (see [7.2.2](#)). Only elements that meet all filter conditions actually become imported members of the package. Together, recursive import and filtering provide a general capability for specifying that a package automatically contain a set of elements identified from across a model by their metadata.

7.2.3.2 Packages Concrete Syntax

7.2.3.2.1 Packages Textual Notation

7.2.3.2.1.1 Packages

```

RootNamespace : Namespace =
    PackagedElement(this)*

Package (m : Membership) : Package =
    PackageDeclaration(this, m) PackageBody(this)

PackageDeclaration (m : Membership, p : Package) : Package =
    'package' Identification(p, m)

PackageBody (p : Package) =
    ';' | '{' PackageBodyElement(p)* '}'

PackageBodyElement (p : Package) =
    p.ownedRelationship += OwnedDocumentation
    | p.ownedRelationship += PackageMember(p)
    | p.ownedRelationship += Import

PackageMember (p : Package) : Membership
    ( ownedRelationship += PrefixDocumentation )*
    ( visibility = BasicVisibilityIndicator )?
    ( NonUsagePackageMember(this, p)
    | UsagePackageMember(this) )

NonUsagePackageMember (m : Membership, p : Package) =
    m.ownedMemberElement = DefinitionElement(m, p)
    | ( 'alias' | 'import' ) m.memberElement = [Qualified Name]
    ( 'as' m.memberName = NAME )? ';'

UsagePackageMember (m : Membership) =
    m.ownedMemberElement = UsageElement(m)

Import : Import =
    ( ownedRelationship += PrefixDocumentation )*
    ( visibility = BasicVisibilityIndicator )?
    'import' ( ImportedNamespace(this) |
              ImportedFilterPackage(this) )
    '::' ( '*' | isRecursive ?= '**' ) ';'

ImportedNamespace (i : Import) =
    i.importedNamespace = [Qualified Name]

ImportedFilterPackage (i : Import) :
    i.ownedRelatedElement += FilterPackage

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

FilterPackageImport : Import =
    ImportedNamespace (this)

```

```
FilterPackageMember : ElementFilterMembership =  
    '[' condition = OwnedExpression ']'  
    { visibility = 'private' }  
  
BasicVisibilityIndicator : VisibilityKind =  
    'public' | 'private'
```

The SysML textual concrete syntax for a Package is the same as the KerML textual notation (see [KerML, 7.4.11.2]), except that only legal SysML Elements are allowed in a SysML Package.

As in KerML, the declaration of a root Namespace (see [KerML, 7.2.4.1]) is implicit and no identification of it is provided in the SysML textual notation. Instead, the body of a root Namespace (i.e., a SysML "model") is given simply by the list of representations of its top-level Elements, typically in a single textual document. Other than this implicit declaration of the root Namespace, SysML does not provide any concrete syntax for Namespaces that are not Packages. While the Namespace metaclass is abstract in the KerML abstract syntax, it shall not be directly instantiated in a SysML model.

The KerML rules for name resolution shall also apply for the SysML textual concrete syntax (see [KerML, 7.2.4.2.4]).

7.2.3.2.1.2 Package Elements

```

DefinitionElement (m : Membership) : Element =
    Package(m)
  | Comment(m, p)
  | TextualRepresentation(m, p)
  | AnnotatingFeature(m, p)
  | Dependency(m)
  | AttributeDefinition(m)
  | EnumerationDefinition(m)
  | ItemDefinition(m)
  | PartDefinition(m)
  | IndividualDefinition(m)
  | ConnectionDefinition(m)
  | InterfaceDefinition(m)
  | PortDefinition(m)
  | ActionDefinition(m)
  | CalculationDefinition(m)
  | StateDefinition(m)
  | ConstraintDefinition(m)
  | RequirementDefinition(m)
  | ConcernDefinition(m)
  | StakeholderDefinition(m)
  | CaseDefinition(m)
  | AnalysisCaseDefinition(m)
  | VerificationCaseDefinition(m)
  | ViewDefinition(m)
  | ViewpointDefinition(m)
  | RenderingDefinition(m)

UsageElement (m : Membership) : Usage =
    AttributeUsage(m)
  | EnumerationUsage(m)
  | ItemUsage(m)
  | PartUsage(m)
  | IndividualUsage(m)
  | TimeSliceUsage(m)
  | SnapshotUsage(m)
  | PortUsage(m)
  | ConnectionUsage(m)
  | Connector(m)
  | InterfaceUsage(m)
  | ActionUsage(m)
  | CalculationUsage(m)
  | StateUsage(m)
  | ConstraintUsage(m)
  | RequirementUsage(m)
  | ConcernUsage(m)
  | StakeholderUsage(m)
  | CaseUsage(m)
  | AnalysisCaseUsage(m)
  | VerificationCaseUsage(m)
  | ViewUsage(m)

```

```
| ViewpointUsage (m)
| RenderingUsage (m)
```

A Package body can contain other Packages, various AnnotatingElements (see [7.2.2](#)) and SysML Definitions and Usages (see [7.4](#)).

7.2.3.2.2 Packages Graphical Notation

7.2.3.2.2.1 Packages

```

element-text (p : Package) =
    PackageDeclaration(p, p.owningMembership)

```

```

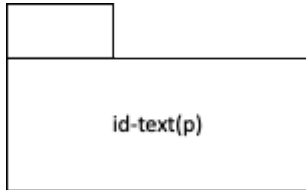
element-symbol (p : Package) =
    package-no-body-symbol (p)
    | package-text-body-symbol (p)
    | package-graphic-body-symbol (p)

```

```

package-no-body-symbol (p : Package) =

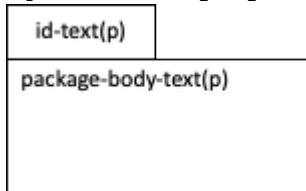
```



```

package-text-body-symbol (p : Package) =

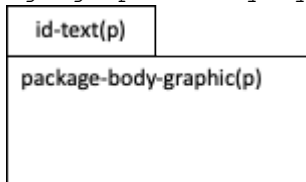
```



```

package-graphic-body-symbol (p : Package) =

```



```

package-body-text (p : Package) =
    ( package-membership-text (p.ownedMembership)
    | package-alias-text (p.ownedMembership)
    | package-import-text (p.ownedImport) ) *

```

```

package-membership-text (m : Membership) =
    visibility-text (m.visibility) ? annotated-element-text (m.ownedMemberElement)

```

```

package-alias-text (m : Membership) =
    visibility-text (m.visibility) ? ( 'alias' | 'import' )
    qualified-name-text (m.memberElement) ( 'as' NAME (m.memberName) ) ?
    element-graphic (i.ownedAnnotation) *

```

```

package-import-text (i : Import) =
    visibility-text (i.visibility) ? 'import'
    qualified-name-text (i.importedPackage) '::<' '*'
    element-graphic (i.ownedAnnotation) *

```

```

package-body-graphic (p : Package) =

```



```
element-graphic(m.ownedMember) *
```

7.2.3.2.2 Memberships

```
element-symbol (m : Membership) =  
  package-membership-symbol(m) | package-alias-symbol(m)  
  
package-membership-symbol (m : Membership) =  
  &element-symbol  
  (m.membershipOwningPackage)  $\oplus$   $\xrightarrow{\text{visibility-text}(m.\text{visibility})?}$  &element-symbol  
  (m.ownedMemberElement)  
  
package-alias-symbol(m : Membership) =  
  &element-symbol  
  (m.membershipOwningPackage)  $\xrightarrow[\text{NAME}(m.\text{memberName})?]{\text{visibility-text}(i.\text{visibility})? \text{'«alias»'}}$  &element-symbol  
  (m.memberElement)  
  
element-symbol (i : Import) =  
  &element-symbol  
  (i.importOwningPackage)  $\xrightarrow[\text{visibility-text}(i.\text{visibility})? \text{'«import»'}]{}$  &element-symbol  
  (i.importedPackage)  
  
visibility-text (VisibilityKind::public) = '+' | 'public'  
visibility-text (VisibilityKind::private) = '-' | 'private'
```

7.2.3.3 Packages Abstract Syntax

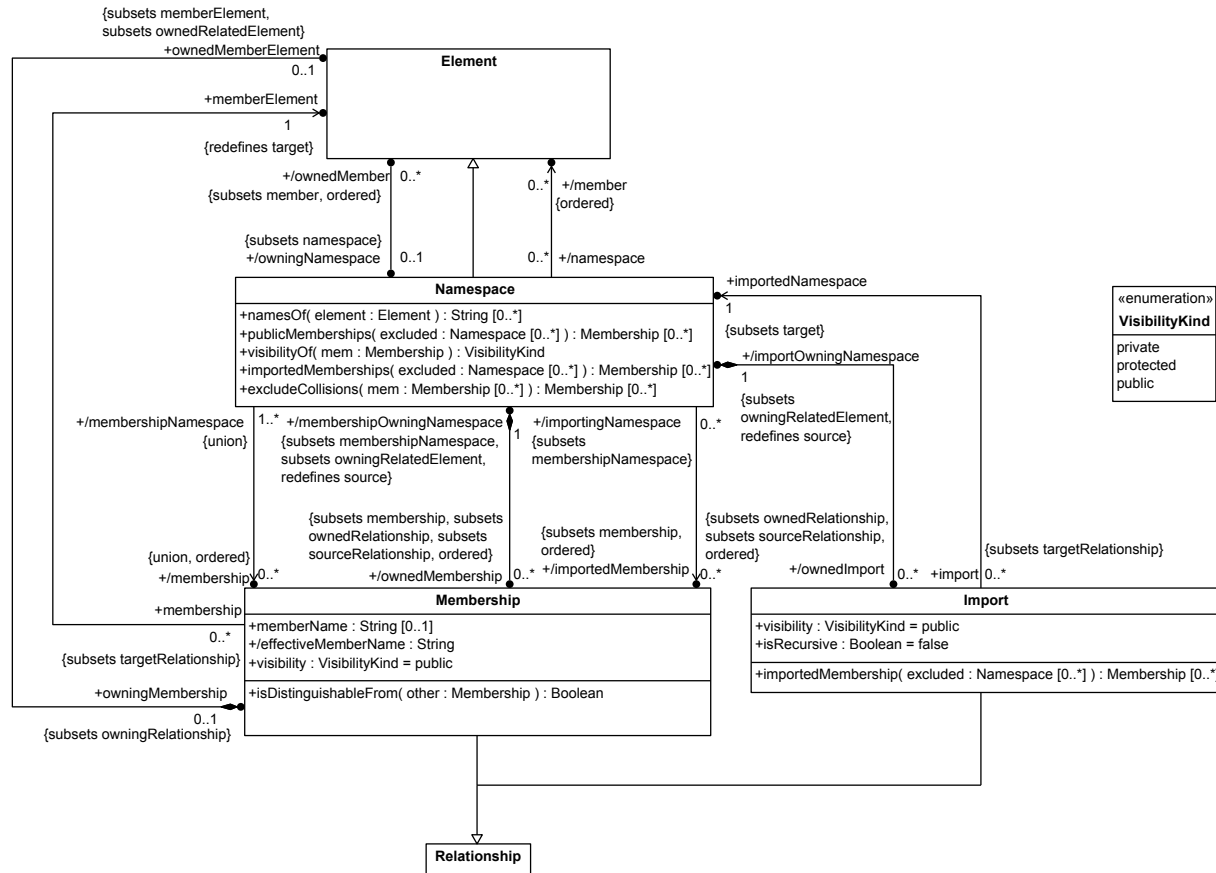


Figure 7. Namespaces

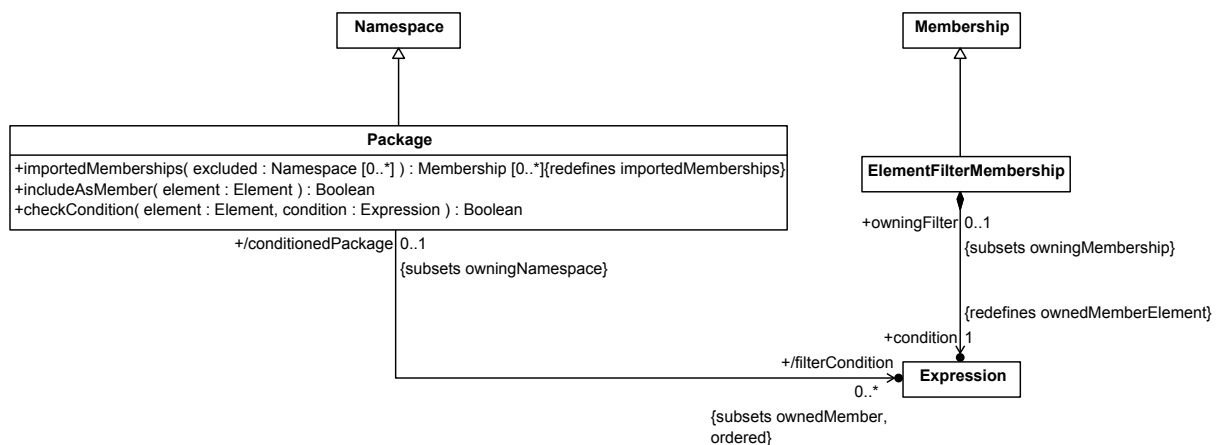


Figure 8. Packages

7.2.4 Types

7.2.4.1 Types Overview

Types

Types are used to classify things in the universe being modeled, which are called the instances of the type. There are two kinds of types: classifiers, which classify individual things (see [7.2.5](#)), and features, which classify how things are related (see [7.2.6](#)). In SysML, definitions are classifiers, while usages are features (see [7.4](#)).

A type is also a kind of namespace and, therefore, can contain members (see [7.2.3](#)). Feature membership is a special kind of membership relationship between a type and its features. Each feature of a type relates the featuring type to other types, mapping from instances of the featuring type to zero or more values of the feature. In this way, a features of type can be used to model properties of the instances classified by that type, whose values are instances classified by the types of the those features.

Since a feature is itself a kind of type, a feature can have its own nested features. This means that the relation defined by a feature is relative to the context of the hierarchy of features in which it is nested.

In KerML, any kind of type can have a multiplicity restricting the cardinality of its instances. However, in SysML multiplicity is only used on features (see [7.2.6](#)), except implicitly to restrict the cardinality for individual definitions (see [7.13](#)).

Generalization

Generalization is a relationships between a specific type and a general one, indicating that all instances of the specific Type are instances of the general one. This means instances of the specific Type have all the features of the general ones, referred to syntactically as *inheriting* features from general to specific Types. It is allowable for generalization relationships to form cycles, which means all the types in the cycle have the same instances.

There are specialized kinds of generalization for classifiers and features, and these are the only kinds used in SysML (see [7.2.5](#) and [7.2.6](#)).

Conjugation

Conjugation is a relationship between two Types in which the conjugated type inherits features from the original type, except that the direction of input and output features is reversed. Features no direction are inherited without change. In SysML, conjugation is only used for ports (see [7.9](#)).

7.2.4.2 Types Concrete Syntax

7.2.4.2.1 Types Textual Notation

7.2.4.2.1.1 Types

```
TypeBody (t : Type) :  
    ';' | '{' TypeBodyElement(t) * '}'  
  
TypeBodyElement (t : Type) : Type =  
    t.ownedRelationship += OwnedDocumentation  
    t.ownedRelationship += NonFeatureTypeMember(t)  
| t.ownedRelationship += FeatureTypeMember  
| t.ownedRelationship += PackageImport  
  
NonFeatureTypeMember (t : Type) : Membership =  
    TypeMemberPrefix(this) DefinitionElement(this, t)  
  
FeatureTypeMember : FeatureMembership =  
    FeatureMember | EndFeatureMember  
  
TypeMemberPrefix (m : Membership) :  
    ( m.ownedRelationship += PrefixAnnotation ) *  
    ( m.visibility = VisibilityIndicator ) ?  
  
VisibilityIndicator : VisibilityKind =  
    PackageVisibilityIndicator | 'protected'
```

SysML does not provide any concrete syntax for generic Types or Generalizations that are not instances of a more specialized metaclass. While the Type and Generalization metaclasses are not abstract in the KerML abstract syntax, they shall not be directly instantiated in a SysML model.

The common syntax defined above is used in the definition of the concrete syntax of the bodies of BindingConnectors (see [7.2.8](#)), Successions (see [7.2.8](#)) and ItemFlows (see [7.2.10](#)), and in part for SysML Definitions and Usages (see [7.4](#)).

Conjugation is used implicitly in the context of the syntax of Ports in SysML (see [7.9](#)). They shall not be used in SysML outside of this context.

7.2.4.2.1.2 Feature Membership

```
FeatureMember : FeatureMembership =  
    TypeMemberPrefix(this) direction = FeatureDirection  
    ownedMemberFeature = FeatureElement(this)  
  
EndFeatureMember : EndFeatureMembership =  
    TypeMemberPrefix(this) 'end' direction = FeatureDirection  
    ownedMemberFeature = FeatureElement(this)  
  
FeatureDirection : FeatureDirectionKind =  
    'in' | 'out' | 'inout'  
  
FeatureElement (m : Membership) : Feature =  
    UsageElement(m)  
    | BindingConnector(m)  
    | Succession(m)  
    | ItemFlow(m)  
    | SuccessionItemFlow(m)
```

7.2.4.2.2 Types Graphical Notation

7.2.4.2.2.1 Types

```
element-text (t : Type) =  
    variation-keyword(t)? abstract-keyword(t)? ref-keyword(t)?  
    kind-keyword(t) type-decl-text(t)  
  
variation-keyword (t : Type) = none  
  
abstract-keyword (t : Type) =  
    'abstract'(t.isAbstract)  
  
kind-keyword (t : Type) = ...  
  
ref-keyword (t : Type) = none  
  
type-decl-text (t : Type) = ...
```

Notes

- A KerML Type cannot in general be a variation. However, SysML Definitions and Usages can, and Type is the common superclass of Definition and Usage (see [7.4](#)). Therefore, it is convenient to provide a placeholder for the **variation** keyword in the general textual representation for a Type, even though this is never used for Types that are not Definitions or Usages.
- The *kind keyword* and *type declaration text* for a Type are specified for specific kinds of Types in following subclauses.
- The **ref** keyword is only applicable for Features (see [7.2.6](#)), but it is convenient to provide a placeholder for it in the general textual representation for a Type.

7.2.4.2.2.2 Feature Membership

```
package-membership-text (m : FeatureMembership) =  
  visibility-text(m.visibility)? end-keyword(m)? direction-keyword(m)?  
  annotated-element-text(m.ownedMemberElement)
```

```
visibility-text (VisibilityKind::protected) = '#' | 'protected'
```

Note: The visibility 'protected' is allowed only for members of Types.

```
end-keyword (m : FeatureMembership) = none
```

```
end-keyword (m : EndFeatureMembership) = 'end'
```

```
direction-keyword (m : Membership) = direction-text(m.direction)
```

```
direction-text (FeatureDirectionKind::in) = 'in'
```

```
direction-text (FeatureDirectionKind::out) = 'out'
```

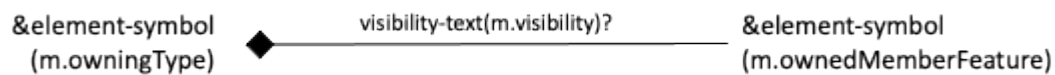
```
direction-text (FeatureDirectionKind::inout) = 'inout'
```

```
direction-keyword (m : ReturnParameterMembership) = 'return'
```

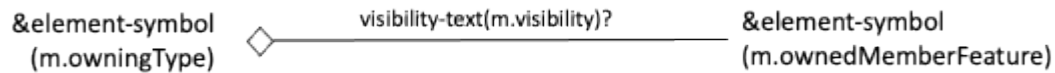
```
package-membership-graphic (m : FeatureMembership) =
```

```
  feature-membership-graphic(m, m.isComposite)
```

```
feature-membership-graphic (m : FeatureMembership, true) =
```



```
feature-membership-graphic (m : FeatureMembership, false) =
```



OMG Systems Modeling Language (SysML) v2.0, Submission



Figure 9. Types

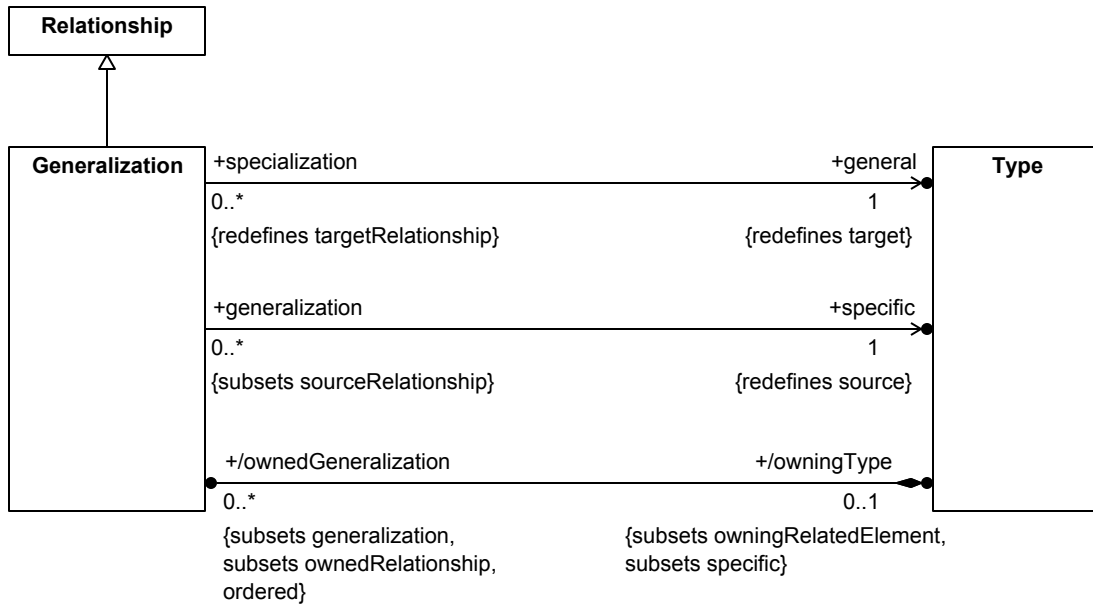


Figure 10. Generalization

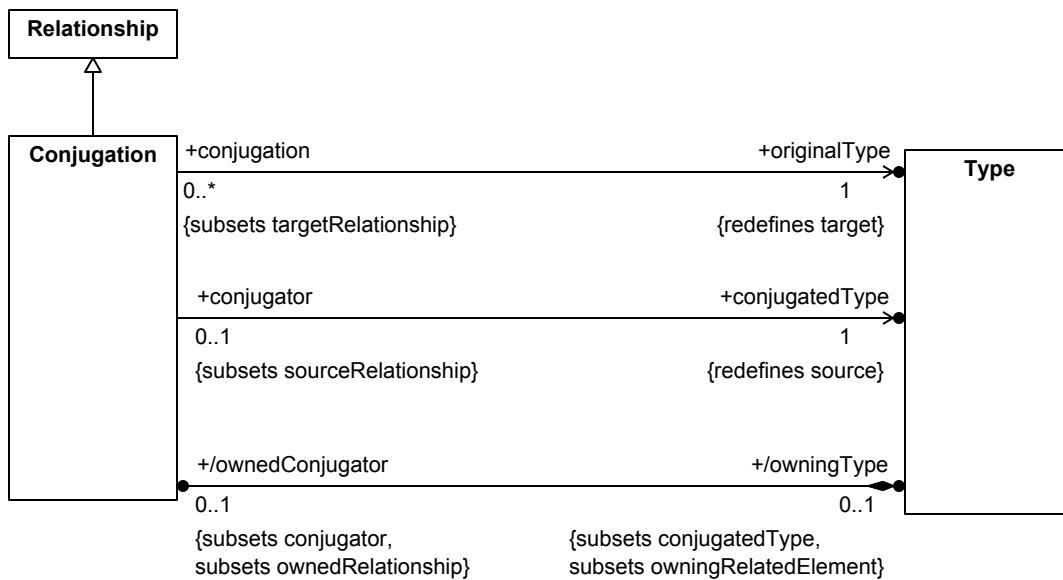


Figure 11. Conjugation

7.2.5 Classifiers

7.2.5.1 Classifiers Overview

Classifiers are types that classify things in the modeled universe, regardless of how features related them. In SysML, all definitions are classifiers (see [7.4](#)).

Superclassing is a kind of generalization between classifiers. SysML allows superclassing between definitions.

Data types are classifiers that classify data values, that is, values that do not change over time. Data values are generally distinguishable when they differ in how they are related to other things. However, data values for some data types are directly identified (enumerated), in which case they are distinguishable regardless of their relationship to other things, including the primitive types defined in the Kernel Model Library *ScalarValues* package (see [KerML, 8.10]), and any subtypes of those. In SysML, attribute definitions are data types (see [7.5](#)),

Classes are classifiers that classify occurrences. Every occurrence is considered to occur over some period of time, and possibly also over some extent in space. The relationships that an occurrence has with other things in the modeled universe (via features) may vary over time and space. That is, a single Feature may be interpreted to map different portions of the temporal and spacial extent of an occurrence to different results.

There are two kinds of classes, structures that classify objects and behaviors that classify performances (behaviors are discussed in [7.2.9](#)). Objects may be involved in and be acted on by the performance of a behavior. An object may also be the performer of a behavior. In SysML, item definitions are structures (see [7.7](#)), and part definitions (a kind of item definition) may, in addition, specify the performance of one or more behaviors (see [7.8](#)).

7.2.5.2 Classifiers Concrete Syntax

7.2.5.2.1 Classifiers Textual Notation

```
SuperclassingPart (c : Classifier) =  
    ( '>' | 'specializes' ) c.ownedSuperclassing += OwnedSuperclassing  
    ( ',' c.ownedSuperclassing += OwnedSuperclassing ) *  
  
OwnedSuperclassing : Superclassing =  
    superclass = [Qualified Name]
```

SysML only provides concrete syntax for classifiers that are kinds of definitions (see [7.4](#)). The notation for Superclassing is consistent with that of KerML.

7.2.5.2.1.1 Classifiers

SysML only provides concrete syntax for classifiers that are kinds of definitions (see [7.4](#)).

7.2.5.2.1.2 Superclassing

```
SuperclassingPart (c : Classifier) =  
    ( '>' | 'specializes' ) c.ownedRelationship += OwnedSuperclassing  
    ( ',' c.ownedRelationship += OwnedSuperclassing ) *  
  
OwnedSuperclassing : Superclassing =  
    superclass = [Qualified Name]
```

7.2.5.2.2 Classifiers Graphical Notation

7.2.5.2.2.1 Classifiers

SysML only provides concrete syntax for classifiers that are kinds of definitions (see [7.4](#)).

7.2.5.2.2 Superclassing

```

element-symbol (s : Superclassing) =
    &element-symbol (s.subclass) -----> &element-symbol (s.superclass)
  
```

7.2.5.3 Classifiers Abstract Syntax

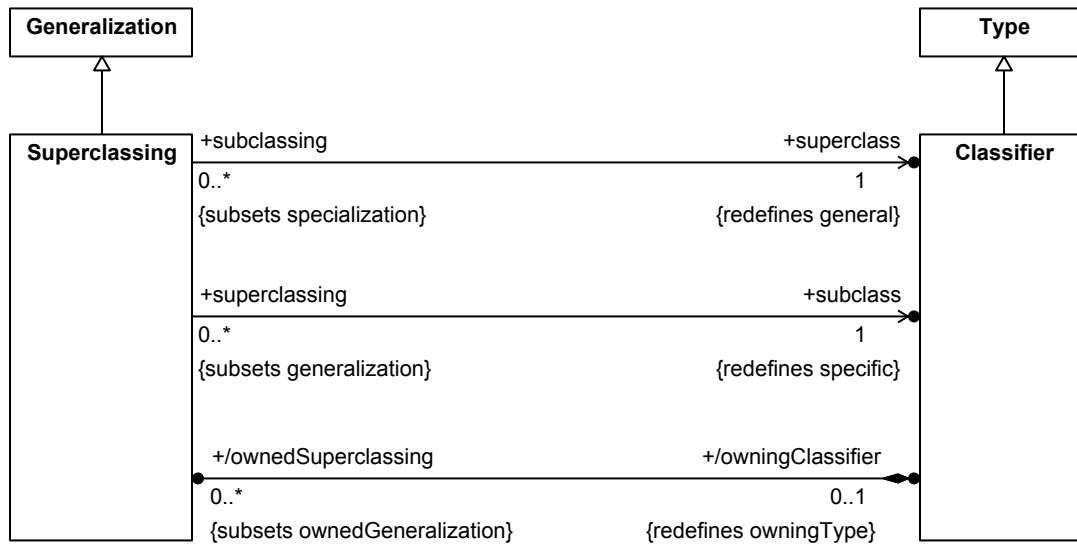


Figure 12. Classifiers

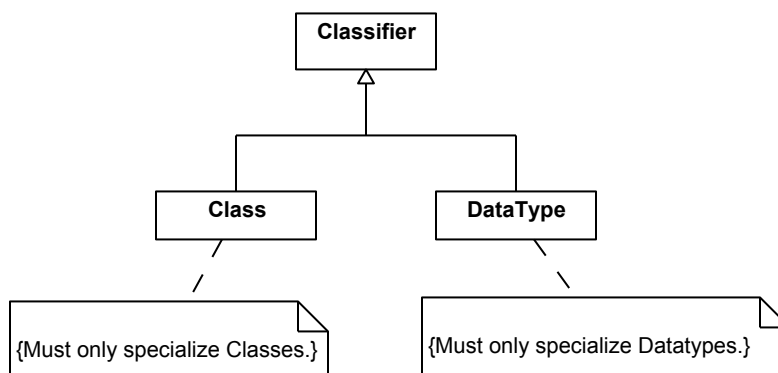


Figure 13. Classification

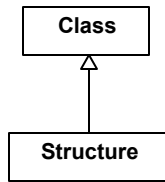


Figure 14. Structures

7.2.6 Features

7.2.6.1 Features Overview

A feature is a type that classifies how things in the modeled universe are related, including by chains of relationships. Relations between things can themselves be treated as things, allowing relations between relations (recurring as many times as needed). In SysML, all usages are features.

Multiplicity

A feature may have a multiplicity that constrains its cardinality, that is, the allowed number of values it may have for any instance of its featuring type. The multiplicity is specified as a range, giving the lower and upper bound expressions that are evaluated to determine the lower and upper bounds of the specified range. The lower bound must be a natural number, while the upper bound must be an unlimited natural number, that is, a natural number of the unbounded value *. An upper bound value of * indicates that the range is unbounded, that is, it includes all numbers greater than or equal to the lower bound value. If a lower bound is not given, then the lower bound is taken to be the same as the upper bound, unless the upper bound is *, in which case the lower bound is taken to be 0.

Submission Note. Allowing more kinds of Multiplicities than just ranges (e.g., sets of cardinalities like `[2, 4, 6]`) will be considered for the revised submission.

Feature Typing

Feature typing is a kind of generalization between a feature and a type. Feature typing is used to declare the types that values of the typed feature must have.

Subsetting

Subsetting is a kind of generalization between two features, in which the values of the subsetting feature are a subset of the values of the subsetted feature. The subsetting feature inherits the feature typings of the subsetted feature, but may specify additional feature typings, whose types further constraint the values of the subsetting feature. The subsetting feature also by default inherits the multiplicity of the subsetted feature, but may further constraint that multiplicity.

Redefinition

Redefinition is a kind of subsetting in which the redefined feature is a feature of a direct or indirect generalization of the owning type of the redefining feature that would otherwise be inherited. In this case, the otherwise inheritable redefined feature is not inherited into the namespace of the owning type and is, instead, replaced by the redefining feature. As for regular subsetting, the redefined feature may further restrict the types and/or the multiplicity of the redefined feature.

Feature Values

A feature value is a special kind of membership that relates a feature to an expression (see [7.2.12](#)). A feature can have at most one feature value. The result of the expression is bound to the feature itself (see [7.2.8](#) on binding connectors), effectively asserting that the values of the feature are always determined by the result of the given Expression.

7.2.6.2 Features Concrete Syntax

7.2.6.2.1 Features Textual Notation

7.2.6.2.1.1 Features

```
FeatureDeclaration (f : Feature, m : Membership) =
    Identification(f, m) FeatureSpecializationPart(f)?

FeatureSpecializationPart (f : Feature) =
    FeatureSpecialization(f)+ MultiplicityPart(f)? FeatureSpecialization(f)*
    | MultiplicityPart(f) FeatureSpecialization(f)*

MultiplicityPart (f : Feature) =
    f.ownedRelationship += MultiplicityMember
    ( f.isOrdered ?= 'ordered' ( !f.isUnique ?= 'nonunique' )?
    | !f.isUnique ?= 'nonunique' ( isOrdered ?= 'ordered' )? )?

MultiplicityMember : Membership =
    ownedMemberFeature = Multiplicity

Multiplicity : MultiplicityRange =
    '[' ( ownedRelationship += LiteralExpressionMember '..' )?
    ownedRelationship += LiteralExpressionMember ']'

LiteralExpressionMember : Membership =
    ownedMemberElement = OwnedExpression

ValuePart (f : Feature) =
    '=' f.ownedRelationship += FeatureValue

FeatureValue : FeatureValue =
    value = OwnedExpression
```

Other than for certain kinds of Connectors (see [7.2.8](#) and [7.2.10](#)), SysML only provides concrete syntax for Features that are kinds of Usages (see [7.4](#)). The common syntax for Feature declaration defined above is used consistently in the notation for all the specific kinds of Features in SysML.

7.2.6.2.1.2 Feature Specialization

```
FeatureSpecialization (f : Feature) =
    Typings(f) | Subsettings(f) | Redefinitions(f)

Typings (f : Feature) =
    TypedBy(f) ( ',' f.ownedTyping += OwnedFeatureTyping )*

TypedBy (f : Feature) =
    ( ':' | 'defined' 'by' ) f.ownedTyping += OwnedFeatureTyping

OwnedFeatureTyping : FeatureTyping =
    type = [Qualified Name]

Subsettings (f : Feature) =
    Subsets(f) ( ',' f.ownedSubsetting += OwnedSubsetting )*

Subsets (f : Feature) =
    ( '>' | 'subsets' ) f.ownedSubsetting += OwnedSubsetting

OwnedSubsetting : Subsetting =
    subsettingFeature = [Qualified Name]

Redefinitions (f : Feature) =
    Redefines(f) ( ',' f.ownedRedefinition += OwnedRedefinition )*

Redefines (f : Feature) =
    ( '>>' | 'redefines' ) ownedRelationship += OwnedRedefinition

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

7.2.6.2.2 Features Graphical Notation

7.2.6.2.2.1 Features

```
ref-keyword (f : Feature) =
    ref-text (f.isComposite)

ref-text (true) = none
ref-text (false) = 'ref'

type-dcl-text (f : Feature) =
    FeatureDeclaration(f, f.owningMembership)

feature-label (f : Feature) =
    visibility-keyword(f)? abstract-keyword(f)? ( '«' kind-keyword(f) '»' )?
    id-text(f)? feature-modifier-text(f)*
(Note: As an alternative to using the 'abstract' keyword, the entire label
can be rendered in italics.)

feature-modifier-text (f : Feature) =
    feature-multiplicity-text(f)
| feature-typing-text(f)
| feature-subsetting-text(f)
| feature-redefinition-text(f)
| feature-ordered-text(f)
| feature-nonunique-text(f)

feature-multiplicity-text (f : Feature) =
    element-text(f.multiplicity)

element-text (m : MultiplicityRange) =
    ( OwnedExpression(f.lowerBound) '..' )? OwnedExpression(f.upperBound)

feature-typing-text (f : Feature) = Typings(f)

feature-subsetting-text (f : Feature) = Subsettings(f)

feature-redefinition-text (f : Feature) = Redefinitions(f)

feature-ordering-text (f : Feature) = 'ordered'(f.isOrdered)

feature-nonuniqueness-text (f : Feature) = 'nonunique'(!f.isUnique)
```

7.2.6.2.2 Feature Specialization

```

element-symbol (f : FeatureTyping) =
    &element-symbol (f.typedFeature) —————▶ &element-symbol (f.type)

element-symbol (s : Subsetting) =
    &element-symbol (s.subsettingFeature) —————▶ &element-symbol (s.subsettedFeature)

element-symbol (r : Redefinition) =
    &element-symbol (r.redefiningFeature) —————▶ &element-symbol (r.redefinedFeature)

```

7.2.6.3 Features Abstract Syntax

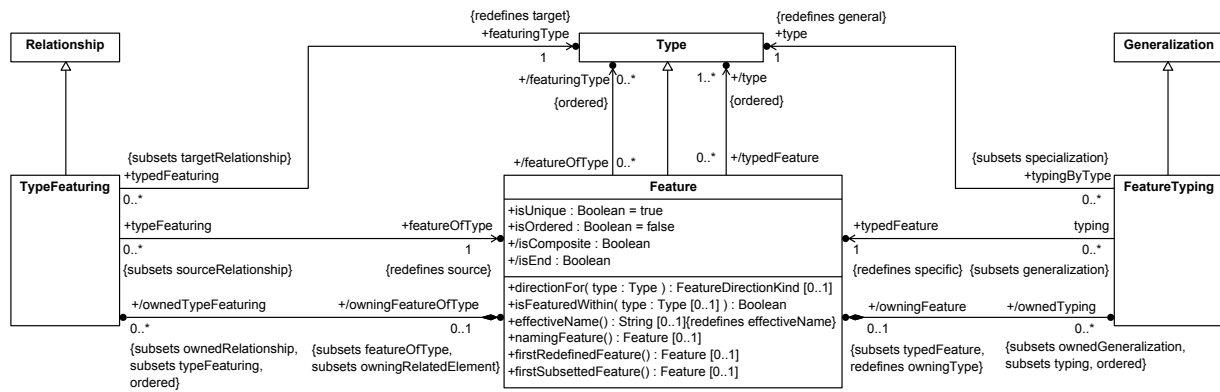


Figure 15. Features

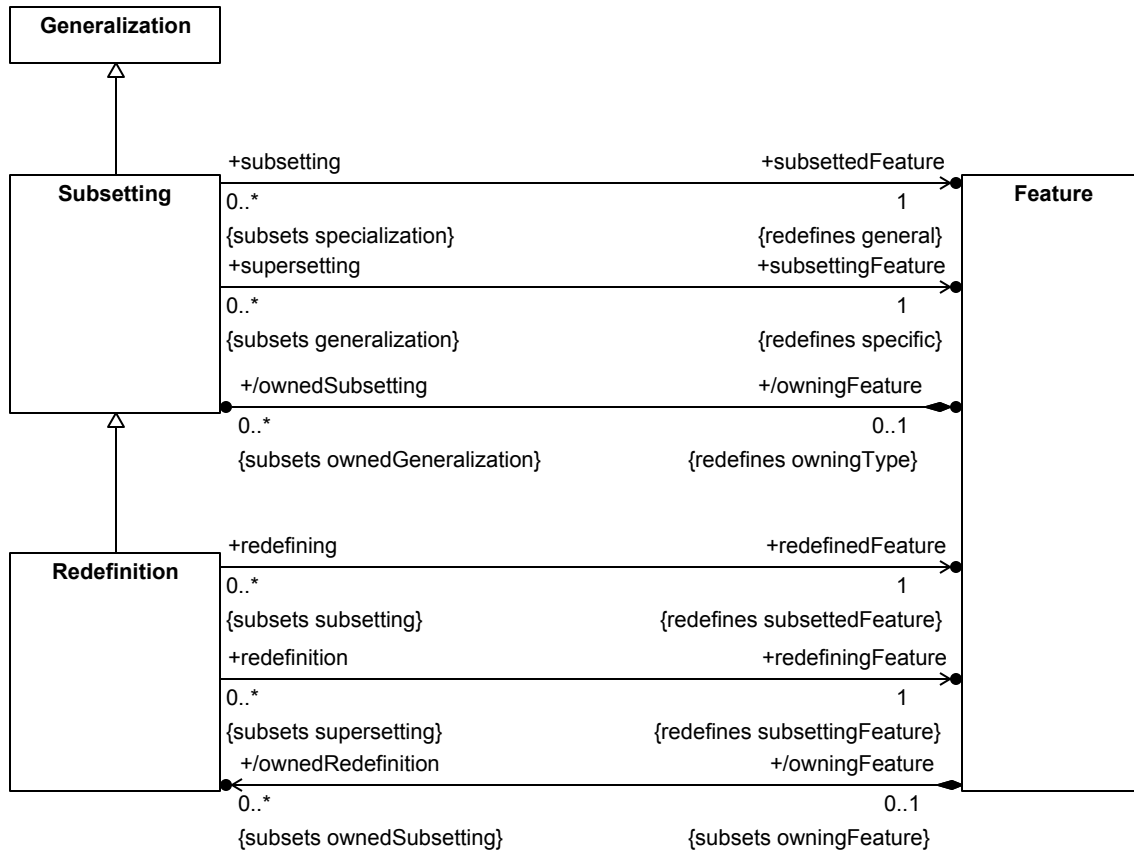


Figure 16. Subsetting

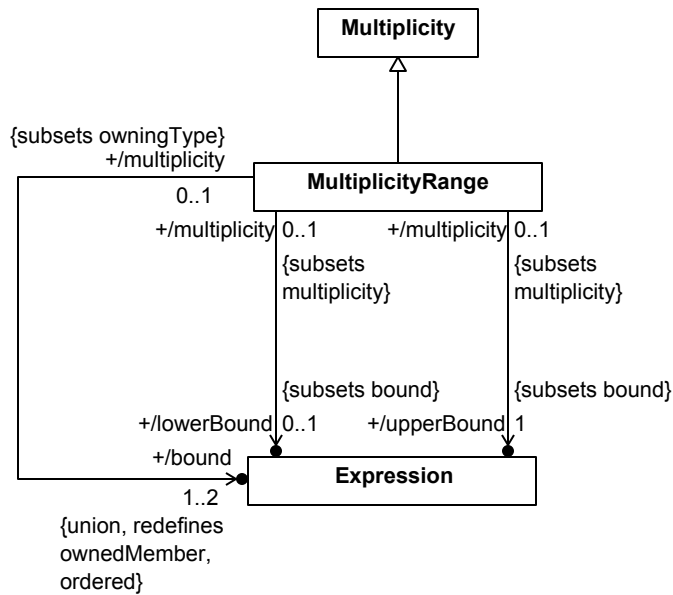


Figure 17. Multiplicities

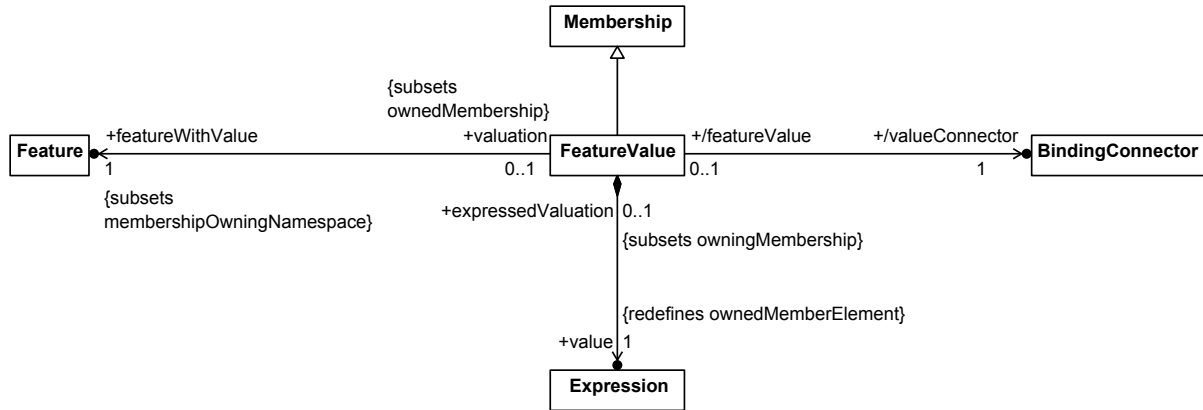


Figure 18. Feature Values

7.2.7 Associations

7.2.7.1 Associations Overview

Associations are classes that classify links between things in the modeled universe, and how they are related by features to those (other) things. SysML connection definitions are associations.

At least two of the owned features of an association must be association ends, related to the association by special end feature memberships. Associations with exactly two association ends classify binary Associations, and they classify binary links.

An association is also a relationship between the types of its association ends. The links of the associations are between instances of these related types.

The features of an association that are not association ends characterize links separately from the linked things. The values of these non-end features can potentially change over time. However, the values of association ends do not change over time (though they can potentially be objects that themselves have features whose values change over time).

7.2.7.2 Associations Concrete Syntax

SysML does not provide any concrete syntax for Associations other than `ConnectionDefinition` (see [7.10](#)) and its specialization `InterfaceDefinition` (see [7.11](#)). The Association metaclass shall not be directly instantiated in a SysML model.

7.2.7.3 Associations Abstract Syntax

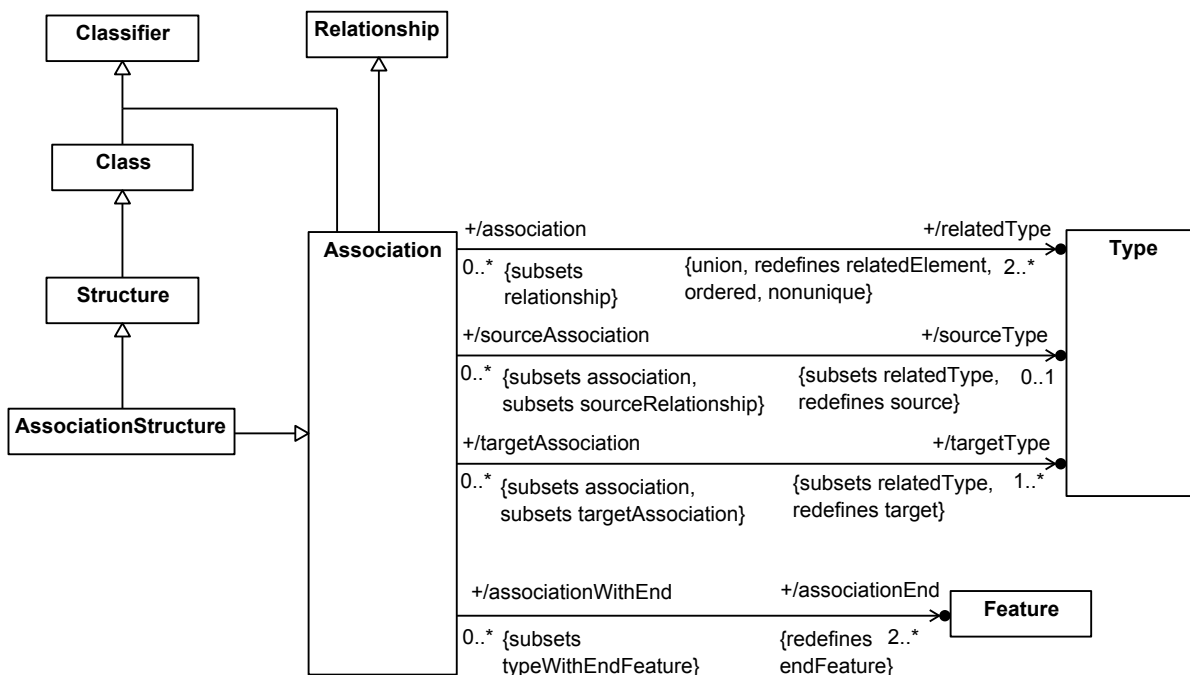


Figure 19. Associations

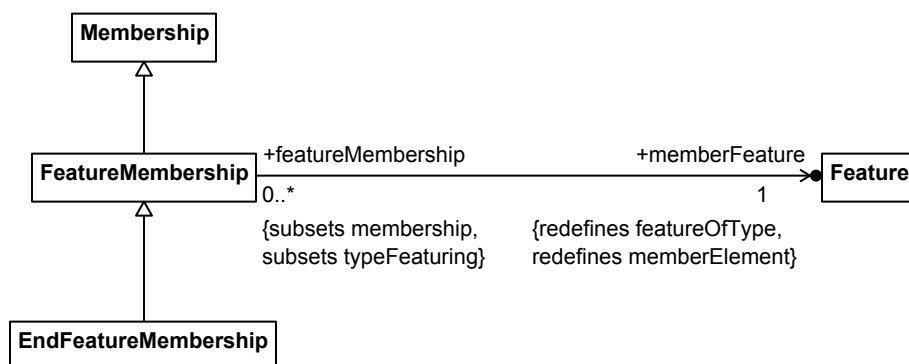


Figure 20. EndFeatureMembership

7.2.8 Connectors

7.2.8.1 Connectors Overview

Connectors

A connector is a kind of feature that is typed by one or more associations. The associations specify the kinds of things that can be linked by the connector, and the connector further restricts those links to be between the values of specific features in its context, known as its related features. A connector is also a relationship between its related features.

All associations typing a connector must have the same number of association ends, which also must be the number of end features of the connector, its connector ends. Each connector end redefines one association end from each of

its types and subsets one of the related features of the connector. Connectors typed by binary associations are called binary connectors.

In SysML, connection usages are connectors between any kinds of items (see [7.10](#)), while interface usages ([7.11](#)) are connectors between port usages.

Binding Connectors

A binding connector is a binary connector that requires its two related features to have the same values. Since data types and classes have disjoint values, a feature typed by data types can only be bound to another feature typed by data types and a feature typed by classes can only be bound to another feature typed by classes. The binding of features typed by classes indicates that the same objects play the roles represented by the related features.

Successions

A succession is a binary connector that requires its two related features to have values that are occurrences that happen completely separated in time, with the first occurrence happening before the second. The linked occurrences can either be objects (instances of Classes, see [7.2.5](#)) or performances (instance of Behaviors, see [7.2.9](#)).

7.2.8.2 Connectors Concrete Syntax

7.2.8.2.1 Connectors Textual Notation

7.2.8.2.1.1 Connectors

```
ConnectorEndMember : EndFeatureMembership :  
    ( memberName = NAME '=>' )? ownedMemberFeature = ConnectorEnd  
  
ConnectorEnd : Feature =  
    ( ownedRelationship += OwnedSubsetting  
    | ownedRelationship += FeaturePathExpressionMember )  
    ( ownedRelationship += MultiplicityMember )?  
  
FeaturePathExpressionMember : FeatureMembership =  
    ownedMemberFeature = FeaturePathExpression  
  
FeaturePathExpression : InvocationExpression =  
    FeaturePathPrefix(this) '.'  
    ownedRelationship += FeatureReferenceExpressionMember  
  
FeaturePathPrefix (e : InvocationExpression) =  
    ( e.ownedRelationship += FeatureReferenceExpressionMember  
    | e.ownedRelationship += FeaturePathExpressionMember )  
    { e.ownedRelationship += [['collect']] }
```

SysML does not provide concrete syntax for generic Connectors that are not instances of some specialization of Connector. The Connector metaclass shall not be directly instantiated in a SysML model. SysML provides concrete syntax for special kinds of Connectors, including BindingConnectors (see [7.2.8.2.1.2](#)), Successions (see [7.2.8.2.1.3](#)), ItemFlows (see [7.2.10](#)), ConnectionUsages (see [7.10](#)), and InterfaceUsages (see [7.11](#)).

7.2.8.2.1.2 Binding Connectors

```
BindingConnector (m : Membership) : BindingConnector =
  ( isAbstract ?= 'abstract' )? 'bind'
  BindingConnectorDeclaration(this, m) TypeBody(m)

BindingConnectorDeclaration (c : BindingConnector, m : Membership) =
  ( FeatureDeclaration(c, m)? 'as' )?
  c.ownedRelationship += ConnectorEndMember '='
  c.ownedRelationship += ConnectorEndMember
```

7.2.8.2.1.3 Successions

```
Succession (m : Membership) : Succession =
  ( isAbstract ?= 'abstract' )? 'succession'
  SuccessionDeclaration(this, m) TypeBody(this)

SuccessionDeclaration (s : Succession, m : Membership) : Succession :
  ( FeatureDeclaration(s, m)? 'first' )?
  s.ownedFeatureMembership += ConnectorEndMember 'then'
  s.ownedFeatureMembership += ConnectorEndMember
```

7.2.8.2.2 Connectors Graphical Notation

7.2.8.2.2.1 Binding Connectors

```
kind-keyword (c : BindingConnector) = 'bind'

type-dcl-text (c : BindingConnector) =
  BindingConnectorDeclaration(c, c.owningMembership)

element-graphic (c : BindingConnector) =
  &element-symbol (c.relatedElement)  $\frac{\text{feature-label (c.connectorEnd)} \quad \text{( feature-label(c) | '=' )? } \quad \text{feature-label (c.connectorEnd)}}{\text{feature-label (c.relatedElement)}}$  &element-symbol (c.relatedElement)
```

Note: The relatedElement and connectorEnd correspond on each end.

7.2.8.2.2.2 Successions

```
kind-keyword (s : Succession) = 'succession'

type-dcl-text (s : Succession) =
    SuccessionDeclaration(s, s.owningMembership)

element-graphic (s : Succession) =
    &element-symbol (s.sourceFeature) -----> ( feature-label(b) | '«then»' )? &element-symbol (s.targetFeature)
```

7.2.8.3 Connectors Abstract Syntax

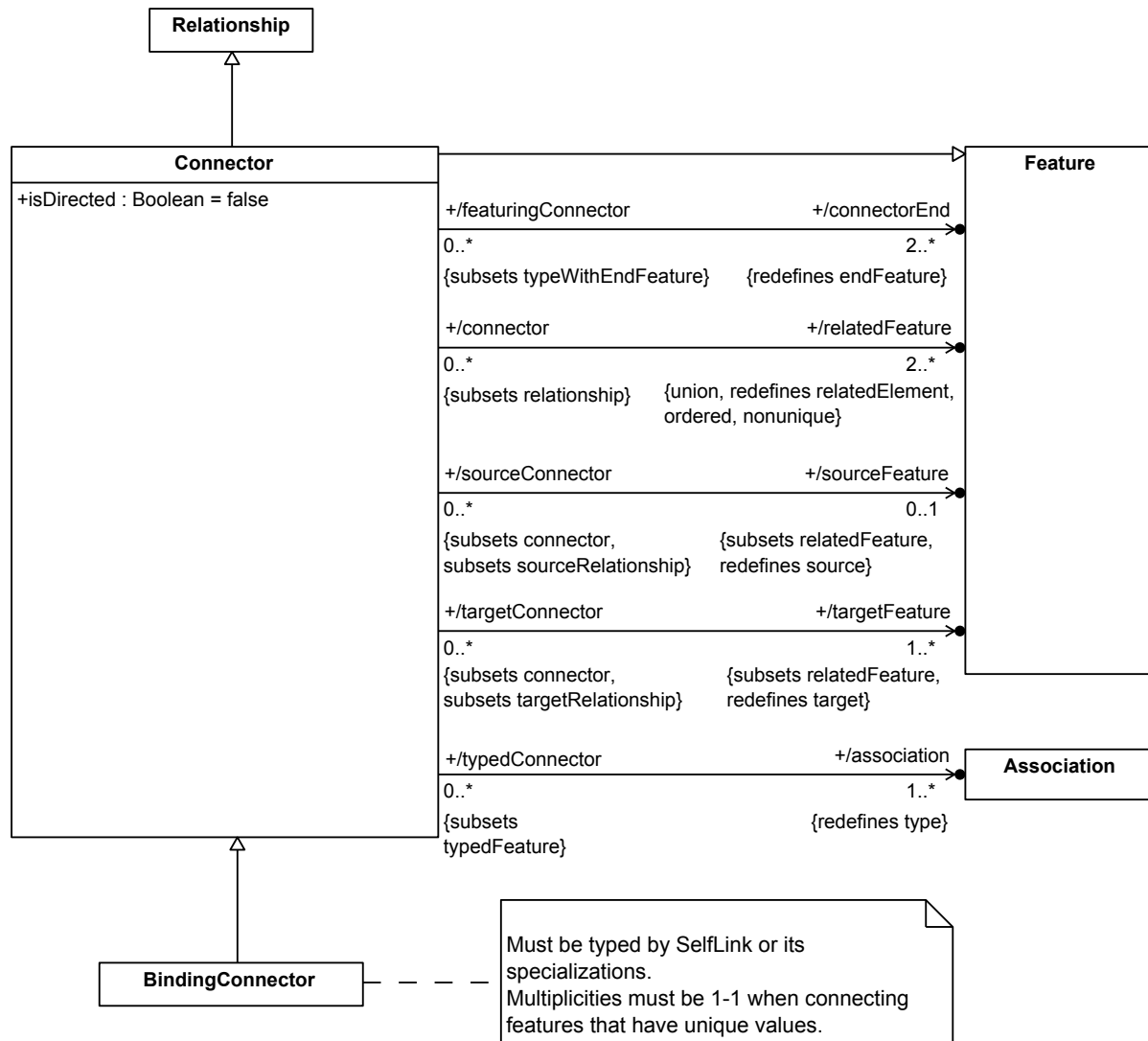


Figure 21. Connectors

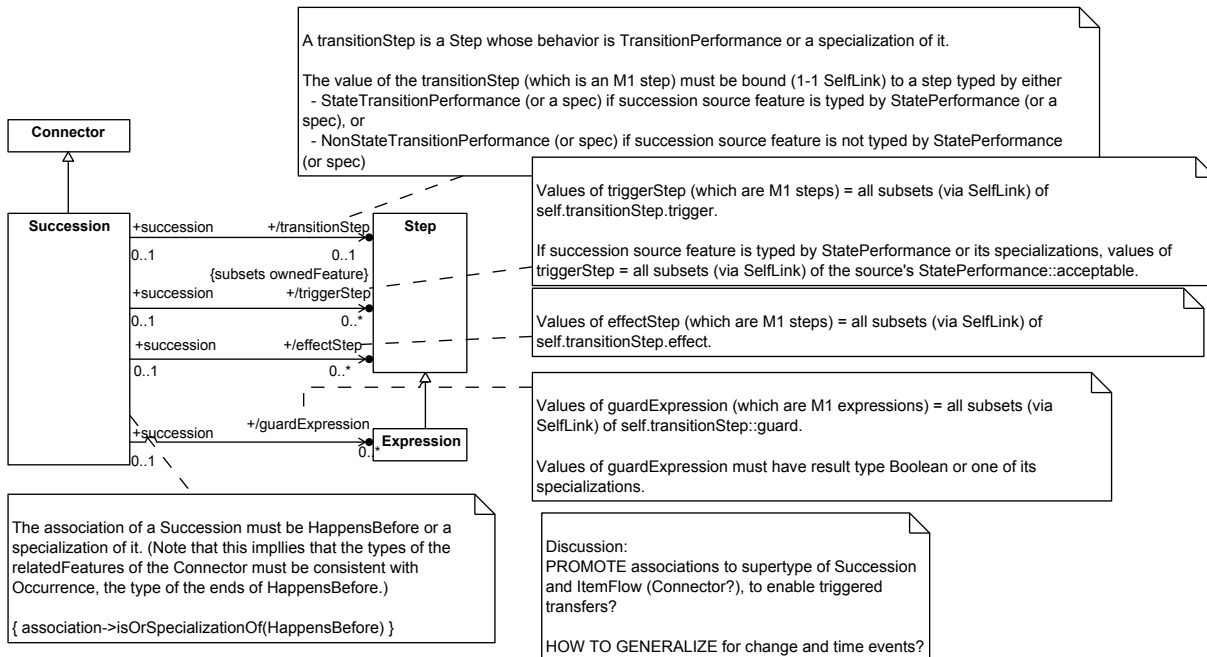


Figure 22. Successions

7.2.9 Behaviors

7.2.9.1 Behaviors Overview

Behaviors

Behaviors are classifiers that classify performances. Performances are occurrences over time that can coordinate the performance of other behaviors and generate effects on objects involved in the performance (including those object's existence and relation to other things), and/or to produce some result before the performance is completed.

A behavior may declare a list of parameters, which are simply distinguished features of the behavior that identify specific information passed into and/or out of the behavior during its performance. A parameter is related to some owning behavior by a special kind of feature membership that is required to have a direction specified, defaulting to "in".

Steps

Steps are features that are typed by behaviors, and, therefore, have performances as their values. The features of a behavior that are steps specify a refinement of the performance of the behavior into performances represented by each of the steps. The steps of a behavior can be connected by successions (see 7.2.8) to order their performances in time and they can be connected by item flows (see 7.2.10) to pass information between their parameters.

Steps inherit the parameters of their behaviors, or they can define their own parameters to augment or redefine those of their behaviors. They can also have nested steps to augment or redefine the steps inherited from their behaviors.

7.2.9.2 Behaviors Concrete Syntax

SysML does not provide any concrete syntax for Behaviors other than ActionDefinitions (see 7.14), ConstraintDefinitions (see 7.17), and their various specializations, nor does it provide any concrete Syntax for Steps other than ItemFlows (see 7.2.10), Expressions (see 7.2.12), ActionUsages (see 7.14), ConstraintUsages (see 7.17), and their various specializations. The Behavior and Step metaclasses shall not be instantiated in SysML models.

7.2.9.3 Behaviors Abstract Syntax

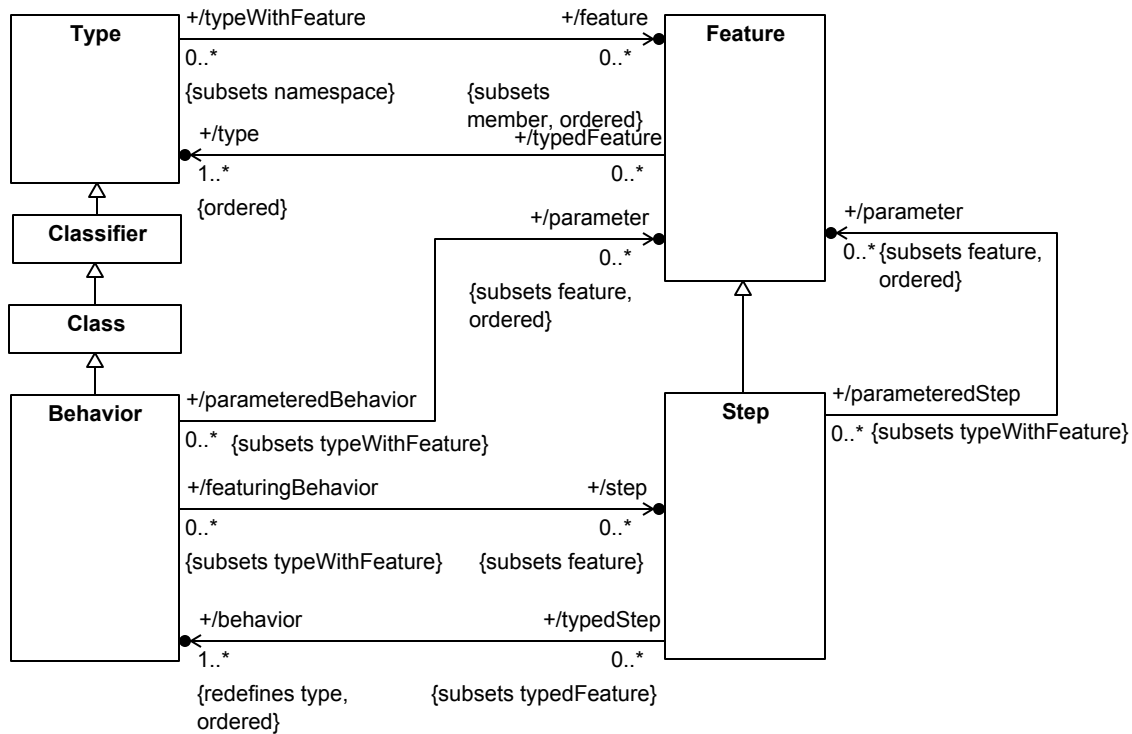


Figure 23. Behaviors

7.2.10 Interactions

7.2.10.1 Interactions Overview

Interactions

Interactions are behaviors that are also associations (see [7.2.9](#) and [7.2.7](#), respectively), whose instances are performances and also links between occurrences. They used to specify how participants affect each other and collaborate. Transfers are a kind of interaction between two participants that specifies when items are provided by one occurrence (via one of its feature) and accepted by another (also via a feature).

Item Flows

Item flows are steps that are also binary connectors (see [7.2.9](#) and [7.2.8](#), respectively) typed as a transfer. An item flow specifies a transfer of items between the Occurrences identified by its first connector end (the transfer source) and its second (the transfer target).

Succession item flows are item flows that are also successions (see [7.2.8](#)), requiring time ordering between their transfer source, the transfer itself, and their transfer target. That is, the transfer happens in the time between the end of the source occurrence and the beginning of the target occurrence.

7.2.10.2 Interactions Concrete Syntax

7.2.10.2.1 Interactions Textual Notation

7.2.10.2.1.1 Interactions

SysML does not provide any concrete syntax for Interactions. The Interaction metaclass shall not be instantiated in a SysML model.

7.2.10.2.1.2 Item Flows

```
ItemFlow (m : Membership) : ItemFlow =
    ( isAbstract ?= 'abstract' )? 'stream'
    ItemFlowDeclaration(this, m) TypeBody(this)

SuccessionItemFlow (m : Membership) : SuccessionItemFlow =
    ( isAbstract ?= 'abstract' )? 'flow'
    ItemFlowDeclaration(this, m) TypeBody(this)

ItemFlowDeclaration (i : ItemFlow, m : Membership) :
    ( FeatureDeclaration(i, m)
      ( 'of' i.ownedRelationship += ItemFeatureMember
        | i.ownedRelationship += EmptyItemFeatureMember )
      'from'
      | i.ownedRelationship += EmptyItemFeatureMember
    )
    i.ownedRelationship += ItemFlowEndMember 'to'
    i.ownedRelationship += ItemFlowEndMember

ItemFeatureMember : FeatureMembership =
    ( memberName = NAME ':' )? ownedMemberFeature = ItemFeature

ItemFeature : Feature =
    ownedRelationship += OwnedFeatureTyping
    ( ownedRelationship += MultiplicityMember )?
    | ownedRelationship += MultiplicityMember
    ( ownedRelationship += OwnedFeatureTyping )?

EmptyItemFeatureMember : FeatureMembership =
    ownedMemberFeature = EmptyItemFeature

EmptyItemFeature : Feature =
    {}

ItemFlowEndMember : FeatureMembership =
    ownedMemberFeature = ItemFlowEnd

ItemFlowEnd : Feature =
    ownedRelationship += ItemFlowFeatureMember

ItemFlowFeatureMember : FeatureMembership =
    ownedMemberFeature = ItemFlowFeature

ItemFlowFeature : Feature =
    ownedRelationship += Redefinition
```

7.2.10.2.2 Interactions Graphical Notation

7.2.10.2.2.1 Interactions

SysML does not provide any concrete syntax for Interactions.

7.2.10.2.2.2 Item Flows

```
kind-keyword (i : ItemFlow) = 'stream'
kind-keyword (i : SuccessionItemFlow) = 'flow'
```

```
type-dcl-text (i : ItemFlow) =
  ItemFlowDeclaration (i, i.owningMembership)
```

```
item-feature-label (f : Feature) =
  element-name-text(f)? feature-typing-text(f)?
  feature-multiplicity-text(f)?
```

```
element-symbol (i : ItemFlow) =
```

```
      &element-symbol (i.sourceOutputFeature)
      feature-label(i)?
      item-feature-label(i.itemFeature)?
      &element-symbol (s.targetInputFeature)
```

Note: Both the feature and item feature labels may be placed on either side of the arrow. If both labels are included, then the kind keyword shall be included in the feature label.

7.2.10.3 Interactions Abstract Syntax

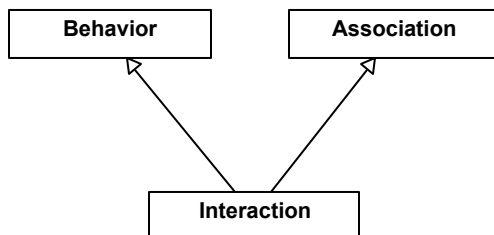


Figure 24. Interactions

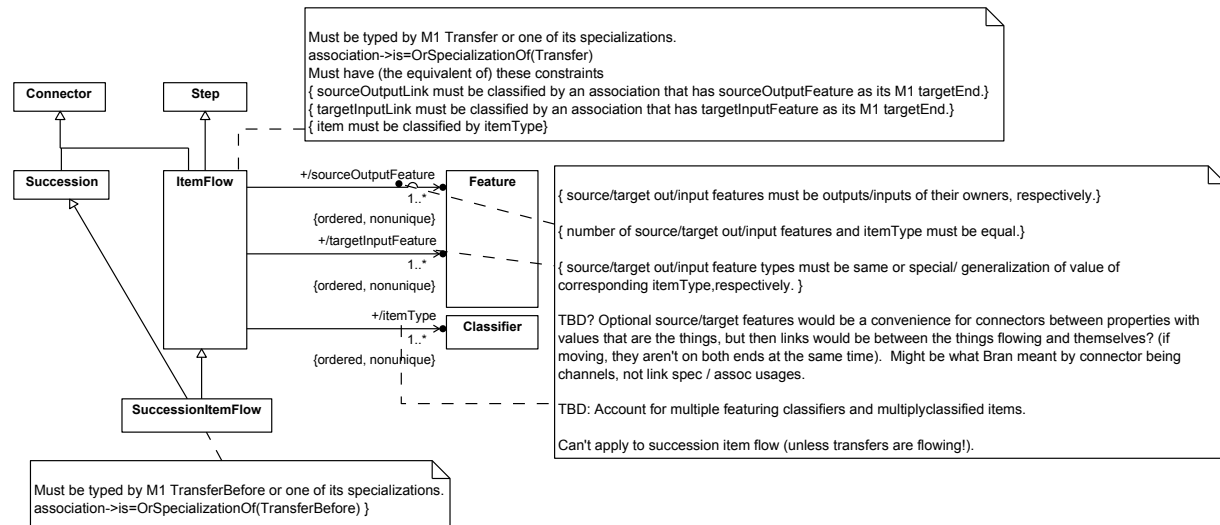


Figure 25. Item Flows

7.2.11 Functions

7.2.11.1 Functions Overview

Functions

Functions are behaviors that designate a single output parameter as their result using a special kind of parameter membership. Functions classify evaluations, which are kind of performances expected to produce values for their result. In general, functions can have output parameter other than their result, and function evaluations can also change involved objects. However, functions with no output parameters other than their result, and which do not change objects during their evaluation, essentially represent mathematical computations (for example, the numerical functions in the Kernel Model Library, see [KerML, 8.16]).

Expressions

Expressions are steps typed only by a single function. They can be steps in any behavior, including functions, in which case one such expression can be designated as specifying the result of the function using a special result-expression membership. The result parameter of a result expression is then bound to the result parameter of the containing function. Expressions can also have their own (nested) parameters, to augment or redefine those of their functions, including the result.

Expressions are commonly organized into tree structures in which the input parameters of each expression are connected by binding connectors to the result of each of its child expressions (its arguments). KerML textual syntax includes traditional operator notation for constructing such Expression trees, which is adopted directly into SysML (see 7.2.12).

Predicates

Predicates are Functions that whose result is a Boolean value. Predicates determine whether the values of their input parameters meet particular conditions at the time of evaluation, returning *true* if they do, and *false* otherwise.

Boolean Expressions and Invariants

Boolean expressions are expressions whose function is a predicate. As such, a Boolean expression must similarly have Boolean result. Invariants are Boolean expressions that must always evaluation to true—a Boolean expression

in general can evaluate to true or false, but an invariant must always evaluate to true in a valid model. (Boolean expressions should not be confused with literal Booleans, which are also expressions, but which are not typed by predicates and always evaluate to *true* or *false*—see 7.2.12).

7.2.11.2 Functions Concrete Syntax

SysML does not provide any concrete syntax for Functions other than CalculationDefinitions (see 7.16), ConstraintDefinitions (see 7.17). SysML adopts the KerML operator notation for Expressions (see 7.2.12) and provides SysML-specific syntax for CalculationUsages (see 7.16), ConstraintUsages (see 7.17), and their various specializations. The Function, Predicate, BooleanExpression and Invariant metaclasses shall not be instantiated in a SysML model.

7.2.11.3 Functions Abstract Syntax

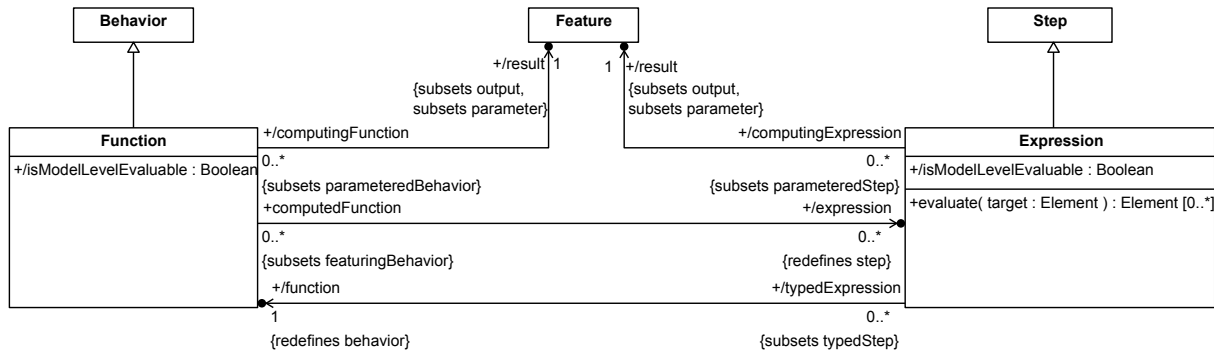


Figure 26. Functions

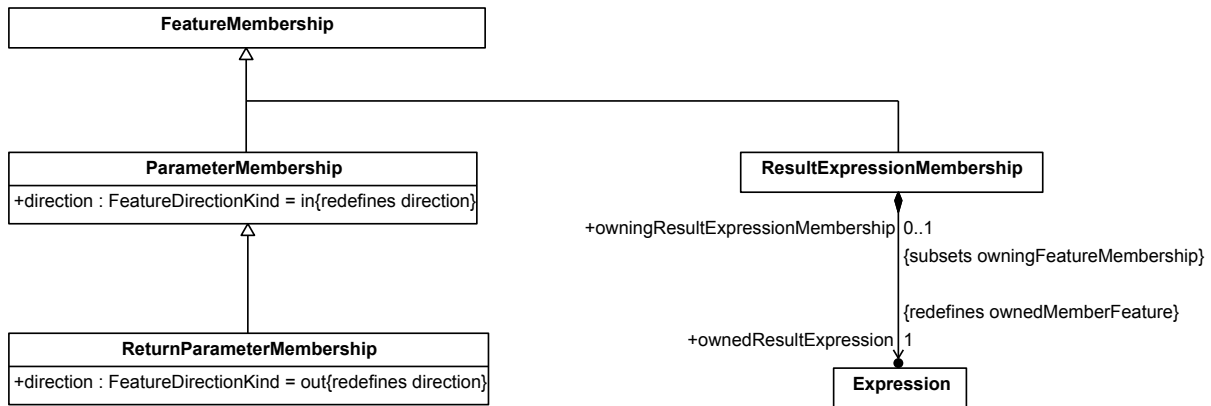


Figure 27. Function Memberships

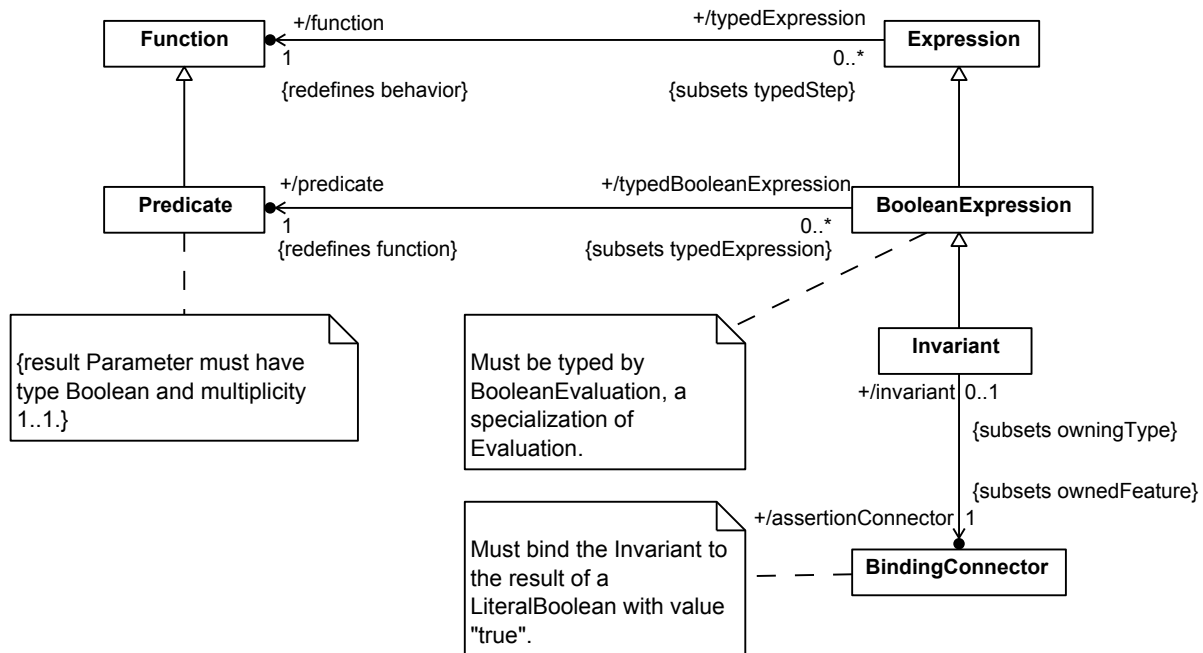


Figure 28. Predicates

7.2.12 Expressions

7.2.12.1 Expressions Overview

Expressions (see Functions) are commonly organized into tree structures to specify compound computations. KerML includes extensive textual syntax for constructing Expression trees, including traditional operator notations for Functions in the Kernel Model Library (see [KerML, Clause 8]), which is adopted in its entirety into SysML. However, these concrete syntax notations map entirely to an abstract syntax involving just a few specialized Expressions:

- The non-leaf nodes of an Expression tree are invocations of function,s with inputs specified as owned argument expressions, one for each of the input parameters of the 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 whose results are values of a reference feature.
 - Literal expressions that result in the literal value of one of the primitive data types from the *ScalarValues* model library (see [KerML, 8.11]).
 - Null expression that result in an empty set of values.

7.2.12.2 Expressions Concrete Syntax

```

OwnedExpressionMember : FeatureMembership =
    ownedFeatureMember = OwnedExpression

OwnedExpression : Expression =
    NullExpression
    | LiteralExpression
    | FeatureReferenceExpression
    | InvocationExpression
    | ExpressionBody
    | '(' OwnedExpression ')'

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

FeatureReferenceExpression : FeatureReferenceExpression =
    ownedRelationship += FeatureReferenceMember

FeatureReferenceMember : ReturnParameterMembership =
    ownedMemberParameter = FeatureReference

FeatureReference : Feature =
    ownedRelationship += Subset

InvocationExpression : InvocationExpression =
    ownedRelationship += OwnedFeatureTyping '(' ArgumentList(this)? ')'

ArgumentList (e : InvocationExpression) =
    PositionalArgumentList(e) | NamedArgumentList(e)

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

NamedArgumentList (e : InvocationExpression) =
    e.ownedRelationship += NamedExpressionMember
    ( ',' e.ownedRelationship += NamedExpressionMember )*

NamedExpressionMember : FeatureMembership =
    memberName = NAME '=>' ownedMemberFeature = OwnedExpression

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

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

LiteralUnbounded : LiteralUnbounded =
    '*'

ExpressionBody : Expression =
    CalculationBody(this)

```

The above grammar defines a basic functional notation for InvocationExpressions, FeatureReferenceExpressions, NullExpressions, LiteralExpressions and in-line Expressions bodies. KerML also provides a rich operator notation for InvocationExpressions (see [KerML, 7.4.8.2]). This notation may also be used in SysML anyplace an InvocationExpression is valid (except that Expression bodies are represented as SysML Calculation bodies, rather than KerML Function bodies).

There is no graphical notation for Expressions in SysML, other than that provided for CalculationUsages (see [7.16](#)), ConstraintUsages (see [7.17](#)), and their various specializations.

7.2.12.3 Expressions Abstract Syntax

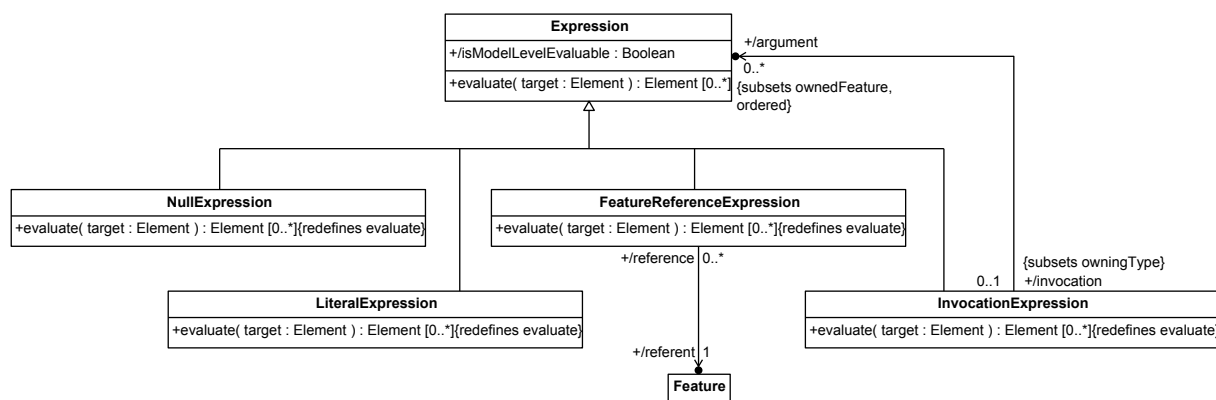


Figure 29. Expressions

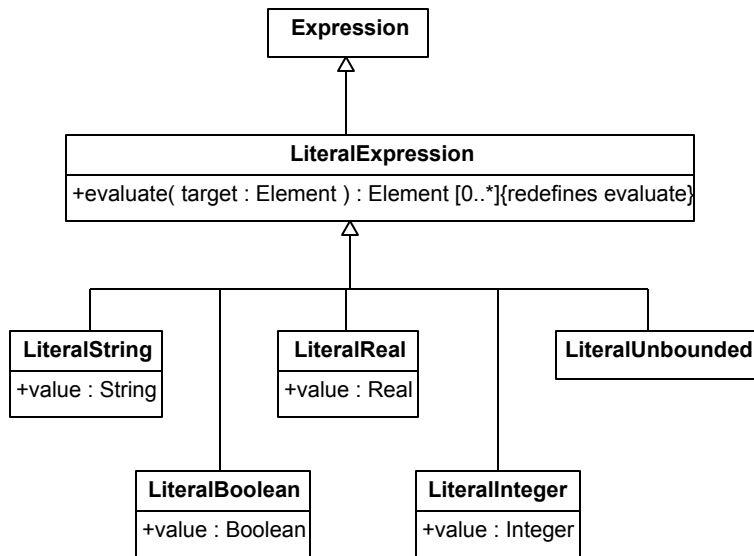


Figure 30. Literal Expressions

7.3 Dependencies

7.3.1 Dependencies Overview

A Dependency is a kind of Relationship between any two elements where the element on the client end depends on the element on the supplier end. This implies that a change to the element on the supplier end may result in a change to the element on the client end.

Dependencies can be useful for representing relationships between elements in an abstract way. For example, a dependency can be used to represent that an upper layer of an architecture stack may depend on a lower layer of the stack. Another example is using a dependency to represent a simplified cause-effect relationship that abstracts away much of the details underlying this relationship. The analysis of cross-model dependencies can support impact assessment and help identify potentially undesired circular dependencies.

7.3.2 Dependencies Concrete Syntax

7.3.2.1 Dependencies Textual Notation

```

Dependency (m : Membership) : Dependency =
    'dependency' DependencyDeclaration(this, m) ';'

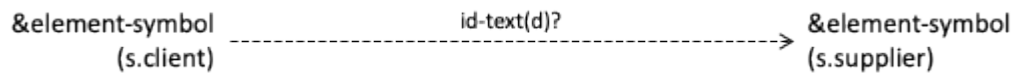
DependencyDeclaration (d : Declaration, m : Membership) =
    ( Identification(this, m) 'from' )?
    client += [QualifiedName] ( ',' client += [QualifiedName] )* 'to'
    supplier += [QualifiedName] ( ',' supplier += [QualifiedName] )*
  
```

7.3.2.2 Dependencies Graphical Notation

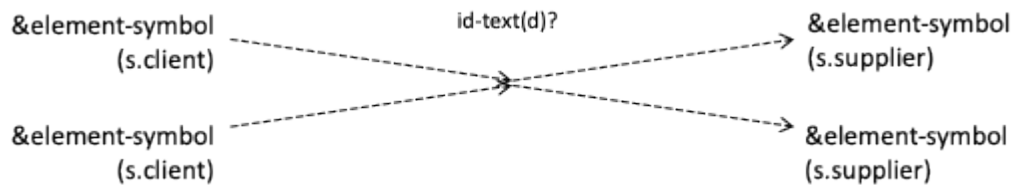
```
element-text (d : Dependency) =  
  visibility-keyword(d)? 'dependency'  
  DependencyDeclaration(d, d.owningMembership)
```

```
element-symbol (d : Dependency) =  
  dependency-binary-symbol(d)  
  | dependency-nary-symbol(d)  
  | dependency-nary-dot-symbol(d)
```

```
dependency-binary-symbol(d) =
```

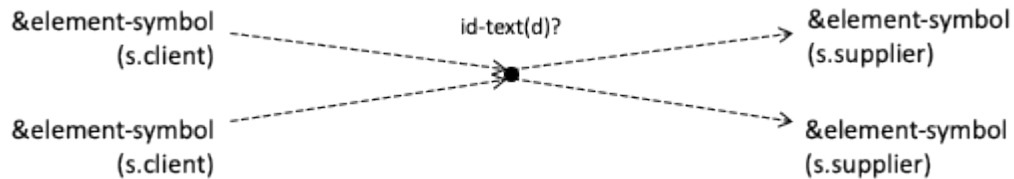


```
dependency-nary-symbol(d) =
```



Note: There may be one or more branches on either side.

```
dependency-nary-dot-symbol(d) =
```



Note: There may be one or more branches on either side.

7.3.3 Dependencies Abstract Syntax

7.3.3.1 Overview

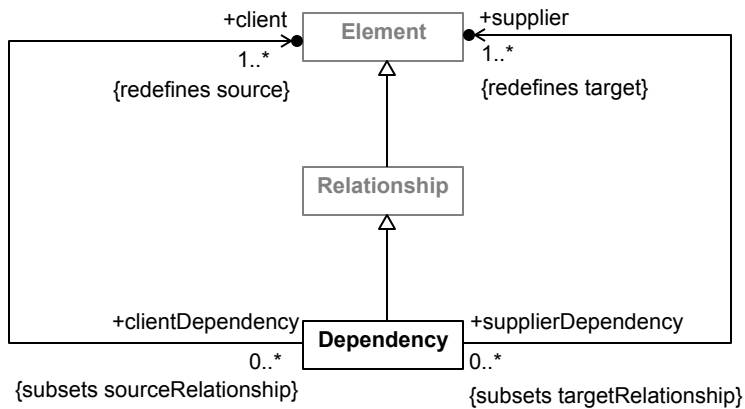


Figure 31. Dependencies

7.3.3.2 Dependency

Description

A Dependency is a Relationship that indicates that one or more `client` Elements require one more `supplier` Elements for their complete specification. In general, this means that a change to one of the `supplier` Elements may necessitate a change to, or re-specification of, the `client` Elements.

Note that a Dependency is entirely a model-level Relationship, without instance-level semantics.

General Classes

Relationship

Attributes

`client` : Element [1..*] {redefines source}

The Element or Elements dependent on the `supplier` elements.

`supplier` : Element [1..*] {redefines target}

The Element or Elements on which the `client` Elements depend in some respect.

Operations

No operations.

Constraints

No constraints.

7.4 Definition and Usage

7.4.1 Definition and Usage Overview

Definitions and Usages

The modeling capabilities of SysML facilitate reuse in different contexts. The Definition and Usage elements provide a consistent foundation for many SysML language constructs to provide this capability. This includes definition and usage elements for attributes, items, parts, ports, connections, interfaces, individuals, actions, states, calculations, constraints, requirements, analysis case, verification case, variants, views, and other specializations.

In general, a definition element is a Classifier that classifies a certain kind of element (e.g., a classification of attributes, parts, actions, etc.). A definition element may then have owned usage elements nested in it, referred to as its Features. A feature has a multiplicity that specifies the number of owned usage elements of this kind. For example, a Vehicle definition could include a usage element called wheels with multiplicity 4, meaning each Vehicle has exactly four wheels, or a less restrictive constraint, such as a multiplicity of 2..8, meaning each Vehicle can have 2 to 8 wheels.

A definition element can be specialized using the Generalization relationship. The specialized definition element (i.e., subclass) must also conform to the general definition element (i.e., superclass). A specialized definition element inherits the features from the more general definition element and can add other features. For example, if Vehicle has a feature called fuel, that is defined by Fuel, and Truck is a specialized kind of Vehicle, then Truck inherits the feature fuel that is defined by Fuel. An inherited feature can be over-ridden using redefinition or subsetting as described below. The *Truck* can also add its own features such as cargoSize.

A usage element is a usage of a definition element in a certain context. A usage element must always be defined by at least one definition element that corresponds to its usage kind. For example, a part usage is defined by a part definition, and an action usage is defined by an action definition. If no definition is specified explicitly, then the usage is defined implicitly by the most general definition element of the appropriate kind from the Systems Model Library (see [8.1](#)). For example, a part usage is implicitly defined by the most general part definition *Part* from the model library package Parts.

A usage element inherits the features from its definition element in the same way that a definition element can specialize a more general definition element. For example, if a part usage vehicle is defined by a part definition Vehicle, and Vehicle has a mass defined by MassValue, then vehicle inherits the feature mass defined by MassValue. In some cases, a usage element may have more than one definition element, in which case the usage element combines the features from each of its definition elements.

A usage element can add its own features, and redefine or subset its inherited features. This enables each usage element to be modified for its context. The redefined or subsetted feature must conform to the constraints of the inherited feature.

To redefine an inherited feature, the definition of the redefined feature must be a subclass of the definition of its inherited feature. For the example above, Vehicle contains a feature called fuel that is defined by Fuel. Truck inherits fuel from Vehicle. A part usage called truck that is defined by a part definition called Truck inherits fuel from Truck. The part usage truck can redefine its fuel to be defined by DieselFuel, which is a subclass of Fuel.

To subset an inherited feature, there can be one or more subsetted features subject to the multiplicity constraints of the inherited feature. For the example above, Truck inherits the feature wheels with multiplicity 2..8 from Vehicle. The part usage truck further inherits wheels with multiplicity 2..8 from Truck. The part usage truck can subset wheels by defining frontLeftWheel, frontRightWheel, rearLeftWheel1, rearLeftWheel2, rearRightWheel1, and rearRightWheel2 as subsets of wheels, as long as the number of wheels is between 2 and 8.

A usage element may be a feature of an owning definition element as described above. However, a usage element may also be contained directly in an owning package. In this case, the context for the usage element is the most

general kernel type Anything. That is, a package-level usage is essentially a generic feature that can be applied in any context. This usage enables specializations in specific contexts.

A usage element may also have nested usage elements. In this case, the context for the nested usages is the containing usage. A simple example is illustrated by a parts tree which is defined by a hierarchy of part usages.. A vehicle usage defined by Vehicle could contain part usages for engine, transmission, frontAxle, and rearAxle. Each part usage has its own part definition. A usage with nested usages, such as this vehicle parts tree, can be reused by subsetting the part usage vehicle. Subsetting the part usage vehicle is analogous to specializing the part definition Vehicle. For this example, assume vehicle1 is a part usage that subsets vehicle. This would enable vehicle1 to inherit the features and structure of vehicle. The part usage vehicle1 can be further modified by adding other part usages to it, such as a body and chassis. The part usage vehicle1 could also redefine the parts from vehicle as needed. For example, vehicle1 may redefine engine to be a 4-cylinder engine. The original part vehicle remains unchanged, but vehicle1 is now a unique design configuration. Other part usages such as vehicle2 could be created in a similar way to represent other design configurations.

It should be noted, that if the part definition Vehicle is modified, the modification will propagate down through the specializations described above. However, it is expected that if Vehicle is baselined in a configuration management tool, than a change to Vehicle is a new revision, and it is up to the modelers to determine whether to retain the previous version of Vehicle or move to the next revision.

A usage element may be a reference or composite feature of another usage or definition element. A reference usage element represents a simple reference from one usage to another. A composite usage element indicates that the usage at any point in time is integral to the structure of the containing definition or usage element. As such, when a containing definition or usage element is destroyed, then any of its contained usages are also destroyed. This is commonly known as *composite destruction semantics*.

As noted above, the definition and usage pattern applies broadly to many different kinds of elements, such as attributes, items, ports, actions, states, requirements, and others. This capability provides the ability to reuse models, while at the same time providing the flexibility to make further modifications for each context.

Variability

Variation and *variant* are used to model variability typically associated with a family of design configurations. A variation is commonly referred to as a variation point. A variation identifies an element in the model that can vary from one design configuration to another. One example of a variation is an engine in a vehicle. For each variation, there are design choices called variants. For this example, where the engine is designated as a point of variation, the design choices are a 4-cylinder engine variant or a 6-cylinder engine variant.

A variation can apply to any usage element in the model, such as a part, port, action, attribute, and requirement. The variation can also be applied to a definition element such as a part definition, port definition, action definition, attribute definition, and requirement definition. Variation can also be used to specify an optional choice where the variant is either included or not. An example may be the option to include a sunroof or not in the vehicle.

All possible variants (i.e., choices) must be enumerated for each variation point. For example, the enumerated variants for the engine variation are the 4-cylinder engine and the 6-cylinder engine. Variants can be logically combined to represent additional choices. For example, the enumerated variants can include A, B, and (A and B). Each of these choices are exclusive, meaning only one choice is valid.

A variant is a subset of the variation usage. For example, the 4-cylinder engine and the 6-cylinder engine are subsets of all possible engines. A variant can only apply to a usage element, and not to a definition element.

Variations can be nested within other variations. For example, the 6-cylinder engine may have choices for cylinder bore diameter. The engine is a variation with a 4-cylinder variant and a 6-cylinder variant. The 6-cylinder variant in turn contains a variation point for bore diameter that includes variants for small-bore diameter and large-bore

diameter. There is no restriction on the number of levels of nesting. The bore diameter variation point could also be applied more generally to the cylinder of engine, enabling both the 4-cylinder engine and the 6-cylinder engine to have this variation point.

A model with variability is a model that includes variation and variants. This variability model can be quite complex since the variation can extend to many other aspects of the model including its structure, behavior, requirements, analysis, and verification. Also, the selection of a particular variant often impacts many other design choices that include other parts, connections, actions, states, and attributes. Constraints can be used to constrain the available choices for a given variant. For example, the choice of a 6-cylinder engine may constrain the choice of transmission to be an automatic transmission, whereas the choice of a 4-cylinder engine may allow for both an automatic transmission or a manual transmission.

As noted above, variation and variants are used to construct a model that is sometimes referred to as a superset model which includes the variants to configure all possible design configurations. A particular configuration is selected by selecting a variant for each variation. SysML is intended to provide validation rules that can evaluate whether a particular configuration is a valid configuration based on the choices and constraints provided in the superset model. Variability modeling in SysML can augment other external variability modeling applications, which provide robust capabilities for managing variability across multiple kinds of models such as CAD, CAE, and analysis models, and auto-generating the variant design configurations based on the selections.

The approach to variability modeling in SysML is intended to align with industry standards such as ISO 26580 Feature-based Product Line Engineering.

7.4.2 Definition and Usage Concrete Syntax

7.4.2.1 Definition and Usage Textual Notation

7.4.2.1.1 Definitions

```

DefinitionPrefix (d : Definition) =
    d.isAbstract ?= 'abstract' | d.isVariation ?= 'variation'

Definition (d : Definition, m: Membership) =
    DefinitionDeclaration(d, m) DefinitionBody(d)

DefinitionDeclaration (d : Definition, m : Membership)
    Identification(d, m) SuperclassingPart(d)?

DefinitionBody (t : Type) =
    ';' | '{' DefinitionBodyItem(t)* '}'

DefinitionBodyItem (t : Type) =
    t.ownedRelationship += OwnedDocumentation
    | t.ownedRelationship += NestedDefinitionMember(t)
    | t.ownedRelationship += VariantUsageMember
    | t.ownedRelationship += NestedUsageMember
    | t.ownedRelationship += IndividualUsageMember
    | t.ownedRelationship += IndividualSuccessionMember(t)
    | t.ownedRelationship += PackageImport

NestedDefinitionMember (t : Type) : Membership =
    NonFeatureTypeMember(t)

VariantUsageMember : VariantMembership =
    DefinitionMemberPrefix(this) 'variant'
    ownedVariantUsage = VariantUsageElement(this)

NestedUsageMember : FeatureMembership =
    StructureUsageMember | BehaviorUsageMember | FlowUsageMember

StructureUsageMember : FeatureMembership =
    DefinitionMemberPrefix(this)
    ownedMemberFeature = StructureUsageElement(this)

BehaviorUsageMember : FeatureMembership =
    DefinitionMemberPrefix(this)
    ownedMemberFeature = BehaviorUsageElement(this)

IndividualUsageMember : FeatureMembership =
    DefinitionMemberPrefix(this)
    ownedFeatureMember = IndividualUsageElement(this)

IndividualSuccessionMember (t : Type) : FeatureMembership =
    s = SourceSuccessionMember(t)
    IndividualUsageMember
    TargetEndFor(s.ownedMemberFeature, ownedMemberFeature)

FlowUsageMember : FeatureMembership =
    DefinitionMemberPrefix(this)
    direction = FeatureDirection

```



```
ownedFeatureMember = FlowUsageElement(this)

DefinitionMemberPrefix (m : Membership) =
  TypeMemberPrefix(m)
```

7.4.2.1.2 Usages

```

UsagePrefix (u : Usage) =
    u.isAbstract ?= 'abstract' | u.isVariation ?= 'variation'

Usage (u : Usage, m : Membership) =
    UsageDeclaration(this, m) UsageCompletion(this)

UsageDeclaration (u : Usage, m : Membership) =
    FeatureDeclaration(u, m)

UsageCompletion (u : Usage) =
    ValueorFlowPart(this)? UsageBody(this)

UsageBody (u : Usage) =
    DefinitionBody(u)

ValueOrFlowPart (u : Usage) =
    ValuePart(u) | FlowPart(u)

FlowPart (f : Feature) =
    f.ownedRelationship += SourceItemFlowMember(f)

SourceItemFlowMember (f : Feature) : FeatureMembership =
    ownedMemberFeature = SourceItemFlow(f)

SourceItemFlow (f : Feature) : ItemFlow =
    ItemFlowTo(f) | SuccessionItemFlowTo(f)

ItemFlowTo (f : Feature) : ItemFlow =
    'stream' ownedRelationship += EmptyItemFeatureMember
    'from' ownedRelationship += ItemFlowEndMember
    ownedRelationship += ItemFlowEndMemberFor(f)

SuccessionItemFlowTo (f : Feature) : SuccessionItemFlow =
    'flow' ownedRelationship += EmptyItemFeatureMember
    'from' ownedRelationship += ItemFlowEndMember
    ownedRelationship += ItemFlowEndMemberFor(f)

ItemFlowEndMemberFor (f : Feature) : FeatureMembership =
    ownedMemberFeature = ItemFlowEndFor (f)

ItemFlowEndFor (f : Feature) : Feature =
    ownedRelationship += ItemFlowFeatureMemberFor(f)

ItemFlowFeatureMemberFor (f : Feature) : FeatureMembership =
    ownedMemberFeature = ItemFlowFeatureFor(f)

ItemFlowFeatureFor (f : Feature) : Feature =
    ownedRelationship += OwnedRedefinitionFor(f)

OwnedRedefinitionFor (f : Feature) : Redefinition =
    { redefinedFeature = f }

```

7.4.2.1.3 Reference Usages

```
ReferenceUsage (m : Membership) : ReferenceUsage =  
    UsagePrefix(this)? 'ref' Usage(this, m)  
  
ReferenceEndUsage (m : Membership) : ReferenceUsage =  
    UsagePrefix(this)? 'ref'? Usage(this, m)  
  
ReferenceVariantUsage (m : Membership) : ReferenceUsage =  
    ReferenceUsage  
    | ownedRelationship += OwnedSubsetting  
    FeatureSpecialization* UsageBody
```

7.4.2.1.4 Body Elements

```

StructureUsageElement (m : Membership) : Usage =
    ReferenceUsage (m)
  | AttributeUsage (m)
  | EnumerationUsage (m)
  | ItemRefUsage (m)
  | PartRefUsage (m)
  | StakeholderRefUsage (m)
  | ViewRefUsage (m)
  | RenderingRefUsage (m)
  | PortUsage (m)
  | ConnectionUsage (m)
  | Connector (m)
  | InterfaceUsage (m)
  | BindingConnector (m)
  | Succession (m)
  | ItemFlow (m)
  | SuccessionItemFlow (m)

IndividualUsageElement (m : Membership) : IndividualUsage =
    IndividualRefUsage (m)
  | TimeSliceRefUsage (m)
  | SnapshotRefUsage (m)

BehaviorUsageElement (m : Membership) : Usage =
    ActionRefUsage (m)
  | CalculationRefUsage (m)
  | StateRefUsage (m)
  | ConstraintRefUsage (m)
  | RequirementRefUsage (m)
  | ConcernRefUsage (m)
  | CaseRefUsage (m)
  | AnalysisCaseRefUsage (m)
  | VerificationCaseRefUsage (m)
  | ViewpointRefUsage (m)
  | PerformActionUsage (m)
  | ExhibitStateUsage (m)
  | AssertConstraintUsage (m)
  | SatisfyRequirementUsage (m)

VariantUsageElement (m) : Usage =
    ReferenceVariantUsage (m)
  | AttributeVariantUsage (m)
  | ItemRefUsage (m)
  | PartRefUsage (m)
  | StakeholderRefUsage (m)
  | PortUsage (m)
  | ConnectionUsage (m)
  | Connector (m)
  | InterfaceUsage (m)
  | IndividualUsageElement (m)
  | BehaviorUsageElement (m)

```

```

FlowUsageElement (m : Membership) : Usage :
    ReferenceUsage (m)
  | AttributeUsage (m)
  | ItemFlowUsage (m)
  | PartFlowUsage (m)
  | StakeholderFlowUsage (m)
  | ViewFlowUsage (m)
  | RenderingFlowUsage (m)
  | ActionFlowUsage (m)
  | CalculationFlowUsage (m)
  | StateFlowUsage (m)
  | ConstraintFlowUsage (m)
  | RequirementFlowUsage (m)
  | ConcernFlowUsage (m)
  | CaseFlowUsage (m)
  | AnalysisCaseFlowUsage (m)
  | VerificationCaseFlowUsage (m)
  | ViewpointFlowUsage (m)

```

7.4.2.2 Definition and Usage Graphical Notation

7.4.2.2.1 Definitions

```

variation-keyword (d : Definition) =
    'variation' (d.isVariation)

end-keyword (d : Definition) = none

type-decl-text (d : Definition) =
    DefinitionDeclaration(d, d.owningMembership)

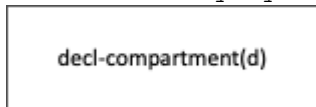
element-symbol (d : definition) =
    ( definition-no-body-symbol(d)
    | definition-with-body-symbol(d)
    ) boundary-feature-graphic(d.usage, d)*

```

```

definition-no-body-symbol (d : Definition) =

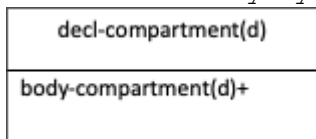
```



```

definition-with-body-symbol (d : Definition) =

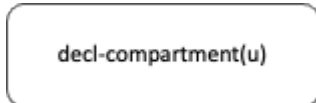
```



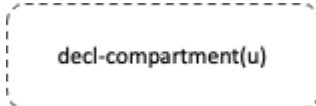
7.4.2.2.2 Usages

```
variation-keyword (u : Usage) =  
    'variation'(u.isVariation)  
  
end-keyword (u : Usage) =  
    'end' (u.isEnd)  
  
type-decl-text (u : Usage) =  
    UsageDeclaration(u, u.owningMembership)  
  
element-symbol (u : Usage) =  
    ( usage-no-body-symbol(u, u.isComposite)  
    | usage-with-body-symbol(u, u.isComposite)  
    ) boundary-feature-graphic(u.usage, u)*
```

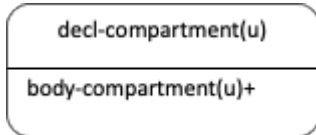
```
usage-no-body-symbol (u : Usage, true) =
```



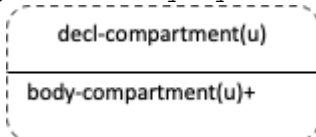
```
usage-no-body-symbol (u : Usage, false) =
```



```
usage-with-body-symbol (u : Usage, true) =
```



```
usage-with-body-symbol (u : Usage, false) =
```



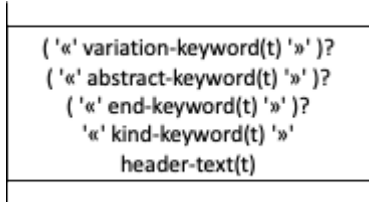
```
kind-keyword (u : ReferenceUsage) = 'ref'
```

```
ref-keyword (u : ReferenceUsage) = none
```

Note: A ReferenceUsage is never composite and a ref keyword would be redundant with its kind keyword.

7.4.2.2.3 Compartments

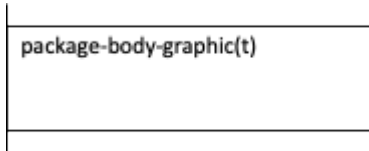
```
header-text (t : Type) =  
    visibility-keyword(t)? direction-keyword(t)? type-dcl-text(t)  
  
direction-keyword (d : Definition) = none  
direction-keyword (u : Usage) = direction-keyword(u.owningFeatureMembership)  
  
decl-compartment (t : Type) =
```



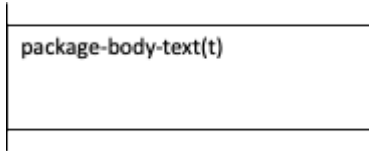
(Note: As an alternative to using the 'abstract' keyword, the header text can be rendered in italics.)

```
body-compartment (t : Type) =  
    graphic-body-compartment (t) | text-body-compartment (t) | ...
```

```
graphic-body-compartment (t : Type) =
```



```
text-body-compartment (t : Type) =
```



```
compartment-element-text (t : Type) =  
    visibility-keyword(t) variation-keyword(t)? abstract-keyword(t)?  
    end-keyword(t)? ref-keyword(t)? type-decl-text(t)  
    element-graphic(t.ownedAnnotation) *
```

```
keyword-element-text (t : Type, k : String) =  
    visibility-keyword(t) variation-keyword(t)? abstract-keyword(t)?  
    KEYWORD(k) ref-keyword(t)? type-decl-text(t)  
    element-graphic(t.ownedAnnotation) *
```

Notes

- Additional kinds of body compartments are given in subsequent subclauses.
- *Compartment element text* does not include the kind keyword. It is used in compartments in which the element kind is already determined by the compartment kind.

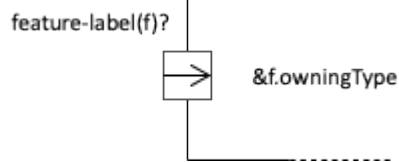
- *Keyword element text* is used in compartments for element declarations that require an extra keyword to be inserted.

7.4.2.2.4 Boundary Features

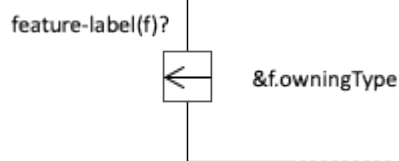
```
boundary-feature-graphic (f : Feature, t : Type) =
  boundary-feature-symbol(f, f.directionFor(t))
  boundary-feature-graphic(f.usage, f)
  element-graphic(f.ownedAnnotation) *
```

```
boundary-feature-symbol (f : Feature, null) = none
```

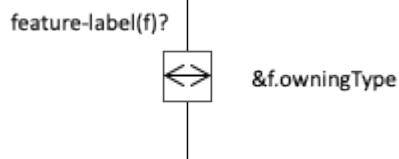
```
boundary-feature-symbol (f : Feature, FeatureDirectionKind::in) =
```



```
boundary-feature-symbol (f : Feature, FeatureDirectionKind::out) =
```



```
boundary-feature-symbol (f : Feature, FeatureDirectionKind::inout) =
```



Note: Boundary Feature symbols can appear anywhere on the boundary of the owningType symbol. The in and out arrows are relative to the inside and outside of the owningType symbol.

Any flowFeature (i.e., a Feature with a non-null direction) of a Definition or Usage can be shown using a *boundary feature symbol*. By default, Features without direction cannot be shown using such symbols, though Ports can be (see [7.9.2.2](#)).

Note that internal features may not be shown on a boundary feature symbols, but it may have nested boundary features. The outer boundary feature symbol can be resized as necessary to accommodate this.

7.4.2.2.5 Variant Memberships

```

package-membership-graphic (m : VariantMembership) =
    &element-symbol (m.membershipOwningPackage) ⊕ visibility-text(m.visibility)? '«variant»' &element-symbol (m.ownedVariantUsage)

```

7.4.3 Definition and Usage Abstract Syntax

7.4.3.1 Overview

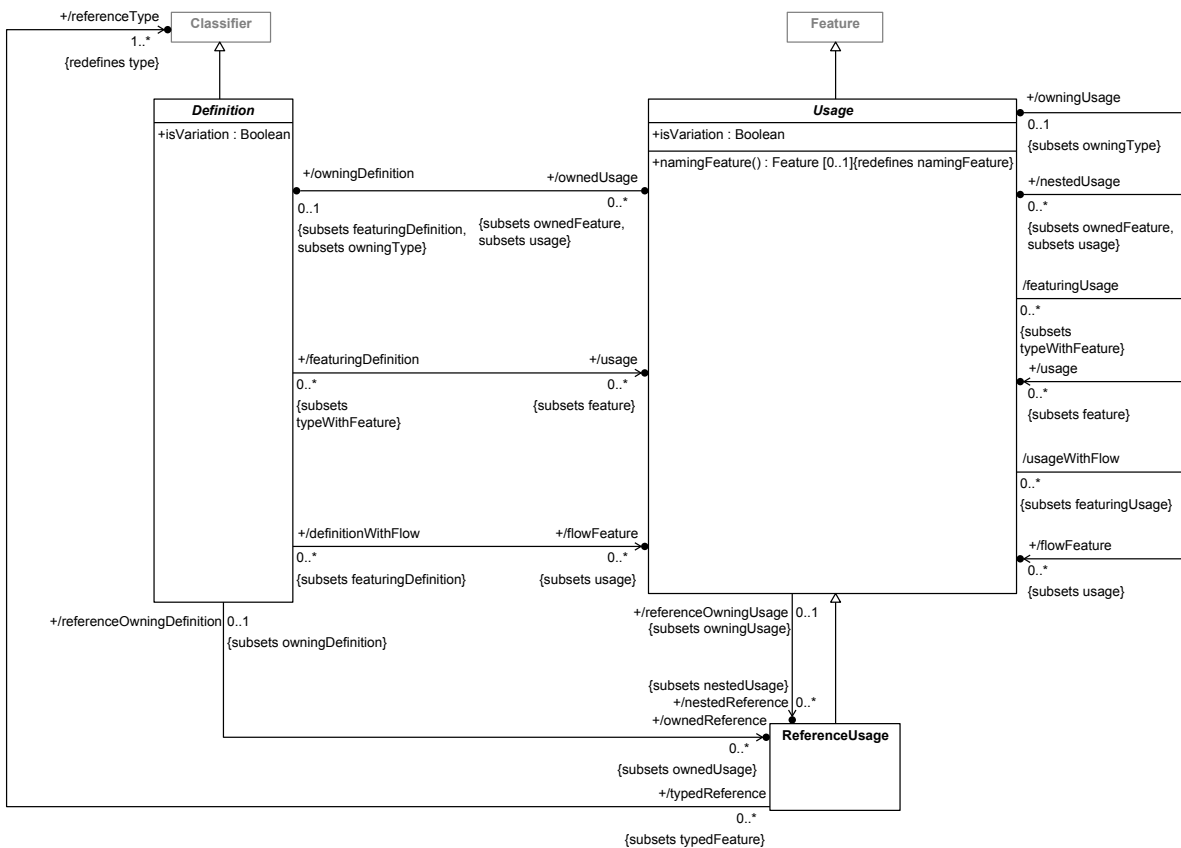


Figure 32. Definition and Usage

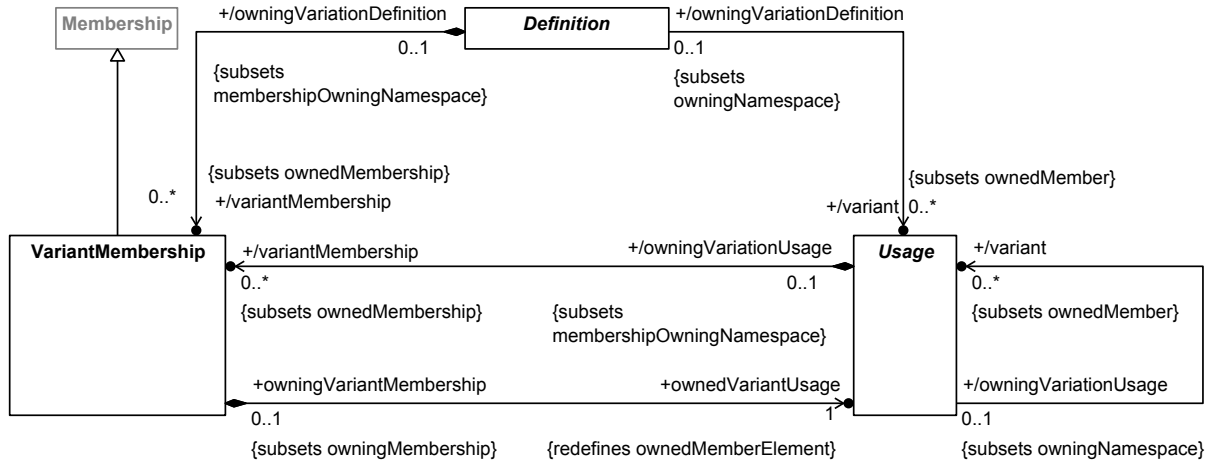


Figure 33. Variant Membership

7.4.3.2 Definition

Description

A Definition is a Classifier of Usages. The actual kinds of Definitions that may appear in a model are given by the concrete subclasses of Definition.

Normally, a Definition has owned Usages that model *features* of the thing being defined. A Definition may also have other Definitions nested in it, but this has no semantic significance, other than the nested scoping resulting from the Definition being considered as a Namespace for any nested Definitions.

However, if a Definition has `isVariation = true`, then it represents a *variation point* Definition. In this case, all of its members must be *variant* Usages, related to the Definition by VariantMembership Relationships. Rather than being *features* of the Definition, *variant* Usages model different concrete alternatives that can be chosen to fill in for an abstract Usage of the variation point Definition.

General Classes

Classifier

Attributes

/flowFeature : Usage [0..*] {subsets usage}

The usages of this Definition that have a non-null direction.

isVariation : Boolean

Whether this Definition is for a variation point or not. If true, then all the memberships of the Definition must be VariantMemberships.

/ownedAction : ActionUsage [0..*] {subsets ownedUsage}

The ActionUsages that are ownedUsages of this Definition.

/ownedAllocation : AllocationUsage [0..*] {subsets ownedConnection}

The AllocationUsages that are ownedUsages of this Definition.

/ownedAnalysisCase : AnalysisCaseUsage [0..*] {subsets ownedCase}

The AnalysisCaseUsages that are ownedUsages of this Definition.

/ownedAttribute : AttributeUsage [0..*] {subsets ownedUsage}

The AttributeUsages that are ownedUsages of this Definition.

/ownedCalculation : CalculationUsage [0..*] {subsets ownedAction}

The CalculationUsages that are ownedUsages of this Definition.

/ownedCase : CaseUsage [0..*] {subsets ownedCalculation}

The CaseUsages that are ownedUsages of this Definition.

/ownedConcern : ConcernUsage [0..*] {subsets ownedRequirement}

The ConcernUsages that are ownedUsages of this Definition.

/ownedConnection : ConnectionUsage [0..*] {subsets ownedPart}

The ConnectionUsages that are ownedUsages of this Definition.

/ownedConstraint : ConstraintUsage [0..*] {subsets ownedUsage}

The ConstraintUsages that are ownedUsages of this Definition.

/ownedEnumeration : EnumerationUsage [0..*] {subsets ownedAttribute}

The EnumerationUsages that are ownedUsages of this Definition.

/ownedIndividual : IndividualUsage [0..*] {subsets ownedItem}

The IndividualUsages that are ownedUsages of this Definition.

/ownedInterface : InterfaceUsage [0..*] {subsets ownedConnection}

The InterfaceUsages that are ownedUsages of this Definition.

/ownedItem : ItemUsage [0..*] {subsets ownedUsage}

The ItemUsages that are ownedUsages of this Definition.

/ownedPart : PartUsage [0..*] {subsets ownedItem}

The PartUsages that are ownedUsages of this Definition.

/ownedPort : PortUsage [0..*] {subsets ownedUsage}

The PortUsages that are ownedUsages of this Definition.

/ownedReference : ReferenceUsage [0..*] {subsets ownedUsage}

The ReferenceUsages that are ownedUsages of this Definition.

/ownedRendering : RenderingUsage [0..*] {subsets ownedPart}

The usages of this Definition that are RenderingUsages.

/ownedRequirement : RequirementUsage [0..*] {subsets ownedConstraint}

The RequirementUsages that are ownedUsages of this Definition.

/ownedStakeholder : StakeholderUsage [0..*] {subsets ownedPart}

The StakeholderUsages that are ownedUsages of this Definition.

/ownedState : StateUsage [0..*] {subsets ownedUsage}

The StateUsages that are ownedUsages of this Definition.

/ownedTransition : TransitionUsage [0..*] {subsets ownedUsage}

The TransitionUsages that are ownedUsages of this Definition.

/ownedUsage : Usage [0..*] {subsets ownedFeature, usage}

The Usages that are ownedFeatures of this Definition.

/ownedVerificationCase : VerificationCaseUsage [0..*] {subsets ownedCase}

The ownedUsages of this Definition that are VerificationCaseUsages.

/ownedView : ViewUsage [0..*] {subsets ownedPart}

The ownedUsages of this Definition that are ViewUsages.

/ownedViewpoint : ViewpointUsage [0..*] {subsets ownedRequirement}

The ownedUsages of this Definition that are ViewpointUsages.

/usage : Usage [0..*] {subsets feature}

The Usages that are features of this Definition (not necessarily owned).

/variant : Usage [0..*] {subsets ownedMember}

The Usages which represent the variants of this Definition as a variation point Definition, if isVariation = true. If isVariation = false, then there must be no variants.

/variantMembership : VariantMembership [0..*] {subsets ownedMembership}

The ownedMemberships of this Definition that are VariantMemberships. If isVariation = true, then this must be all ownedMemberships of the Definition. If isVariation = false, then variantMembership must be empty.

Operations

No operations.

Constraints

definitionIsVariationMembership

[no documentation]

isVariation implies variantMembership = ownedMembership

definitionNonVariationMembership

[no documentation]

not isVariation implies variantMembership->isEmpty()

definitionVariant

[no documentation]

variant = variantMembership.ownedVariantUsage

definitionVariantMembership

[no documentation]

variantMembership = ownedMembership->selectByKind(VariantMembership)

7.4.3.3 ReferenceUsage

Description

A ReferenceUsage is a Usage that specifies a non-compositional (`isComposite = false`) reference to something. The type of a ReferenceUsage can be any kind of Classifier, with the default being the top-level Classifier Anything from the Kernel library. This allows the specification of a generic reference without distinguishing if the thing referenced is an attribute value, item, action, etc. All features of a ReferenceUsage must also have `isComposite = false`.

General Classes

Usage

Attributes

/referenceType : Classifier [1..*] {redefines type}

The types of this ReferenceUsage, which must all be Classifiers.

Operations

No operations.

Constraints

No constraints.

7.4.3.4 Usage

Description

A Usage is a usage of a Definition. A Usage may only be an `ownedFeature` of a Definition or another Usage.

A Usage may have `nestedUsages` that model features that apply in the context of the `owningUsage`. A Usage may also have Definitions nested in it, but this has no semantic significance, other than the nested scoping resulting from the Usage being considered as a Namespace for any nested Definitions.

However, if a Usage has `isVariation = true`, then it represents a *variation point* Usage. In this case, all of its members must be `variant` Usages, related to the Usage by `VariantMembership` Relationships. Rather than being features of the Usage, `variant` Usages model different concrete alternatives that can be chosen to fill in for the variation point Usage.

General Classes

Feature

Attributes

`/flowFeature : Usage [0..*] {subsets usage}`

The usages of this Usage that have a non-null `direction`.

`isVariation : Boolean`

Whether this Usage is for a variation point or not. If true, then all the `memberships` of the Usage must be `VariantMemberships`.

`/nestedAction : ActionUsage [0..*] {subsets nestedUsage}`

The `ActionUsages` that are `ownedUsages` of this Usage.

`/nestedAllocation : AllocationUsage [0..*] {subsets nestedConnection}`

The `AllocationUsages` that are `nestedUsages` of this Usage.

`/nestedAnalysisCase : AnalysisCaseUsage [0..*] {subsets nestedCase}`

The `AnalysisCaseUsages` that are `ownedUsages` of this Usage.

`/nestedAttribute : AttributeUsage [0..*] {subsets nestedUsage}`

The `AttributeUsages` that are `ownedUsages` of this Usage.

`/nestedCalculation : CalculationUsage [0..*] {subsets nestedAction}`

The `CalculationUsage` that are `ownedUsages` of this Usage.

`/nestedCase : CaseUsage [0..*] {subsets nestedCalculation}`

The CaseUsages that are ownedUsages of this Usage.

/nestedConcern : ConcernUsage [0..*] {subsets nestedRequirement}

The ConcernUsages that are ownedUsages of this Usage.

/nestedConnection : ConnectionUsage [0..*] {subsets nestedPart}

The ConnectionUsages that are nestedUsages of this Usage.

/nestedConstraint : ConstraintUsage [0..*] {subsets nestedUsage}

The ConstraintUsages that are ownedUsages of this Usage.

/nestedEnumeration : EnumerationUsage [0..*] {subsets nestedAttribute}

The EnumerationUsages that are nestedUsages of this Usage.

/nestedIndividual : IndividualUsage [0..*] {subsets nestedItem}

nestedUsages of this Usage.

/nestedInterface : InterfaceUsage [0..*] {subsets nestedConnection}

The InterfaceUsages that are ownedUsages of this Usage.

/nestedItem : ItemUsage [0..*] {subsets nestedUsage}

The ItemUsages that are nestedUsages of this Usage.

/nestedPart : PartUsage [0..*] {subsets nestedItem}

The PartUsages that are nestedUsages of this Usage.

/nestedPort : PortUsage [0..*] {subsets nestedUsage}

The PortUsages that are ownedUsages of this Usage.

/nestedReference : ReferenceUsage [0..*] {subsets nestedUsage}

/nestedRendering : RenderingUsage [0..*] {subsets nestedPart}

The nestedUsages

of this Usage that are RenderingUsages.

/nestedRequirement : RequirementUsage [0..*] {subsets nestedConstraint}

The RequirementUsages that are ownedUsages of this Usage.

/nestedStakeholder : StakeholderUsage [0..*] {subsets nestedPart}

The StakeholderUsages that are nestedUsages of this Usage.

/nestedState : StateUsage [0..*] {subsets nestedAction}

The StateUsages that are ownedUsages of this Usage.

/nestedTransition : TransitionUsage [0..*] {subsets nestedUsage}

The TransitionUsages that are ownedUsages of this Usage.

/nestedUsage : Usage [0..*] {subsets ownedFeature, usage}

The Usages that are ownedFeatures of this Usage.

/nestedVerificationCase : VerificationCaseUsage [0..*] {subsets nestedCase}

The nestedUsages of this Usage that are VerificationCaseUsages

/nestedView : ViewUsage [0..*] {subsets nestedPart}

The nestedUsages of this Usage that are ViewUsages.

/nestedViewpoint : ViewpointUsage [0..*] {subsets nestedRequirement}

The nestedUsages

of this Usage that are ViewpointUsages.

/owningDefinition : Definition [0..1] {subsets owningType, featuringDefinition}

The Definition that owns this Usage (if any).

/owningUsage : Usage [0..1] {subsets owningType}

The Usage in which this Usage is nested (if any).

/usage : Usage [0..*] {subsets feature}

The Usages that are features of this Usage (not necessarily owned).

/variant : Usage [0..*] {subsets ownedMember}

The Usages which represent the variants of this Usage as a variation point Usage, if isVariation = true. If isVariation = false, then there must be no variants.

/variantMembership : VariantMembership [0..*] {subsets ownedMembership}

The ownedMemberships of this Usage that are VariantMemberships. If isVariation = true, then this must be all memberships of the Usages. If isVariation = false, then variantMembership must be empty.

Operations

namingFeature() : Feature [0..1]

If this Usage is a variant, then its naming Feature is its first subsetted Feature (unless that Feature is the owner of the usage).

```

body: if not owningMembership.ocIsKindOf(VariantMembership) then
    self.ocAsType(Feature).namingFeature()
else
    let namingFeature : Feature = firstSubsettedFeature() in
    if namingFeature = owner then null
    else namingFeature
endif

```

Constraints

usageVariantMembership

[no documentation]

```
variantMembership = ownedMembership->selectByKind(VariantMembership)
```

usageIsVariationMembership

[no documentation]

```
isVariation implies variantMembership = ownedMembership
```

usageNonVariationMembership

[no documentation]

```
not isVariation implies variantMembership->isEmpty()
```

usageVariant

[no documentation]

```
variant = variantMembership.ownedVariantUsage
```

7.4.3.5 VariantMembership

Description

A VariantMembership is a Membership between a variation point Definition or Usage and a Usage that represents a variant in the context of that variation. The membershipOwningNamespace for the VariantMembership must be either a Definition or a Usage with isVariation = true.

General Classes

Membership

Attributes

```
ownedVariantUsage : Usage {redefines ownedMemberElement}
```

The Usage that represents a variant in the context of the owningVariationDefinition or owningVariationUsage.

Operations

No operations.

Constraints

No constraints.

7.4.4 Definition and Usage Semantics

7.5 Attributes

7.5.1 Attributes Overview

An `AttributeDefinition` is a `DataType` (see [7.2.5](#)) that defines a set of data values, such as numbers, quantitative values with units, qualitative values such as text strings, or a composite structure of such values. An `AttributeUsage` is a kind of `Usage` element. An attribute usage is a usage of an attribute definition.

The data values of an attribute usage are constrained to be in the range specified by its definition. The range of data values for an attribute definition can be further restricted using constraints (see [7.17](#)). An enumeration definition is a specialized kind of attribute definition that further restricts the values of the data type to a discrete set of data values (see [7.6](#)).

Attribute usages can be defined by KerML data types as well as SysML attribute definitions. This allows them to be typed by primitive data types from the Kernel Model Library, such as `String`, `Boolean`, and numeric types including `Integer`, `Rational`, `Real` and `Complex`. The Kernel Model Library also includes basic structured data types for collections and non-scalar values, such as vectors and tensors.

Attribute usages representing quantities with units are defined using the SysML Quantities and Units Model Library or extensions of the elements in this library (see [8.2](#)). The `QuantityValue` attribute definition in the library provides the base for all such quantities, along with other models that specify the full set of international standard quantity kinds and units. Fundamental to this approach is the principle that only the kind of unit (e.g., `MassUnit`, `LengthUnit`, `TimeUnit`, etc.) is associated with an attribute definition, while a specific unit (e.g., kilogram, meter, second, etc.) is only given with an actual quantity value. This means that an attribute usage for a quantity value is independent of the specific units used, allowing for automatic conversion and interoperability between different units of the same kind (e.g., kilograms and pounds mass, meters and feet, etc.).

Attributes definition and usages can specify state variables that can vary over time (see [7.13](#)).

7.5.2 Attributes Concrete Syntax

7.5.2.1 Attributes Textual Notation

```
AttributeDefinition (m : Membership) : AttributeDefinition =  
    DefinitionPrefix(this)? ( 'attribute' 'def' | 'value' 'type' )  
    DefinitionDeclaration(this, m) DefinitionBody(this)  
  
AttributeUsage (m : Membership) : AttributeUsage =  
    UsagePrefix(this)? ( 'attribute' | 'value' )?  
    Usage(this, m)  
  
AttributeVariantUsage (m : Membership) : AttributeUsage =  
    UsagePrefix(this)? ( 'attribute' | 'value' )  
    Usage(this, m)
```

7.5.2.2 Attributes Graphical Notation

```

kind-keyword (a : AttributeDefinition) = 'attribute' 'def' | 'value' 'type'
kind-keyword (a : AttributeUsage) = 'attribute' | 'value'

body-compartment (t : Type) = ... | attributes-compartment(t)

attributes-compartment (t : Type) =
    |
    | 'attributes' | 'values'
    | attributes-compartment-text(t)
    |
    |

attributes-compartment-text (d : Definition) =
    compartment-element-text(d.ownedAttribute) *

attributes-compartment-text (u : Usage) =
    compartment-element-text(u.nestedAttribute) *

```

7.5.3 Attributes Abstract Syntax

7.5.3.1 Overview

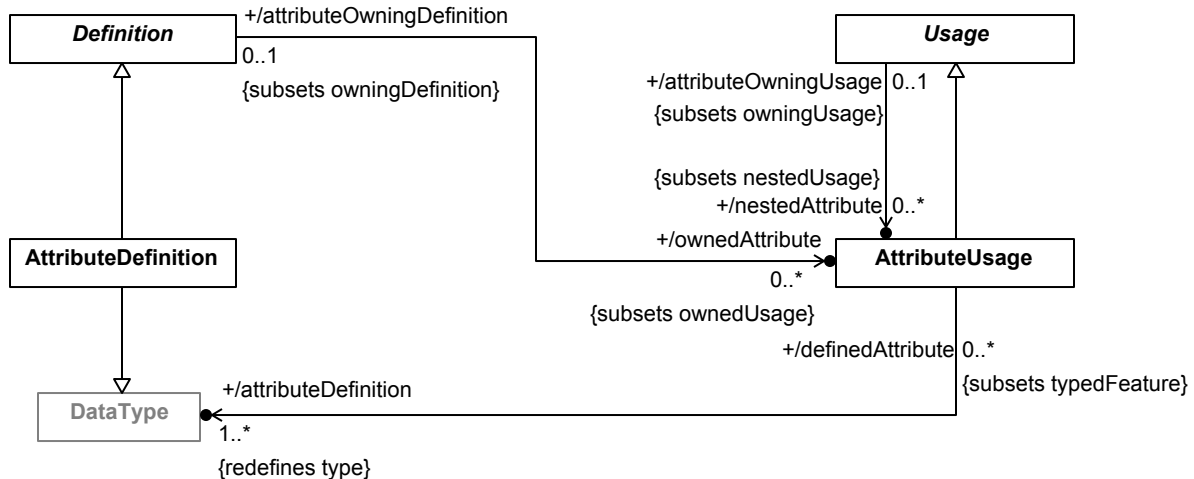


Figure 34. Attribute Definition and Usage

7.5.3.2 AttributeUsage

Description

An AttributeUsage is a Usage whose type is a DataType. Nominally, if the type is an AttributeDefinition, an AttributeUsage is a usage of a AttributeDefinition to represent the value of some system quality or characteristic. However, other kinds of kernel DataTypes are also allowed, to permit use of DataTypes from the Kernel Library. An AttributeUsage itself as well as all its nested features must have `isComposite = false`.

An `AttributeUsage` must subset, directly or indirectly, the base `AttributeUsage` `attributeValues` from the Systems model library.

General Classes

Usage

Attributes

/attributeDefinition : `DataType` [1..*] {redefines type}

The `DataTypes` that are the types of this `AttributeUsage`. Nominally, these are `AttributeDefinitions`, but other kinds of kernel `DataTypes` are also allowed, to permit use of `DataTypes` from the Kernel Library.

Operations

No operations.

Constraints

No constraints.

7.5.3.3 AttributeDefinition

Description

An `AttributeDefinition` is a `Definition` and a `DataType` of information about a quality or characteristic of a system or part of a system that has no independent identity other than its value. All *features* of an `AttributeDefinition` must have `isComposite = false`.

An `AttributeDefinition` must subclass, directly or indirectly, the base `AttributeDefinition` `AttributeValue` from the Systems model library.

General Classes

Definition
DataType

Attributes

No attributes.

Constraints

No constraints.

7.5.4 Attributes Semantics

7.6 Enumerations

7.6.1 Enumerations Overview

An `EnumerationDefinition` is a kind of `AttributeDefinition` ([8.1.2](#)) whose instances are limited to specific set of *enumerated values*. An `EnumerationUsage` is an `AttributeUsage` that is required to have a single definition that is an `EnumerationDefinition`.

An enumeration usage is restricted to only the set of enumerated values specified in its definition. Since an enumeration definition is a kind of attribute definition, it can also be used to define a regular attribute usage. Even if the attribute usage is not syntactically an enumeration usage, it is still semantically restricted to take on only the values allowed by its enumeration definition.

An enumeration definition can specialize an attribute definition that is not itself an enumeration definition. In this case, the enumerated values of the enumeration definition will be a subset of the attribute values of the specialized attribute definition. Which enumerated values correspond to which attribute values may be specified by binding the enumerated values to expressions that evaluate to the desired values of the specialized attributed definition. In this case, the results of all the expressions shall be distinct (typically they will just be literals).

An enumeration definition may not contain anything other than the declaration of its enumerated values. However, if the enumeration definition specializes an attribute definition with nested usages, then those nested usages will be inherited by the enumeration definition, and they may be bound to specific values within each enumerated value of the enumeration definition.

An enumeration definition may not specialize another enumeration definition. This is because the semantics of generalization require that the set instances classified by a definition be a subset of the instances of classified by any definition it specializes. The enumerated values defined in an enumeration definition, however, would *add* to the set of enumerated values allowed by any enumeration definition it specialized, which is inconsistent with the semantics of generalization.

Release Note. It is expected that the restriction of enumerations to just attribute definitions will be removed in a future release.

7.6.2 Enumerations Concrete Syntax

7.6.2.1 Enumerations Textual Notation

```
EnumerationDefinition (m : Membership) : EnumerationDefinition =  
    'enum' 'def' DefinitionDeclaration(this, m) EnumerationBody(this)  
  
EnumerationBody (e : EnumerationDefinition) =  
    ';' |  
    | '{' ( e.ownedRelationship += EnumerationUsageMember ) * '}'  
  
EnumerationUsageMember : VariantMembership =  
    DefinitionMemberPrefix(this) ownedVariantUsage = EnumeratedValue(this)  
  
EnumeratedValue (m : Membership) : EnumerationUsage =  
    'enum'? Usage(m)  
  
EnumerationUsage (m : Membership) : EnumerationUsage =  
    'enum' Usage(m)
```

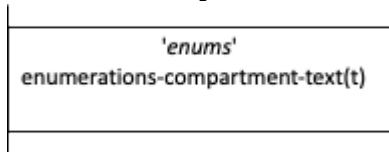
7.6.2.2 Enumerations Graphical Notation

```
kind-keyword (e : EnumerationDefinition) = 'enum' 'def'
```

```
kind-keyword (a : EnumerationUsage) = 'enum'
```

```
body-compartment (t : Type) = ... | enumerations-compartment(t)
```

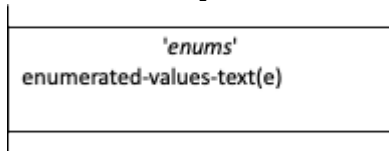
```
enumerations-compartment (t : Type) =
```



```
enumerations-compartment-text (d : Definition) =  
    enumeration-element-text(d.ownedEnumeration)*
```

```
enumerations-compartment-text (u : Usage) =  
    compartment-element-text(u.nestedEnumeration)*
```

```
enumerations-compartment (e : EnumerationDefinition) =
```



```
enumerated-values-text (e : EnumerationDefinition) =  
    compartment-element-text(e.enumeratedValue)
```

7.6.3 Enumerations Abstract Syntax

7.6.3.1 Overview

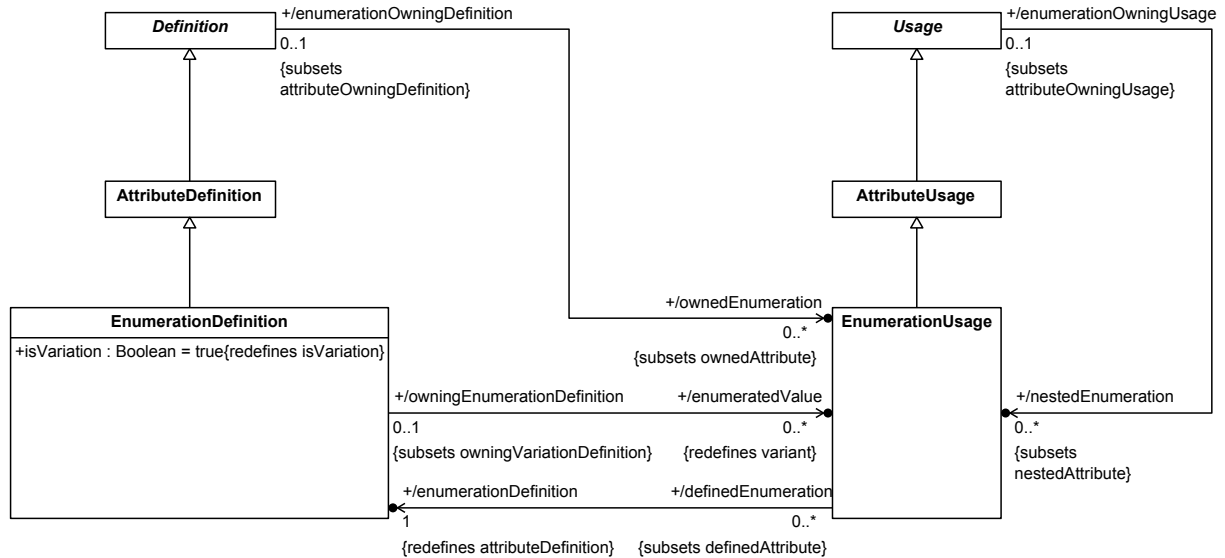


Figure 35. Enumeration Definition and Usage

7.6.3.2 EnumerationDefinition

Description

An EnumerationDefinition is an AttributeDefinition all of whose instances are given by an explicit list of enumeratedValues.

An EnumerationDefinition must subclass, directly or indirectly, the base EnumerationDefinition EnumerationValue from the Systems model library.

General Classes

AttributeDefinition

Attributes

/enumeratedValue : EnumerationUsage [0..*] {redefines variant}

A EnumerationUsage of this EnumerationDefinition with a fixed value, distinct from the value of all other enumerationValues, which specifies one of the allowed instances of the EnumerationDefinition.

isVariation : Boolean {redefines isVariation}

An EnumerationDefinition is considered semantically to be a variation whose allowed variants are its enumerationValues.

Operations

No operations.

Constraints

No constraints.

7.6.3.3 EnumerationUsage

Description

An EnumerationUsage is an AttributeUsage whose `attributeDefinition` is an EnumerationDefinition.

An EnumerationUsage must subset, directly or indirectly, the base EnumerationUsage `enumerationValues` from the Systems model library.

General Classes

AttributeUsage

Attributes

`/enumerationDefinition : EnumerationDefinition {redefines attributeDefinition}`

The single EnumerationDefinition that is the type of this EnumerationUsage.

Operations

No operations.

Constraints

No constraints.

7.6.4 Enumerations Semantics

7.7 Items

7.7.1 Items Overview

An ItemDefinition is a DefinitionElement that defines a class of things that may be acted on, but which do not necessarily perform actions. An ItemUsage is a kind of Usage element.

An item usage is a usage of one more item definitions. Item usages are used to represent inputs and outputs to actions such as water, fuel or electrical signals, Item usages such as fuel, may flow through a system, and be stored by a system. An item definition and usage may have attributes, states (see [7.15](#)), and be decomposed into nested item usages.

An individual item is an object that has an extent in time and may have spatial extent. The extent of an individual item in time is known as its lifetime, which covers the period in time from the item's creation to its destruction. An individual item maintains its identity over its lifetime, while the values of its features may change over time. (see [7.13](#) on the modeling of individuals, time slices and snapshots).

7.7.2 Items Concrete Syntax

7.7.2.1 Items Textual Notation

```
ItemDefinition (m : Membership) : PartDefinition =
    DefinitionPrefix? 'item' 'def' Definition(this, m)

ItemUsage (m : Membership) : ItemUsage =
    UsagePrefix? 'item' Usage(this, m)

ItemFlowUsage (m : Membership) : ItemUsage =
    UsagePrefix? 'ref'? 'item' Usage(this, m)

ItemRefUsage (m : Membership) : ItemUsage =
    UsagePrefix? ( 'ref' 'item' | isComposite ?= 'item' )
    Usage(this, m)
```

7.7.2.2 Items Graphical Notation

```
kind-keyword (a : ItemDefinition) = 'item' 'def'
kind-keyword (a : AttributeUsage) = 'item'
body-compartment (t : Type) = ... | items-compartment(t)
items-compartment (t : Type) =
    

|                                             |
|---------------------------------------------|
| <i>'items'</i><br>items-compartment-text(t) |
|---------------------------------------------|



items-compartment-text (d : Definition) =
    item-compartment-element-text(d.ownedItem) *

items-compartment-text (u : Usage) =
    item-compartment-element-text(u.nestedItem) *

items-compartment-element-text(i : ItemUsage) =
    compartment-element-text(t)

item-compartment-element-text(i : PartUsage) =
    keyword-element-text(i, "part")
```

7.7.3 Items Abstract Syntax

7.7.3.1 Overview

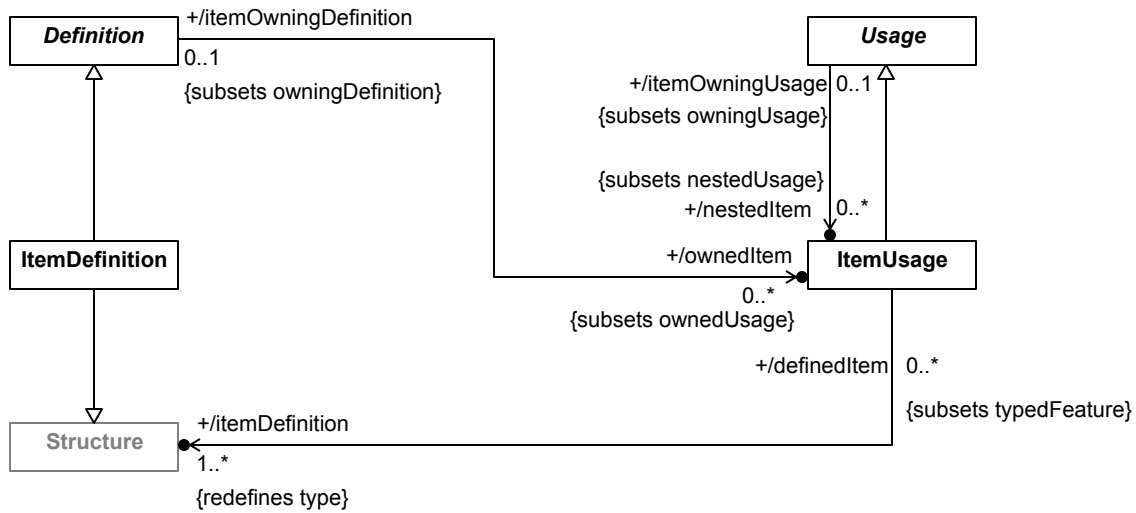


Figure 36. Item Definition and Usage

7.7.3.2 ItemDefinition

Description

An ItemDefinition is a Definition of the Structure of things that may be acted on by a system or parts of a system, which do not necessarily perform actions themselves. This includes items that can be exchanged between parts of a system, such as water or electrical signals.

An ItemDefinition must subclass, directly or indirectly, the base ItemDefinition Item from the Systems model library.

General Classes

Definition
Structure

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

7.7.3.3 ItemUsage

Description

An ItemUsage is a Usage whose type is a Class. Nominally, if the type is an ItemDefinition, an ItemUsage is a Usage of that ItemDefinition within a system. However, other types of Kernel Structure are also allowed, to permit use of Structures from the Kernel Library.

An ItemUsage must subset, directly or indirectly, the base ItemUsage `items` from the Systems model library.

General Classes

Usage

Attributes

/itemDefinition : Structure [1..*] {redefines type}

The Structures that are the types of this ItemUsage. Nominally, these are ItemDefinitions, but other kinds of Kernel Structures are also allowed, to permit use of Structures from the Kernel Library.

Operations

No operations.

Constraints

No constraints.

7.7.4 Items Semantics

7.8 Parts

7.8.1 Parts Overview

A PartDefinition represents a modular unit of structure such as a system, system component, or external entity that may directly or indirectly interact with the system. A PartDefinition is a kind of ItemDefinition (see [7.7](#)) and, as such, defines a class of part objects with temporal (and possibly spatial) extent. A PartUsage is a kind of ItemUsage.

A part usage is a usage of a part definition. A part usage may be a reference part usage or composite part usage (see [7.7](#)). A part usage may also have multiple definitions. At least one definition must be a part definition, but all other definitions can be either part definitions or item definitions. This allows a part to be treated like an item in some cases (e.g., when an engine under assembly flows through an assembly line) and as a part in other cases (e.g., when an assembled engine is installed in a vehicle).

A system is modeled as a composite part, and its part usages may themselves have further composite structure. The parts of a system may have attributes (see [7.5](#)) that represent different performance, physical and other quality characteristics. The parts may have ports (see [7.9](#)) that define the points at which those parts may be interconnected (see [7.10](#) and [7.11](#)). Parts may also *perform* actions (see [7.14](#)) resulting in items flowing across the connections between them, and *exhibit* states (see [7.15](#)) that enable different actions.

A part can represent any level of abstraction, such as a purely logical component without implementation constraints, or a physical component with a part number, or some intermediate abstraction. Parts can also be used to represent different kinds of system components such as a hardware component, a software component, a facility, an organization, or a user of a system.

7.8.2 Parts Concrete Syntax

7.8.2.1 Parts Textual Notation

```
PartDefinition (m : Membership) : PartDefinition =  
    DefinitionPrefix? ( 'part' 'def' | 'block' )  
    DefinitionDeclaration(this, m) DefinitionBody(m)  
  
PartUsage (m : Membership) : PartUsage =  
    UsagePrefix? 'part' Usage(this, m)  
  
PartFlowUsage (m : Membership) : PartUsage =  
    UsagePrefix? 'ref'? 'part' Usage(this, m)  
  
PartRefUsage (m : Membership) : PartUsage =  
    UsagePrefix? ( 'ref' 'part' | isComposite ?= 'part' )  
    Usage(this, m)
```

7.8.2.2 Parts Graphical Notation

```
kind-keyword (p : PartDefinition) = 'part' 'def' | 'block'  
  
kind-keyword (p : PortUsage) = 'part'  
  
body-compartment (t : Type) = ... | parts-compartment(t)  
  
parts-compartment (t : Type) =  


|                                             |
|---------------------------------------------|
| <i>'parts'</i><br>parts-compartment-text(t) |
|---------------------------------------------|

  
  
parts-compartment-text (d : Definition) =  
    compartment-element-text(d.ownedPart) *  
  
parts-compartment-text (u : Usage) =  
    compartment-element-text(u.nestedPart) *
```

7.8.3 Parts Abstract Syntax

7.8.3.1 Overview

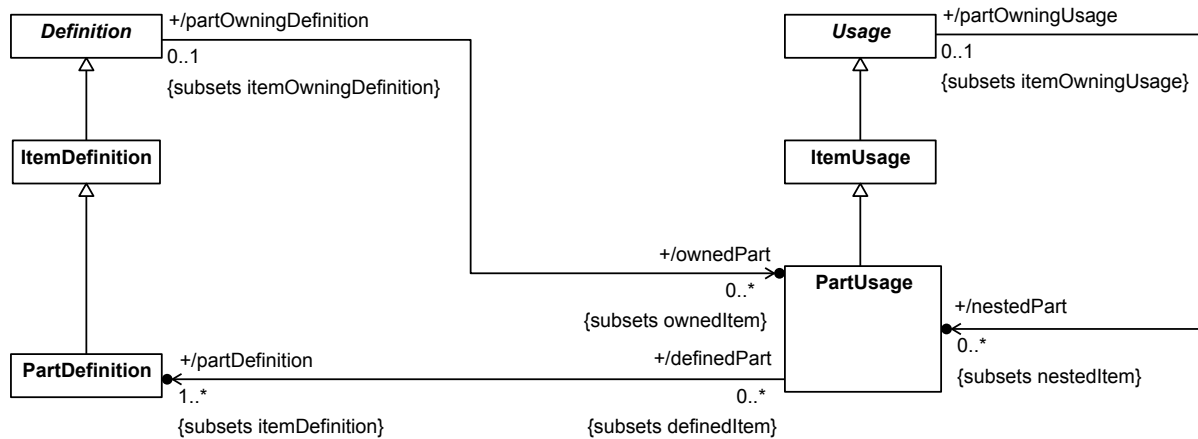


Figure 37. Part Definition and Usage

7.8.3.2 PartDefinition

Description

A PartDefinition is a ItemDefinition of a Class of systems or parts of systems. Note that all parts may be considered items for certain purposes, but not all items are parts that can perform actions within a system.

A PartDefinition must subclass, directly or indirectly, the base PartDefinition Part from the Systems model library.

General Classes

ItemDefinition

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

7.8.3.3 PartUsage

Description

A PartUsage is a usage of a PartDefinition to represent a system or a part of a system. At least one of the types of the PartUsage must be a PartDefinition.

A PartUsage must subset, directly or indirectly, the base PartUsage parts from the Systems model library.

General Classes

ItemUsage

Attributes

/partDefinition : PartDefinition [1..*] {subsets itemDefinition}

The itemDefinitions of this PartUsage that are PartDefinitions.

Operations

No operations.

Constraints

No constraints.

7.8.4 Parts Semantics

7.9 Ports

7.9.1 Ports Overview

Ports are connection points on parts that enable interactions between parts. A PortDefinition is a kind of Definition element that defines the kind of connection point on a part definition or part usage. PortUsage is a kind of Usage element that is a usage of a port definition.

A part definition or part usage can contain port usages, where each port usage can be connected to one or more port usages on other part usages. These connections enable interactions between part usages that conform to their port usages. Port usages and port definitions can contain nested port usages.

The port definition, like other definition elements, can contain features that specify the kind of interaction that a port usage can support. The port usage can redefine, subset, and/or add to the features of its port definition.

For signal flow, a common feature on a port definition is an item usage that specifies the kind of item that can flow through a port and its direction of flow. For physical interactions, such as when specifying an interface between electrical components, a common pattern is to specify across and through variables for each port. The across and through variables are attributes of the port that are defined as voltage and current quantities respectively (see [7.11](#)).

Each port definition has a *conjugated* port definition that contains features whose direction is reversed from the original port definition. A feature with direction in is changed to direction out, and a feature with direction out is changed to direction in, and a feature with direction in-out is not changed. A conjugate port usage (e.g., conjugate port) is a usage of the conjugated port definition. The conjugate port can represent a compatible port on another part where the only change is the feature direction. A simple example is the specification of two connected ports, where an item flows out of one port and into another port. In this case, the port definition contains an item usage whose direction is out, and its conjugate port definition contains the same item usage, but whose direction is in.

7.9.2 Ports Concrete Syntax

7.9.2.1 Ports Textual Notation

7.9.2.1.1 Port Definitions

```
PortDefinition (m : Membership) : PortDefinition =
  DefinitionPrefix(this)? 'port' 'def' Definition
  ownedRelationship += ConjugatedPortDefinitionMember(this)

ConjugatedPortDefinitionMember (p : PortDefinition) : Membership =
  ownedMemberFeature = ConjugatedPortDefinition(p)

ConjugatedPortDefinition (p : PortDefinition) : ConjugatedPortDefinition =
  ownedRelationship += PortConjugation(p)

PortConjugation (p : PortDefinition) : PortConjugation =
  originalPortDefinition = p
```

7.9.2.1.2 Port Usages

```
PortUsage (m : Membership) : PortUsage =
  UsagePrefix(this)? 'port'
  PortDeclaration(this, m) ValuePart(this)? UsageBody(this)

PartEndUsage (m : Membership) : PortUsage =
  UsagePrefix(this)? 'port'?
  PortDeclaration(this, m) ValuePart(this)? UsageBody(this)

PortDeclaration (p : PortDefinition, m : Membership) =
  Identification(p, m) PortSpecializationPart(p)

PortSpecializationPart (p : Port) =
  PortSpecialization(p) * MultiplicityPart(p)? PortSpecialization(p) *

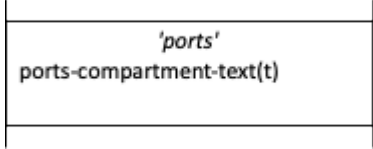
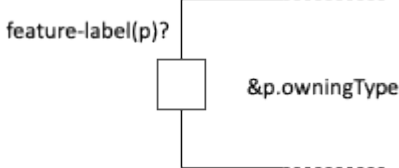
PortSpecialization (p : PortDefinition) =
  PortTypings(p) | Subsettings(p) | Redefinitions(p)

PortTypings (p : PortDefinition) =
  ( ':' | 'defined' 'by' ) p.ownedTyping += PortTyping
  ( ',' p.ownedTyping += PortTyping ) *

PortTyping : FeatureTyping =
  OwnedFeatureTyping | ConjugatedPortTyping

ConjugatedPortTyping : ConjugatedPortTyping =
  '~' originalPortDefinition = [Qualified Name]
```

7.9.2.2 Ports Graphical Notation

```
kind-keyword (p : PortDefinition) = 'port' 'def'
kind-keyword (p : PortUsage) = 'port'
body-compartment (t : Type) = ... | ports-compartment(t)
ports-compartment (t : Type) =
    
ports-compartment-text (d : Definition) =
    compartment-element-text(d.ownedPort)*
ports-compartment-text (u : Usage) =
    compartment-element-text(u.nestedPort)*
boundary-feature-symbol (p : Port, null) =
    
```

Note. A port can be shown using a *boundary feature symbol*, even if it does not have a specified direction (see also [7.4.2.2.4](#) on boundary feature graphical notation in general).

7.9.3 Ports Abstract Syntax

7.9.3.1 Overview

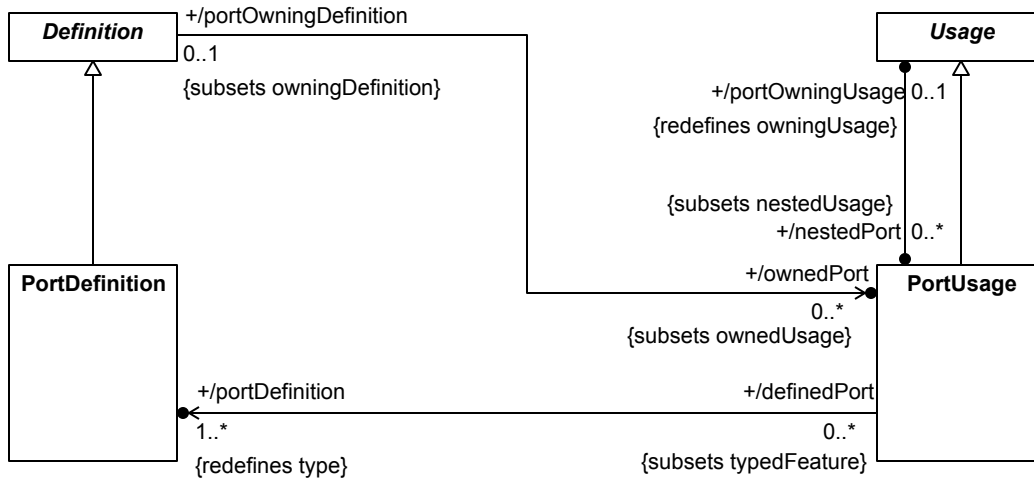


Figure 38. Port Definition and Usage

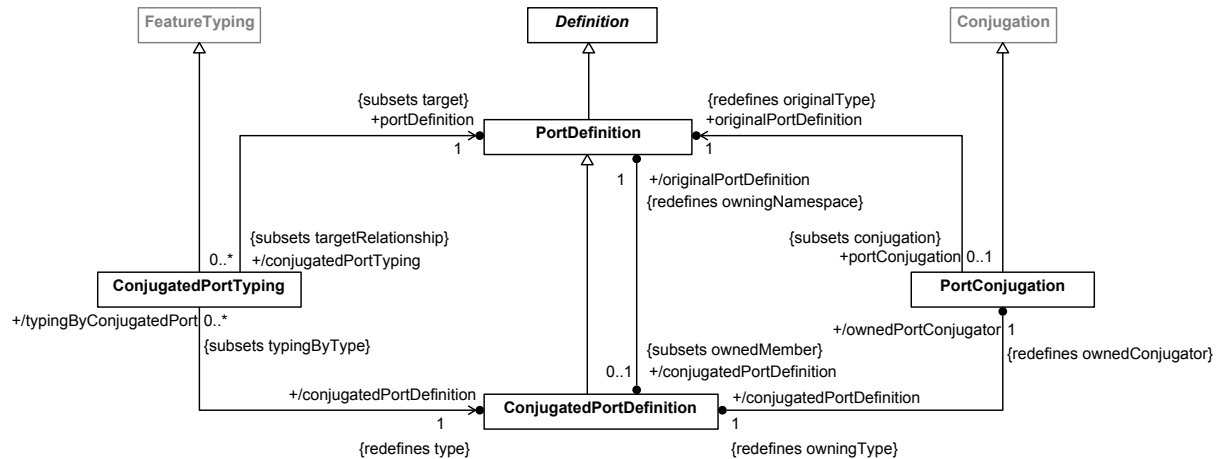


Figure 39. Port Conjugation

7.9.3.2 ConjugatedPortDefinition

Description

A **ConjugatedPortDefinition** is a **PortDefinition** that is a **PortConjugation** of its original **PortDefinition**. That is, a **ConjugatedPortDefinition** inherits all the *features* of the original **PortDefinition**, but input *flows* of the original **PortDefinition** become outputs on the **ConjugatedPortDefinition** and output *flows* of the original **PortDefinition** become inputs on the **ConjugatedPortDefinition**. Every **PortDefinition** has exactly one corresponding **ConjugatedPortDefinition**, whose name is the same as that of the original **PortDefinition**, with the character `~` prepended.

General Classes

PortDefinition

Attributes

`/originalPortDefinition : PortDefinition {redefines owningNamespace}`

The original PortDefinition for this ConjugatedPortDefinition.

`/ownedPortConjugator : PortConjugation {redefines ownedConjugator}`

The PortConjugation that is the ownedConjugator of this ConjugatedPortDefinition, linking it its originalPortDefinition.

Operations

No operations.

Constraints

`conjugatedPortDefinitionConjugatedPortDefinitionIsEmpty`

[no documentation]

`conjugatedPortDefinition = null`

`conjugatedPortDefinitionOriginalPortDefinition`

[no documentation]

`originalPortDefinition = ownedPortConjugator.originalPortDefinition`

7.9.3.3 ConjugatedPortTyping

Description

A ConjugatedPortTyping is a FeatureTyping in which the type is derived as the conjugatedPortDefinition of a given PortDefinition. A ConjugatedPortTyping allows a PortUsage to be related directly to a PortDefinition, but to be effectively typed by the conjugation of the referenced PortDefinition.

Note that ConjugatedPortTyping is a *ternary* Relationship, with portDefinition being a third relatedElement, in addition to type and typedFeature from FeatureTyping.

General Classes

FeatureTyping

Attributes

`/conjugatedPortDefinition : ConjugatedPortDefinition {redefines type}`

The conjugatedPortDefinition of the portDefinition of this ConjugatedPortTyping, which is the derived type of the ConjugatedPortTyping considered as a FeatureTyping.

`portDefinition : PortDefinition {subsets target}`

The PortDefinition whose conjugatedPortDefinition is to be the derived type of this ConjugatedPortTyping.

Operations

No operations.

Constraints

conjugatedPortTypingConjugatedPortDefinition

[no documentation]

```
conjugatedPortDefinition = portDefinition.conjugatedPortDefinition
```

7.9.3.4 PortConjugation

Description

A PortConjugation is a Conjugation Relationship between a PortDefinition and its corresponding ConjugatedPortDefinition. As a result of this Relationship, the ConjugatedPortDefinition inherits all the features of the original PortDefinition, but input flows of the original PortDefinition become outputs on the ConjugatedPortDefinition and output flows of the original PortDefinition become inputs on the ConjugatedPortDefinition.

General Classes

Conjugation

Attributes

/conjugatedPortDefinition : ConjugatedPortDefinition {redefines owningType}

The ConjugatedPortDefinition that is conjugate to the originalPortDefinition.

originalPortDefinition : PortDefinition {redefines originalType}

The PortDefinition being conjugated.

Operations

No operations.

Constraints

No constraints.

7.9.3.5 PortDefinition

Description

A PortDefinition defines a point at which external entities can connect to and interact with a system or part of a system. Any ownedUsages of a PortDefinition must not be composite.

General Classes

Definition

Attributes

/conjugatedPortDefinition : ConjugatedPortDefinition [0..1] {subsets ownedMember}

The ConjugatedPortDefinition that is conjugate to this PortDefinition.

Operations

No operations.

Constraints

portDefinitionConjugatedPortDefinition

[no documentation]

conjugatedPortDefinition = ownedMember->select (oclIsKindOf (ConjugatedPortDefinition))

7.9.3.6 PortUsage

Description

A PortUsage is a usage of a PortDefinition. A PortUsage must be owned by a PartDefinition, a PortDefinition, a PartUsage or another PortUsage. Any ownedUsages of a PortUsage must not be composite.

A PortUsage must subset, directly or indirectly, the PortUsage ports from the Systems model library.

General Classes

Usage

Attributes

/portDefinition : PortDefinition [1..*] {redefines type}

The types of this PortUsage, which must all be PortDefinitions.

/portOwningUsage : Usage [0..1] {redefines owningUsage}

The Usage in which the nestedPort is nested (if any).

Operations

No operations.

Constraints

No constraints.

7.9.4 Ports Semantics

7.10 Connections

7.10.1 Connections Overview

Connections enable features such as items and parts to be connected in a context. A ConnectionDefinition is a kind of KerML AssociationStructure (see [7.2.7](#)) and also a kind of PartDefinition (see [7.8](#)). This enables a connection

definition to be a kind of relationship, but at the same time to have structure and other features similar to a PartDefinition. A connection definition can associate two or more definition elements. A ConnectionUsage is a kind of KerML Connector (see [7.2.8](#)) and a kind of PartUsage (see [7.8](#)). A connection usage is a usage of a connection definition that connects usage elements such as items and parts. Connection definitions and connection usages contain connection ends that are bound to their respective connected elements.

A connection usage that connects parts is often a logical connection that abstracts away details of how they are connected. For example, plumbing that includes pipes and fittings may be used to connect a pump and a tank. It is sometimes desired to connect the pump to the tank at a more abstract level without including the plumbing. This is viewed as a logical connection between the pump and the tank. Alternatively, the plumbing can be modeled as a part, sometimes referred to as an interface medium, where the pump connects to the plumbing, and the plumbing connects to the tank.

Since connections are definition and usage elements, they can also have structure. The connection can contain the plumbing either as an owned member, or as a reference, to the plumbing that is owned by a higher level pump-tank system context. In this way, the logical connection without structure can be transformed to a physical connection.

A BindingConnector is also a kind of KerML Connector, which is adopted directly for use in SysML and is *not* a kind of ConnectionUsage. A binding connector is a special kind of connector that asserts that the values of the features on either end are equal. This is used to bind attributes together to assert two attributes have the same value. A binding connector is also used to bind a reference feature of one context to an owned feature of another context to assert they are the same thing. For example, the steering wheel in a vehicle may be considered part of the interior of the car, while at the same time it is considered part of steering subsystem. The steering wheel can be a composite part usage of the interior, and a reference part usage of the steering subsystem, and these two usages can be bound together with a binding connector to assert that they are the same part. (See [7.2.8](#) for further description of binding connectors.)

Submission Note. Adding a BindingConnection specialization of BindingConnector for use in SysML is being considered for the revised submission.

7.10.2 Connections Concrete Syntax

7.10.2.1 Connections Textual Notation

7.10.2.1.1 Connection Definitions

```
ConnectionDefinition (m : Membership) : ConnectionDefinition =
  DefinitionPrefix(this)? ('connection' 'def' | 'assoc' 'block' )
  DefinitionDeclaration(this, m) ConnectionBody(this)

ConnectionBody (t : Type) =
  ';' | '{' ConnectionBodyItem(t)* '}'

ConnectionBodyItem (t : Type) =
  DefinitionBodyItem(t)
  | t.ownedRelationship += ConnectionEndMember

ConnectionEndMember : EndFeatureMembership =
  DefinitionMemberPrefix(this) 'end'
  ownedMemberFeature = ConnectionEndElement(this)

ConnectionEndElement (m : Membership) : Usage =
  ReferenceEndUsage(m)
  | ItemRefUsage(m)
  | PartRefUsage(m)
  | PortUsage(m)
  | ActionRefUsage(m)
  | CalculationRefUsage(m)
  | StateRefUsage(m)
```

7.10.2.1.2 Connection Usages

```
ConnectionUsage (m : Membership) : ConnectionUsage =
  UsagePrefix(this)?
  ( ( 'connection' | 'link' ) UsageDeclaration(this, m)
    ( 'connect' ConnectorPart(this) )? )
  | 'connect' ConnectorPart(this)
  )
  ConnectionBody(this)

ConnectorPart (c : ConnectionUsage ) =
  BinaryConnectorPart(c) | NaryConnectorPart(c)

BinaryConnectorPart (c : ConnectionUsage) =
  c.ownedFeatureMembership += ConnectorEndMember 'to'
  c.ownedFeatureMembership += ConnectorEndMember

NaryConnectorPart (c : ConnectionUsage) =
  '(' c.ownedFeatureMembership += ConnectorEndMember ','
  c.ownedFeatureMembership += ConnectorEndMember
  ( ',' c.ownedFeatureMembership += ConnectorEndMember )* ')'
```


7.10.2.2 Connections Graphical Notation

```
kind-keyword (c : ConnectionDefinition) = 'connection' 'def'
kind-keyword (c : ConnectionUsage) = 'connection'
body-compartment (t : Type) = ... | connections-compartment(t) | ends-compartment(t)
connections-compartment (t : Type) =
    
connections-compartment-text (d : Definition) =
    connections-compartment-element-text(d.ownedConnection)*
connections-compartment-text (u : Usage) =
    connections-compartment-element-text(u.nestedConnection)*
connections-compartment-element-text (c : Connection) =
    compartment-element-text(c)
connection-compartment-element-text (i : Interface) =
    keyword-element-text(i, "interface")
ends-compartment (t : Type) =
    
ends-compartment-text (t : Type) =
    ends-compartment-element-text(t.endFeature)*
end-compartment-element-text (f : Feature) =
    visibility-keyword(f) variation-keyword(f)? abstract-keyword(f)?
    ref-keyword(f)? kind-keyword(f) type-decl-text(f)
    element-graphic(f.ownedAnnotation)*
```

7.10.3 Connections Abstract Syntax

7.10.3.1 Overview

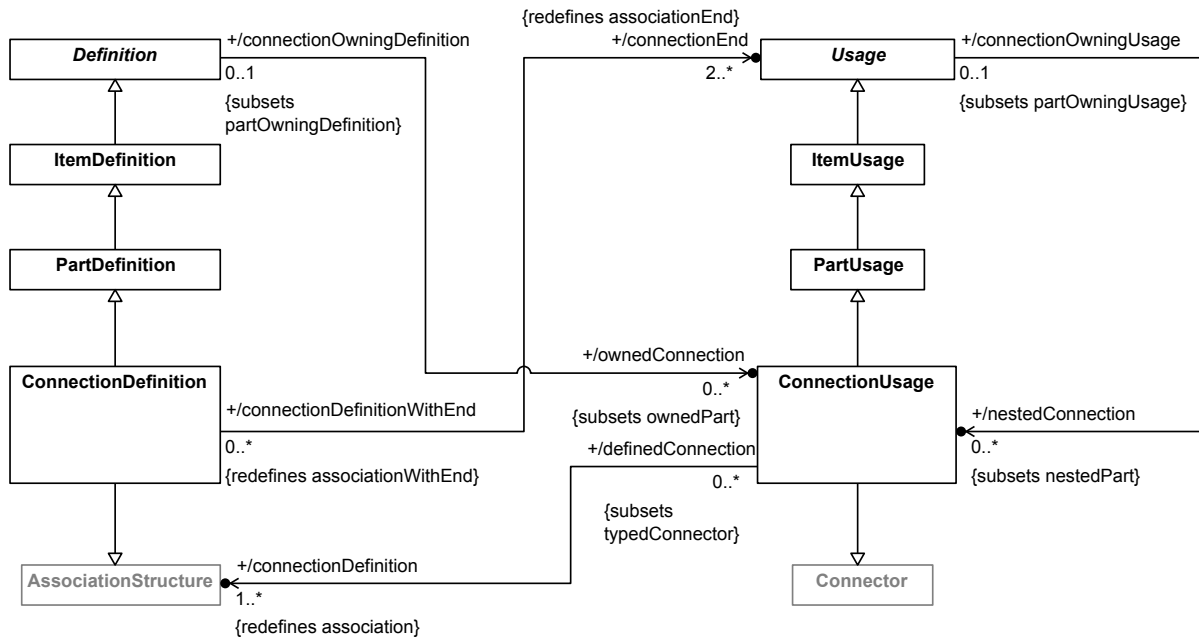


Figure 40. Connection Definition and Usage

7.10.3.2 ConnectionDefinition

Description

A ConnectionDefinition is a PartDefinition that is also an AssociationStructure, with two or more end features. The `associationEnds` of a ConnectionDefinition must be Usages.

A ConnectionDefinition must subclass, directly or indirectly, the base ConnectionDefinition Connection from the Systems model library.

General Classes

PartDefinition
AssociationStructure

Attributes

`/connectionEnd : Usage [2..*] {redefines associationEnd}`

The Usages that define the things related by the ConnectionDefinition.

Operations

No operations.

Constraints

No constraints.

7.10.3.3 ConnectionUsage

Description

A ConnectionUsage is a Connector that is also a Usage. Nominally, if its type is a ConnectionDefinition, then a ConnectionUsage is a Usage of that ConnectionDefinition, representing a connection between parts of a system. However, other kinds of kernel AssociationStructures are also allowed, to permit use of Associations from the Kernel Library (such as the default BinaryLink).

A ConnectionUsage must subset the base ConnectionUsage `connections` from the Systems model library.

General Classes

Connector
PartUsage

Attributes

/connectionDefinition : AssociationStructure [1..*] {redefines association}

The Associations that are the types of this ConnectionUsage. Nominally, these are ConnectionDefinitions, but other kinds of Kernel Associations are also allowed, to permit use of Associations from the Kernel Library.

Operations

No operations.

Constraints

No constraints.

7.10.4 Connections Semantics

7.11 Interfaces

7.11.1 Interfaces Overview

An interface facilitates the specification and reuse of compatible connections between parts. An InterfaceDefinition is a kind of ConnectionDefinition whose ends are restricted to be port definitions. An InterfaceUsage is a kind of ConnectionUsage that is usage of an interface definition. The ends of an interface usage are restricted to be port usages. Quite simply, an interface is a connection with ports on either end.

Interface definitions provide an important mechanism for specifying and reusing interfaces. A logical interface may be defined that can then be specialized to represent more specific physical interfaces. For example, an interface can be defined to represent the power interface connection between an appliance and the wall power. The port on one side of the interface can represent the appliance power connection point, and the port on the other side can represent the power outlet connection point. This interface can then be defined, specialized as necessary, and used for connecting many different appliances to wall power.

When modeling physical interactions, the interface definition and interface usage can contain constraints to constrain the values of the attributes on the ports on either end (see [7.9](#)). These attributes are referred to as across and through variables, which are constrained by conservation laws across the interface (e.g., Kirchhoff's Laws). For example, when specifying an interface between electrical components, the across and through variables specified

on each port are attributes defined as voltage and current quantities respectively. The attribute values on either port are constrained such that the voltages must be equal, and the sum of the currents must equal zero.

7.11.2 Interfaces Concrete Syntax

7.11.2.1 Interfaces Textual Syntax

7.11.2.1.1 Interface Definitions

```
InterfaceDefinition (m : Membership) : InterfaceDefinition =  
    DefinitionPrefix(this) 'interface' 'def'  
    DefinitionDeclaration(this, m) InterfaceBody(this)  
  
InterfaceBody (t : Type) =  
    ';' | '{' InterfaceBodyItem(t)* '}'  
  
InterfaceBodyItem (t : Type) =  
    DefinitionBodyItem(t)  
    | t.ownedRelationship += InterfaceEndMember  
  
InterfaceEndMember : EndFeatureMembership =  
    DefinitionMemberPrefix(this) 'end'  
    ownedMemberFeature = PortEndUsage
```

7.11.2.1.2 Interface Usages

```
InterfaceUsage (m : Membership) : InterfaceUsage =  
    UsagePrefix(this) 'interface'  
    ( UsageDeclaration(this, m) ( 'connect' ConnectorPart(this) )?  
    | ConnectorPart(this)  
    )  
    InterfaceBody(this)
```

7.11.2.2 Interfaces Graphical Syntax

```
kind-keyword (i : InterfaceDefinition) = 'interface' 'def'

kind-keyword (i : InterfaceUsage) = 'interface'

body-compartment (t : Type) = ... | interfaces-compartment(t)

connections-compartment (t : Type) =
    

|                                                              |
|--------------------------------------------------------------|
| <i>'interfaces'</i><br><b>interfaces-compartment-text(t)</b> |
|--------------------------------------------------------------|



interfaces-compartment-text (d : Definition) =
    compartment-element-text(d.ownedInterface)*

interfaces-compartment-text (u : Usage) =
    compartment-element-text(u.nestedInterface)*
```

7.11.3 Interfaces Abstract Syntax

7.11.3.1 Overview

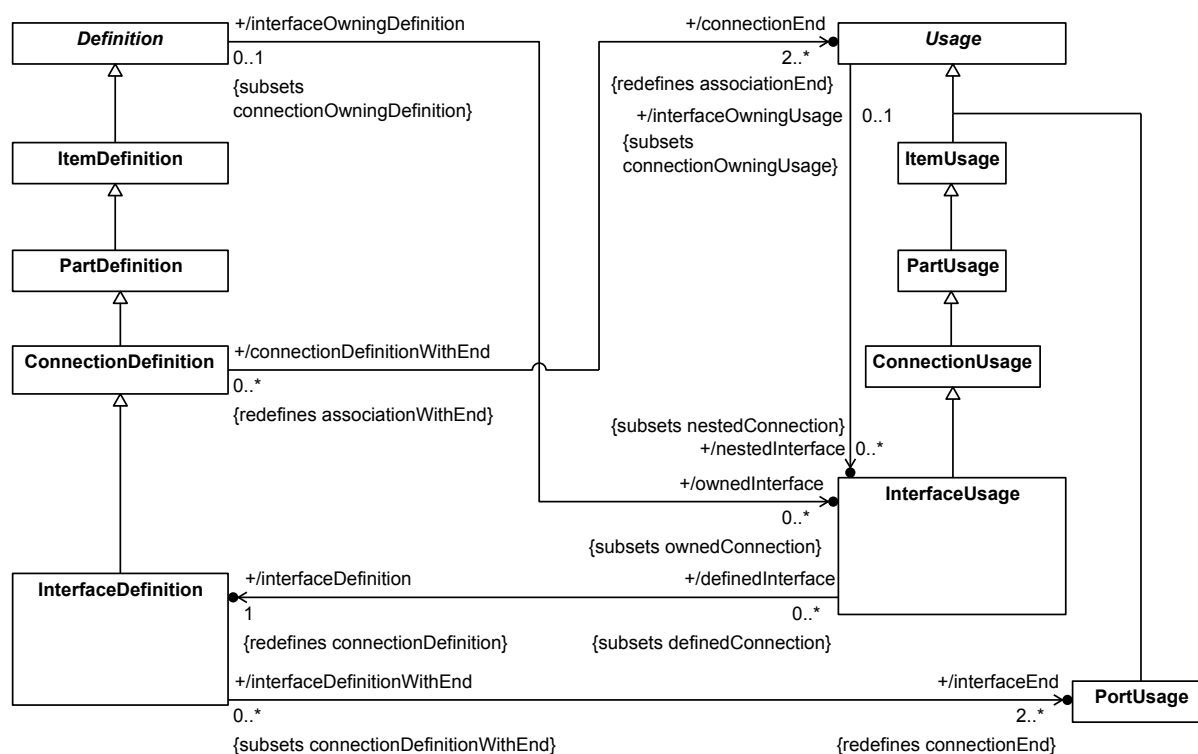


Figure 41. Interface Definition and Usage

The `InterfaceUsages` that are typed by a certain `InterfaceDefinition`.

7.11.3.2 InterfaceDefinition

Description

An InterfaceDefinition is a ConnectionDefinition all of whose ends are PortUsages, defining an interface between elements that interact through such ports.

An InterfaceDefinition must subclass, directly or indirectly, the base InterfaceDefinition Interface from the Systems model library.

General Classes

ConnectionDefinition

Attributes

/interfaceEnd : PortUsage [2..*] {redefines connectionEnd}

The PortUsages that are the associationEnds of this InterfaceDefinition.

Operations

No operations.

Constraints

No constraints.

7.11.3.3 InterfaceUsage

Description

An InterfaceUsage is a Usage of an InterfaceDefinition to represent an interface connecting parts of a system through specific ports.

An InterfaceUsage must subset, directly or indirectly, the base InterfaceUsage interfaces from the Systems model library.

General Classes

ConnectionUsage

Attributes

/interfaceDefinition : InterfaceDefinition {redefines connectionDefinition}

The InterfaceDefinition that is the single type of this InterfaceUsage.

Operations

No operations.

Constraints

No constraints.

7.11.4 Interfaces Semantics

7.12 Allocations

7.12.1 Allocations Overview

An AllocationDefinition is a ConnectionDefinition (see [7.10](#)) that specifies that some or all of the responsibility to realize the intent of a source element is allocated to a target element. An AllocationUsage is a usage of one or more AllocationDefinitions. An AllocationDefinition or AllocationUsage can be further refined using nested AllocationUsages that give a finer-grained decomposition of the containing allocation.

As used by systems engineers, an allocation denotes a "mapping" across the various structures and hierarchies of a system model. This concept of "allocation" requires flexibility suitable for abstract system specification, rather than a particular constrained method of system or software design. System modelers often associate various elements in a user model in abstract, preliminary, and sometimes tentative ways. Allocations can be used early in the design as a precursor to more detailed rigorous specifications and implementations.

Allocations can provide an effective means for navigating a model by establishing cross relationships and ensuring that various parts of the model are properly integrated. By making these relationships instantiable connections, they can also be semantically related to other such relationships, including satisfying requirements, performing actions and exhibiting states. Modelers can also create specialized allocation definitions to reflect conventions for allocation on specific projects or within certain system models.

Release Note. The library model for allocations currently does not provide any specializations of the most general definition of an Allocation. Consideration will be given to including specializations of Allocation in a future release to cover similar areas as in SysML v1 (i.e., behavior, structure and flows) and the relationship of these to new capabilities in SysML v2 for performing actions, etc.

7.12.2 Allocations Concrete Syntax

7.12.2.1 Allocations Textual Notation

```
AllocationDefinition (m : Membership) : AllocationDefinition =  
    DefinitionPrefix(this)? 'allocation' 'def'  
    DefinitionDeclaration(this, m) ConnectionBody(this)  
  
AllocationUsage (m : Membership) : AllocationUsage =  
    UsagePrefix(this)?  
    ( 'allocation' UsageDeclaration(this, m)  
      ( 'allocate' ConnectorPart(this) )? )  
    | 'allocate' ConnectorPart(this)  
    )  
    ConnectionBody(this)
```

7.12.2.2 Allocations Graphical Notation

7.12.3 Allocations Abstract Syntax

7.12.3.1 Overview

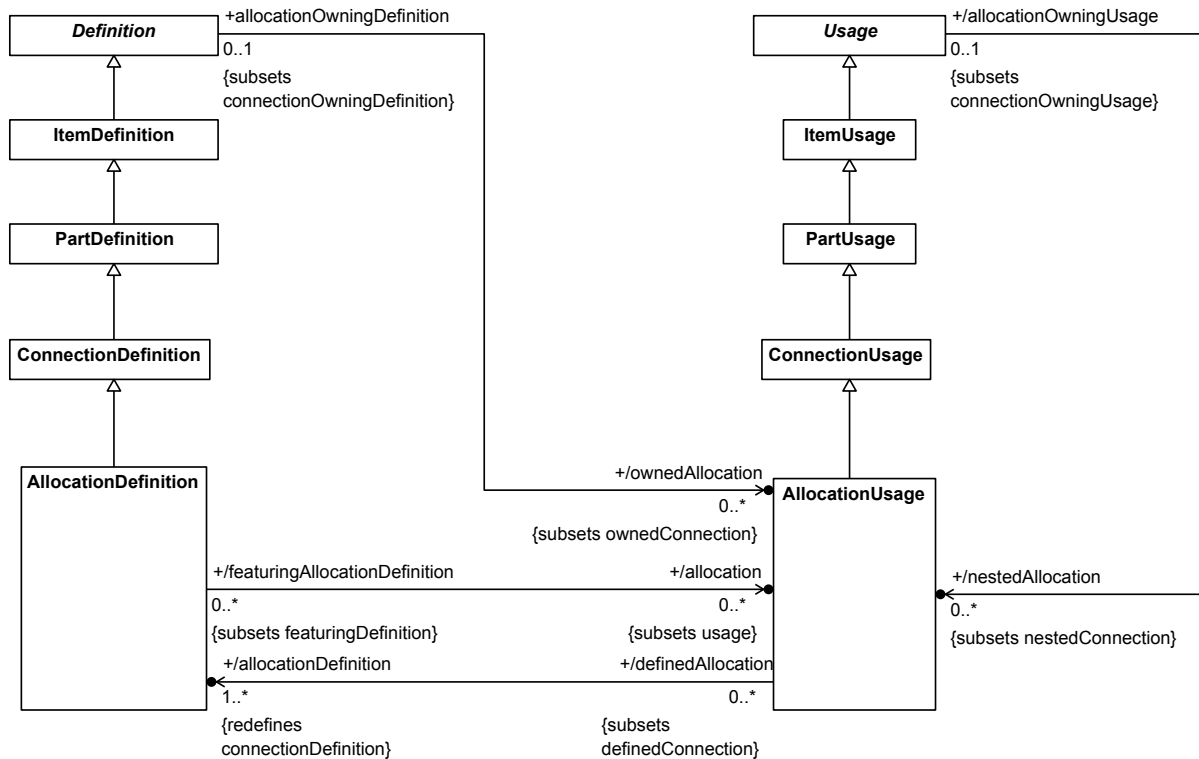


Figure 42. Allocation Definition and Usage

7.12.3.2 AllocationDefinition

Description

An **AllocationDefinition** is a **ConnectionDefinition** that specifies that some or all of the responsibility to realize the intent of the `source` is allocated to the `target` instances. Such allocations define mappings across the various structures and hierarchies of a system model, perhaps as a precursor to more rigorous specifications and implementations. An **AllocationDefinition** can itself be refined using nested allocations that give a finer-grained decomposition of the containing allocation mapping.

An **AllocationDefinition** must subclass, directly or indirectly, the base **AllocationDefinition** **Allocation** from the Systems model library.

General Classes

ConnectionDefinition

Attributes

/allocation : **AllocationUsage** [0..*] {subsets usage}

The **ActionUsages** that refine the allocation mapping defined by this **AllocationDefinition**.

Operations

No operations.

Constraints

No constraints.

7.12.3.3 AllocationUsage

Description

An AllocationUsage is a usage of an AllocationDefinition asserting the allocation of the `source` feature to the `target` feature.

An AllocationUsage must subset, directly or indirectly, the base AllocationUsage allocations from the Systems model library.

General Classes

ConnectionUsage

Attributes

/allocationDefinition : AllocationDefinition [1..*] {redefines connectionDefinition}

The AllocationDefinitions that are the types of this AllocationUsage.

Operations

No operations.

Constraints

No constraints.

7.12.4 Allocations Semantics

7.13 Individuals

7.13.1 Individuals Overview

Individuals

An IndividualDefinition is a kind of ItemDefinition. An individual definition is intended to represent a real or perceived object with a unique identity.

A simple example is the representation of an individual car called Car1 with a vehicle identification number. Car1 is a subclass of the part definition Car. As such, Car1 inherits all the features of Car. Car1 is composed of an individual engine, individual transmission, individual chassis, and 4 individual wheels, each of which is defined by a uniquely identifiable subclass of a part definition.

An IndividualUsage is a kind of Item Usage. An individual usage is a usage of an individual definition. For example, the individual definition Car1 can be used in different contexts, such as the usage of Car1 when it is in for service and the usage of Car1 when it is used for normal operations. The usage of Car1 is called car1InService when it is in for service to have its tires rotated. For this usage, car1InService has four wheels that play different roles, including front-left, front-right, rear-left, and rear-right. Each of the four wheels of Car1 is an individual wheel usage defined by an individual wheel definition that is named Wheel1, Wheel2, Wheel3, and Wheel4 respectively. Each individual wheel definition is a subclass of Wheel. When car1_InService enters the shop, the front-left wheel is initially

defined by Wheel1, but after the tires are rotated, the front-left wheel is defined by Wheel2. An individual usage is a role that an individual definition plays for some period of time.

Lifetimes

An Individual definition has a lifetime with a beginning and an end. The beginning of the lifetime occurs when the identity of the individual is established, and the end of the lifetime occurs when the individual loses its identity. For example, the lifetime of a car could begin when it leaves the production-line, or when its vehicle identification number is assigned to the car. Similarly, the lifetime of a car could end when the car is disassembled or demolished. During its lifetime, Car1 can play different roles, such as the car1InService role and the car1InOperation role.

Time Slices

The individual's lifetime can be partitioned into time slices which correspond to some duration of time. These time slices can represent periods or phases of a lifetime, such as the deployment or operational phase. Time slices can be further partitioned into other time slices.

The lifetime and any of its time slices can be actual or projected. For example, the individual car Car1 may be purchased as a used car. Car1 has had an actual lifetime up to that time. A mechanic may perform diagnostics and obtain some measurements, and may estimate the remaining life of the car or its parts based on the measurements. For example, the mechanic may estimate the remaining lifetime of the tires, based on the tread measurements and the estimated tire wear rate.

Snapshots

A time slice with zero time is a snapshot. The start and end snapshot, and intermediate snapshots can be defined for any time slice to represent particular points in time in the individual's lifetime. At a given point in time, the state (i.e., condition) of the individual usage can be specified by the values of its attributes. As an example, the state of car1_inOperation at different points in time can be specified in terms of its acceleration, velocity, and position. In addition, its finite (i.e., discrete) state can be specified at different points in time as off or on, and any nested state such as forward or reverse. The state (i.e., condition) of the car can continue to change over its lifetime, and can be specified as a function of discrete and/or continuous time.

7.13.2 Individuals Concrete Syntax

7.13.2.1 Individuals Textual Notation

7.13.2.1.1 Individual Definitions

```
IndividualDefinition (m : Membership) : IndividualDefinition =
  DefinitionPrefix(m) 'individual' 'def' Definition(this, m)
  ownedRelationship += LifeClassMembership(this)

LifeClassMembership (i : IndividualDefinition) : Membership =
  ownedMemberElement = LifeClass(i)

LifeClass (i : IndividualDefinition) : Membership =
  ownedRelationship += TargetedSuperclassing(i)
  ownedRelationship += SingletonMultiplicityMember

TargetedSuperclassing (c : Classifier) : Superclassing =
  { general = c }

SingletonMultiplicityMember : FeatureMembership =
  ownedMemberFeature = SingletonMultiplicity

SingletonMultiplicity : MultiplicityRange =
  lowerBound += LiteralZero upperBound += LiteralOne

LiteralZero : LiteralInteger =
  { value = 0 }

LiteralOne : LiteralInteger =
  { value = 1 }
```

7.13.2.1.2 Individual Usages

```
IndividualUsage (m : Membership) : IndividualUsage =
    UsagePrefix(this)? 'individual' Usage(this, m)

IndividualRefUsage (m : Membership) : IndividualUsage =
    UsagePrefix(this)? ( 'ref' 'individual' | isComposite ?= 'individual' )
    Usage(this, m)

TimeSliceUsage (m : Membership) : IndividualUsage =
    UsagePrefix(this)? 'timeslice' Usage(this, m)
    ownedRelationship += TimeSliceFeatureMember(this)

TimeSliceRefUsage (m : Membership) : IndividualUsage =
    UsagePrefix(this)? ( 'ref' 'timeslice' | isComposite ?= 'timeslice' )
    Usage(this, m)
    ownedRelationship += TimeSliceFeatureMember(this)

TimeSliceFeatureMember (i : IndividualUsage) : FeatureMembership =
    ownedMemberFeature = TimeSliceFeature(i)

TimeSliceFeature (i : IndividualUsage) : Feature =
    ownedRelationship += FeatureTypingTo(i.individualDefinition)

SnapshotUsage (m : Membership) : IndividualUsage =
    UsagePrefix(this)? 'snapshot' Usage(this, m)
    ownedRelationship += SnapshotFeatureMember(this)

SnapshotRefUsage (m : Membership) : IndividualUsage =
    UsagePrefix(this)? ( 'ref' 'snapshot' | isComposite ?= 'snapshot' )
    Usage(this, m)
    ownedRelationship += SnapshotFeatureMember(this)

SnapshotFeatureMember (i : IndividualUsage) : FeatureMembership =
    ownedMemberFeature = SnapshotFeature(i)

SnapshotFeature (i : Individual) : Feature =
    ownedRelationship += FeatureTypingTo(i.individualDefinition)

FeatureTypingTo (t : Type) : FeatureTyping =
    { type = t }
```

7.13.2.1.3 Individual Successions

```
SourceSuccessionMember (t : Type) : FeatureMembership =
  ownedMemberFeature = SourceSuccession(t)
  { t.ownedRelationship += this }

SourceSuccession (t : Type) : Succession =
  'then' ownedRelationship += SourceEndMemberFrom(t)

SourceEndMemberFrom (t : Type) : EndFeatureMembership =
  ConnectorEndMemberFor(sourceFeature(t))

ConnectorEndMemberFor (f : Feature) : EndFeatureMembership =
  ownedMemberFeature = ConnectorEndFor(f)

ConnectorEndFor (f : Feature) : Feature =
  ownedSubsetting += SubsettingTo(f)
  ( ownedRelationship += MultiplicityMember )?

TargetEndMemberFor (f : Feature) : EndFeatureMembership =
  ownedMemberFeature = TargetEndFor(f)

TargetEndFor (f : Feature) : Feature =
  ownedRelationship += SubsettingTo(f)

SubsettingTo (f : Feature) : Subsetting =
  { subsettingFeature = f }
```

7.13.2.2 Individuals Graphical Notation

```
kind-keyword (a : IndividualDefinition) = 'individual' 'def'
kind-keyword (a : IndividualUsage) = 'individual'
body-compartment (t : Type) = ... | individuals-compartment(t)
individuals-compartment (t : Type) =
    

|                                                                |
|----------------------------------------------------------------|
| <i>'individuals'</i><br><b>individuals-compartment-text(t)</b> |
|----------------------------------------------------------------|


individuals-compartment-text (d : Definition) =
    individuals-compartment-element-text(d.ownedIndividual)*
individuals-compartment-text (u : Usage) =
    individuals-compartment-element-text(u.nestedIndividual)*
individuals-compartment-element-text (i : IndividualUsage) =
    individual-usage-element-text(i, i.isTimeSlice, i.isSnapshot)
individuals-compartment-element-text (i : IndividualUsage, false, false) =
    compartment-element-text(i)
individuals-compartment-element-text (i : IndividualUsage, true, false) =
    keyword-element-text(i, 'timeslice')
individuals-compartment-element-text (i : IndividualUsage, false, true) =
    keyword-element-text(i, 'snapshot')
```

7.13.3 Individuals Abstract Syntax

7.13.3.1 Overview

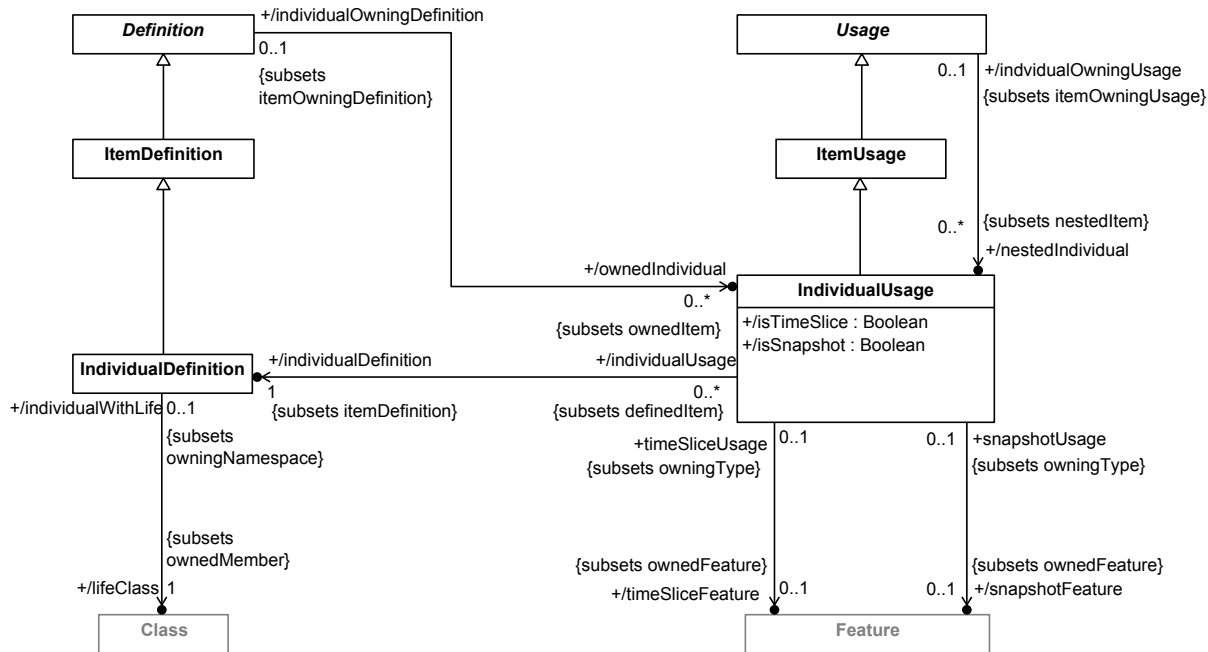


Figure 43. Individual Definition and Usage

7.13.3.2 IndividualDefinition

Description

An IndividualDefinition is an ItemDefinition that is constrained to represent an individual thing. The instances of an IndividualDefinition include all spatial and temporal portions of the individual being represented, but only one of these can be the complete Life of the individual. All other instances must be portions of the "maximal portion" that is single Life instance, capturing the conception that all of the instances represent one individual with a single "identity".

General Classes

ItemDefinition

Attributes

/lifeClass : Class {subsets ownedMember}

An Class that is an ownedMember of this IndividualDefinition, which specializes both the IndividualDefinition and the Base::Life Class from the Kernel Library and has a multiplicity of 0..1. This constrains the IndividualDefinition to have at most one instance that is a complete Life.

Operations

No operations.

Constraints

No constraints.

7.13.3.3 IndividualUsage

Description

An IndividualUsage is a ItemUsage exactly one of whose types is an IndividualDefinition, specifying the usage of the represented individual, or a portion of it, in a certain context.

General Classes

ItemUsage

Attributes

/individualDefinition : IndividualDefinition {subsets itemDefinition}

The one type of the IndividualUsage that is the IndividualDefinition defining the individual thing being represented. (Note that the IndividualUsage may have other types in addition to this that are not IndividualDefinitions.)

/isSnapshot : Boolean

Whether this IndividualUsage represents a temporal portion of the entire spacial extent of an individual at one instance of time, as indicated by whether or not it has a snapshotFeature.

/isTimeSlice : Boolean

Whether this IndividualUsage represents a temporal portion of the entire spacial extent of an individual over some duration of time, as indicated by whether or not it has a timeSliceFeature.

/snapshotFeature : Feature [0..1] {subsets ownedFeature}

An ownedFeature of this IndividualUsage that is a redefinition of the Feature Occurrence::snapshotOf and is typed by the individualDefinition. An IndividualUsage with such a feature is restricted to represent a snapshot of the represented individual.

/timeSliceFeature : Feature [0..1] {subsets ownedFeature}

An ownedFeature of this IndividualUsage that is a redefinition of the Feature Occurrence::timeSliceOf and is typed by the individualDefinition. An IndividualUsage with such a feature is restricted to represent a time slice of the represented individual.

Operations

No operations.

Constraints

individualUsageIsTimeSlice

[no documentation]

isTimeSlice = (timeSliceFeature <> null)

individualUsageIsSnapshotOf

[no documentation]


```
isSnapshot = (snapshotFeature <> Null)
```

7.13.4 Individuals Semantics

7.14 Actions

7.14.1 Actions Overview

Actions

An ActionDefinition is a kind of KerML Behavior (see [7.2.9](#)) and a Definition element that can specify what a part does in terms of its transformation of input items to output items. An ActionUsage is a kind of KerML Step (see [7.2.9](#)) and a Usage of an ActionDefinition. When an action usage executes, it transforms its input item usages to its output item usages. Action definition and action usages describe function-based behavior, which is a kind of causal chain behavior.

Action definitions and action usages follow the same patterns that apply to structural elements. Action definitions and action usages can be decomposed into lower-level action usages to create an action tree, and action usages can be referenced by other actions. In addition, an action definition can be specialized, and an action usage can be subsetted and redefined. This provides enhanced flexibility to modify a hierarchy of action usages to adapt to its context.

A top-level action has a lifetime with a start event and completion event. A child action usage can only execute when its parent action usage is executing.

An action usage can be a feature of a part definition or a part usage, which can perform an action either by reference to an action usage or by an owned action usage. Whether owned or referenced, the action usage that is performed can represent a top action in a hierarchy of action usages.

Control Flows and Control Nodes

Action usages can be connected by control flows and item flows. A basic control flow is a KerML Succession connector, which simply sequences the actions (see [7.2.8](#)). However, a control flow may also have a guard condition, meaning that the succession is only valid if the guard condition evaluates to *true*.

The sequencing of action usages may be further controlled using control nodes, which are special kinds of action usages that impose additional constraints on action sequencing. Control nodes are always connected to other actions usages by incoming and outgoing control flows. The kinds of control nodes include the following.

- A fork node has one incoming control flow and one or more outgoing control flows. The actions connected to the outgoing control flows cannot begin to execute until the action connected to the incoming control flow has completed.
- A join node has one or more incoming control flows and one outgoing control flow. The action connected to the outgoing control flow cannot begin to execute until all the actions connected to the incoming control flows have completed.
- A decision node has one incoming control flow and one or more outgoing control flows. Exactly one of the actions connected to an outgoing control flow can begin to execute after the action connected to the incoming control flow has completed. Which of the downstream actions is executed can be controlled by placing guards on the outgoing control flows.
- A merge node has one or more incoming control flows and one outgoing control flow. The action connected to the outgoing control flow cannot begin to execute until any one of the actions connected to an incoming control flow has completed.

Items Flows and Transfers

In addition to the sequencing of action usages, an output of an action usage can be connected to an input of another action usage using the concept of ItemFlow (see [7.2.10](#)). Each output parameter from one action can be connected to an input parameter of another action by an item flow. The item flow results in a transfer of values from the source output parameter to the target input parameter. A streaming flow is an item flow in which this transfer can be ongoing while both the source and target actions are executing. A succession flow is an item flow that imposes an additional succession constraint, that the transfer of values across the item flow cannot begin until the source action completes execution, and the target action cannot begin executing until the transfer has completed.

Transfers can also be performed using transfer actions. In this case, the source and target of the transfer do not have to be explicitly connected with a flow. Instead, the source of the transfer is specified using a send action from some source part or action, while the target is given by an accept action in some destination part or action (which may be the same as or different than the source). A send action includes an expression that is evaluated to provide the values to be transferred, and it specifies the destination to which those values are to be sent (possibly delegated through a port and across one or more interfaces -- see also [7.9](#) and [7.11](#) on interfaces between ports). An accept action specifies the type of values that can be received by the action. When a send action executing in the source is matched with a compatible accept action executing in the destination, then the transfer of values from the origin to the destination can be completed.

7.14.2 Actions Concrete Syntax

7.14.2.1 Actions Textual Notation

7.14.2.1.1 Action Definitions

```

ActionDefinition (m : Membership) : ActionDefinition =
    DefinitionPrefix(m) ( 'action' 'def' | 'activity' )
    ActionDefinitionDeclaration (this, m) ActionBody(this)

ActionDeclaration (a : ActionDefinition, m : Membership) =
    DefinitionDeclaration (a, m) ParameterList(a)?

ParameterList (t : Type) =
    '(' ( t.ownedRelationship += ParameterMember
        ( ',' t.ownedRelationship += ParameterMember ) * )? ')'

ParameterMember : ParameterMembership =
    ( direction = FeatureDirection )?
    ownedMemberParameter = ParameterDeclaration(this)

ParameterDeclaration(m : Membership) : Feature =
    ReferenceParameterDeclaration(m)
    | ItemParameterDeclaration(m)
    | PartParameterDeclaration(m)
    | ActionParameterDeclaration(m)
    | CalculationParameterDeclaration(m)
    | ConstraintParameterDeclaration(m)

ReferenceParameterDeclaration (m : Membership) : ReferenceUsage =
    'ref'? Identification(this, m) ParameterSpecializationPart(this)

ItemParameterDeclaration (m : Membership) : ItemUsage =
    'item' Identification(this, m) ParameterSpecializationPart(this)

PartParameterDeclaration (m : Membership) : PartUsage =
    'part' Identification(this, m) ParameterSpecializationPart(this)

ActionParameterDeclaration (m : Membership) : ActionUsage =
    'action' Identification(this, m) ParameterSpecializationPart(this)

CalculationParameterDeclaration (m : Membership) : CalculationUsage =
    'calc' Identification(this, m) ParameterSpecializationPart(this)

ConstraintParameterDeclaration (m : Membership) : ConstraintUsage =
    'constraint' Identification(this, m) ParameterSpecializationPart(this)

ParameterSpecializationPart (u : Usage) =
    ParameterSpecialization(u) * MultiplicityPart(u)? ParameterSpecialization(u) *

ParameterSpecialization (u : Usage) =
    TypedBy(u) | Subsets(u) | Redefines(u)

ActionBody (t : Type) =
    ';' | '{' ActionBodyItem(t) * '}'

ActionBodyItem (t : Type) =

```

```

        NonBehaviorBodyItem(t)
    | t.ownedRelationship += ActionBehaviorMember(t)
      ( t.ownedRelationship += ActionTargetSuccessionMember(t) ) *
    | t.ownedRelationship += GuardedSuccessionMember
    | t.ownedRelationship += PackageImport

NonBehaviorBodyItem (t : Type) =
    t.ownedRelationship += OwnedDocumentation
    | t.ownedRelationship += NestedDefinitionMember(t)
    | t.ownedRelationship += ParameterFlowUsageMember
    | t.ownedRelationship += VariantUsageMember
    | t.ownedRelationship += StructureUsageMember
    | t.ownedRelationship += IndividualUsageMember
    | t.ownedRelationship += IndividualSuccessionMember(t)

ActionBehaviorMember (t : Type) : FeatureMembership =
    BehaviorUsageMember
    | InitialNodeMember
    | ActionNodeMember
    | ActionBehaviorSuccessionMember(t)

ParameterFlowUsageMember : ParameterMembership =
    DefinitionMemberPrefix(this) direction = FeatureDirection
    ownedMemberParameter = FlowUsageElement(this)

InitialNodeMember : FeatureMembership =
    DefinitionMemberPrefix(this) 'first' memberFeature = [Qualified Name] ';'

ActionNodeSuccessionMember (t : Type) : FeatureMembership =
    s = SourceSuccessionMember(t)
    ( BehaviorUsageMember | ActionNodeMember )
    TargetEndFor(s.ownedMemberFeature, ownedMemberFeature)

ActionNodeMember : FeatureMembership =
    DefinitionMemberPrefix(this) ownedMemberFeature = ActionNode(this)

ActionTargetSuccessionMember (t : Type) : FeatureMembership =
    DefinitionMemberPrefix(this) ownedMemberFeature = ActionTargetSuccession(t) ';'

GuardedSuccessionMember : FeatureMembership =
    DefinitionMemberPrefix(this) ownedMemberFeature = GuardedSuccession(this) ';'

```

7.14.2.1.2 Action Usages

```
ActionUsage (m : Membership) : ActionUsage =
    UsagePrefix(this)? 'action'
    ActionUsageDeclaration(this, m) ActionBody(this)

ActionFlowUsage (m : Membership) : ActionUsage =
    UsagePrefix(this)? 'ref'? 'action'
    ActionUsageDeclaration(this, m) ActionBody(this)

ActionRefUsage (m : Membership) : ActionUsage =
    UsagePrefix(this)? ( 'ref' 'action' | isComposite ?= 'action' )
    ActionUsageDeclaration(this, m) ActionBody(this)

ActionUsageDeclaration (a : ActionUsage, m : Membership) : ActionUsage =
    UsageDeclaration(this, m) ( ValuePart(this) | ActionUsageParameterList(this) )?

PerformActionUsage (m : Membership) : PerformActionUsage =
    UsagePrefix(this)? 'perform'
    PerformActionUsageDeclaration (this, m) ActionBody(this)

PerformActionUsageDeclaration (a : PerformActionUsage, m : Membership) =
    ( ownedRelationship += OwnedSubsetting | 'action' Identification(a, m) )
    FeatureSpecializationPart(a)?
    ( ValuePart(a) | ActionUsageParameterList(a) )?

ActionUsageParameterList (f : Feature) =
    '(' ( f.ownedRelationship += ActionUsageParameterMember
        ( ',' f.ownedRelationship += ActionUsageParameterMember )* )? ')'

ActionUsageParameterMember : ParameterMembership =
    ( direction = FeatureDirection )?
    ownedMemberParameter = ActionUsageParameter(this)

ActionUsageParameter (m : Membership) : Usage =
    ParameterDeclaration(m) ValueOrFlowPart(this)?
```

7.14.2.1.3 Action Nodes

```
ActionNode (m : Membership) : ActionUsage =
    SendNode(m) | AcceptNode(m) | ControlNode(m)

AcceptNode (m : Membership) : AcceptActionUsage =
    UsagePrefix(m)? AcceptNodeDeclaration(this, m) ActionBody(this)

AcceptNodeDeclaration (a : AcceptActionUsage, m : Membership) =
    a.ownedRelationship += EmptyParameterMember
    ( 'action' UsageDeclaration(a, m) )?
    'accept' a.ownedRelationship += ItemFeatureMember

SendNode (m : Membership) : SendActionUsage =
    UsagePrefix(this)? SendNodeDeclaration(this, m) ActionBody(this)

SendNodeDeclaration (a : SendActionUsage, m : Membership) =
    ownedRelationship += EmptyParameterMember
    ownedRelationship += EmptyItemFeatureMember
    ( 'action' UsageDeclaration(a, m) )?
    'send' ownedRelationship += OwnedExpressionMember
    'to' ownedRelationship += OwnedExpressionMember

ControlNode (m : Membership) : ControlNode =
    MergeNode(m) | DecisionNode(m) | JoinNode(m) | ForkNode(m)

MergeNode (m : Membership) : MergeNode =
    UsagePrefix(this)? isComposite ?= 'merge' UsageDeclaration(this, m) ';'

DecisionNode (m : Membership) : DecisionNode =
    UsagePrefix(this)? isComposite ?= 'decide' UsageDeclaration(this, m) ';'

JoinNode (m : Membership) : JoinNode =
    UsagePrefix(this)? isComposite ?= 'join' UsageDeclaration(this, m) ';'

ForkNode (m : Membership) : ForkNode =
    UsagePrefix(this)? isComposite ?= 'fork' UsageDeclaration(this, m) ';'

EmptyParameterMember : FeatureMembership =
    ownedMemberFeature = EmptyParameter

EmptyParameter : ReferenceUsage :
    {}
```

7.14.2.1.4 Action Successions

```

ActionTargetSuccession (t : Type) : Feature =
    TargetSuccession(t) | GuardedTargetSuccession(t) | DefaultTargetSuccession(t)

TargetSuccessionMember (t : Type) : FeatureMembership =
    ownedMemberFeature = TargetSuccession(t)

TargetSuccession (t : Type) : Succession =
    ownedRelationship += SourceEndMemberFor(t)
    ownedRelationship += ConnectorEndMember

GuardedTargetSuccession (t : Type) : TransitionUsage =
    ownedRelationship += GuardExpressionMember
    'then' ownedRelationship += TargetSuccessionMember(t)

DefaultTargetSuccession (t : Type) : TransitionUsage =
    'else' ownedRelationship += TargetSuccessionMember(t)

GuardedSuccession (m : Membership) : TransitionUsage =
    'succession' ( UsageDeclaration(this, m) 'first' )?
    f = [Qualified Name]
    ownedRelationship += GuardExpressionMember
    'then' ownedRelationship += TransitionSuccessionMember(f)

```

7.14.2.2 Actions Graphical Notation

7.14.3 Actions Abstract Syntax

7.14.3.1 Overview

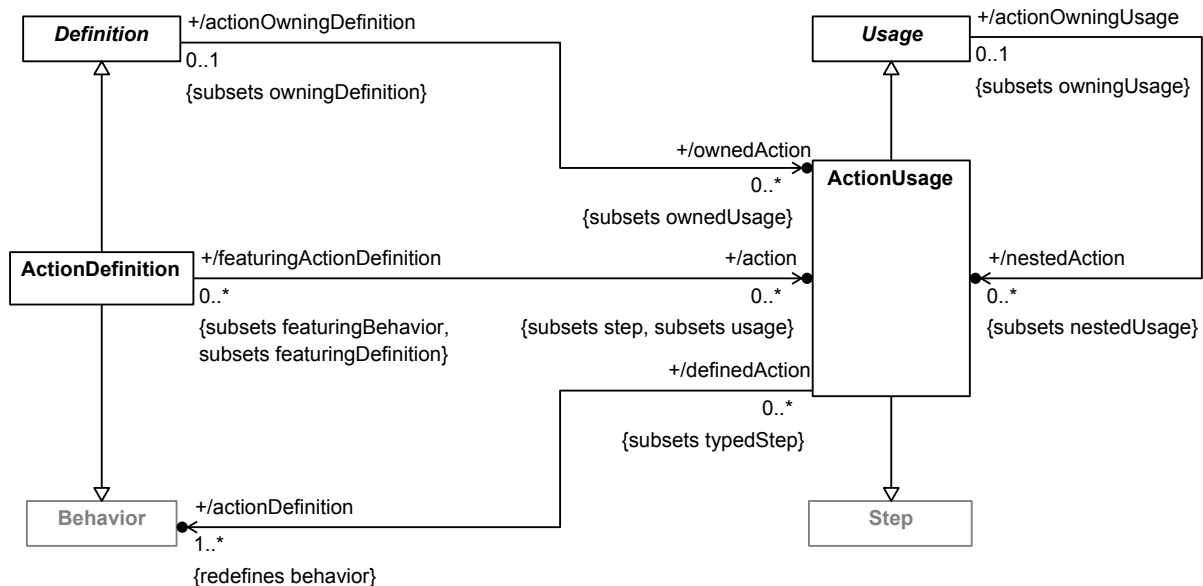


Figure 44. Action Definition and Usage

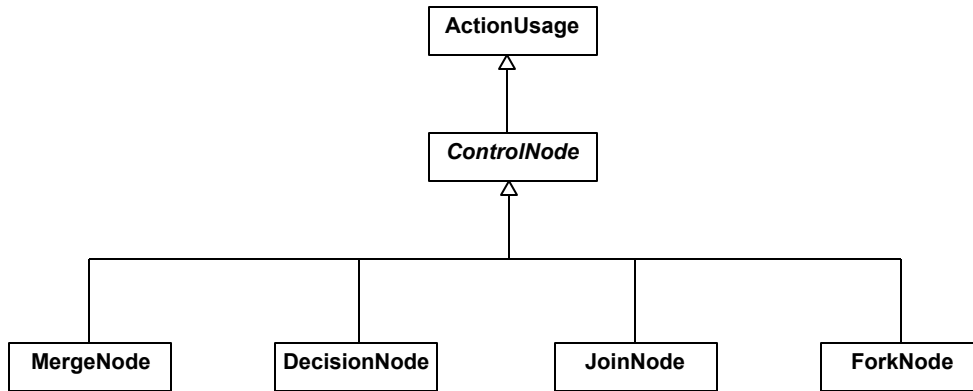


Figure 45. Control Nodes

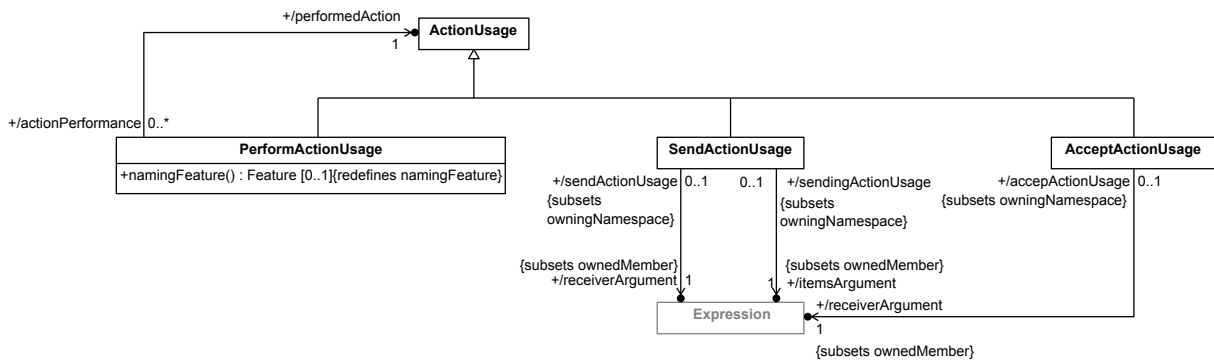


Figure 46. Perform, Send and Accept Actions

7.14.3.2 AcceptActionUsage

Description

An AcceptActionUsage is an ActionUsage that is typed, directly or indirectly, by the ActionDefinition AcceptAction from the Systems model library. It specifies the acceptance of an incomingTransfer from the Occurrence given by the result of its receiverArgument Expression. The payload of the accepted Transfer is output on its items parameter.

General Classes

ActionUsage

Attributes

/receiverArgument : Expression {subsets ownedMember}

An Expression whose result is bound to the receiver input parameter of this AcceptActionUsage.

Constraints

No constraints.

7.14.3.3 ActionDefinition

Description

An ActionDefinition is a Definition that is also a Behavior that defines an action performed by a system or part of a system.

An ActionDefinition must subclass, directly or indirectly, the base ActionDefinition Action from the Systems model library.

General Classes

Behavior
Definition

Attributes

/action : ActionUsage [0..*] {subsets step, usage}

The ActionUsages that are Steps in this Activity, which define the actions that specify the behavior of the Activity.

Operations

No operations.

Constraints

No constraints.

7.14.3.4 ActionUsage

Description

An ActionUsage is a Usage that is also a Step, and, so, is typed by a Behavior. Nominally, if the type is an ActionDefinition, an ActionUsage is a Usage of that ActionDefinition within a system. However, other kinds of kernel Behaviors are also allowed, to permit use of Behaviors from the Kernel Library.

An ActionUsage (other than a PerformActionUsage owned by a Part) must subset, directly or indirectly, either the base ActionUsage actions from the Systems model library, if it is not a composite feature, or the ActionUsage subactions inherited from its owner, if it is a composite feature.

General Classes

Usage
Step

Attributes

/actionDefinition : Behavior [1..*] {redefines behavior}

The Behaviors that are the types of this ActionUsage. Nominally, these would be ActionDefinitions, but other kinds of Kernel Behaviors are also allowed, to permit use of Behaviors from the Kernel Library.

Operations

No operations.

Constraints

No constraints.

7.14.3.5 ControlNode

Description

A ControlNode is an ActionUsage that does not have any inherent behavior but provides constraints on incoming and outgoing Succession Connectors that are used to control other Actions.

A ControlNode must be a composite owned feature of an ActionDefinition or ActionUsage, subsetting, directly or indirectly, the ActionUsage `Action::controls`. This implies that the ControlNode must be typed by ControlAction from the Systems model library, or a subtype of it.

All outgoing Successions from a ControlNode must have source multiplicity of 1..1. All incoming Succession must have target multiplicity of 1..1.

General Classes

ActionUsage

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

7.14.3.6 DecisionNode

Description

A DecisionNode is a ControlNode that makes a selection from its outgoing Successions. All outgoing Successions must be must have a target multiplicity of 0..1 and subset the Feature `DecisionAction::outgoingHBLink`. A DecisionNode may have at most one incoming Succession.

A DecisionNode must subset, directly or indirectly, the ActionUsage `Action::decisions`, implying that it is typed by DecisionAction from the Systems model library (or a subtype of it).

General Classes

ControlNode

Attributes

No attributes.

Operations

No operations.

Constraints

decisionNodeIncomingSuccession

A DecisionNode may have at most one incoming Succession Connector.

7.14.3.7 ForkNode

Description

A ForkNode is a ControlNode that must be followed by successor Actions as given by all its outgoing Successions. All outgoing Successions must have a target multiplicity of 1..1. A ForkNode may have at most one incoming Succession.

A ForkNode must subset, directly or indirectly, the ActionUsage `Action::forks`, implying that it is typed by ForkAction from the Systems model library (or a subtype of it).

General Classes

ControlNode

Attributes

No attributes.

Operations

No operations.

Constraints

forkNodeIncomingSuccession

A ForkNode may have at most one incoming Succession Connector.

7.14.3.8 JoinNode

Description

A JoinNode is a ControlNode that waits for the completion of all the predecessor Actions given by incoming Successions. All incoming Successions must have a source multiplicity of 1..1. A JoinNode may have at most one outgoing Succession.

A JoinNode must subset, directly or indirectly, the ActionUsage `Action::joins`, implying that it is typed by JoinAction from the Systems model library (or a subtype of it).

General Classes

ControlNode

Attributes

No attributes.

Operations

No operations.

Constraints

joinNodeOutgoingSuccession

A JoinNode may have at most one outgoing Succession Connector.

7.14.3.9 MergeNode

Description

A MergeNode is a ControlNode that asserts the merging of its incoming Successions. All incoming Successions must have a source multiplicity of 0..1 and subset the Feature `MergeAction::incomingHBLink`. A MergeNode may have at most one outgoing Succession.

A MergeNode must subset, directly or indirectly, the ActionUsage `Action::merges`, implying that it is typed by MergeAction from the Systems model library (or a subtype of it).

General Classes

ControlNode

Attributes

No attributes.

Operations

No operations.

Constraints

mergeNodeOutgoingSuccession

A MergeNode may have at most one outgoing Succession Connector.

7.14.3.10 PerformActionUsage

Description

A PerformActionUsage is an ActionUsage that represents the performance of an ActionUsage. The ActionUsage to be performed (which may be the PerformActionUsage itself) is related to the PerformActionUsage by a Subsetting relationship.

If the PerformActionUsage is owned by a Part, then it also subsets the performedAction property of that Part (as defined in the library model for Part), otherwise it subsets either `actions` or `subactions`, as required for a regular ActionUsage.

General Classes

ActionUsage

Attributes

/performedAction : ActionUsage

The ActionUsage to be performed by this PerformedActionUsage. It is the subsettedFeature of the first owned Subsetting Relationship of the PerformedActionUsage.

Constraints

No constraints.

7.14.3.11 SendActionUsage

Description

A SendActionUsage is an ActionUsage that is typed, directly or indirectly, by the ActionDefinition SendAction from the Systems model library. It specifies the sending of a payload given by the result of its itemsArgument Expression via a Transfer that becomes an incomingTransfer of the Occurrence given by the result of its receiverArgument Expression.

General Classes

ActionUsage

Attributes

/itemsArgument : Expression {subsets ownedMember}

An Expression whose result is bound to the items input parameter of this SendActionUsage.

/receiverArgument : Expression {subsets ownedMember}

An Expression whose result is bound to the receiver input parameter of this SendActionUsage.

Constraints

No constraints.

7.14.3.12 TransferActionUsage

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

7.14.4 Actions Semantics

7.15 States

7.15.1 States Overview

States

A `StateDefinition` is a kind of `ActionDefinition` that defines the conditions under which other actions can execute. A `StateUsage` is a usage of a state definition. State definition and usages are used to describe state-based behavior, where the execution of any particular state is triggered by events.

A state can contain action usages which can only execute while the state is activated. An *entry* action begins execution when the state is activated. A *do* action begins execution after the entry action completes. An *exit* action begins execution after completion of the do action. All actions must execute prior to deactivation of the state.

State definitions and state usages follow the same patterns that apply to structural elements (see [7.4](#)). States can be decomposed into lower-level states to create a hierarchy of state usages, and states can be referenced by other states. In addition, a state definition can be specialized, and a state usage can be subsetted and redefined. This provides enhanced flexibility to modify a state hierarchy to adapt to its context.

A state has a lifetime with a start event and completion event. A child state usage can only be activated when its parent state is activated.

A state usage can be a feature of a part definition or a part usage, which can exhibit a state by referencing the state usage or by containing an owned state usage. Whether owned or referenced, the state usage that the part exhibits can represent a top state in a hierarchy of state usages.

Transitions

State usages can be connected by transition usages, which can activate and deactivate a state usage. The triggering of a transition usage from its source state usage to its target state usage deactivates the source state and activates the target state. If two child state usages are connected by a transition usage, only one of the child states can be activated at any time. If a transition usage does not connect a child state usage to another child state usage, the child state is activated whenever the parent state usage is activated.

A transition usage can be triggered by a triggering event such as when a signal is received, when a change in attribute value is determined, or when a do behavior of a source state of the transition completes its execution. The transition usage can contain a guard condition that must evaluate to true for the transition to occur. In addition, a transition usage may specify an action usage that can begin execution if the transition is triggered, after the source state is deactivated, and must complete execution before the target state is activated.

Submission Note. The transition syntax currently only supports triggering a transition on receipt of a signal. Support for triggering transitions by change and time events will be included in the revised submission. Consideration will also be given to allowing the declaration of transition definitions.

7.15.2 States Concrete Syntax

7.15.2.1 States Textual Notation

7.15.2.1.1 State Definitions


```

StateDefinition (m : Membership) : StateDefinition =
    DefinitionPrefix(this)? 'state' 'def'
    ActionDeclaration (this, m) StateBody(this)

StateBody (t : Type) =
    ';'
    | '{' EntryActionMember(t)?
        ( t.ownedRelationship += DoActionMember )?
        ( t.ownedRelationship += ExitActionMember )?
        StateBodyItem(t)* '}'

EntryActionMember (t : Type) : StateSubactionMembership =
    DefinitionMemberPrefix(this) kind = 'entry' ownedMemberFeature = StateActionUsage
    { t.ownedRelationship += this }
    ( t.ownedRelationship += EntryTransitionMember(this) ) *

DoActionMember : StateSubactionMembership =
    DefinitionMemberPrefix(this) kind = 'do' ownedMemberFeature = StateActionUsage

ExitActionMember : StateSubactionMembership =
    DefinitionMemberPrefix(this) kind = 'exit' ownedMemberFeature = StateActionUsage

EntryTransitionMember (f : Feature) : FeatureMembership :
    DefinitionMemberPrefix(this)
    ( ownedMemberFeature = GuardedTargetSuccession(f)
    | 'then' ownedMemberFeature_comp = TargetTransitionSuccession(f)
    ) ';'

StateActionUsage (m : Membership) : ActionUsage :
    EmptyActionUsage ';'
    | StatePerformActionUsage(m)
    | StateAcceptActionUsage(m)
    | StateSendActionUsage(m)

EmptyActionUsage : ActionUsage =
    {}

StatePerformActionUsage (m : Membership) : PerformActionUsage =
    PerformActionUsageDeclaration(this, m) ActionBody(this)

StateAcceptActionUsage (m : Membership) : AcceptAction =
    AcceptNodeDeclaration(this, m) ActionBody(this)

StateSendActionUsage (m : Membership) : SendAction
    SendNodeDeclaration(this, m) ActionBody(this)

StateBodyItem (t : Type) =
    NonBehaviorBodyItem(t)
    | t.ownedRelationship = BehaviorUsageMember
        ( t.ownedRelationship = TargetTransitionUsageMember(t) ) *
    | t.ownedRelationship += TransitionUsageMember

```

```

TransitionUsageMember : FeatureMembership :
    DefinitionMemberPrefix(this) ownedMemberFeature = TransitionUsage(this) ';'

TargetTransitionUsageMember (t : Type) : FeatureMembership =
    DefinitionMemberPrefix(this) ownedMemberFeature = TargetTransitionUsage(t) ';'

```

7.15.2.1.2 State Usages

```

StateUsage (m : Membership) : StateUsage =
    UsagePrefix(this)? 'state'
    ActionUsageDeclaration(this, m) StateBody(this)

StateFlowUsage (m : Membership) : StateUsage =
    UsagePrefix(this)? 'ref'? 'state'
    ActionUsageDeclaration(this, m) StateBody(this)

StateRefUsage (m : Membership) : ActionUsage =
    UsagePrefix(this)? ( 'ref' 'state' | isComposite ?= 'state' )
    ActionUsageDeclaration(this, m) StateBody(this)

ExhibitStateUsage (m : Membership) : ExhibitStateUsage =
    UsagePrefix(this)? 'exhibit'
    ( ownedRelationship += OwnedSubsetting | 'state' Identification(this, m) )
    FeatureSpecializationPart(this)?
    ( ValuePart(this) | ActionUsageParameterList(this) )?
    StateBody(this)

```

7.15.2.1.3 Transition Usages

```
TransitionUsage (m : Membership) : TransitionUsage =
  'transition' ( UsageDeclaration(this, m) 'first' )?
  f = [QualifiedName]
  ( ownedRelationship += TriggerActionMember )?
  ( ownedRelationship += GuardExpressionMember )?
  ( ownedRelationship += EffectBehaviorMember )?
  'then' ownedRelationship += TransitionSuccessionMember(f)

TargetTransitionUsage (t : Type) : TransitionUsage =
  ( ownedRelationship += TriggerActionMember )?
  ( ownedRelationship += GuardExpressionMember )?
  ( ownedRelationship += EffectBehaviorMember )?
  'then' ownedRelationship += TargetSuccessionMember(t)

TriggerActionMember : TransitionFeatureMembership =
  'accept' { kind = 'trigger' } ownedMemberFeature = TriggerAction

TriggerAction : AcceptActionUsage =
  ownedRelationship += EmptyParameterMember
  ownedRelationship += ItemFeatureMember

GuardExpressionMember : TransitionFeatureMembership =
  'if' { kind = 'guard' } ownedMemberFeature = OwnedExpression

EffectBehaviorMember : TransitionFeatureMembership =
  'do' { kind = 'effect' } ownedMemberFeature = EffectBehaviorUsage

EffectBehaviorUsage : ActionUsage =
  EmptyActionUsage
  | TransitionPerformActionUsage(m)
  | TransitionAcceptActionUsage(m)
  | TransitionSendActionUsage(m)

TransitionPerformActionUsage (m : Membership) : PerformActionUsage =
  PerformActionDeclaration(this, m) ( '{' ActionBodyItem* '}' )?

TransitionAcceptActionUsage (m : Membership) : AcceptActionUsage =
  AcceptNodeDeclaration(this, m) ( '{' ActionBodyItem* '}' )?

TransitionSendActionUsage (m : Membership) : SendActionUsage =
  SendNodeDeclaration(this, m) ( '{' ActionBodyItem* '}' )?

TransitionSuccessionMember (f : Feature) : FeatureMembership =
  ownedMemberFeature = TransitionSuccession(f)

TransitionSuccession (f : Feature) : Succession =
  ownedRelationship += ConnectorEndMemberFor(f)
  ownedRelationship += ConnectorEndMember
```

7.15.2.2 States Graphical Notation

7.15.3 States Abstract Syntax

7.15.3.1 Overview

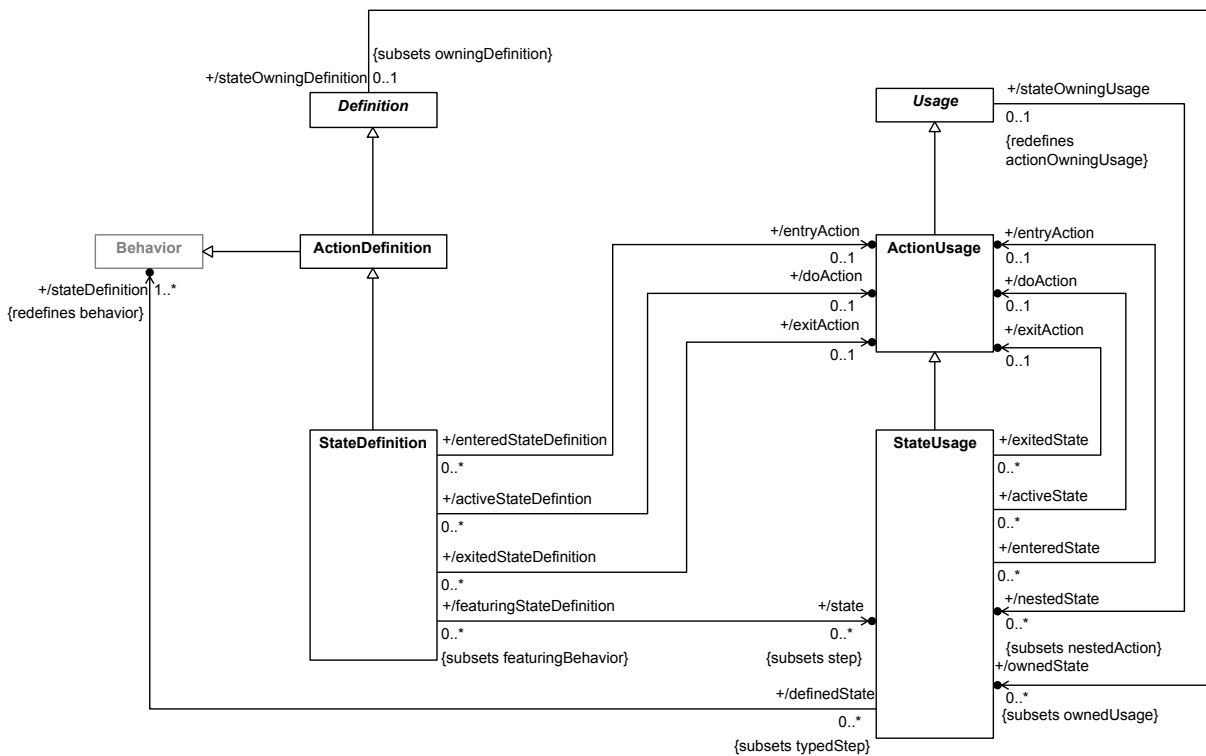


Figure 47. State Definition and Usage

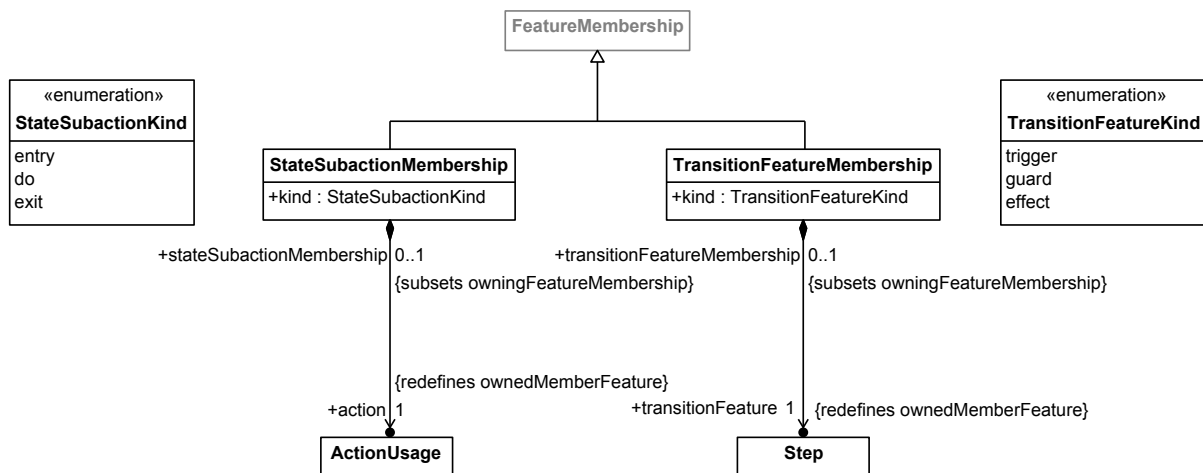


Figure 48. State Membership

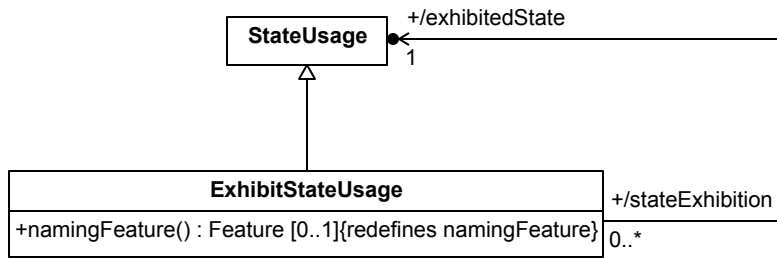


Figure 49. State Exhibition

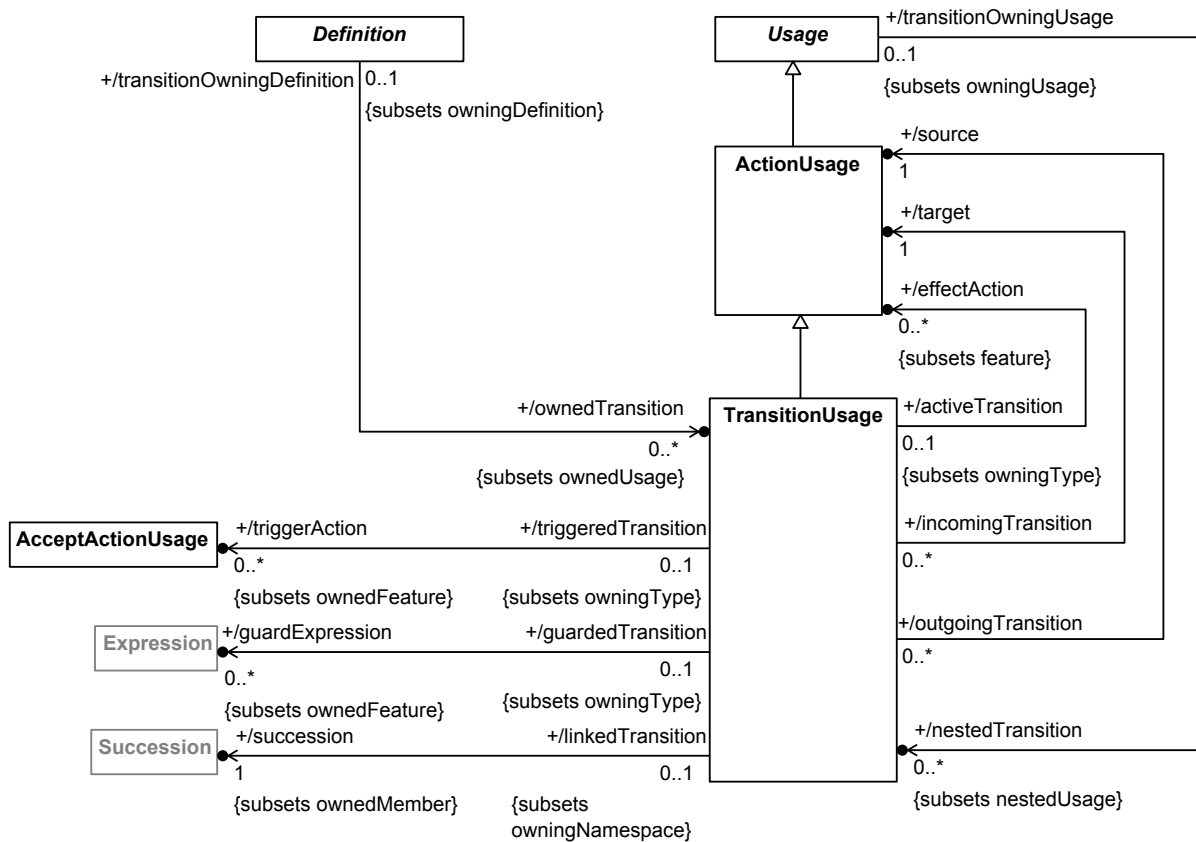


Figure 50. Transition Usage

7.15.3.2 ExhibitStateUsage

Description

An ExhibitStateUsage is a StateUsage that represents the exhibiting of a StateUsage. The StateUsage to be exhibited (which may be the ExhibitStateUsage itself) is related to the ExhibitStateUsage by a Subsetting Relationship.

If the ExhibitStateUsage is owned by a Part, then it also subsets the exhibitedStates property of that Part (as defined in the library model for Part), otherwise it subsets either states or substates, as required for a regular StateUsage.

General Classes

StateUsage

Attributes

/exhibitedState : StateUsage

The StateUsage to be exhibited by the ExhibitStateUsage. It is the `subsettedFeature` of the first owned Subsetting Relationship of the ExhibitStateUsage.

Operations

namingFeature() : Feature [0..1]

The naming Feature of an ExhibitStateUsage is its `exhibitedState`.

body: `exhibitedState`

Constraints

No constraints.

7.15.3.3 StateSubactionKind

Description

A StateSubactionKind indicates whether the `action` of a StateSubactionMembership is an entry, do or exit action.

General Classes

No general classes.

Literal Values

do

Indicates that a subaction of a StateUsage is a do action.

entry

Indicates that a subaction of a StateUsage is an entry action.

exit

Indicates that a subaction of a StateUsage is an exit action.

7.15.3.4 StateSubactionMembership

Description

A StateSubactionMembership is a FeatureMembership for an entry, do or exit ActionUsage of a StateDefinition or StateUsage. The `ownedMemberFeature` of a StateSubactionMembership must be an ActionUsage.

General Classes

FeatureMembership

Attributes

`action : ActionUsage {redefines ownedMemberFeature}`

The `ActionUsage` that is the `ownedMemberFeature` of this `StateSubactionMembership`.

`kind : StateSubactionKind`

Whether this `StateSubactionMembership` is for an entry, do or exit `ActionUsage`.

Operations

No operations.

Constraints

No constraints.

7.15.3.5 StateDefinition

Description

A `StateDefinition` is the Definition of the Behavior of a system or part of a system in a certain state condition.

A State Definition must subclass, directly or indirectly, the base `StateDefinition StateAction` from the Systems model library.

A `StateDefinition` may be related to up to three of its ownedFeatures by `StateBehaviorMembership` Relationships, all of different `kinds`, corresponding to the entry, do and exit actions of the `StateDefinition`.

General Classes

`ActionDefinition`

Attributes

`/doAction : ActionUsage [0..1]`

The `ActionUsage` of this `StateDefinition` to be performed while in the state defined by the `StateDefinition`. This is derived as the owned `ActionUsage` related to the `StateDefinition` by a `StateSubactionMembership` with `kind = do`.

`/entryAction : ActionUsage [0..1]`

The `ActionUsage` of this `StateDefinition` to be performed on entry to the state defined by the `StateDefinition`. This is derived as the owned `ActionUsage` related to the `StateDefinition` by a `StateSubactionMembership` with `kind = entry`.

`/exitAction : ActionUsage [0..1]`

The `ActionUsage` of this `StateDefinition` to be performed on exit from the state defined by the `StateDefinition`. This is derived as the owned `ActionUsage` related to the `StateDefinition` by a `StateSubactionMembership` with `kind = exit`.

`/state : StateUsage [0..*] {subsets step}`

The StateUsages that are the steps of the StateDefinition, which specify the discrete states in the Behavior defined by the StateDefinition.

Operations

No operations.

Constraints

No constraints.

7.15.3.6 StateUsage

Description

A StateUsage is an ActionUsage that is nominally the Usage of a StateDefinition. However, other kinds of kernel Behaviors are also allowed as types, to permit use of Behaviors from the Kernel Library.

A StateUsage (other than an ExhibitStateUsage owned by a Part) must subset, directly or indirectly, either the base StateUsage stateActions from the Systems model library, if it is not a composite feature, or the StateUsage substates inherited from its owner, if it is a composite feature.

A StateUsage may be related to up to three of its ownedFeatures by StateBehaviorMembership Relationships, all of different kinds, corresponding to the entry, do and exit actions of the StateUsage.

General Classes

ActionUsage

Attributes

/doAction : ActionUsage [0..1]

The ActionUsage of this StateUsage to be performed while in the state specified by the StateUsage. This is derived as the owned ActionUsage related to the StateDefinition by a StateSubactionMembership with kind = do.

/entryAction : ActionUsage [0..1]

The ActionUsage of this StateUsage to be performed on entry to the state specified by the StateUsage. This is derived as the owned ActionUsage related to the StateDefinition by a StateSubactionMembership with kind = entry.

/exitAction : ActionUsage [0..1]

The ActionUsage of this StateUsage to be performed on exit from the state specified by the StateUsage. This is derived as the owned ActionUsage related to the StateDefinition by a StateSubactionMembership with kind = exit.

/stateDefinition : Behavior [1..*] {redefines behavior}

The Behaviors that are the types of this StateUsage. Nominally, these would be StateDefinitions, but non-Activity Behaviors are also allowed, to permit use of Behaviors from the Kernel Library.

Operations

No operations.

Constraints

No constraints.

7.15.3.7 TransitionFeatureKind

Description

A TransitionActionKind indicates whether the `transitionFeature` of a TransitionFeatureMembership is a trigger, guard or effect.

General Classes

No general classes.

Literal Values

effect

Indicates that a member Step of a TransitionUsage represents an effect.

guard

Indicates that a member Expression of a TransitionUsage represents a guard.

trigger

Indicates that a member Transfer of a TransitionUsage represents a trigger.

7.15.3.8 TransitionFeatureMembership

Description

A TransitionFeatureMembership is a FeatureMembership for a trigger, guard or effect of a TransitionUsage. The `ownedMemberFeature` must be a Step. For a trigger, the `ownedMemberFeature` must more specifically be a Transfer, while for a guard it must be an Expression with a result type of Boolean.

General Classes

FeatureMembership

Attributes

kind : TransitionFeatureKind

Whether this TransitionFeatureMembership is for a trigger, guard or effect.

transitionFeature : Step {redefines ownedMemberFeature}

The Step that is the `ownedMemberFeature` of this TransitionFeatureMembership.

Operations

No operations.

Constraints

No constraints.

7.15.3.9 TransitionUsage

Description

A TransitionUsage is an ActionUsage that is a behavioral Step representing a transition between ActionUsages or StateUsages.

A TransitionUsage must subset, directly or indirectly, the base TransitionUsage `transitionActions`, if it is not a composite feature, or the TransitionUsage `subtransitions` inherited from its owner, if it is a composite feature.

A TransitionUsage may be related to some of its `ownedFeatures` using TransitionFeatureMembership Relationships, corresponding to the triggers, guards and effects of the TransitionUsage.

General Classes

ActionUsage

Attributes

`/effectAction : ActionUsage [0..*] {subsets feature}`

The ActionUsages that define the effects of this TransitionUsage, derived as the `ownedFeatures` of this TransitionUsage related to it by a TransitionFeatureMembership with `kind = effect`.

`/guardExpression : Expression [0..*] {subsets ownedFeature}`

The Expressions that define the guards of this TransitionUsage, derived as the `ownedFeatures` of this TransitionUsage related to it by a TransitionFeatureMembership with `kind = guard`.

`/source : ActionUsage`

The source ActionUsage of this TransitionUsage, derived as the `source` of the `succession` for the TransitionUsage.

`/succession : Succession {subsets ownedMember}`

The Succession that is the `ownedFeature` of this TransitionUsage that redefines `TransitionPerformance::transitionLink`.

`/target : ActionUsage`

The target ActionUsage of this TransitionUsage, derived as the `target` of the `succession` for the TransitionUsage.

`/triggerAction : AcceptActionUsage [0..*] {subsets ownedFeature}`

The `AcceptActionUsages` that define the triggers of this `TransitionUsage`, derived as the `ownedFeatures` of this `TransitionUsage` related to it by a `TransitionFeatureMembership` with `kind = trigger`.

Operations

No operations.

Constraints

No constraints.

7.15.4 States Semantics

7.16 Calculations

7.16.1 Calculations Overview

A `CalculationDefinition` is a kind of `ActionDefinition` (see [7.14](#)) one of whose parameters is always a designated output parameter called the result. The calculation definition specifies a reusable computation that returns the result. A `CalculationUsage` is a usage of a calculation definition.

Calculations are often used to define mathematical functions that have multiple inputs and return one output. The inputs are ordered in a parameter list, and the output is designated as a result. Such a calculation should be stateless in that a set of inputs always produce the same output. The calculation definition and usage can contain one or more equations that define the computation to be carried, but must return a single result.

A `CalculationDefinition` is also a KerML Function and a `CalculationUsage` is a KerML Expression (see [7.2.11](#)). This allows a calculation definition to also be invoked using the notation of an invocation expression (see [7.2.12](#)).

7.16.2 Calculations Concrete Syntax

7.16.2.1 Calculations Textual Notation

7.16.2.1.1 Calculation Definitions

```
CalculationDefinition (m : Membership) : CalculationDefinition =
    DefinitionPrefix(this)? 'calc' 'def' DefinitionDeclaration(this, m)
    ( ParameterList(this) ReturnParameterPart(this)? )?
    ( CalculationBody(this)
    | '=' ownedRelationship += ResultExpressionMember ';'
    )

CalculationDeclaration (c : CalculationDefinition, m : Membership) =
    DefinitionDeclaration(this, m) ( ParameterList(this) ReturnParameterPart(this)? )?

ReturnParameterPart (t : Type) =
    t.ownedRelationship += ReturnParameterMember

ReturnParameterMember : ReturnParameterMembership =
    'return'? ownedMemberParameter = ParameterDeclaration(this)

CalculationBody (t : Type) =
    ';'
    | '{' CalculationBodyItem(t)*
      ( t.ownedRelationship += ResultExpressionMember )?
    '}'

CalculationBodyItem (t : Type) =
    ActionBodyItem(t)
    | t.ownedRelationship += ReturnParameterFlowUsageMember

ReturnParameterFlowUsageMember : ReturnParameterMembership =
    DefinitionMemberPrefix(this)? 'return' ownedMemberParameter = FlowUsageElement

ResultExpressionMember : ResultExpressionMembership =
    DefinitionMemberPrefix(this)? ownedResultExpression = OwnedExpression
```

7.16.2.1.2 Calculation Usages

```

CalculationUsage (m : Membership) : CalculationUsage =
  UsagePrefix? 'calc'
  CalculationUsageDeclaration(this, m) CalculationBody(this)

CalculationFlowUsage (m : Membership) : CalculationUsage =
  UsagePrefix? 'ref'? 'calc'
  CalculationUsageDeclaration(this, m) CalculationBody(this)

CalculationRefUsage (m : Membership) : CalculationUsage =
  UsagePrefix? ( 'ref' 'calc' | isComposite ?= 'calc' )
  CalculationUsageDeclaration(this, m) CalculationBody(this)

CalculationUsageDeclaration (u : Usage, m : Membership) =
  UsageDeclaration(this, m) CalculationParameterPart(u)?

CalculationParameterPart (u : Usage) =
  ValuePart(u)
  | ActionUsageParameterList(u)
  ( ownedFeatureMembership += CalculationReturnParameterMember )?

CalculationReturnParameterMember : ReturnParameterMembership =
  'return'? ownedMemberParameter = ActionUsageParameter(this)

```

7.16.2.2 Calculations Graphical Notation

7.16.3 Calculations Abstract Syntax

7.16.3.1 Overview

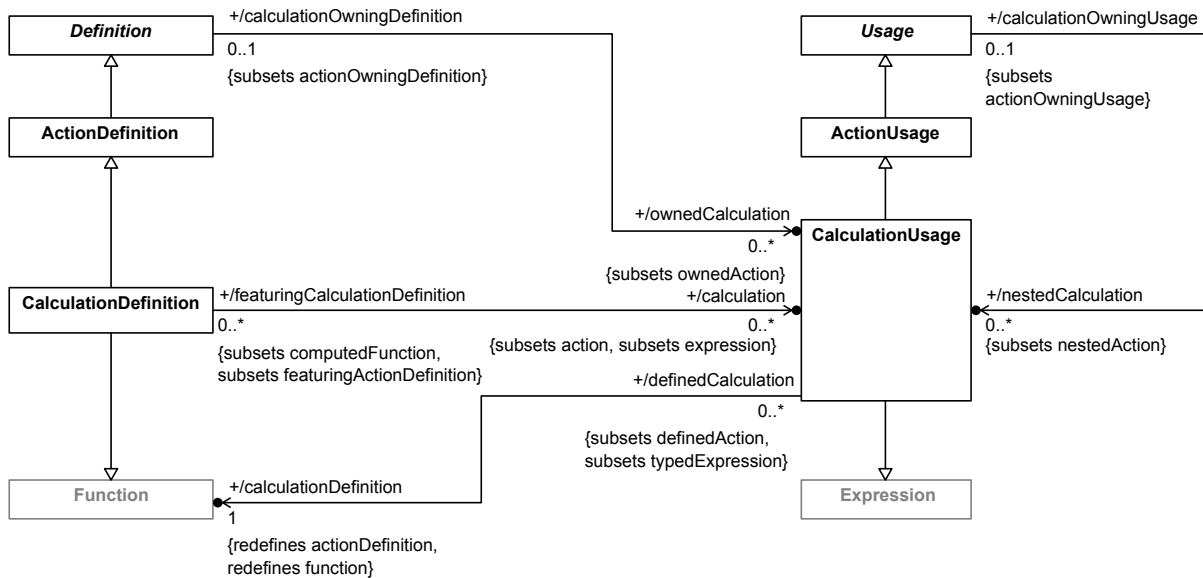


Figure 51. Calculation Definition and Usage

7.16.3.2 CalculationDefinition

Description

A CalculationDefinition is an ActionDefinition that also defines a Function producing a result.

A CalculationDefinition must subclass, directly or indirectly, the base CalculationDefinition Calculation from the Systems model library.

General Classes

ActionDefinition
Function

Attributes

/calculation : CalculationUsage [0..*] {subsets action, expression}

The CalculationUsage that are actions in this CalculationDefinition.

Operations

No operations.

Constraints

No constraints.

7.16.3.3 CalculationUsage

Description

A CalculationUsage is an ActionUsage that is also an Expression, and, so, is typed by a Function. Nominally, if the type is a CalculationDefinition, a CalculationUsage is a Usage of that CalculationDefinition within a system. However, other kinds of kernel Functions are also allowed, to permit use of Functions from the Kernel Library.

A CalculationUsage must subset, directly or indirectly, either the base CalculationUsage calculations from the Systems model library, if it is not a composite feature, or the CalculationUsage subcalculations inherited from its owner, if it is a composite feature.

General Classes

Expression
ActionUsage

Attributes

/calculationDefinition : Function {redefines function, actionDefinition}

The Function that is the type of this CalculationUsage. Nominally, this would be a CalculationDefinition, but a kernel Function is also allowed, to permit use of Functions from the Kernel Library.

Operations

No operations.

Constraints

No constraints.

7.16.4 Calculations Semantics

7.17 Constraints

7.17.1 Constraints Overview

A `ConstraintDefinition` is a KerML Predicate (see [7.2.11](#)) and a kind of Definition (see [7.4](#)). A constraint definition is a logical predicate that is a parameterized Boolean-valued expression used to constrain features. A simple example is $\{x < y\}$, which is a Boolean-valued expression in which the parameters x and y are related by the relational operator $<$. Such an expression evaluates to either true or false depending on the values of its parameters. For the constraint $\{x < y\}$, if x is 3 and y is 5, then the Boolean expression evaluates to true. In the general case, the expression used to define a constraint can be arbitrarily complicated, as long as the overall expression returns a Boolean value.

A `ConstraintUsage` is a kind of Usage element. A constraint usage is a usage of a constraint definition. The parameters of a constraint usage may be bound to specific features whose values can be constrained by the constraint expression. For the constraint expression $\{x < y\}$, the constraint usage may bind x to the diameter of a bolt and bind y to the diameter of a hole that the bolt must fit into. This constraint can then be evaluated to be true or false.

For a given set of parameter values, a constraint usage is satisfied if its expression evaluates to true and *violated* otherwise. In general, a constraint may be satisfied sometimes and violated other times. However, a constraint usage can also be *asserted* to be true, which requires that the assert constraint *always* be satisfied for the model to be valid. Constraints associated with the laws of physics, for example, should be asserted to be true, because they cannot be violated in any valid model of the real world. However, the constraint $\{\text{fuel} > 0\}$ may be evaluated to be false if the fuel equals zero $\{\text{fuel} == 0\}$, but this is still a valid model.

7.17.2 Constraints Concrete Syntax

7.17.2.1 Constraints Textual Notation

```

ConstraintDefinition (m : Membership) : ConstraintDefinition =
  DefinitionPrefix(this)? 'constraint' 'def'
  ConstraintDeclaration(this, m) CalculationBody(this)

ConstraintDeclaration (c : ConstraintDefinition, m : Membership) =
  DefinitionDeclaration(this, m) ParameterList(this)?

ConstraintUsage (m : Membership) : ConstraintUsage =
  UsagePrefix(this)? 'constraint'
  CalculationDeclaration(this, m) CalculationBody(this)

ConstraintFlowUsage (m : Membership) : ConstraintUsage =
  UsagePrefix(this)? 'ref'? 'constraint'
  CalculationDeclaration(this, m) CalculationBody(this)

ConstraintRefUsage (m : Membership) : ConstraintUsage =
  UsagePrefix(this)? ( 'ref' 'constraint' | isComposite ?= 'constraint' )
  CalculationDeclaration(this, m) CalculationBody(this)

AssertConstraintUsage (m : Membership) : AssertConstraintUsage =
  UsagePrefix(this)? 'assert'
  ( ownedRelationship += OwnedSubsetting | 'constraint' Identification(this, m) )
  FeatureSpecializationPart(this)?
  CalculationParameterPart(this) CalculationBody(this)

```

7.17.2.2 Constraints Graphical Notation

7.17.3 Constraints Abstract Syntax

7.17.3.1 Overview

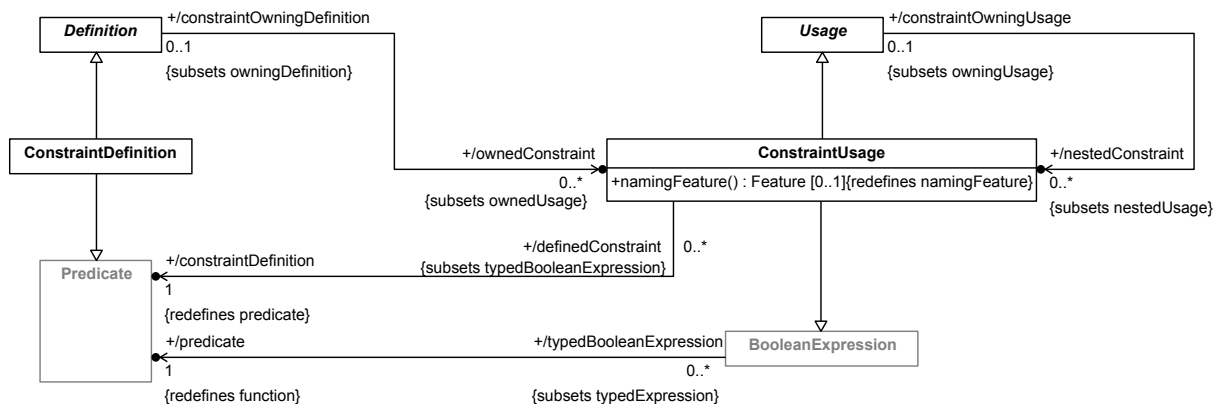


Figure 52. Constraint Definition and Usage

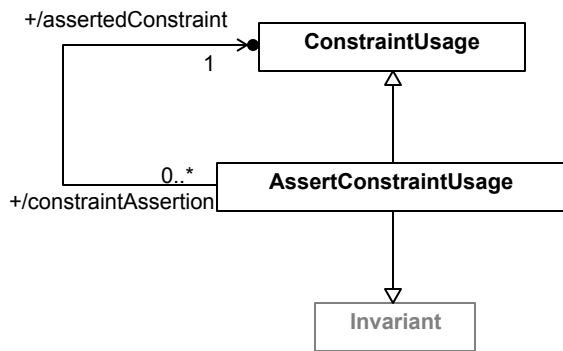


Figure 53. Constraint Assertion

7.17.3.2 AssertConstraintUsage

Description

An AssertConstraintUsage is a ConstraintUsage that is also an Invariant and, so, is asserted to be true. The asserted ConstraintUsage (which may be the AssertConstraintUsage itself) is related to the AssertConstraintUsage by a Subsetting relationship.

If the AssertConstraintUsage is owned by a Part, then it also subsets the `assertedConstraints` property of that Part (as defined in the library model for Part), otherwise it subsets `constraintChecks`, as required for a regular ConstraintUsage.

General Classes

Invariant
ConstraintUsage

Attributes

`/assertedConstraint` : ConstraintUsage

The ConstraintUsage to be performed by the AssertConstraintUsage. It is the `subsettingFeature` of the first owned Subsetting Relationship of the AssertConstraintUsage.

Operations

No operations.

Constraints

No constraints.

7.17.3.3 ConstraintDefinition

Description

A ConstraintDefinition is a Definition that is also a Predicate that defines a constraint that may be asserted to hold on a system or part of a system.

A ConstraintDefinition must subclass, directly or indirectly, the base ConstraintDefinition ConstraintCheck from the Systems model library.

General Classes

Definition
Predicate

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

7.17.3.4 ConstraintUsage

Description

A ConstraintUsage is a Usage that is also a BooleanExpression, and, so, is typed by a Predicate. Nominally, if the type is a ConstraintDefinition, a ConstraintUsage is a Usage of that ConstraintDefinition. However, other kinds of kernel Predicates are also allowed, to permit use of Predicates from the Kernel Library.

A ConstraintUsage (other than an AssertConstraintUsage owned by a Part) must subset, directly or indirectly, the base ConstraintUsage constraintChecks from the Systems model library.

General Classes

Usage
BooleanExpression

Attributes

/constraintDefinition : Predicate {redefines predicate}

The (single) Predicate the is the type of this Constraint Usage. Nominally, this will be ConstraintDefinition, but non-ConstraintDefinition Predicates are also allowed, to permit use of Predicates from the Kernel Library.

Operations

namingFeature() : Feature [0..1]

If this ConstraintUsage is an assumedConstraint or requiredConstraint of a RequirementUsage, then its naming Feature is the first subsetted Feature that is not a default subsetting from the model library, if any.

Constraints

No constraints.

7.17.4 Constraints Semantics

7.18 Requirements

7.18.1 Requirements Overview

Requirements

A requirement specifies stakeholder-imposed constraints that a design solution must satisfy to be a valid solution. A RequirementDefinition is a kind of ConstraintDefinition. The requirement definition can contain features that include a combination of text statements (also known as "shall" statements), and more formally specified constraints with constraint expressions. A constraint expressions generally increases the precision of the requirement, and facilitates automated evaluation.

A RequirementUsage is a kind of ConstraintUsage. A requirement usage is a usage of a requirement definition in some context. The context for multiple requirements can be provided by a package, another requirement, or a part. A design solution must satisfy the requirement and all of its member requirements and constraints to be a valid solution.

Like any usage element, the features of the requirement usage can redefine the features of the requirement definition. For example, a requirement definition called MaximumMass may include the constraint {massActual <= massRequired}, and the attributes for massActual and massRequired. The value of the attribute massRequired may not be specified for the requirement definition. The requirement usage called maximumVehicleMass is defined by MaximumMass, and can include the value of the massRequired to be 2000 kilograms. In this way, the requirement definition serves as a requirement template that can be reused and tailored to each context of use.

The top-level requirement can compose a hierarchy of requirement usages, where each nested requirement usage can contain further nested requirement usages. A requirement definition or usage can own zero or more requirement usages, or can contain references to other requirement usages. For a requirement usage to be satisfied, all of its owned and referenced requirements must be satisfied.

A requirement definition and requirement usage have a subject, which if not specified, is the most general thing. A requirement usage can only be satisfied by an entity that is a subset of the subject. For example, if the subject is of type Vehicle, then a standard vehicle model or sports vehicle model can satisfy the requirement, as long as the type of both vehicles is a subclass of the subject Vehicle. The subject type can be restricted to certain kinds of definition elements if it is desired to constrain what kind of entity can satisfy the requirement. For example, the subject type can be restricted to be an action definition, if it is desired to constrain the requirement to be satisfied by action usages. A satisfy relationship between an entity that satisfies a requirement and the requirement must always share a common context with the satisfying entity and the requirement.

Since a requirement is a kind of constraint, a requirement can be evaluated to be true or false. A requirement is satisfied when it is evaluated to be true. A satisfy relationship is established between a requirement usage and the entity that satisfies the requirement. The satisfy relationship must always be contained in a context that also contains the requirement and the entity that is asserted to satisfy it. For the maximumVehicleMass requirement above, the massActual attribute of the requirement is bound to the mass of the specific vehicle that is asserted to satisfy the requirement. The requirement is satisfied if the required constraint {mass <= massRequired} evaluates to true.

A requirement can also contain assumptions, where an assumption is a kind of constraint. For a requirement to be satisfied, all of its assumed constraints must also evaluate to true. For example, the maximumVehicleMass requirement may contain an assumption that the vehicle's fuel tank is full of gas, which can be expressed as {fuelMass == maximumFuelMass}. This assumed constraint must evaluate to be true for the requirement to be satisfied.

Stakeholders and Concerns

Stakeholders and their concerns can also be explicitly modeled in association with requirements. A StakeholderDefinition is a kind of PartDefinition representing an entity (such as a person or group, perhaps in a specific role) with concerns to be addressed. A StakeholderUsage is then a kind of PartUsage.

A ConcernDefinition is a kind of RequirementDefinition that represents a stakeholder concern and a ConcernUsage is a kind of RequirementUsage. A concern definition or usage references the stakeholders it affects, that is, those stakeholders who require the concern to be addressed. A more specific requirement may then frame one or more stakeholder concerns, which become required constraints of the requirement, so that the referenced concerns can be considered addressed when the requirement is satisfied.

Note. Stakeholder and concern modeling is frequently used in the context of view and viewpoint modeling (see [7.22](#)). A viewpoint is a kind of requirement that frames certain stakeholder concerns to be addressed by one more views satisfying the viewpoint.

7.18.2 Requirements Concrete Syntax

7.18.2.1 Requirements Textual Notation

7.18.2.1.1 Requirement Definitions

```
RequirementDefinition (m : Membership) : RequirementDefinition =
    DefinitionPrefix(this)? 'requirement' 'def'
    ConstraintDeclaration(this, m) RequirementBody(this)?

RequirementBody (t : Type) =
    ';' | '{' RequirementBodyItem(t)* '}'

RequirementBodyItem (t : Type) =
    DefinitionBodyItem(t)
    | t.ownedRelationship += SubjectMember
    | t.ownedRelationship += RequirementConstraintMember
    | t.ownedRelationship += AddressedConcernMember
    | t.ownedRelationship += RequirementVerificationMember

SubjectMember : SubjectMembership =
    DefinitionMemberPrefix(this) ownedSubjectParameter = SubjectUsage(this)

SubjectUsage (m : Membership) : ReferenceUsage =
    Usage(m)

RequirementConstraintMember : RequirementConstraintMembership =
    DefinitionMemberPrefix(this)? RequirementKind(this)
    ownedMemberFeature = RequirementConstraintUsage(this)

RequirementKind (m : RequirementConstraintMembership) =
    'assume' { m.kind = 'assumption' }
    | 'require' { m.kind = 'requirement' }

RequirementConstraintUsage (m : Membership) : ConstraintUsage =
    ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this)?
    CalculationParameterPart(this) RequirementBody(this)
    | 'constraint' CalculationUsageDeclaration(this, m) CalculationBody(this)

AddressedConcernMember : AddressedConcernMembership =
    DefinitionMemberPrefix(this)? 'frame'
    ownedMemberFeature = AddressedConcernUsage(this)

AddressedConcernUsage (m : Membership) : ConcernUsage =
    ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this)?
    CalculationParameterPart(this) RequirementBody(this)
    | 'concern' CalculationUsageDeclaration(this, m) CalculationBody(this)
```

7.18.2.1.2 Requirement Usages

```
RequirementUsage (m : Membership) : RequirementUsage =
  UsagePrefix(this)? 'requirement'
  CalculationUsageDeclaration(this, m) RequirementBody(this)

RequirementFlowUsage (m : Membership) : RequirementUsage =
  UsagePrefix(this)? 'ref'? 'requirement'
  CalculationUsageDeclaration(this, m) RequirementBody(this)

RequirementRefUsage (m : Membership) : RequirementUsage =
  UsagePrefix(this)? ( 'ref' 'requirement' | isComposite ?= 'requirement' )
  CalculationUsageDeclaration(this, m) RequirementBody(this)

SatisfyRequirementUsage : SatisfyRequirementUsage =
  UsagePrefix(this)? 'satisfy'
  ( ownedSubsetting += OwnedSubsetting | 'requirement' Identification(this, m) )
  FeatureSpecializationPart(this) ( ValuePart(this) | ActionParameterList(this) )
  ( 'by' ownedMembership += SatisfactionConnectorMember(this) )?
  RequirementBody(this)

SatisfactionConnectorMember (s : SatisfyRequirementUsage) : Membership =
  ownedMemberFeature = SatisfactionConnector(s)

SatisfactionConnector (s : SatisfyRequirementUsage ) : BindingConnector =
  ownedFeatureMembership += ConnectorEndMemberFor(s.subjectParameter)
  ownedFeatureMembership += ConnectorEndMember
```

7.18.2.1.3 Concern Definitions

```
ConcernDefinition (m : Membership) : ConcernDefinition =
    DefinitionPrefix(this)? 'concern' 'def'
    ConstraintDeclaration(this, m) ConcernBody(this)?

ConcernBody (t : Type) =
    ';' | '{' ConcernBodyItem(t)* '}'

ConcernBodyItem (t : Type) =
    RequirementBodyItem(t)
    | t.ownedRelationship += AffectedStakeholderMember

AffectedStakeholderMember : FeatureMembership =
    DefinitionMemberPrefix(this)? 'affect'
    ownedMemberFeature = AffectedStakeholderUsage(this)

AffectedStakeholderUsage (m : Membership) : StakeholderUsage =
    ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this)?
    UsageBody(this)
    | 'stakeholder' UsageDeclaration(this, m) UsageBody(this)

AddressedConcernMember : AddressedConcernMembership =
    DefinitionMemberPrefix(this)? 'frame'
    ownedMemberFeature = AddressedConcernUsage(this)
```

7.18.2.1.4 Concern Usages

```
ConcernUsage (m : Membership) : ConcernUsage =
    UsagePrefix(this)? 'concern'
    CalculationUsageDeclaration(this, m) ConcernBody(this)

ConcernFlowUsage (m : Membership) : ConcernUsage =
    UsagePrefix(this)? 'ref'? 'concern'
    CalculationUsageDeclaration(this, m) ConcernBody(this)

ConcernRefUsage (m : Membership) : ConcernUsage =
    UsagePrefix(this)? ( 'ref' 'concern' | isComposite ?= 'concern' )
    CalculationUsageDeclaration(this, m) ConcernBody(this)
```

7.18.2.1.5 Stakeholders

```

StakeholderDefinition (m : Membership) : StakeholderDefinition =
    DefinitionPrefix(this)? 'stakeholder' 'def'
    Definition(this, m)

StakeholderUsage (m : Membership) : StakeholderUsage =
    UsagePrefix(this)? 'stakeholder'
    Usage(this, m)

StakeholderFlowUsage (m : Membership) : StakeholderUsage =
    UsagePrefix(this)? 'ref'? 'stakeholder'
    Usage(this, m)

StakeholderRefUsage (m : Membership) : StakeholderUsage =
    UsagePrefix(this)? ( 'ref' 'stakeholder' | isComposite ?= 'stakeholder' )
    Usage(this, m)

```

7.18.2.1.6 Stakeholder Usage

7.18.2.2 Requirements Graphical Notation

7.18.3 Requirements Abstract Syntax

7.18.3.1 Overview

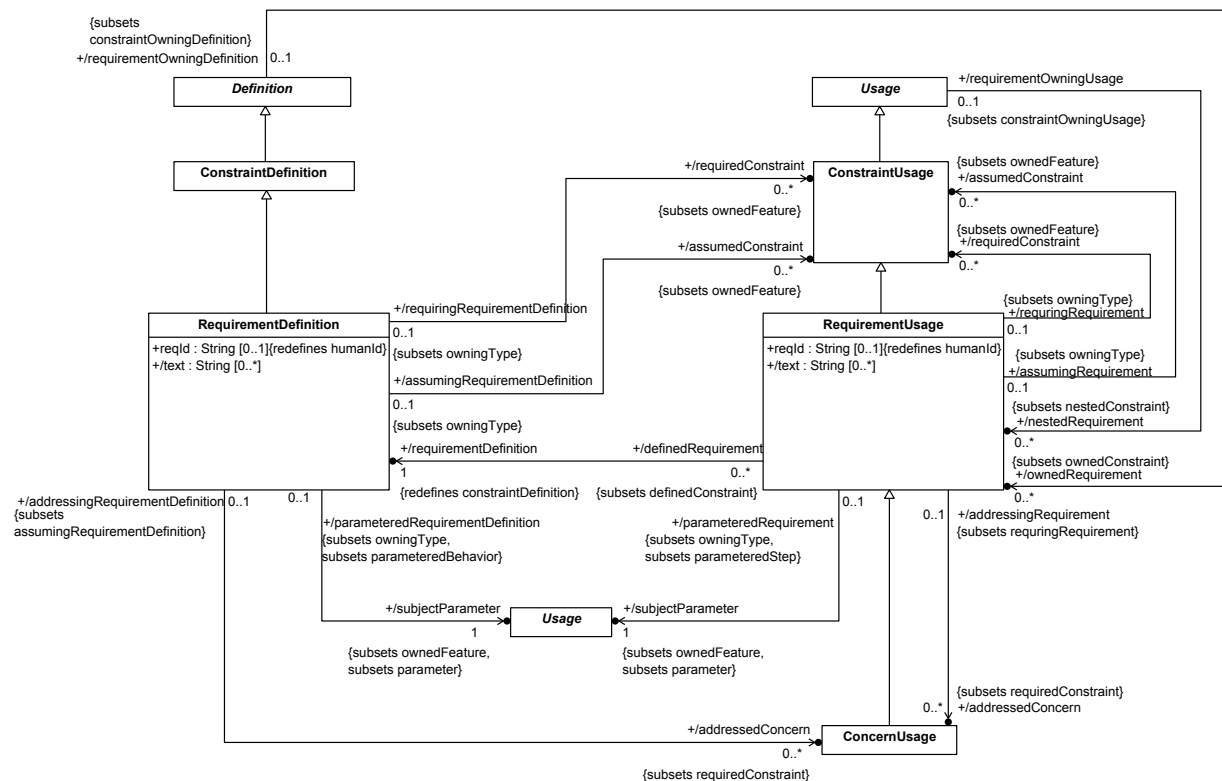


Figure 54. Requirement Definition and Usage

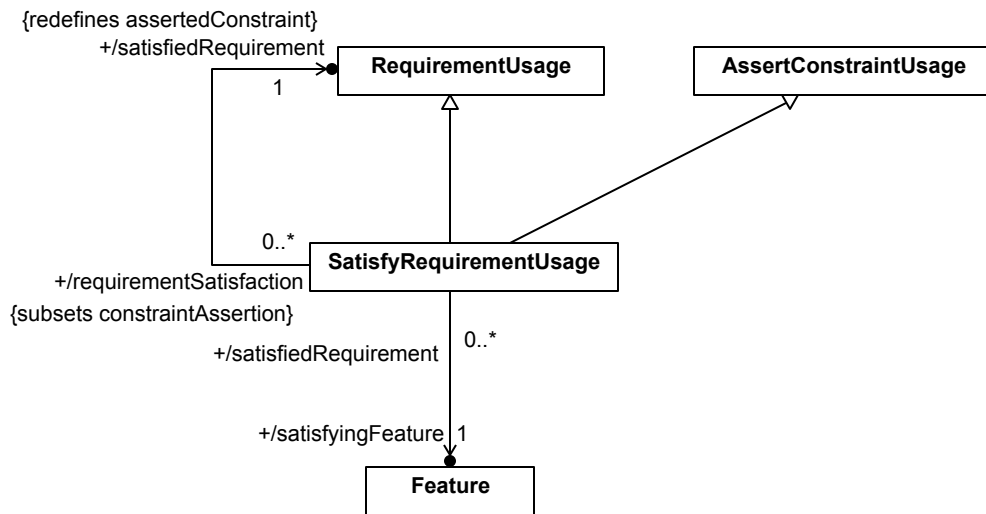


Figure 55. Requirement Satisfaction

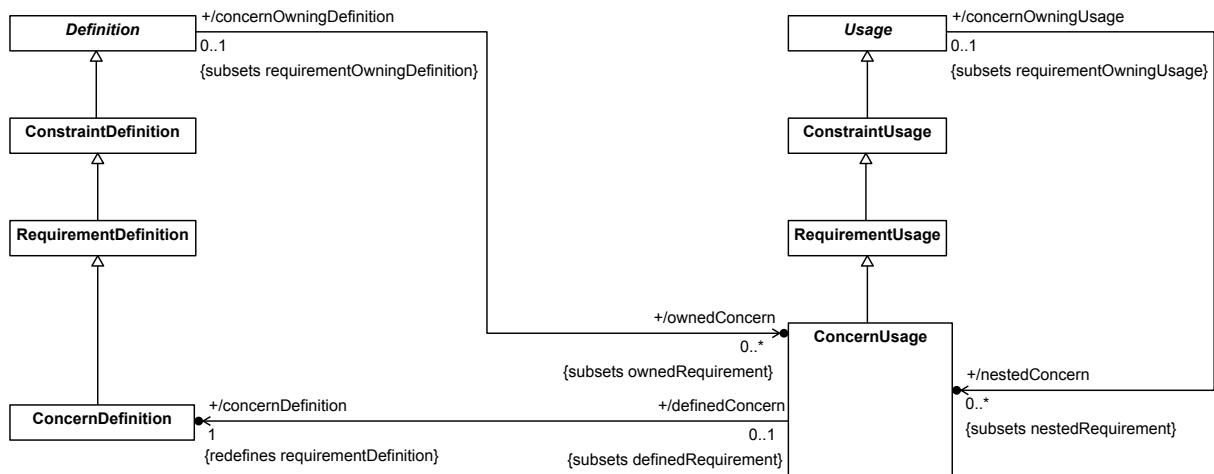


Figure 56. Concern Definition and Usage

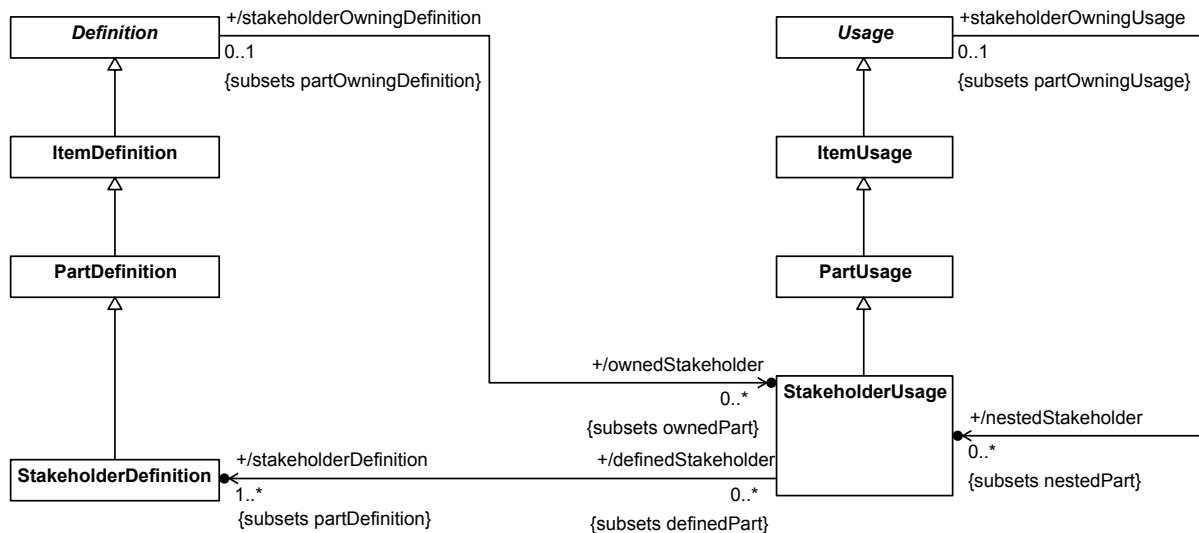


Figure 57. Stakeholder Definition and Usage

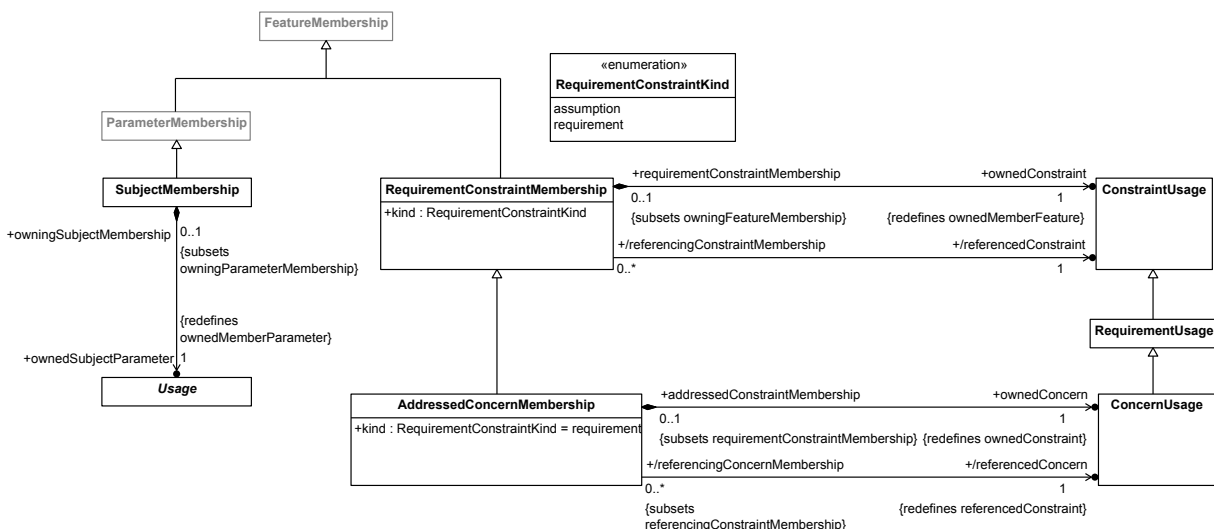


Figure 58. Requirement Membership

7.18.3.2 AddressConcernMembership

Description

An AddressedConcernMembership is a RequirementConstraintMembership for an addressed ConcernUsage of a RequirementDefinition or RequirementUsage. The ownedConstraint of an AddressedConcernMembership must be a ConcernUsage.

General Classes

RequirementConstraintMembership

Attributes

kind : RequirementConstraintKind

The kind of an AddressedConcernMembership must be requirement.

ownedConcern : ConcernUsage {redefines ownedConstraint}

The ConcernUsage that is the ownedConstraint of this AddressedConcernMembership.

/referencedConcern : ConcernUsage {redefines referencedConstraint}

The ConcernUsage that is referenced through this AddressedConcernMembership. This is derived as being the first ConcernUsage subset by the ownedConcern, if there is one, and, otherwise, the ownedConcern itself.

Operations

No operations.

Constraints

No constraints.

7.18.3.3 ConcernDefinition

Description

A ConcernDefinition is a RequirementDefinition that one or more stakeholders may be interested in having addressed. These stakeholders are identified by the ownedStakeholders of the ConcernDefinition.

A ConcernDefinition must subclass, directly or indirectly, the base ConcernDefinition ConcernCheck from the Systems model library. The ownedStakeholder features of a ConcernDefinition shall all subset the ConcernCheck::concernedStakeholders feature.

General Classes

RequirementDefinition

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

7.18.3.4 ConcernUsage

Description

A ConcernUsage is a Usage of a ConcernDefinition.

A ConcernUsage must subset, directly or indirectly, the base ConcernUsage concernChecks from the Systems model library. The ownedStakeholder features of the ConcernUsage shall all subset the ConcernCheck::concernedStakeholders feature. If the ConcernUsage is an ownedFeature of a

StakeholderDefinition or StakeholderUsage, then the ConcernUsage shall have an `ownedStakeholder` feature that is bound to the `self` feature of its owner.

General Classes

RequirementUsage

Literal Values

7.18.3.5 RequirementConstraintKind

Description

A RequirementConstraintKind indicates whether a ConstraintUsage is an assumption or a requirement in a RequirementDefinition or RequirementUsage.

General Classes

No general classes.

Literal Values

assumption

Indicates that a member ConstraintUsage of a RequirementDefinition or RequirementUsage represents an assumption.

requirement

Indicates that a member ConstraintUsage of a RequirementDefinition or RequirementUsage represents an requirement.

7.18.3.6 RequirementConstraintMembership

Description

A RequirementConstraintMembership is a FeatureMembership for an assumed or required ConstraintUsage of a RequirementDefinition or RequirementUsage. The `ownedMemberFeature` of a RequirementConstraintMembership must be a ConstraintUsage.

General Classes

FeatureMembership

Attributes

`kind` : RequirementConstraintKind

Whether the RequirementConstraintMembership is for an assumed or required ConstraintUsage.

`ownedConstraint` : ConstraintUsage {redefines `ownedMemberFeature`}

The ConstraintUsage that is the `ownedMemberFeature` of this RequirementConstraintMembership.

`/referencedConstraint` : ConstraintUsage

The ConstraintUsage that is referenced through this RequirementConstraintMembership. This is derived as being the first ConstraintUsage subset by the ownedConstraint, if there is one, and, otherwise, the ownedConstraint itself.

Operations

No operations.

Constraints

No constraints.

7.18.3.7 RequirementDefinition

Description

A RequirementDefinition is a ConstraintDefinition that defines a requirement as a constraint that is used in the context of a specification that a valid solution must satisfy.

A RequirementDefinition must subclass, directly or indirectly, the base RequirementDefinition RequirementCheck from the Systems model library.

General Classes

ConstraintDefinition

Attributes

/addressedConcern : ConcernUsage [0..*] {subsets requiredConstraint}

The Concerns addressed by this RequirementDefinition, derived as the ownedConcerns of all AddressedConcernMemberships of the RequirementDefinition.

/assumedConstraint : ConstraintUsage [0..*] {subsets ownedFeature}

The owned ConstraintUsages that represent assumptions of this RequirementDefinition, derived as the ownedConstraints of the RequirementConstraintMemberships of the RequirementDefinition with kind = assumption.

reqId : String [0..1] {redefines humanId}

An optional modeler-specified identifier for this RequirementDefinition (used, e.g., to link it to an original requirement text in some source document), derived as the modeledId for the RequirementDefinition.

/requiredConstraint : ConstraintUsage [0..*] {subsets ownedFeature}

The owned ConstraintUsages that represent requirements of this RequirementDefinition, derived as the ownedConstraints of the RequirementConstraintMemberships of the RequirementDefinition with kind = requirement.

/subjectParameter : Usage {subsets parameter, ownedFeature}

The `parameter` of this `RequirementDefinition` that is owned via a `SubjectMembership`, which must redefine, directly or indirectly, the `subject` parameter of the base `RequirementDefinition` `RequirementCheck` from the Systems model library.

`/text : String [0..*]`

An optional textual statement of the requirement represented by this `RequirementDefinition`, derived as the `bodies` of the `documentaryComments` of the `RequirementDefinition`.

Operations

No operations.

Constraints

No constraints.

7.18.3.8 RequirementUsage

Description

A `RequirementUsage` is a `Usage` of a `RequirementDefinition`.

A `RequirementUsage` (other than a `SatisfyRequirementUsage` owned by a `Part`) must subset, directly or indirectly, the base `RequirementUsage` `requirementChecks` from the Systems model library.

General Classes

`ConstraintUsage`

Attributes

`/addressedConcern : ConcernUsage [0..*] {subsets requiredConstraint}`

The `Concerns` addressed by this `RequirementUsage`, derived as the `ownedConcerns` of all `AddressedConcernMemberships` of the `RequirementUsage`.

`/assumedConstraint : ConstraintUsage [0..*] {subsets ownedFeature}`

The owned `ConstraintUsages` that represent assumptions of this `RequirementUsage`, derived as the `ownedConstraints` of the `RequirementConstraintMemberships` of the `RequirementUsage` with `kind = assumption`.

`reqId : String [0..1] {redefines humanId}`

An optional modeler-specified identifier for this `RequirementUsage` (used, e.g., to link it to an original requirement text in some source document), derived as the `modeledId` for the `RequirementUsage`.

`/requiredConstraint : ConstraintUsage [0..*] {subsets ownedFeature}`

The owned `ConstraintUsages` that represent requirements of this `RequirementUsage`, derived as the `ownedConstraints` of the `RequirementConstraintMemberships` of the `RequirementUsage` with `kind = requirement`.

/requirementDefinition : RequirementDefinition {redefines constraintDefinition}

The RequirementDefinition that is the single type of this RequirementUsage.

/subjectParameter : Usage {subsets parameter, ownedFeature}

The parameter of this RequirementUsage that is owned via a SubjectMembership, which must redefine, directly or indirectly, the subject parameter of the base RequirementDefinition RequirementCheck from the Systems model library.

/text : String [0..*]

An optional textual statement of the requirement represented by this RequirementUsage, derived as the bodies of the documentaryComments of the RequirementDefinition.

Operations

No operations.

Constraints

No constraints.

7.18.3.9 SatisfyRequirementUsage

Description

A SatisfyRequirementUsage is an AssertConstraintUsage that asserts that a satisfied RequirementUsage is true for a specific satisfyingSubject. The satisfied RequirementUsage is related to the SatisfyRequirementUsage by a Subsetting relationship.

General Classes

AssertConstraintUsage
RequirementUsage

Attributes

/satisfiedRequirement : RequirementUsage {redefines assertedConstraint}

The RequirementUsage that is satisfied by the satisfyingSubject of this SatisfyRequirementUsage. It is the subsettingFeature of the first owned Subsetting Relationship of the SatisfyRequirementUsage.

/satisfyingFeature : Feature

The Feature that represents the actual subject that is asserted to satisfy the satisfiedRequirement. The satisfyingFeature must be the target of a BindingConnector from the subjectParameter of the satisfiedRequirement.

Operations

No operations.

Constraints

No constraints.

7.18.3.10 StakeholderDefinition

Description

A StakeholderDefinition is a kind of PartDefinition representing an entity that has concerns that are required to be addressed. These concerns may be identified as `ownedConcerns` of the StakeholderDefinition and/or by reference via a StakeholderUsage in the corresponding ConcernDefinition or ConcernUsage.

A StakeholderDefinition must subclass, directly or indirectly, the base StakeholderDefinition Stakeholder from the Systems model library.

General Classes

PartDefinition

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

7.18.3.11 StakeholderUsage

Description

A StakeholderUsage is a usage of a StakeholderDefinition. At least one of the types of a StakeholderUsage must be a StakeholderDefinition, but a StakeholderUsage may also be typed by PartDefinitions and ItemDefinitions that are not StakeholderDefinitions.

A StakeholderUsage must subset, directly or indirectly, the base StakeholderUsage `stakeholders` from the Systems model library.

General Classes

PartUsage

Attributes

`/stakeholderDefinition : StakeholderDefinition [1..*] {subsets partDefinition}`

The `partDefinitions` of this StakeholderUsage that are StakeholderDefinitions.

Operations

No operations.

Constraints

No constraints.

7.18.3.12 SubjectMembership

Description

A SubjectMembership is a ParameterMembership that indicates that its `ownedSubjectParameter` is the subject Parameter for its `owningType`. The `owningType` of a SubjectMembership must be a CaseDefinition, CaseUsage, RequirementDefinition or RequirementUsage.

General Classes

ParameterMembership

Attributes

`ownedSubjectParameter` : Usage {redefines `ownedMemberParameter`}

The Usage that is the `ownedMemberParameter` of this SubjectMembership.

Operations

No operations.

Constraints

No constraints.

7.18.4 Requirements Semantics

7.19 Cases

7.19.1 Cases Overview

A case is a general concept that provides the basis for more specific cases including an analysis case and a verification case. A CaseDefinition is a specialized kind of CalculationDefinition with a specified objective, that is a RequirementUsage. The objective is used to specify what the case is intended to achieve regarding the subject, which generally includes some combination of collecting information about the subject and evaluating the subject.. The subject is an input to the case, and the return result is the designated output of the case. A case definition can contain a series of actions needed to achieve the case objective. A CaseUsage is a usage of a case definition.

7.19.2 Cases Concrete Syntax

7.19.2.1 Cases Textual Notation

7.19.2.1.1 Case Definitions

```
CaseDefinition (m : Membership) : CaseDefinition =
    DefinitionPrefix(this)? 'case' 'def'
    CalculationDeclaration(this, m) CaseBody(this)

CaseBody (t : Type) =
    ';'
    | '{' CaseBodyItem(t)*
      ( t.ownedRelationship += ResultExpressionMember )?
    '}'

CaseBodyItem (t : Type) =
    ActionBodyItem(t)
    | t.ownedRelationship += SubjectMember
    | t.ownedRelationship += ObjectiveMember

ObjectiveMember : ObjectiveMembership =
    DefinitionMemberPrefix(this) 'objective'
    ownedObjectiveRequirement = ObjectiveRequirementUsage(this)

ObjectiveRequirementUsage (m : Membership) : RequirementUsage =
    CalculationUsageDeclaration(this, m) RequirementBody(this)
```

7.19.2.1.2 Case Usages

```
CaseUsage (m : Membership) : CaseUsage =
    UsagePrefix(this)? 'case'
    CalculationUsageDeclaration(this, m) CaseBody(this)

CaseFlowUsage (m : Membership) : CaseUsage =
    UsagePrefix(this)? 'ref'? 'case'
    CalculationUsageDeclaration(this, m) CaseBody(this)

CaseRefUsage (m : Membership) : CaseUsage =
    UsagePrefix(this)? ( 'ref' 'case' | isComposite ?= 'case' )
    CalculationUsageDeclaration(this ,m) CaseBody(this)
```

7.19.2.2 Cases Graphical Notation

7.19.3 Cases Abstract Syntax

7.19.3.1 Overview

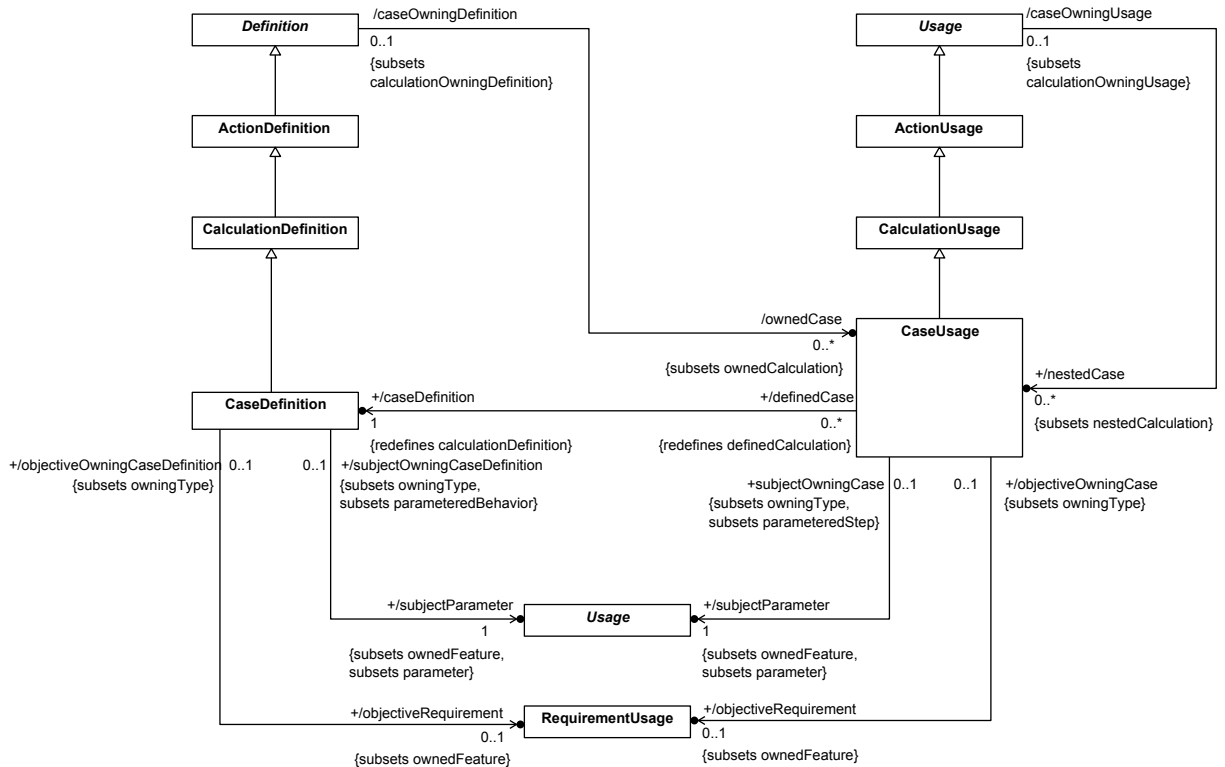


Figure 59. Case Definition and Usage

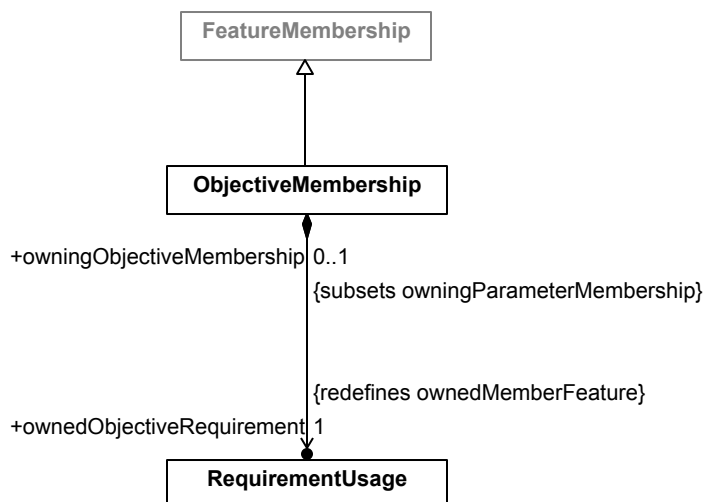


Figure 60. Case Membership

7.19.3.2 CaseDefinition

Description

A CaseDefinition is a CalculationDefinition for a process, often involving collecting evidence or data, relative to a subject, producing a result that meets an objective.

A CaseDefinition must subclass, directly or indirectly, the base CaseDefinition Case from the Systems model library.

General Classes

CalculationDefinition

Attributes

/objectiveRequirement : RequirementUsage [0..1] {subsets ownedFeature}

The

ownedFeature

of this CaseDefinition that is owned via an ObjectiveMembership, and that must redefine, directly or indirectly, the `objectiveRequirementUsage` of the base CaseDefinition Case from the Systems model library.

/subjectParameter : Usage {subsets parameter, ownedFeature}

The `parameter` of this CaseDefinition that is owned via a SubjectMembership, which must redefine, directly or indirectly, the `subjectParameter` of the base CaseDefinition Case from the Systems model library.

Operations

No operations.

Constraints

No constraints.

7.19.3.3 CaseUsage

Description

A CaseUsage is a Usage of a CaseDefinition.

A CaseUsage must subset, directly or indirectly, either the base CaseUsage `cases` from the Systems model library, if it is not owned by a CaseDefinition or CaseUsage, or the CaseUsage `subcases` inherited from its owner, otherwise.

General Classes

CalculationUsage

Attributes

/caseDefinition : CaseDefinition {redefines calculationDefinition}

The CaseDefinition that is the type of this CaseUsage.

/objectiveRequirement : RequirementUsage [0..1] {subsets ownedFeature}

The

ownedFeature

of this CaseUsage that is owned via an ObjectiveMembership, and that must redefine, directly or indirectly, the objective RequirementUsage of the base CaseDefinition Case from the Systems model library.

/subjectParameter : Usage {subsets parameter, ownedFeature}

The parameter of this CaseUsage that is owned via a SubjectMembership, which must redefine, directly or indirectly, the subject parameter of the base CaseDefinition Case from the Systems model library.

Operations

No operations.

Constraints

No constraints.

7.19.3.4 ObjectiveMembership

Description

An ObjectiveMembership is a FeatureMembership that indicates that its ownedObjectiveRequirement is the objective RequirementUsage for its owningType. The owningType of an ObjectiveMembership must be a CaseDefinition or CaseUsage.

General Classes

FeatureMembership

Attributes

ownedObjectiveRequirement : RequirementUsage {redefines ownedMemberFeature}

The RequirementUsage that is the

ownedMemberFeature

of this RequirementUsage.

Operations

No operations.

Constraints

No constraints.

7.19.4 Cases Semantics

7.20 Analysis Cases

7.20.1 Analysis Cases Overview

An analysis can be specified in SysML and solved by external solvers. The analysis specification may include a combination of equations to be solved, a set of input and output parameters that are used in the equations, initial conditions, and boundary conditions.

An `AnalysisCaseDefinition` is a kind of `CaseDefinition` that enables the specification of an analysis. An `AnalysisCaseUsage` is a kind of `CaseUsage`. An analysis case usage is a usage of an analysis case definition. The analysis case definition and analysis case usage may have an analysis objective to specify what the analysis is intended to achieve. The analysis objective can be defined in terms of one or more questions to be answered. The subject of the analysis identifies what is being analyzed, and is an input to the analysis case.

The subject of the analysis is typically quite general for the analysis case definition, and then made more specific for the analysis case usage. Executing the analysis case usage returns an analysis result about the subject. For example, a fuel economy analysis of a vehicle subject returns the estimated fuel economy of the vehicle, given a set of analysis inputs and assumed conditions. The analysis result can be evaluated to determine whether it satisfies the analysis objective.

The analysis case can include a set of analysis actions, each of which can specify calculations that return results. As an example, the fuel economy analysis referred to above may require both a dynamics analysis and a fuel consumption analysis. The dynamics analysis determines the vehicle trajectory and the required engine power versus time. The fuel consumption analysis determines the fuel consumed to achieve the required engine power. Both the dynamics analysis and the fuel consumption analysis may require multiple calculations.

As noted above, an analysis case is a specification of an analysis that is often performed by solvers external to the SysML modeling tool. The analysis case does not specify how the analysis is executed. For example, the analysis does not specify the type of integration algorithm that is used to solve a differential equation.

An analysis case can also specify a set of simultaneous equations to be solved. This is done defining one or more constraint expressions that logically AND each of the equations, and asserting that the constraint must be true. A solver would be expected to solve the equations such that it returns values that satisfy each equation.

A trade-off analysis is a special kind of analysis used to evaluate and compare alternatives. Each alternative being evaluated is a subject of the trade-off analysis. The trade-off analysis also includes a set of criteria that are evaluated for each alternative, and an objective function that provides an overall evaluation result for each alternative. An example of a trade-off analysis is an analysis that evaluates and compares multiple vehicle design alternatives in terms of their maximum acceleration, reliability, and fuel economy (i.e., criterion). The objective function establishes a relative weighting of each criterion based on its importance to the stakeholder. The evaluation result is computed for each alternative based on a weighted sum of the normalized value for maximum acceleration, reliability, and fuel economy. The evaluation results for each alternative are then compared to determine a preferred solution.

Modeling of trade-off analyses is specifically supported through the use of the Trade Studies library model found in the Analysis domain library (see [8.3.2](#)).

7.20.2 Analysis Cases Concrete Syntax

7.20.2.1 Analysis Cases Textual Notation

```

AnalysisCaseDefinition (m : Membership) : AnalysisCaseDefinition =
  DefinitionPrefix(this)? 'analysis' 'def'
  CalculationDeclaration(this, m) CaseBody(this)

AnalysisCaseUsage (m : Membership) : CaseUsage =
  UsagePrefix(this)? 'analysis'
  CalculationUsageDeclaration(this, m) CaseBody(this)

AnalysisCaseFlowUsage (m : Membership) : CaseUsage =
  UsagePrefix(this)? 'ref'? 'analysis'
  CalculationUsageDeclaration(this, m) CaseBody(this)

AnalysisCaseRefUsage (m : Membership) : AnalysisCaseUsage =
  UsagePrefix(this)? ( 'ref' 'analysis' | isComposite ?= 'analysis' )
  CalculationUsageDeclaration(this ,m) CaseBody(this)

```

Submission Note. Additional concrete syntax for specialized analysis actions is planned for the revised submission.

7.20.2.2 Analysis Cases Graphical Notation

7.20.3 Analysis Cases Abstract Syntax

7.20.3.1 Overview

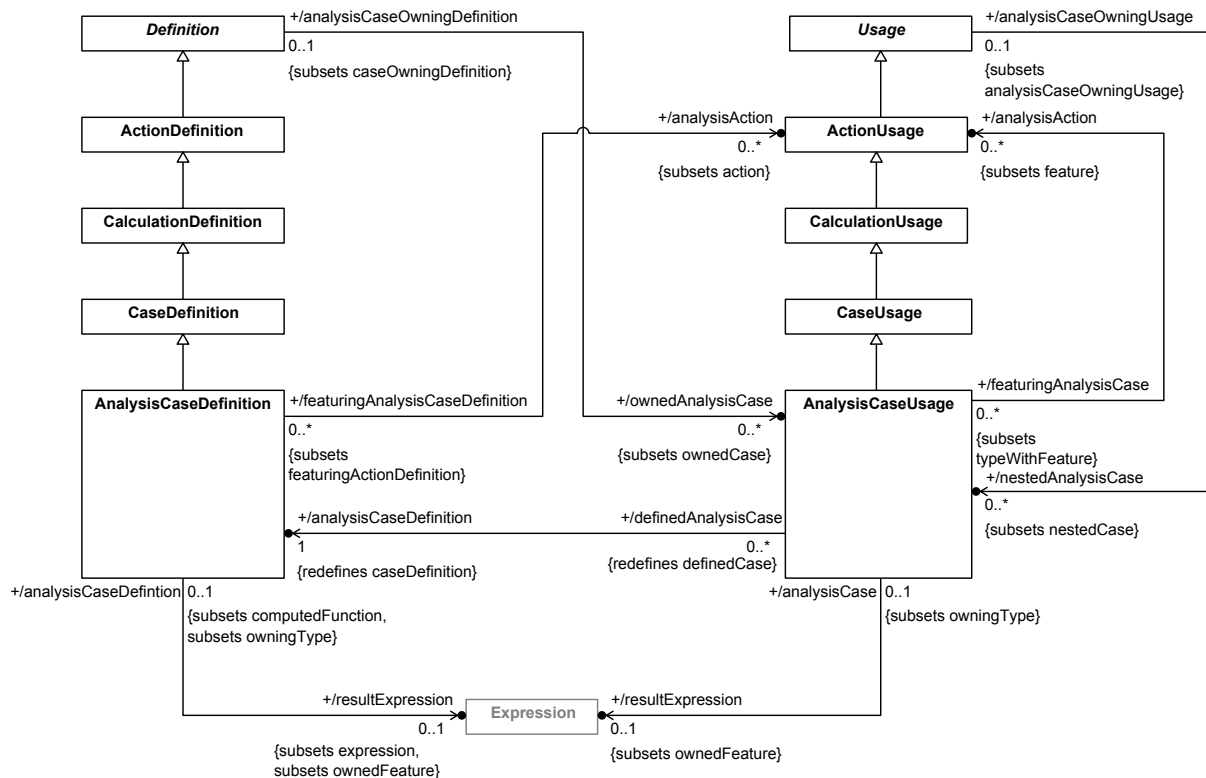


Figure 61. Analysis Case Definition and Usage

Submission Note. Additional abstract syntax for specialize analysis actions is planned for the revised submission.

7.20.3.2 AnalysisCaseDefinition

Description

An AnalysisCaseDefinition is a CaseDefinition for the case of carrying out an analysis.

An AnalysisCaseDefinition must subclass, directly or indirectly, the base AnalysisCaseDefinition AnalysisCase from the Systems model library.

General Classes

CaseDefinition

Attributes

/analysisAction : ActionUsage [0..*] {subsets action}

The `actions` of the AnalysisCaseDefinitions that are typed as AnalysisActions. Each `analysisAction` ActionUsage must subset the `analysisSteps` ActionUsage of the base AnalysisCaseDefinition AnalysisCase from the Systems model library.

/resultExpression : Expression [0..1] {subsets expression, ownedFeature}

The Expression used to compute the `result` of the AnalysisCaseDefinition, derived as the Expression own via a ResultExpressionMembership. The `resultExpression` must redefine directly or indirectly, the `resultEvaluation` Expression of the base AnalysisCaseDefinition AnalysisCase from the Systems model library.

Operations

No operations.

Constraints

No constraints.

7.20.3.3 AnalysisCaseUsage

Description

An AnalysisCaseUsage is a Usage of an AnalysisCaseDefinition.

An AnalysisCaseUsage must subset, directly or indirectly, either the base AnalysisCaseUsage `analysisCases` from the Systems model library, if it is not owned by an AnalysisCaseDefinition or AnalysisCaseUsage, or the AnalysisCaseUsage `subAnalysisCases` inherited from its owner, otherwise.

General Classes

CaseUsage

Attributes

/analysisAction : ActionUsage [0..*] {subsets feature}

The features of the AnalysisCaseUsage that are typed as AnalysisActions. Each analysisAction ActionUsage must subset the analysisSteps ActionUsage of the base AnalysisCaseDefinition AnalysisCase from the Systems model library.

/analysisCaseDefinition : AnalysisCaseDefinition {redefines caseDefinition}

The AnalysisCaseDefinition that is the type of this AnalysisCaseUsage.

/resultExpression : Expression [0..1] {subsets ownedFeature}

The Expression used to compute the result of the AnalysisCaseUsage, derived as the Expression own via a ResultExpressionMembership. The resultExpression> must redefine directly or indirectly, the resultEvaluation Expression of the base AnalysisCaseDefinition AnalysisCase from the Systems model library.

Operations

No operations.

Constraints

No constraints.

7.20.4 Analysis Cases Semantics

7.21 Verification Cases

7.21.1 Verification Cases Overview

Verification cases are used to specify the actions needed to verify that a system or component satisfies its requirements. The verification cases are an important input to verification planning and execution.

A VerificationCaseDefinition is a kind of CaseDefinition. A verification case definition defines a set of verification actions that are needed to verify that a system or other entity satisfies its requirements. A VerificationCaseUsage is a kind of Caseusage. A verification case usage is a usage of a verification case definition.

The verification case has a verification objective and a subject. The subject is the system or other entity that is being evaluated as to whether it satisfies the requirements. The subject is often referred to as the unit under test or unit under verification, and is an input to the verification case. The subject of the verification case usage can subset the subject of the verification case definition to make it more specific. The verification objective can be stated in terms of a satisfying a set of requirements, such as the objective to evaluate whether the system satisfies its environmental requirements.

A typical verification case includes a set of verification actions to collect the data, analyze the data, and then evaluate whether the resulting analysis result satisfies the requirement.

- The first step is to collect the data about the subject needed to support the verification objective, which is typically done using verification methods such as analysis, inspection, demonstration, and test.
- The second step involves an analysis of this collected data. For example, the data may include multiple measurements that span a range of conditions for a particular individual, or measurements of different individuals. This analysis step may need to determine the probability distribution, mean, and standard deviation associated with the measurements.
- The third step is to evaluate whether the results from the data analysis satisfy the requirement or requirements.

Each of the verification actions in the verification case requires a set of resources to perform the actions. This may include verification personnel, test equipment, facilities, and other resources. These resources may be represented in the model as parts that perform actions, or other inputs and outputs of actions.

A verification case can be used to verify that one or more requirements are satisfied. A verify relationship is defined between the verification case and each requirement it is intended to verify (to be included). Executing the verification case will return a verification result, such as pass, fail, or inconclusive.

7.21.2 Verification Cases Concrete Syntax

7.21.2.1 Verification Cases Textual Notation

```
VerificationCaseDefinition (m : Membership) : VerificationCaseDefinition =  
    DefinitionPrefix(this)? 'verification' 'def'  
    CalculationDeclaration(this, m) CaseBody(this)  
  
VerificationCaseFlowUsage (m : Membership) : VerificationCaseUsage =  
    UsagePrefix(this)? 'verification'  
    CalculationUsageDeclaration(this, m) CaseBody(this)  
  
VerificationCaseUsage (m : Membership) : VerificationCaseUsage =  
    UsagePrefix(this)? 'ref'? 'verification'  
    CalculationUsageDeclaration(this, m) CaseBody(this)  
  
VerificationCaseRefUsage (m : Membership) : AnalysisCaseUsage =  
    UsagePrefix(this)? ( 'ref' 'verification' | isComposite ?= 'verification' )  
    CalculationUsageDeclaration(this, m) CaseBody(this)  
  
RequirementVerificationMember : RequirementVerificationMembership =  
    DefinitionMemberPrefix kind = RequirementVerificationKind  
    ownedRequirement = RequirementVerificationUsage  
  
RequirementVerificationKind : RequirementConstraintKind =  
    'verify' { m.kind = 'requirement' }  
  
RequirementVerificationUsage : RequirementUsage =  
    ownedRelationship += OwnedSubsetting FeatureSpecialization(this)*  
    CalculationUsageParameterPart(this)? RequirementBody(this)  
    | 'requirement' CalculationUsageDeclaration(this) RequirementBody(this)
```

7.21.2.2 Verification Cases Graphical Notation

7.21.3 Verification Cases Abstract Syntax

7.21.3.1 Overview

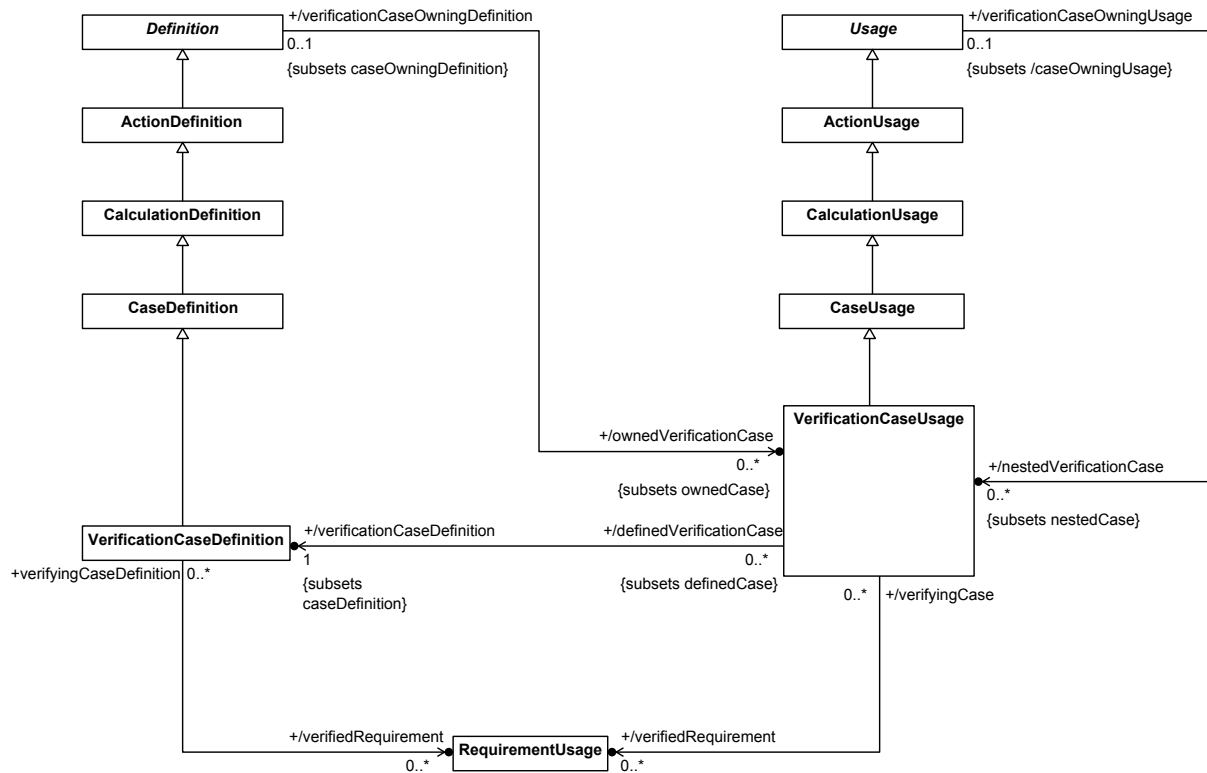


Figure 62. Verification Case Definition and Usage

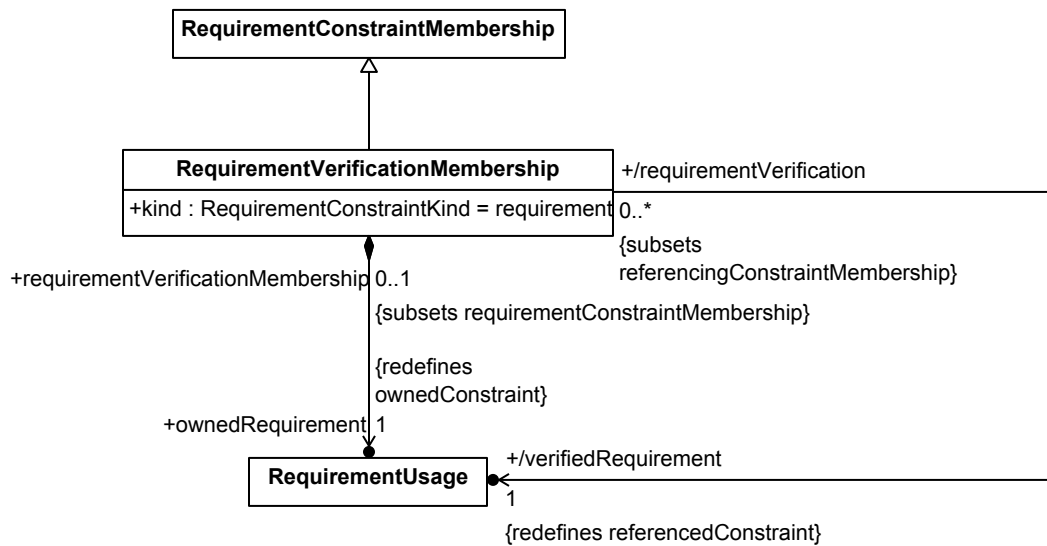


Figure 63. Verification Membership

7.21.3.2 RequirementVerificationMembership

Description

A RequirementVerificationMembership is a RequirementConstraintMembership used in the objective of a VerificationCase to identify a Requirement that is verified by the VerificationCase.

General Classes

RequirementConstraintMembership

Attributes

kind : RequirementConstraintKind

The kind of a RequirementVerificationMembership must be requirement.

ownedRequirement : RequirementUsage {redefines ownedConstraint}

The owned Requirement that acts as the constraint for this RequirementVerificationMembership. This will either be the verifiedRequirement, or it will subset the verifiedRequirement.

/verifiedRequirement : RequirementUsage {redefines referencedConstraint}

The RequirementUsage that is identified as being verified. This is derived as being the first RequirementUsage subset by the ownedRequirement, if there is one, and, otherwise, the ownedRequirement itself.

Operations

No operations.

Constraints

No constraints.

7.21.3.3 VerificationCaseDefinition

Description

A VerificationCaseDefinition is a CaseDefinition for the purpose of verification of the subject of the case against its requirements.

A VerificationCaseDefinition must subclass, directly or indirectly, the base VerificationCaseDefinition VerificationCase from the Systems model library.

General Classes

CaseDefinition

Attributes

/verifiedRequirement : RequirementUsage [0..*]

The RequirementUsages verified by this VerificationCaseDefinition, derived as the verifiedRequirements of all RequirementVerificationMemberships of the objectiveRequirement.

Operations

No operations.

Constraints

No constraints.

7.21.3.4 VerificationCaseUsage

Description

A VerificationCaseUsage is a Usage of a VerificationCaseDefinition.

A VerificationCaseUsage must subset, directly or indirectly, either the base VerificationCaseUsage `verificationCases` from the Systems model library, if it is not owned by a VerificationCaseDefinition or VerificationCaseUsage, or the VerificationCaseUsage `subVerificationCases` inherited from its owner, otherwise.

General Classes

CaseUsage

Attributes

/verificationCaseDefinition : VerificationCaseDefinition {subsets caseDefinition}

The VerificationCase that defines this VerificationCaseUsage.

/verifiedRequirement : RequirementUsage [0..*]

The RequirementUsages verified by this VerificationCaseUsage, derived as the `verifiedRequirements` of all RequirementVerificationMemberships of the `objectiveRequirement`.

Operations

No operations.

Constraints

No constraints.

7.21.4 Verification Cases Semantics

7.22 Views

7.22.1 Views Overview

A view is intended to provide information that addresses some aspect of a system or domain of interest that one or more stakeholders care about. This information can be extracted from the model and rendered in some form, such as a diagram, table, or a document that contains diagrams, tables, and text.

SysML includes a set of standard views that are commonly used to describe systems. The model information presented in the standard views can be rendered in textual, graphical, or tabular form. In addition to the standard views, SysML v2 supports user defined views.

A Viewpoint is a specialized kind of Definition element that specifies the information about a system or domain that is of interest to one or more stakeholders. A ViewDefinition is a specialized kind of Definition element that

specifies how to create a view artifact to satisfy one or more viewpoints. A ViewpointUsage and ViewUsage are specialized kinds of Usage elements.

A view definition can include a query expression to extract the relevant model content, and a rendering specification that specifies how the model content should be rendered in a view artifact. A view usage is a usage of a view definition that can expose a portion of the model to limit the scope of the query. Executing the query expression returns the model elements that meet the query criteria. The query results can be referenced by a query package and persisted in the model. View usages can be nested and ordered within a composite view to generate composite view artifacts. The view definition and view usage also can contain a rendering specification to specify the symbolic representation, style, and layout for a particular view.

Complex view definitions with deeply nested structures can be rendered as documents, where each nested view usage corresponds to a section of a document, and the ordering represents the order of the section within the document. Within each section of the document, the nested view usages can specify the information that is rendered as a combination of text, graphical, and tabular information.

7.22.2 Views Concrete Syntax

7.22.2.1 Views Textual Notation

7.22.2.1.1 View Definitions

```
ViewDefinition (m : Membership) : ViewDefinition =
    DefinitionPrefix(this)? 'view' 'def' DefinitionDeclaration(this, m)
    ViewDefinitionBody(this)

ViewDefinitionBody (v : ViewDefinition) =
    ';' | '{' ViewDefinitionBodyItem(v) * '}'

ViewDefinitionBodyItem (v : ViewUsage) =
    DefinitionBodyItem(v)
    | v.ownedRelationship += ElementFilterMember
    | v.ownedRelationship += ViewRenderingMember

ViewRenderingMember : ViewRenderingMembership =
    DefinitionMemberPrefix(this) 'render'
    ownedRendering = ViewRenderingUsage

ViewRenderingUsage : RenderingUsage =
    ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this)?
    UsageBody(this)
```

7.22.2.1.2 View Usages

```
ViewUsage (m : Membership) : ViewUsage =
  UsagePrefix(this)? 'view' UsageDeclaration(this, m)?
  ValueOrFlowPart(this)? ViewBody(this)

ViewFlowUsage (m : Membership) : ViewUsage =
  UsagePrefix(this)? 'ref'? 'view' UsageDeclaration(this, m)?
  ValueOrFlowPart(this)? ViewBody(this)

ViewRefUsage (m : Membership) : ViewUsage =
  UsagePrefix(this)? ( 'ref' 'view' | isComposite ?= 'view' )
  UsageDeclaration(this, m)? ValueOrFlowPart(this)? ViewBody(this)

ViewBody (v : ViewUsage) =
  ';' | '{' ViewBodyItem(v) * '}'

ViewBodyItem (v : ViewUsage) =
  DefinitionBodyItem(v)
  | v.ownedRelationship += ElementFilterMember
  | v.ownedRelationship += ViewRenderingMember
  | v.ownedRelationship += Expose

Expose : Expose =
  ( ownedRelationship += PrefixDocumentation ) *
  ( visibility = BasicVisibilityIndicator )?
  'expose' ( ImportedNamespace | ImportedFilterPackage ) ';'

```

7.22.2.1.3 Viewpoints

```
ViewpointDefinition (m : Membership) : ViewpointDefinition =
  DefinitionPrefix(this)? 'viewpoint' 'def' ConstraintDeclaration(this, m)
  RequirementBody(this)

ViewpointUsage (m : Membership) : ViewpointUsage =
  UsagePrefix(this)? 'viewpoint' CalculationUsageDeclaration(this, m)
  RequirementBody

ViewpointFlowUsage (m : Membership) : ViewpointUsage =
  UsagePrefix(this)? 'ref'? 'viewpoint' CalculationUsageDeclaration(this, m)
  RequirementBody(this)

ViewpointRefUsage (m : Membership) : ViewpointUsage =
  UsagePrefix(this)? ( 'ref' 'viewpoint' | isComposite ?= 'viewpoint' )
  CalculationUsageDeclaration(this, m) RequirementBody(this)

```

7.22.2.1.4 Renderings

```

RenderingDefinition (m : Membership) : RenderingDefinition =
    DefinitionPrefix(this)? 'rendering' 'def' Definition(this, m)

RenderingUsage (m : Membership) : RenderingUsage =
    UsagePrefix(this)? 'rendering' Usage(this, m)

RenderingFlowUsage (m : Membership) : RenderingUsage =
    UsagePrefix(this)? 'ref' 'rendering' Usage(this, m)

RenderingRefUsage (m : Membership) : RenderingUsage =
    UsagePrefix(this)? ( 'ref' 'rendering' | isComposite ?= 'rendering' )
    Usage(this, m)

```

7.22.2.2 Views Graphical Notation

7.22.3 Views Abstract Syntax

7.22.3.1 Overview

Submission Note. This is a preliminary abstract syntax model for views and viewpoints. A complete model will be provided in the revised submission.

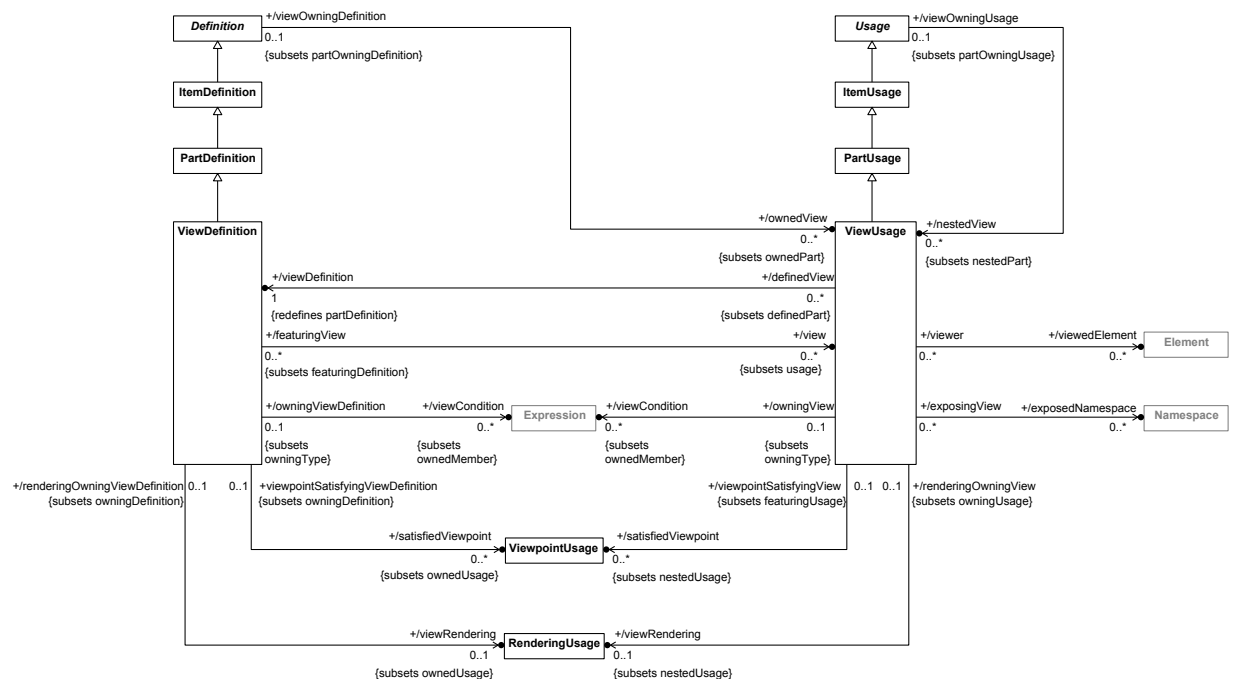


Figure 64. View Definition and Usage

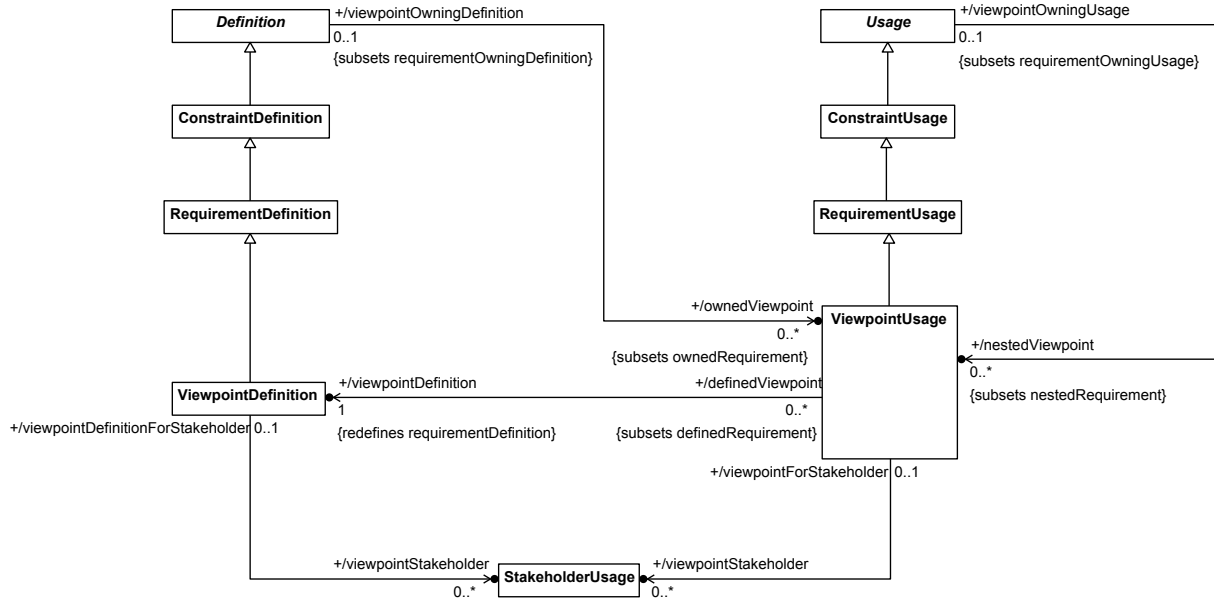


Figure 65. Viewpoint Definition and Usage

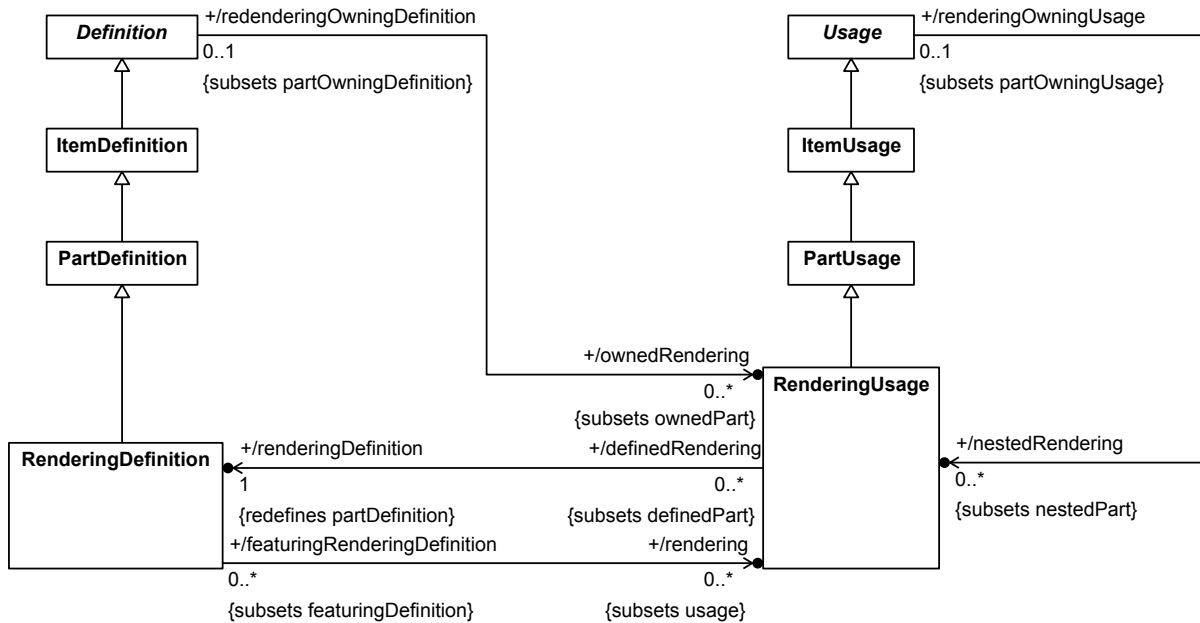


Figure 66. Rendering Definition and Usage

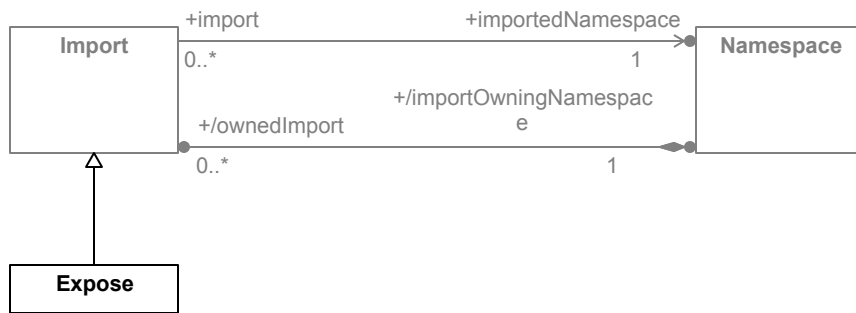


Figure 67. Expose Relationship

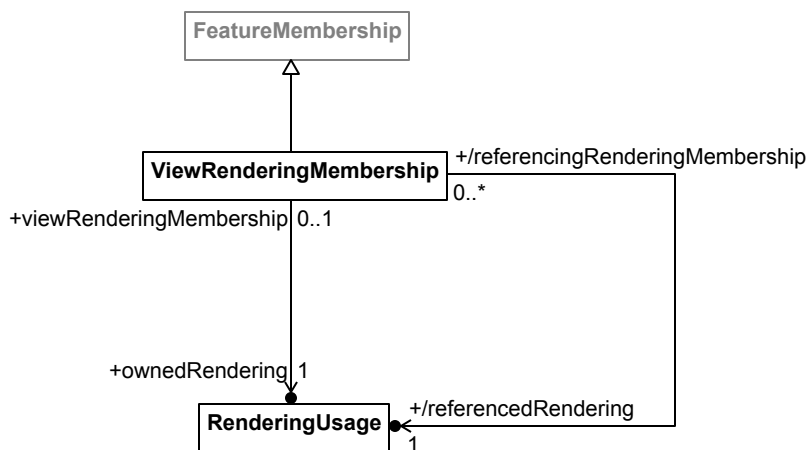


Figure 68. View Rendering Membership

7.22.3.2 Expose

Description

An Expose is an Import of a Namespace into a ViewUsage that provides a root for determining what Elements are to be included in a view.

General Classes

Import

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

7.22.3.3 RenderingDefinition

Description

A RenderingDefinition is a PartDefinition that defines a specific rendering of the content of a model view (e.g., symbols, style, layout, etc.).

A RenderingDefinition must subclass, directly or indirectly, the base RenderingDefinition Rendering from the Systems model library.

General Classes

PartDefinition

Attributes

/rendering : RenderingUsage [0..*] {subsets usage}

The usages of a RenderingDefinition that are RenderingUsages.

Operations

No operations.

Constraints

No constraints.

7.22.3.4 RenderingUsage

Description

A RenderingUsage is the usage of a RenderingDefinition to specify the rendering of a specific model view to produce a physical view artifact.

A RenderingUsage must subset, directly or indirectly, the base RenderingUsage renderings from the Systems model library.

General Classes

PartUsage

Attributes

/renderingDefinition : RenderingDefinition {redefines partDefinition}

The RenderingDefinition that defines this RenderingUsage.

Operations

No operations.

Constraints

No constraints.

7.22.3.5 ViewDefinition

Description

A ViewDefinition is a PartDefinition that specifies how a view artifact is constructed to satisfy a viewpoint. It specifies a `viewConditions` to define the model content to be presented and a `rendering` to define how the model content is presented.

A ViewDefinition must subclass, directly or indirectly, the base ViewDefinition View from the Systems model library.

General Classes

PartDefinition

Attributes

/satisfiedViewpoint : ViewpointUsage [0..*] {subsets ownedUsage}

The `ownedUsages` of this ViewDefinition that are ViewpointUsages for viewpoints satisfied by the ViewDefinition.

/view : ViewUsage [0..*] {subsets usage}

The `usages` of this ViewDefinition that are ViewUsages.

/viewCondition : Expression [0..*] {subsets ownedMember}

The Expressions related to this ViewDefinition by ElementFilterMemberships, which specify conditions on Elements to be rendered in a view.

/viewRendering : RenderingUsage [0..1] {subsets ownedUsage}

The RenderingUsage to be used to render views defined by this ViewDefinition. Derived as the `referencedRendering` of the ViewRenderingMembership of the ViewDefinition. A ViewDefinition may have at most one.

Operations

No operations.

Constraints

No constraints.

7.22.3.6 ViewpointDefinition

Description

A ViewpointDefinition is a RequirementDefinition that specifies one or more stakeholder concerns that to be satisfied by created a view of a model.

A ViewpointDefinition must subclass, directly or indirectly, the base ViewpointDefinition Viewpoint from the Systems model library.

General Classes

RequirementDefinition

Attributes

/viewpointStakeholder : StakeholderUsage [0..*]

The features that identify the stakeholders with concerns addressed by this ViewpointDefinition, derived as the owned and inherited Stakeholders of the `addressedConcerns` of this ViewpointDefinition.

Operations

No operations.

Constraints

No constraints.

7.22.3.7 ViewpointUsage

Description

A ViewpointUsage is a usage of a ViewpointDefinition.

A ViewpointUsage must subset, directly or indirectly, the base ViewpointUsage `viewpoints` from the Systems model library.

General Classes

RequirementUsage

Attributes

/viewpointDefinition : ViewpointDefinition {redefines requirementDefinition}

The ViewpointDefinition that defines this ViewUsage.

/viewpointStakeholder : StakeholderUsage [0..*]

The features that identify the stakeholders with concerns addressed by this ViewpointUsage, derived as the owned and inherited Stakeholders of the `addressedConcerns` of this ViewpointUsage.

Operations

No operations.

Constraints

No constraints.

7.22.3.8 ViewRenderingMembership

Description

A `ViewRenderingMembership` is a `FeatureMembership` that identifies the `viewRendering` of a `View`. The `ownedMemberFeature` of a `RequirementConstraintMembership` must be a `RenderingUsage`.

General Classes

`FeatureMembership`

Attributes

`ownedRendering` : `RenderingUsage`

`/referencedRendering` : `RenderingUsage`

The `RenderingUsage` that is referenced through this `ViewRenderingMembership`. This is derived as being the first `RenderingUsage` subset by the `ownedRendering`, if there is one, and, otherwise, the `ownedRendering` itself.

Operations

No operations.

Constraints

No constraints.

7.22.3.9 ViewUsage

Description

A `ViewUsage` is a usage of a `ViewDefinition` to specify the generation of a view of the members of a collection of `exposedNamespaces`. The `ViewDefinition` can satisfy more `viewpoints` than its definition, and it can specialize the `rendering` specified by its definition.

A `ViewUsage` must subset, directly or indirectly, the base `ViewUsage` views from the Systems model library.

General Classes

`PartUsage`

Attributes

`/exposedNamespace` : `Namespace` [0..*]

The `Namespaces` that are exposed by this `ViewUsage`, derived as the `Namespaces` related to the `ViewUsage` by `Expose Relationships`.

`/satisfiedViewpoint` : `ViewpointUsage` [0..*] {subsets `nestedUsage`}

The `nestedUsages` of this `ViewUsage` that are `ViewpointUsages` for (additional) viewpoints satisfied by the `ViewUsage`.

`/viewCondition` : `Expression` [0..*] {subsets `ownedMember`}

The `Expressions` related to this `ViewUsage` by `ElementFilterMemberships`, which specify conditions on `Elements` to be rendered in a view.

/viewDefinition : ViewDefinition {redefines partDefinition}

The definition of this ViewUsage.

/viewedElement : Element [0..*]

The Elements that are rendered by this ViewUsage, derived as the members of all the exposedNamespaces that met all the owned and inherited viewConditions.

/viewRendering : RenderingUsage [0..1] {subsets nestedUsage}

The RenderingUsage to be used to render views defined by this ViewUsage. Derived as the referencedRendering of the ViewRenderingMembership of the ViewUsage. A ViewUsage may have at most one.

Operations

No operations.

Constraints

No constraints.

7.22.4 Views Semantics

7.23 Language Extension

Submission Note. SysML v2 will include the ability to extend the language, in a similar way to which SysML is build on KerML. This will be fully addressed in the revised submission.

8 Model Libraries

Submission Note. The documentation provided in this clause is currently incomplete. Please refer to the textual notation files for the normative representation of all library models. Currently, the only domain library included is for Quantities and Units. Additional domain libraries are planned for the revised submission, at least for Basic Geometry.

8.1 Systems Model Library

8.1.1 Overview

8.1.2 Attributes

8.1.2.1 Attributes Overview

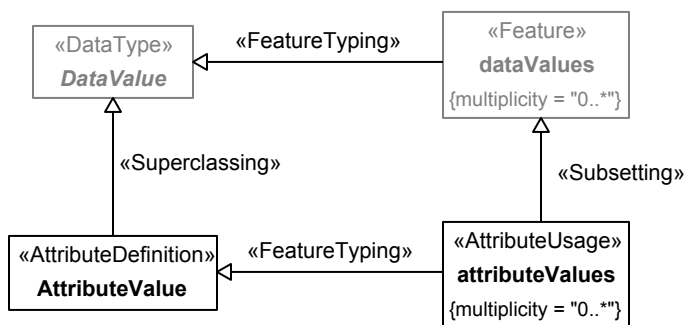


Figure 69. Attributes Model Library

8.1.2.2 Elements

8.1.2.2.1 attributeValues <AttributeUsage>

Description

attributeValues is the base feature for all AttributeUsages.

General Classes

AttributeValue
dataValues

Attributes

No attributes.

Constraints

No constraints.

8.1.2.2.2 AttributeValue <AttributeDefinition>

Description

AttributeValue is the most general type of data values that represent qualities or characteristics of a system or part of a system. AttributeValue is the base type of all AttributeDefinitions.

General Classes

DataValue

Attributes

No attributes.

Constraints

No constraints.

8.1.3 Items

8.1.3.1 Items Overview

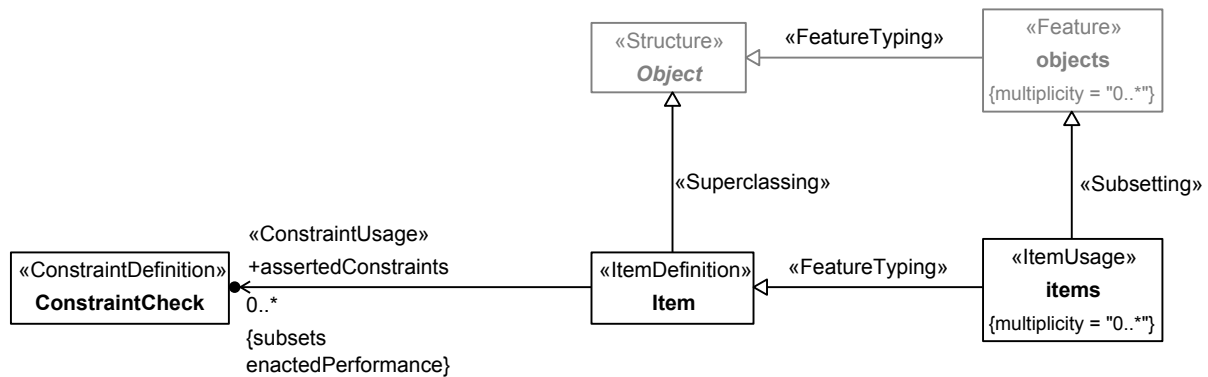


Figure 70. Items Model Library

8.1.3.2 Elements

8.1.3.2.1 Item <ItemDefinition>

Description

Item is the most general class of objects that are part of, exist in or flow through a system. Item is the base type of all ItemDefinitions.

General Classes

Object

Attributes

assertedConstraints : ConstraintCheck [0..*] {subsets enactedPerformance}

Constraints

No constraints.

8.1.3.2.2 items <ItemUsage>

Description

items is the base feature of all ItemUsages.

General Classes

objects
Item

Attributes

No attributes.

Constraints

No constraints.

8.1.4 Parts

8.1.4.1 Parts Overview

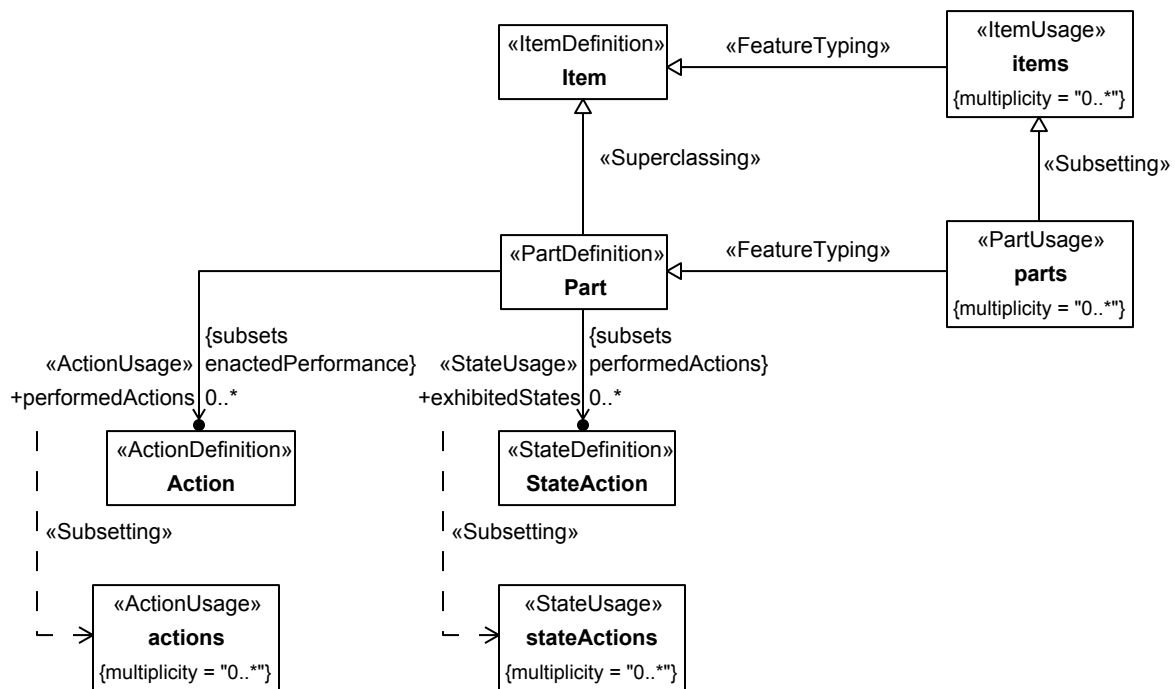


Figure 71. Parts Library Model

8.1.4.2 Elements

8.1.4.2.1 Part <PartDefinition>

Description

Part is the most general class of objects that represent all or a part of a system. Part is the base type of all PartDefinitions.

General Classes

Item

Attributes

exhibitedStates : StateAction [0..*] {subsets performedActions}

performedActions : Action [0..*] {subsets enactedPerformance}

Actions that are performed by this part.

Constraints

No constraints.

8.1.4.2.2 parts <PartUsage>

Description

parts is the base feature of all PartUsages.

General Classes

Part
items

Attributes

No attributes.

Constraints

No constraints.

8.1.5 Ports

8.1.5.1 Ports Overview

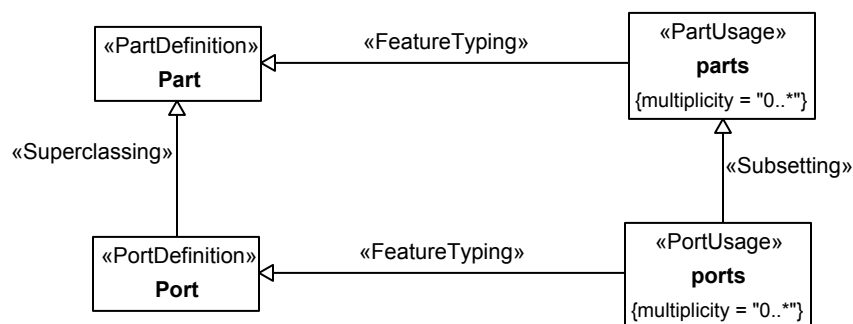


Figure 72. Ports Library Model

8.1.5.2 Elements

8.1.5.2.1 Port <PortDefinition>

Description

Port is the most general class of objects that represent connection points for interacting with a Part. Port is the base type of all PortDefinitions.

General Classes

Part

Attributes

No attributes.

Constraints

No constraints.

8.1.5.2.2 ports <PortUsage>

Description

ports is the base feature of all PortUsages.

General Classes

Port
parts

Attributes

No attributes.

Constraints

No constraints.

8.1.6 Connections

8.1.6.1 Connections Overview

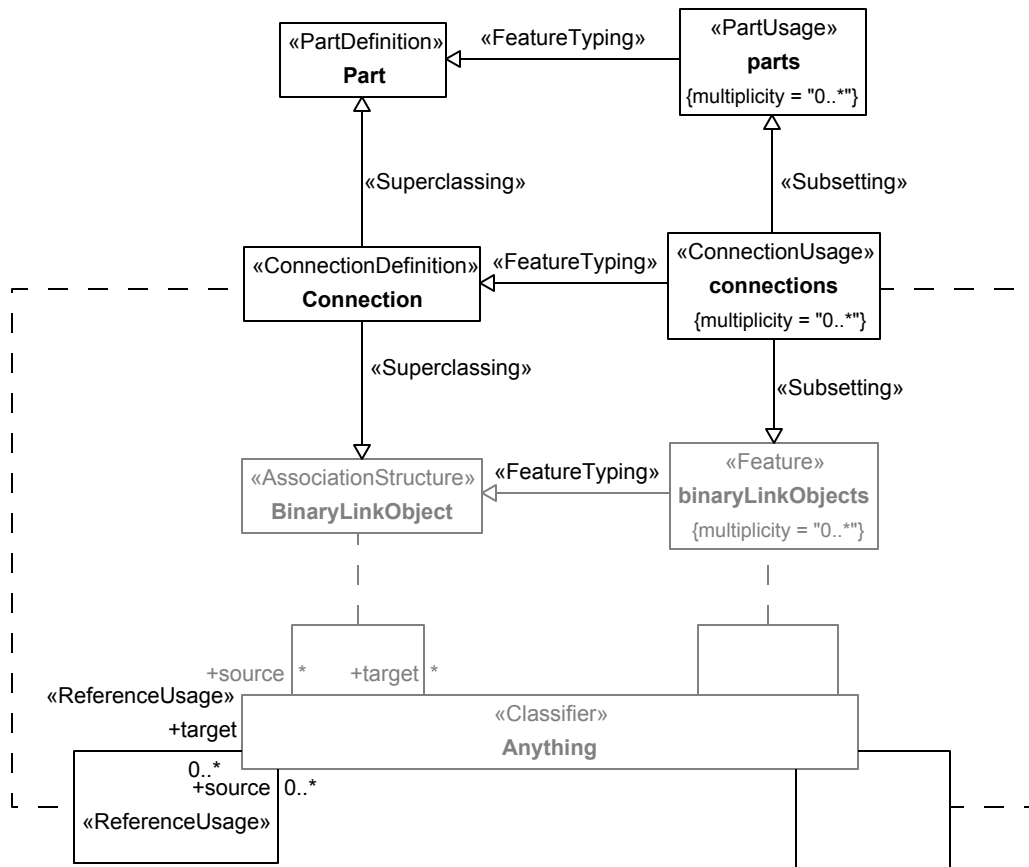


Figure 73. Connections Library Model

8.1.6.2 Elements

8.1.6.2.1 Connection <ConnectionDefinition>

Description

Connection is the most general class of connections between two Parts within some containing structure. Connection is the base type of all ConnectionDefinitions.

(Note that this does not include BindingConnectors, which are typed by the kernel Association SelfLink.)

General Classes

Part
BinaryLinkObject

Attributes

source : Anything [0..*]

target : Anything [0..*]

Constraints

No constraints.

8.1.6.2.2 connections <ConnectionUsage>

Description

connections is the base feature of all ConnectionUsages.

General Classes

binaryLinkObjects

parts

Connection

Attributes

[no name] : Anything

[no name] : Anything

Constraints

No constraints.

8.1.7 Interfaces

8.1.7.1 Interfaces Overview

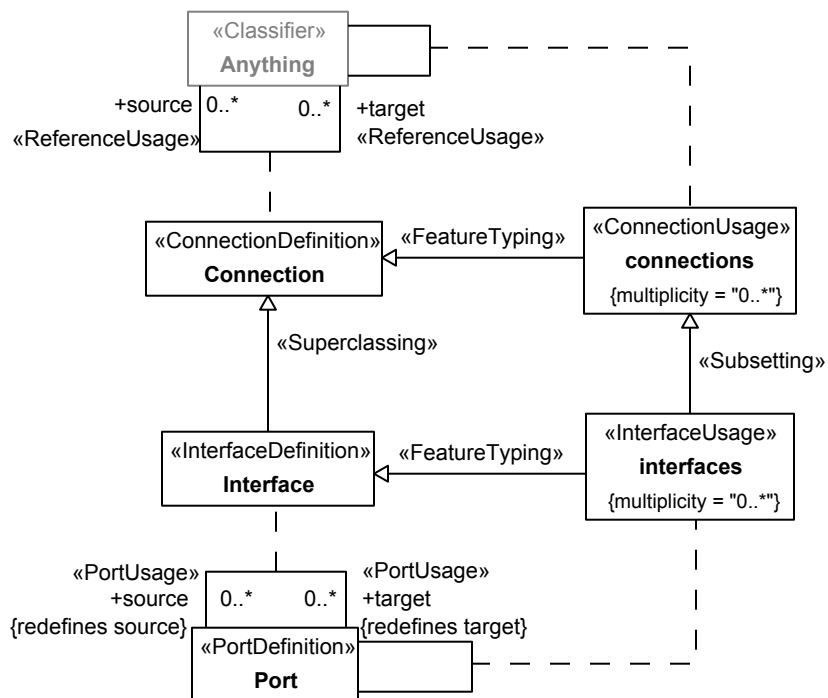


Figure 74. Interfaces Library Model

8.1.7.2 Elements

8.1.7.2.1 Interface <InterfaceDefintion>

Description

Interface is the most general class of connections between two Ports on Parts within some containing structure. Interface is the base type of all InterfaceDefinitions.

General Classes

Connection

Attributes

source : Port [0..*] {redefines source}

target : Port [0..*] {redefines target}

Constraints

No constraints.

8.1.7.2.2 interfaces <InterfaceUsage>

Description

interfaces is the base feature of all InterfaceUsages.

General Classes

Interface
connections

Attributes

[no name] : Port

[no name] : Port

Constraints

No constraints.

8.1.8 Allocations

8.1.8.1 Allocations Overview

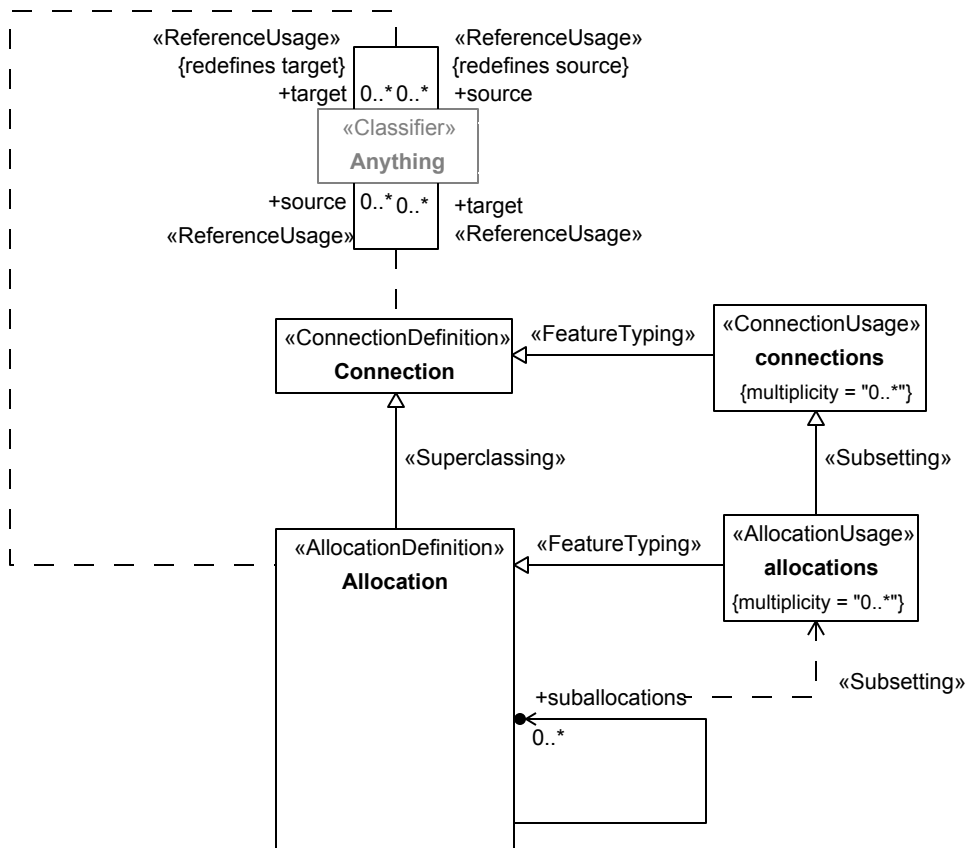


Figure 75. Allocations Library Model

8.1.8.2 Elements

8.1.8.2.1 Allocation <AllocationDefinition>

Description

Allocation is the most general class of allocation, represented as a connection between the source of the allocation and the target. Allocation is the base type of all AllocationDefinitions.

General Classes

Connection

Attributes

source : Anything [0..*] {redefines source}

suballocations : Allocation [0..*]

target : Anything [0..*] {redefines target}

Constraints

No constraints.

8.1.8.2.2 allocations <AllocationUsage>

Description

allocations is the base feature of all ConnectionUsages.

General Classes

Allocation
connections

Attributes

[no name] : Anything

[no name] : Anything

Constraints

No constraints.

8.1.9 Actions

8.1.9.1 Actions Overview

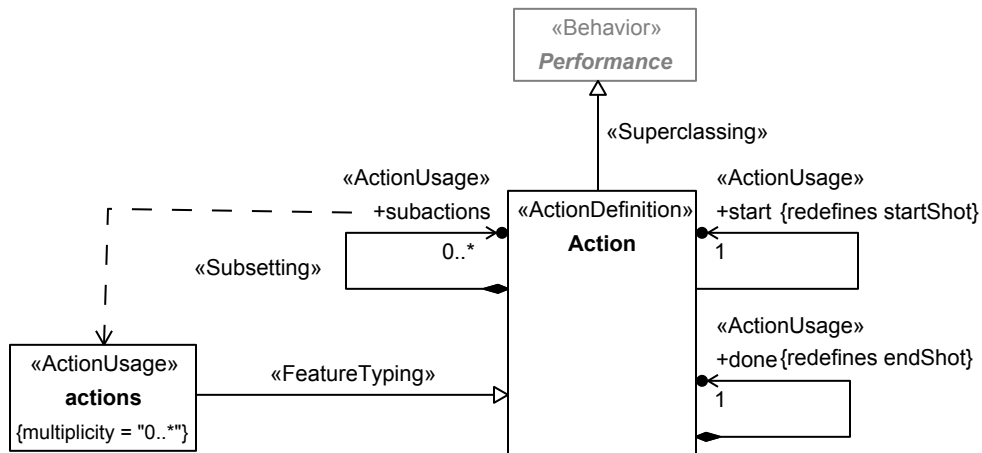


Figure 76. Actions Library Model

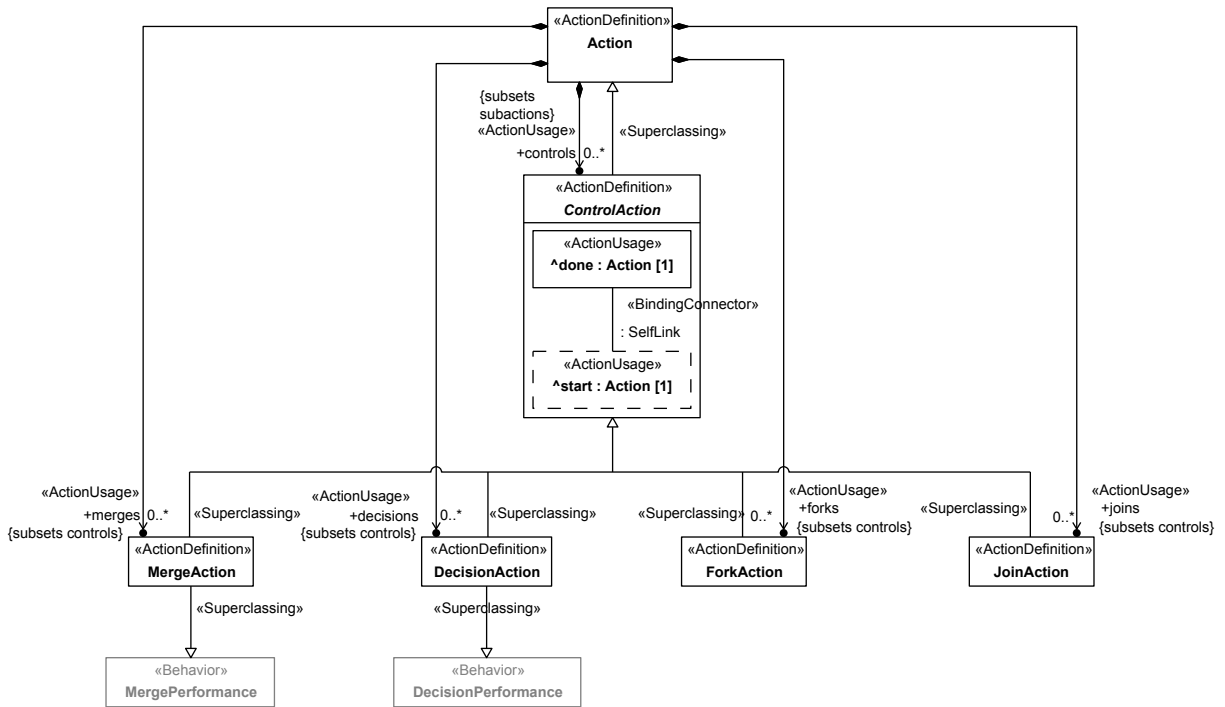


Figure 77. Control Nodes Library Model

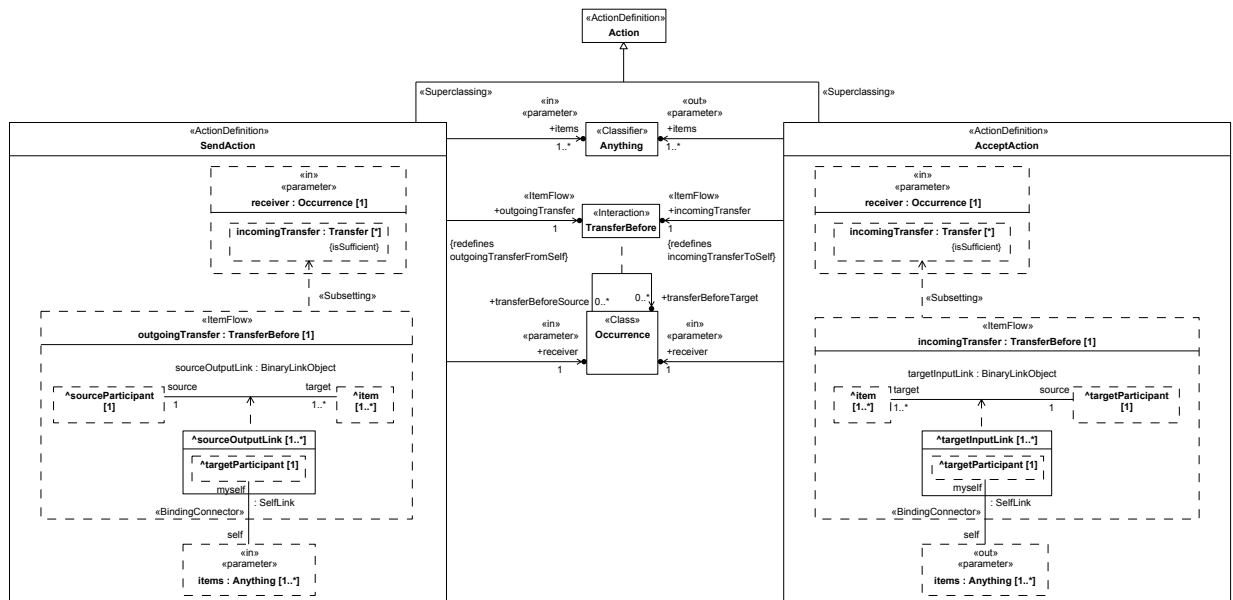


Figure 78. Send and Accept Actions

8.1.9.2 Elements

8.1.9.2.1 AcceptAction <ActionDefinition>

Description

An `AcceptAction` is an `Action` used to type an `AcceptActionUsage`. It completes an `incomingTransferToSelf` that is one of the `incomingTransfers` of a given `receiver Occurrence`, outputting the payload of `items` from the `Transfer`.

General Classes

`Action`

Attributes

`incomingTransfer` : `TransferBefore` {redefines `incomingTransferToSelf`}

The `Transfer` accepted by this `AcceptAction`.

`items` : `Anything` [1..*]

The payload received from the incoming `Transfer`.

`receiver` : `Occurrence`

The `Occurrence` from whose `incomingTransfers` the `incomingTransfer` of the `AcceptAction` is accepted.

Constraints

No constraints.

8.1.9.2.2 Action <ActionDefinition>

Description

`Action` is the most general class of performances of `ActionDefinitions` in a system or part of a system. `Action` is the base class of all `ActionDefinitions`.

General Classes

`Performance`

Attributes

`controls` : `ControlAction` [0..*] {subsets `subactions`}

The `subactions` of this activity that are control actions.

`decisions` : `DecisionAction` [0..*] {subsets `controls`}

The control actions of this activity that are decision actions.

`done` : `Action` {redefines `endShot`}

The ending snapshot of an action.

`forks` : `ForkAction` [0..*] {subsets `controls`}

The control actions of this activity that are fork actions.

joins : JoinAction [0..*] {subsets controls}

The control actions of this activity that are join actions.

merges : MergeAction [0..*] {subsets controls}

The control actions of this activity that are merge actions.

start : Action {redefines startShot}

The starting snapshot of an action.

subactions : Action [0..*]

The subperformances of this action that are actions.

subtransitions : TransitionAction [0..*]

Constraints

No constraints.

8.1.9.2.3 actions <ActionUsage>

Description

actions is the base feature for all ActionUsages.

General Classes

Action

Attributes

No attributes.

Constraints

No constraints.

8.1.9.2.4 ControlAction <ActionDefinition>

Description

A ControlAction is the Action of a ControlNode, which has no inherent behavior.

General Classes

Action

Attributes

No attributes.

Constraints

No constraints.

8.1.9.2.5 DecisionAction <ActionDefinition>

Description

A DecisionAction is the ControlAction for a DecisionNode. It is a DecisionPerformance that selects one outgoing HappensBeforeLink.

General Classes

DecisionPerformance
ControlAction

Attributes

No attributes.

Constraints

No constraints.

8.1.9.2.6 ForkAction <ActionDefinition>

Description

A ForkAction is the ControlAction for a ForkNode.

Note: Fork behavior results from requiring that the target multiplicity of all outgoing succession connectors be 1..1.

General Classes

ControlAction

Attributes

No attributes.

Constraints

No constraints.

8.1.9.2.7 JoinAction <ActionDefinition>

Description

A JoinAction is the ControlAction for a JoinNode.

Note: Join behavior results from requiring that the source multiplicity of all incoming succession connectors be 1..1.

General Classes

ControlAction

Attributes

No attributes.

Constraints

No constraints.

8.1.9.2.8 MergeAction <ActionDefinition>

Description

A MergeAction is the ControlAction for a merge node. It is a MergePerformance that selects exactly one incoming HappensBefore link.

General Classes

ControlAction
MergePerformance

Attributes

No attributes.

Constraints

No constraints.

8.1.9.2.9 SendAction <ActionDefinition>

Description

A SendAction is an Action used to type a SendActionUsage. It initiates an outgoingTransferFromSelf to a designated receiver Occurrence with a given payload of items.

General Classes

Action

Attributes

items : Anything [1..*]

The payload to be sent in the outgoing Transfer.

outgoingTransfer : TransferBefore {redefines outgoingTransferFromSelf}

The Transfer initiated by this SendAction.

receiver : Occurrence

The Occurrence that receives the outgoingTransfer as an incomingTransfer.

Constraints

No constraints.

8.1.10.1 States Overview



8.1.10.2.1 StateAction <StateDefinition>

A `StateAction` is a kind of `Action` that is also a `StatePerformance`. It is the base type for all `StateDefinitions`.

General Classes

StatePerformance
Action

Attributes

subactions : Action [0..*] {subsets middle, redefines subactions}

The subperformances of this StateAction that are Actions, other than the entry and exit Actions. These subactions all take place in the "middle" of the StatePerformance, that is, after the entry Action and before the exit Action.

substates : StateAction [0..*] {subsets subactions}

The subactions of this StateAction that are StateActions. These substates all take place in the "middle" of the StatePerformance, that is, after the entry Action and before the exit Action.

Constraints

No constraints.

8.1.10.2.2 stateActions <StateUsage>

Description

stateActions is the base feature for all StateUsages.

General Classes

actions
StateAction

Attributes

No attributes.

Constraints

No constraints.

8.1.10.2.3 TransitionAction <ActionDefintion>

Description

A TransitionAction is a StateTransitionPerformance whose transitionLinkSource is an Action. It is the base type of all TransitionUsages.

General Classes

StateTransitionPerformance
Action

Attributes

accepter : AcceptAction [0..1] {subsets subactions}
effect : Action [0..*] {subsets subactions, redefines effect}
transitionLinkSource : Action {redefines transitionLinkSource}

Constraints

No constraints.

8.1.10.2.4 transitionActions <TransitionUsage>

Description

transitionActions is the base feature for all TransitionUsages.

General Classes

actions
Action
TransitionAction

Attributes

No attributes.

Constraints

No constraints.

8.1.11 Calculations

8.1.11.1 Calculations Overview

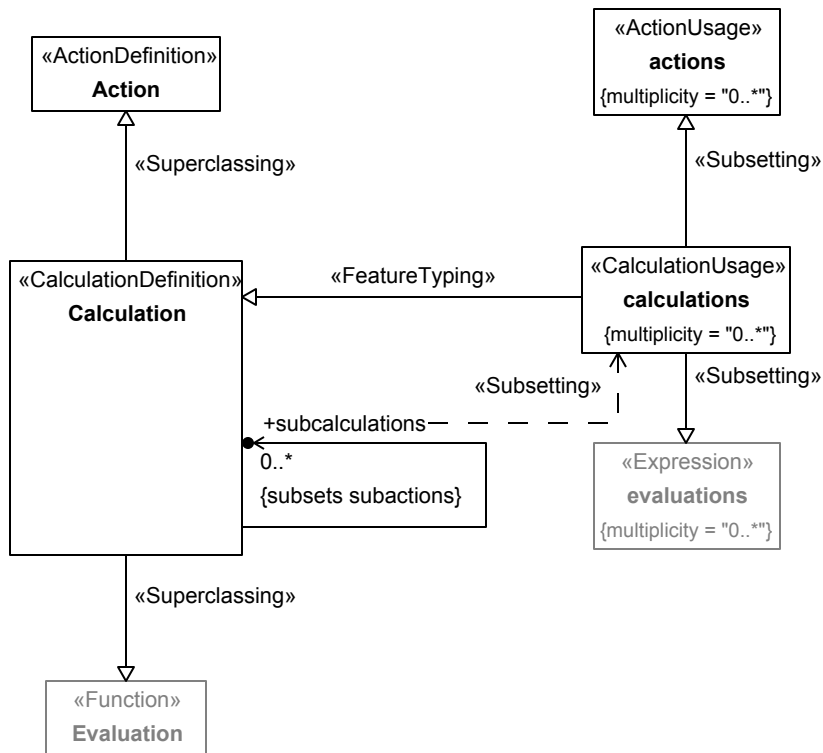


Figure 81. Calculations Library Model

8.1.11.2 Elements

8.1.11.2.1 Calculation <CalculationDefinition>

Description

Calculation is the most general class of evaluations of CalculationDefinitions in a system or part of a system. Calculation is the base class of all CalculationDefinitions.

General Classes

Action
Evaluation

Attributes

subcalculations : Calculation [0..*] {subsets subactions}

The subactions of this FunctionInvocation that are FunctionInvocations.

Constraints

No constraints.

8.1.11.2.2 calculations <CalculationUsage>

Description

calculations is the base Feature for all CalculationUsages.

.

General Classes

Calculation
actions
evaluations

Attributes

No attributes.

Constraints

No constraints.

8.1.12 Constraints

8.1.12.1 Constraints Overview

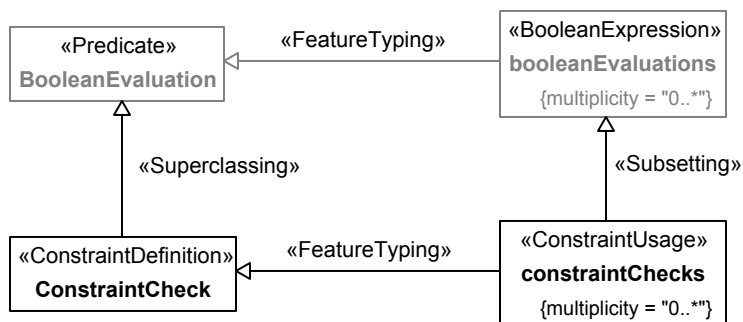


Figure 82. Constraints Library Model

8.1.12.2 Elements

8.1.12.2.1 ConstraintCheck <ConstraintDefinition>

Description

ConstraintCheck is the most general class for constraint checking. ConstraintCheck is the base type of all ConstraintDefinitions.

General Classes

BooleanEvaluation

Attributes

No attributes.

Constraints

No constraints.

8.1.12.2 constraintChecks <ConstraintUsage>

Description

`constraintChecks` is the base feature of all `ConstraintUsages`.

General Classes

`booleanEvaluations`

`ConstraintCheck`

Attributes

No attributes.

Constraints

No constraints.

8.1.13 Requirements

8.1.13.1 Requirements Overview

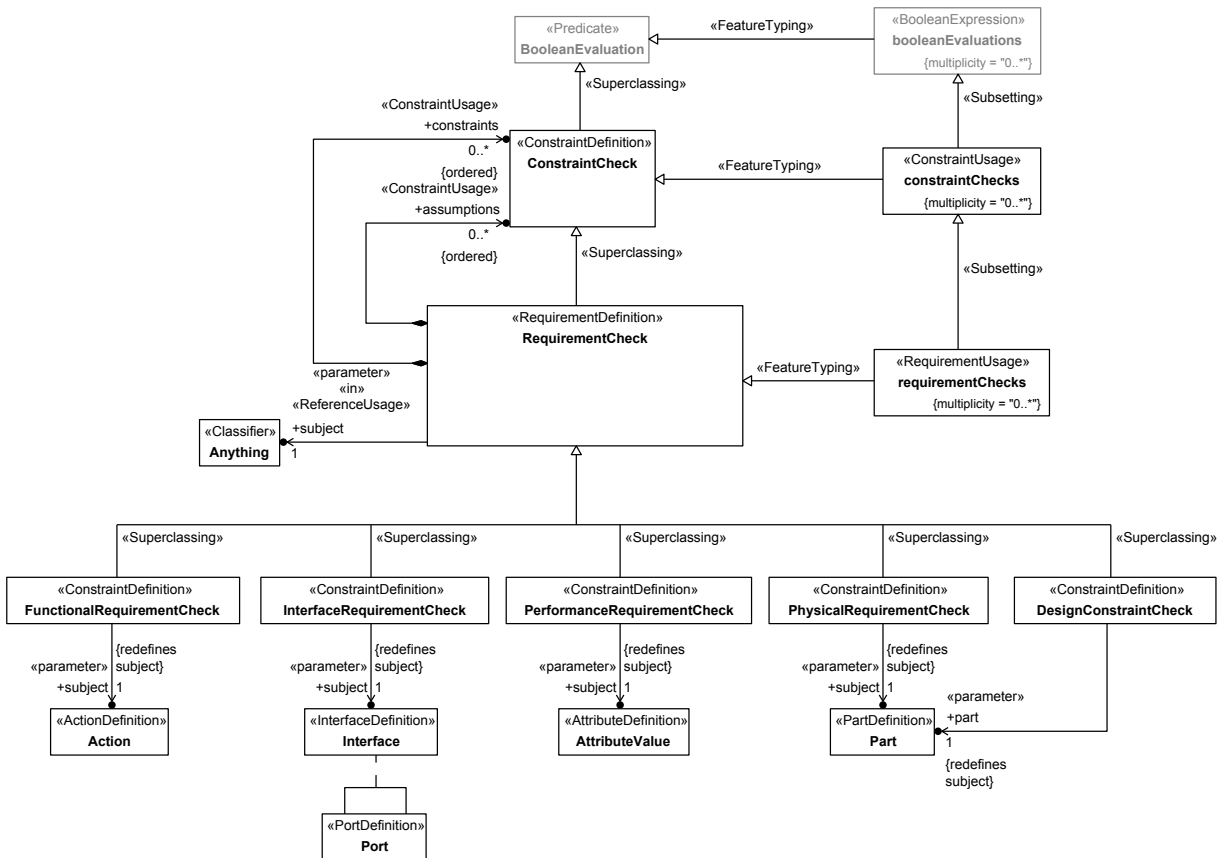


Figure 83. Requirements Library Model

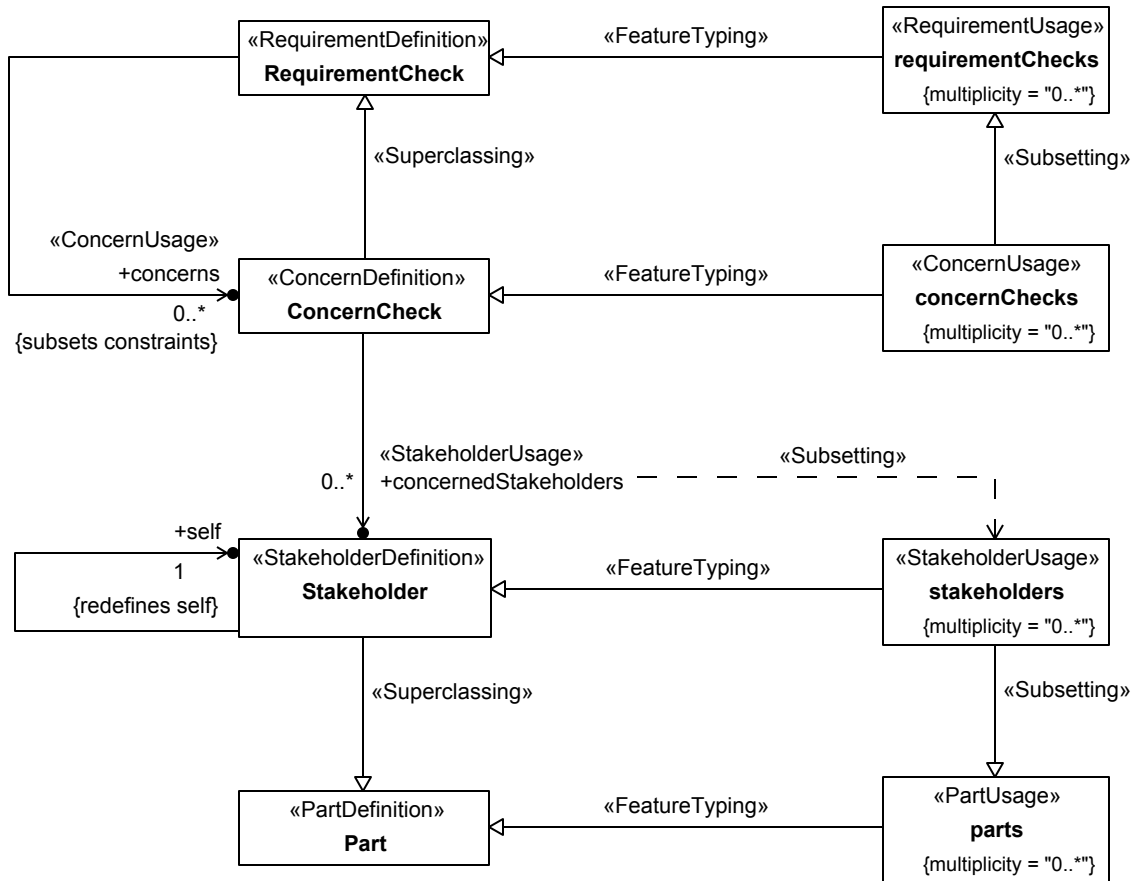


Figure 84. Stakeholders and Concerns Library Model

8.1.13.2 Elements

8.1.13.2.1 ConcernCheck <ConcernDefinition>

Description

ConcernCheck is the most general class for concern checking. ConcernCheck is the base type of all ConcernDefinitions.

General Classes

RequirementCheck

Attributes

concernedStakeholders : Stakeholder [0..*]

The stakeholders interested in the concern being checked.

Constraints

No constraints.

8.1.13.2.2 concernChecks <ConcernUsage>

Description

concernChecks is the base feature of all ConcernUsages.

General Classes

requirementChecks
ConcernCheck

Attributes

No attributes.

Constraints

No constraints.

8.1.13.2.3 DesignConstraintCheck <ConstraintDefinition>

Description

A DesignConstraint specifies a constraint on the implementation of the system or system part, such as the system must use a commercial-off-the-shelf component.

General Classes

RequirementCheck

Attributes

part : Part {redefines subject}

Constraints

No constraints.

8.1.13.2.4 FunctionalRequirementCheck <ConstraintDefinition>

Description

A FunctionalRequirementCheck specifies an action that a system, or part of a system, must perform.

General Classes

RequirementCheck

Attributes

subject : Action {redefines subject}

Constraints

No constraints.

8.1.13.2.5 InterfaceRequirementCheck <ConstraintDefinition>

Description

An InterfaceRequirement Check specifies an Interface for connecting systems and system parts, which optionally may include item flows across the Interface and/or Interface constraints.

General Classes

RequirementCheck

Attributes

subject : Interface {redefines subject}

Constraints

No constraints.

8.1.13.2.6 PerformanceRequirementCheck <ConstraintDefinition>

Description

A PerformanceRequirementCheck quantitatively measures the extent to which a system, or a system part, satisfies a required capability or condition.

General Classes

RequirementCheck

Attributes

subject : AttributeValue {redefines subject}

Constraints

No constraints.

8.1.13.2.7 PhysicalRequirementCheck <ConstraintDefinition>

Description

A PhysicalRequirementCheck specifies physical characteristics and/or physical constraints of the system, or a system part.

General Classes

RequirementCheck

Attributes

subject : Part {redefines subject}

Constraints

No constraints.

8.1.13.2.8 RequirementCheck <RequirementDefinition>

Description

RequirementCheck is the most general class for requirements checking. RequirementCheck is the base type of all RequirementDefinitions.

General Classes

ConstraintCheck

Attributes

assumptions : ConstraintCheck [0..*] {ordered}

The checks of assumptions that must hold for the required constraints to apply.

concerns : ConcernCheck [0..*] {subsets constraints}

The checks of any concerns being addressed (as required constraints).

constraints : ConstraintCheck [0..*] {ordered}

The checks of required constraints.

subject : Anything

The entity that is being check for satisfaction of the required constraints.

Constraints

[no name]

[no documentation]

`allTrue(assumptions) implies allTrue(constraints)`

8.1.13.2.9 requirementChecks <RequirementUsage>

Description

requirementChecks is the base feature of all RequirementUsages.

General Classes

RequirementCheck
constraintChecks

Attributes

No attributes.

Constraints

No constraints.

8.1.13.2.10 Stakeholder <StakeholderDefinition>

Description

Stakeholder is the most general class of objects that represent a stakeholder with identified concerns. Stakeholder is the base type of all StakeholderDefinitions.

General Classes

Part

Attributes

self : Stakeholder {redefines self}

Constraints

No constraints.

8.1.13.2.11 stakeholders <StakeholderUsage>

Description

stakeholders is the base feature of all StakeholderUsages.

General Classes

parts

Stakeholder

Attributes

No attributes.

Constraints

No constraints.

8.1.14 Cases

8.1.14.1 Cases Overview

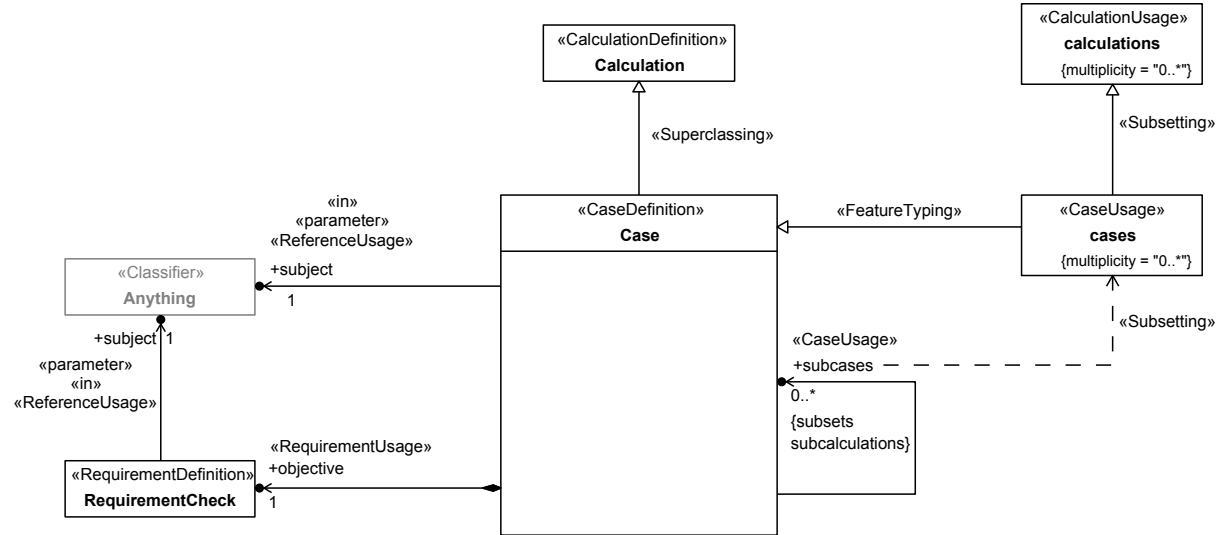


Figure 85. Cases Library Model

8.1.14.2 Elements

8.1.14.2.1 Case <CaseDefinition>

Description

Case is the most general class of performances of CaseDefinitions. Case is the base class of all CaseDefinitions.

General Classes

Calculation

Attributes

objective : RequirementCheck

A check of whether the objective RequirementUsage was satisfied for this Case.

subcases : Case [0..*] {subsets subcalculations}

Other Cases carried out as part of the performance of this Case.

subject : Anything

The subject that was investigated by this Case.

Constraints

No constraints.

8.1.14.2.2 cases <CaseUsage>

Description

cases is the base feature of all CaseUsages.

General Classes

calculations

Case

Attributes

No attributes.

Constraints

No constraints.

8.1.15 Analysis Cases

8.1.15.1 Analysis Cases Overview

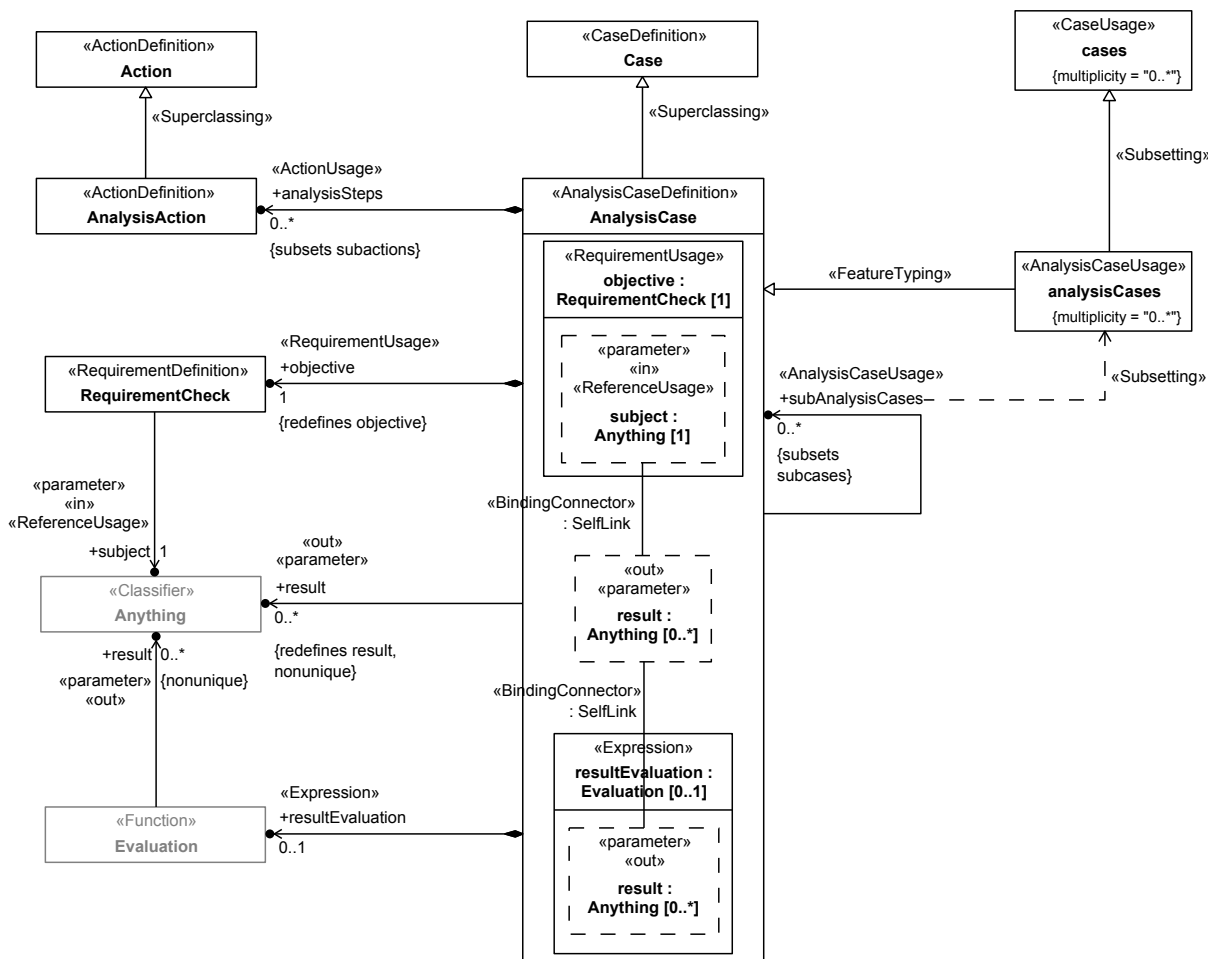


Figure 86. Analysis Case Library Model

8.1.15.2 Elements

8.1.15.2.1 AnalysisAction <ActionDefinition>

Description

An AnalysisAction is a specialized kind of Action used intended to be used as a step in an AnalysisCase.

General Classes

Action

Attributes

No attributes.

Constraints

No constraints.

8.1.15.2.2 AnalysisCase <AnalysisCaseDefinition>

Description

AnalysisCase is the most general class of performances of AnalysisCaseDefinitions. AnalysisCase is the base class of all AnalysisCaseDefinitions.

General Classes

Case

Attributes

analysisSteps : AnalysisAction [0..*] {subsets subactions}

The subactions of this AnalysisCase that are AnalysisActions.

objective : RequirementCheck {redefines objective}

The objective of this AnalysisCase, whose subject is bound to the result of the AnalysisCase.

result : Anything [0..*] {redefines result, nonunique}

The result of this AnalysisCase, which is bound to the result of the resultEvaluation.

resultEvaluation : Evaluation [0..1]

The Evaluation of the resultExpression from the definition of this AnalysisCase.

subAnalysisCases : AnalysisCase [0..*] {subsets subcases}

The subcases

of this AnalysisCase that are AnalysisCaseUsages.

Constraints

No constraints.

8.1.15.2.3 analysisCases <AnalysisCaseUsage>

Description

analysisCases is the base feature of all AnalysisCaseUsages.

General Classes

AnalysisCase
cases

Attributes

No attributes.

Constraints

No constraints.

8.1.16 Verification Cases

8.1.16.1 Verification Cases Overview

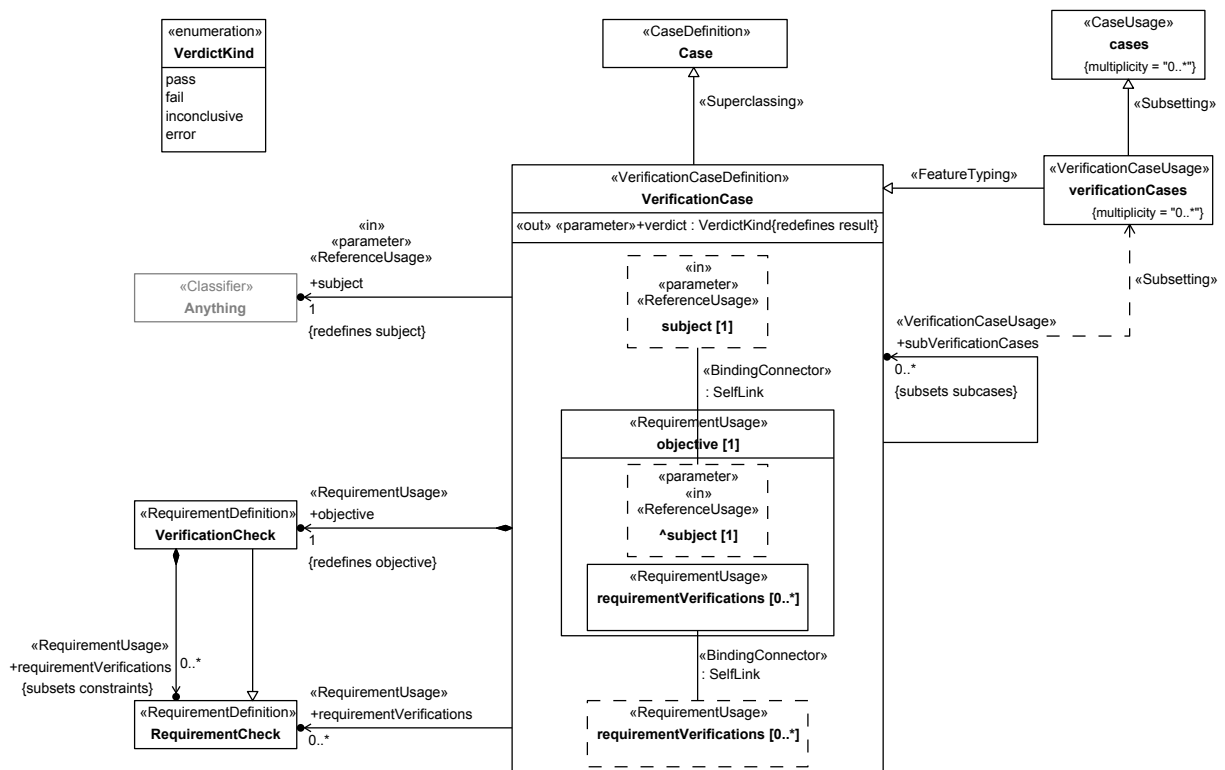


Figure 87. Verification Case Library Model

8.1.16.2 Elements

8.1.16.2.1 VerdictKind <Enumeration>

Description

VerdictKind is an enumeration of the possible results of a VerificationCase.

General Classes

No general classes.

Literal Values

error

An error occurred while evaluating the ValidationCase.

fail

The VerificationCase failed to achieve its objective.

inconclusive

The result of the VerificationCase was inconclusive.

pass

The VerificationCase passed, achieving its objective.

8.1.16.2.2 VerificationCase <VerificationCaseDefinition>

Description

VerificationCase is the most general class of performances of VerificationCaseDefinitions. VerificationCase is the base class of all VerificationCaseDefinitions.

General Classes

Case

Attributes

objective : VerificationCheck {redefines objective}

The objective this VerificationCase, whose `subject` is bound to the `subject` of the VerificationCase and whose `requirementVerifications` are bound to the `requirementVerifications` of the VerificationCase.

requirementVerifications : RequirementCheck [0..*]

Checks on whether the `verifiedRequirements` of the VerificationCase have been satisfied.

subject : Anything {redefines subject}

The subject of this VerificationCase, representing the system under test, which is bound to the `subject` of the `objective` of the VerificationCase.

subVerificationCases : VerificationCase [0..*] {subsets subcases}

The subcases

of this VerificationCase that are VerificationCaseUsages.

verdict : VerdictKind {redefines result}

The result of a VerificationCase must be a VerdictKind.

Constraints

No constraints.

8.1.16.2.3 verificationCases <VerificationCaseUsage>

Description

verificationCases is the base feature of all VerificationCaseUsages.

General Classes

VerificationCase
cases

Attributes

No attributes.

Constraints

No constraints.

8.1.16.2.4 VerificationCheck <RequirementDefinition>

Description

VerificationCheck is a specialization of RequirementCheck used for the objective of a VerificationCase in order to record the evaluations of the RequirementChecks of requirements being verified.

General Classes

RequirementCheck

Attributes

requirementVerifications : RequirementCheck [0..*] {subsets constraints}

Constraints

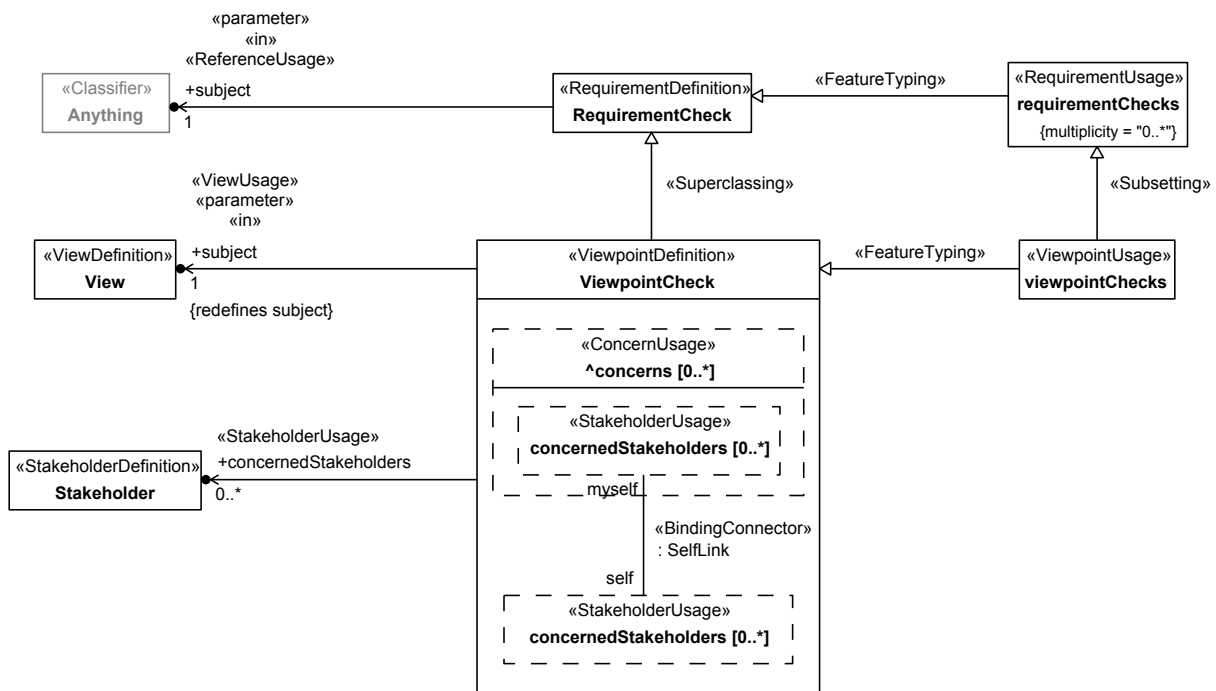
No constraints.

8.1.17 Views

Figure 89. Viewpoints Library Model



OMG Systems Modeling Language (SysML) v2.0, Submission



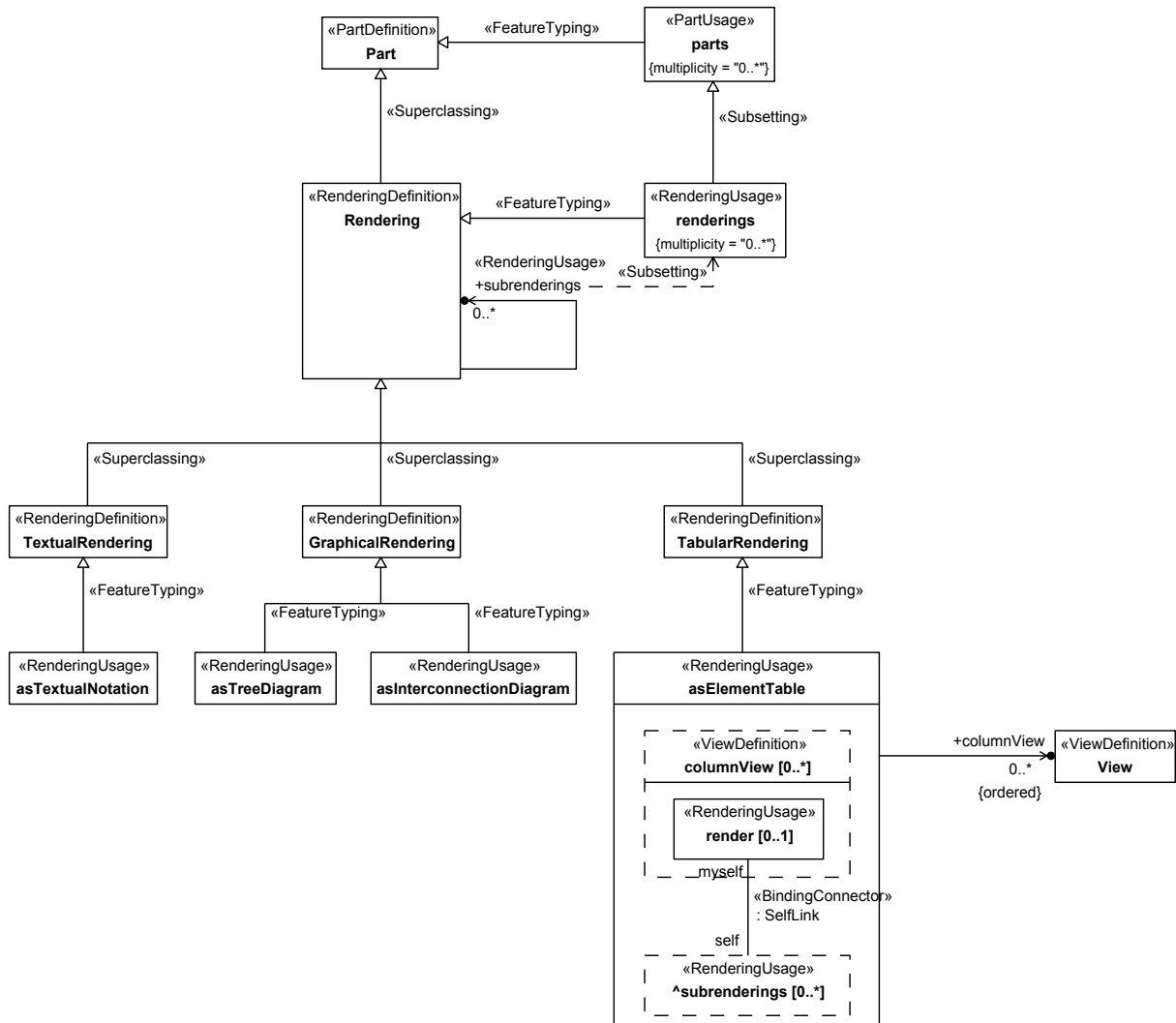


Figure 90. Renderings Library Model

8.1.17.2 Elements

8.1.17.2.1 asElementTable <RenderingUsage>

Description

asElementTable renders a View as a table, with one row for each exposed Element and columns rendered by applying the columnViews in order to the Element in each row.

General Classes

TabularRendering

Attributes

columnView : View [0..*] {ordered}

The Views to be rendered in the column cells, in order, of each rows of the table.

Constraints

No constraints.

8.1.17.2.2 asInterconnectionDiagram <RenderingUsage>

Description

`asInterconnectionDiagram` renders a View as an interconnection diagram, using the graphical notation defined in the SysML specification.

General Classes

GraphicalRendering

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.3 asTextualNotation <RenderingUsage>

Description

`asTextualNotation` renders a View into textual notation as defined in the KerML and SysML specifications.

General Classes

TextualRendering

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.4 asTreeDiagram <RenderingUsage>

Description

`asTreeDiagram` renders a View as a tree diagram, using the graphical notation defined in the SysML specification.

General Classes

GraphicalRendering

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.5 GraphicalRendering <RenderingDefinition>

Description

A GraphicalRendering is a Rendering of a View into a Graphical format.

General Classes

Rendering

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.6 Rendering <RenderingDefinition>

Description

Rendering is the base type of all RenderingDefinitions.

General Classes

Part

Attributes

subrenderings : Rendering [0..*]

Other Renderings used to carry out this Rendering.

Constraints

No constraints.

8.1.17.2.7 renderings <RenderingUsage>

Description

renderings is the base feature of all RenderingUsages.

General Classes

Rendering
parts

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.8 TabularRendering <RenderingDefinition>

Description

A TabularRendering is a Rendering of a View into a tabular format.

General Classes

Rendering

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.9 TextualRendering <RenderingDefinition>

Description

A TextualRendering is a Rendering of a View into a textual format.

General Classes

Rendering

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.10 View <ViewDefinition>

Description

View is the base type of all ViewDefinitions.

General Classes

Part

Attributes

render : Rendering [0..1]

The Rendering of this View.

`self : View {redefines self}`

`subviews : View [0..*]`

Other Views that are used in the rendering of this View.

`viewpointConformance : viewpointConformance`

An assertion that all `viewpointSatisfactions` are true.

`viewpointSatisfactions : ViewpointCheck [0..*]`

Checks that the View satisfies all required ViewpointsUsages.

Constraints

No constraints.

8.1.17.2.11 ViewpointCheck <ViewpointDefinition>

Description

ViewpointCheck is a RequirementCheck for checking if a View meets the concerns of `concernedStakeholders`. It is the base type of all ViewpointDefinitions.

General Classes

RequirementCheck

Attributes

`concernedStakeholders : Stakeholder [0..*]`

The stakeholders interested in the concerns to be addressed by the `subject` View.

`subject : View {redefines subject}`

The subject of this ViewpointCheck, which must be a View.

Constraints

No constraints.

8.1.17.2.12 viewpointChecks <ViewpointUsage>

Description

`viewpointChecks`

is the base feature of all ViewpointUsages.

General Classes

ViewpointCheck
requirementChecks

Attributes

No attributes.

Constraints

No constraints.

8.1.17.2.13 viewpointConformance <SatisfyRequirementUsage>

Description

General Classes

RequirementCheck

Attributes

viewpointSatisfactions : ViewpointCheck [0..*] {subsets constraints}

The required ViewpointChecks.

Constraints

No constraints.

8.1.17.2.14 views <ViewUsage>

Description

views is the base feature of all ViewUsages.

General Classes

parts
View

Attributes

No attributes.

Constraints

No constraints.

8.2 Quantities and Units Domain Library

8.2.1 Overview

The Quantities and Values packages contain syntax and semantics to support well-defined engineering quantities as well as systems of units and non-scalar values.

For any system model, a solid foundation for the representation of quantities, measurement units and scales, quantity dimensions, coordinate systems as well as value conversions is essential. Quantity attributes are needed to specify many characteristics of a system of interest. The foundation should be a shareable resource that can be reused in models within and across organizations and projects in order to facilitate collaboration and model interoperability.

The most widely accepted, scrutinized, and globally used foundational collection of quantities and units is captured and maintained in:

- the International System of Quantities (ISQ)
- the International System of Units (SI)

These systems are formally standardized through the ISO/IEC 80000 series of standards. The top level concepts and semantics defined in this model library are derived from and mapped to the concepts and semantics specified in [ISO 80000-1] and [VIM], as directly as possible, but staying at a generic level. This enables representing the ISQ and SI, but also any other systems of quantities and units.

The data model in this library includes precise representation and unambiguous semantics of the relationships between quantities, units, scales and quantity dimensions. As a result, robust automated conversion between quantity values expressed in compatible measurement units or scales is enabled, as well as support for quantity dimension analysis of expressions and constraints.

The model library further contains lower level packages that represent actual quantities and units as specified in parts 3 to 14 of the ISO/IEC 80000 series. These packages are intended to be used as a basis, and then be extended and tailored for use by particular communities of practitioners and organizations.

Apart from SI, the system of US Customary Units is still in wide industrial use. In order to support this system, the library also contains a package of US Customary Units, their relationship with ISQ quantities, and their conversion factors to and from SI units as specified in [NIST SP-811].

8.2.2 Quantities

8.2.2.1 Quantities Overview

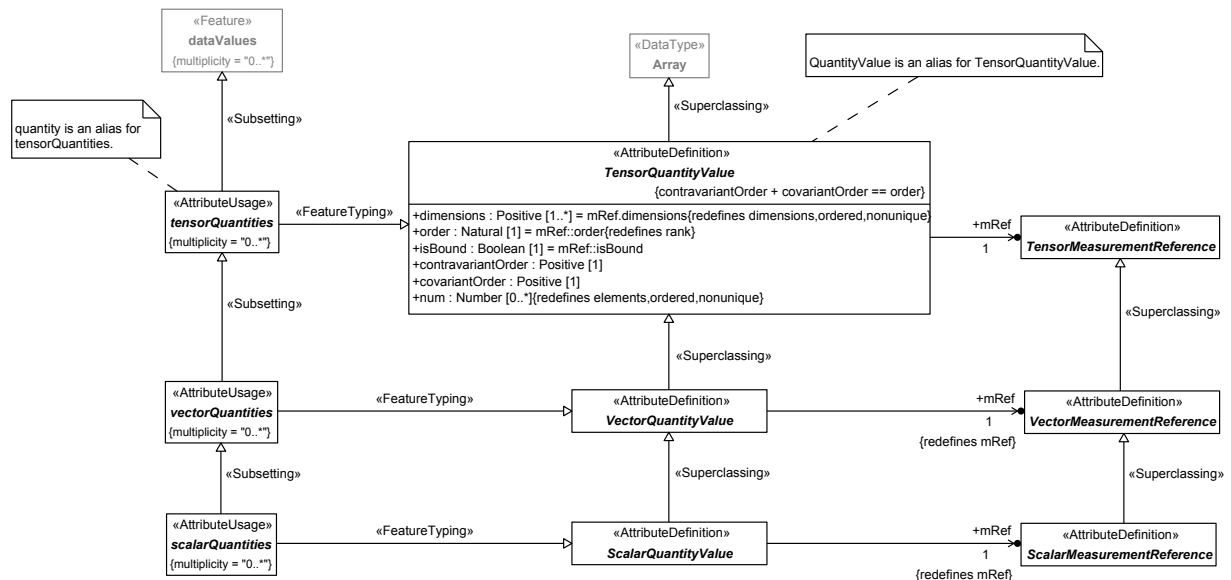


Figure 91. Quantities

8.2.2.2 Elements

8.2.2.2.1 scalarQuantities <AttributeUsage>

Description

`vectorQuantities` is the subset of `tensorQuantities` that are typed by `VectorQuantityValue` or a specialization of it.

General Classes

`vectorQuantities`
`ScalarQuantityValue`

Attributes

No attributes.

Constraints

No constraints.

8.2.2.2.2 ScalarQuantityValue <AttributeDefinition>

Description

A `ScalarQuantityValue` is an `AttributeDefinition` that represents the value of a scalar quantity by a tuple of a `Number` and a `ScalarMeasurementReference`. It is a specialization of `QuantityValue` and has `order` zero.

A scalar quantity can be free (`isBound` is false) or bound (`isBound` is true). A value of a free scalar quantity is expressed using just a measurement unit, as there is no need for particular choice of zero. A bound scalar quantity value must be expressed using an interval scale that includes an explicit specification of zero.

A free quantity is also often referred to as an absolute quantity. Similarly a bound quantity can be referred to as a relative quantity. Examples of free and bound quantity pairs that have the same `QuantityDimension` are: length and position, duration and time instant, kinetic energy and potential energy.

General Classes

`TensorQuantityValue`
`VectorQuantityValue`

Attributes

`mRef` : `ScalarMeasurementReference` {redefines `mRef`}

Attribute `mRef` is the `MeasurementReference` for this `QuantityValue`.

Constraints

`oneElement`

[no documentation]

`dimensions[1] == 1`

8.2.2.2.3 tensorQuantities <AttributeUsage>

Description

Quantities are defined as self-standing features that can be used to consistently specify quantities features of occurrences. Each single quantity feature is subsetting the root feature `tensorQuantities`. In other words, the codomain of a quantity feature is a suitable specialization of `TensorQuantityValue`.

General Classes

`TensorQuantityValue`
`dataValues`

Attributes

No attributes.

Constraints

No constraints.

8.2.2.2.4 TensorQuantityValue <AttributeDefinition>

Description

A `TensorQuantityValue` is an abstract `AttributeDefinition` that represents the value of a tensor, vector or scalar quantity value by a tuple of Numbers and a `MeasurementReference`.

The dimensionality of the quantity is specified in `dimensions`, from which the `order` of the quantity is derived. The order is two or greater for a tensor quantity, one for a vector quantity and zero for a scalar quantity. `QuantityValue` and `MeasurementReference` must have the same `dimensions` and `order`. It is possible to specify the contravariant and covariant order of a tensor or vector quantity through the attributes `contravariantOrder` and `covariantOrder`, of which the sum must equal the overall order.

A `QuantityValue` can be free (`isBound` is false) or bound (`isBound` is true). A value of a free quantity is expressed using a free `MeasurementReference` (which can be a coordinate system), in which there is no particular choice of origin or zero. A value of a bound quantity is expressed using a bound `MeasurementReference` that includes a specified choice of origin or zero.

General Classes

`Array`

Attributes

`contravariantOrder` : Positive

`covariantOrder` : Positive

`dimensions` : Positive [1..*] {redefines dimensions, ordered, nonunique}

`isBound` : Boolean

`mRef : TensorMeasurementReference`

Attribute `mRef` is the `MeasurementReference` for this `QuantityValue`.

`num : Number [0..*] {redefines elements, ordered, nonunique}`

`order : Natural {redefines rank}`

Constraints

`orderSum`

[no documentation]

`contravariantOrder + covariantOrder == order`

`matchingDimensions`

[no documentation]

`dimensions == mRef.dimensions`

8.2.2.2.5 vectorQuantities <AttributeUsage>

Description

`vectorQuantities` is the subset of `tensorQuantities` that are typed by `VectorQuantityValue` or a specialization of it.

General Classes

`tensorQuantities`

`VectorQuantityValue`

Attributes

No attributes.

Constraints

No constraints.

8.2.2.2.6 VectorQuantityValue <AttributeDefinition>

Description

A `VectorQuantityValue` is an `AttributeDefinition` that represents the value of a vector quantity by a tuple of Numbers and a `VectorMeasurementReference`. It is a specialization of `QuantityValue` and has `order` one.

A `VectorQuantityValue` can be free (`isBound` is false) or bound (`isBound` is true). A value of a free vector quantity is expressed using a free `VectorMeasurementReference`, in which there is no particular choice of zero or origin. A value of a bound vector quantity is expressed using a bound `CoordinateSystem` that includes a specified choice of origin.

General Classes

TensorQuantityValue

Attributes

mRef : VectorMeasurementReference {redefines mRef}

Attribute mRef is the MeasurementReference for this QuantityValue.

Constraints

[no name]

[no documentation]

order == 1

8.2.3 Units and Scales

8.2.3.1 Units and Scales Overview

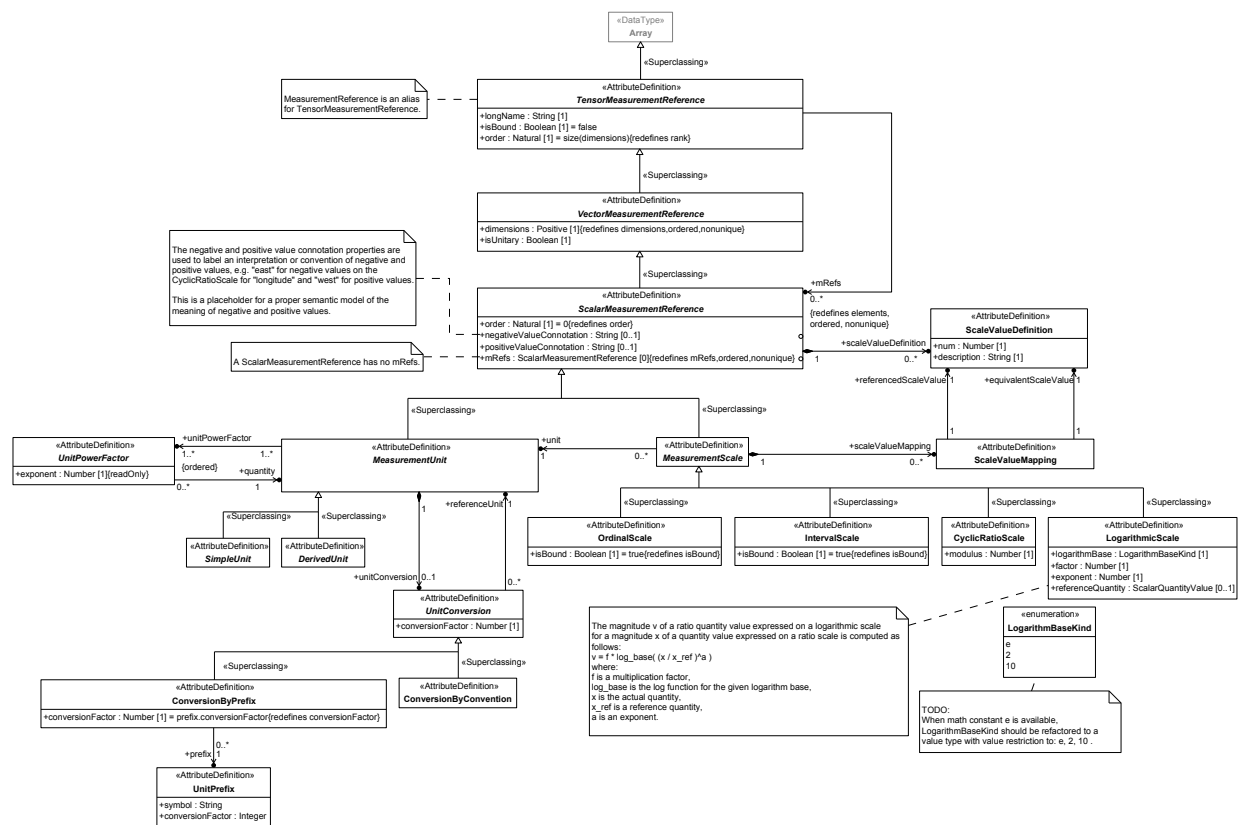


Figure 92. Measurement Units and Scales

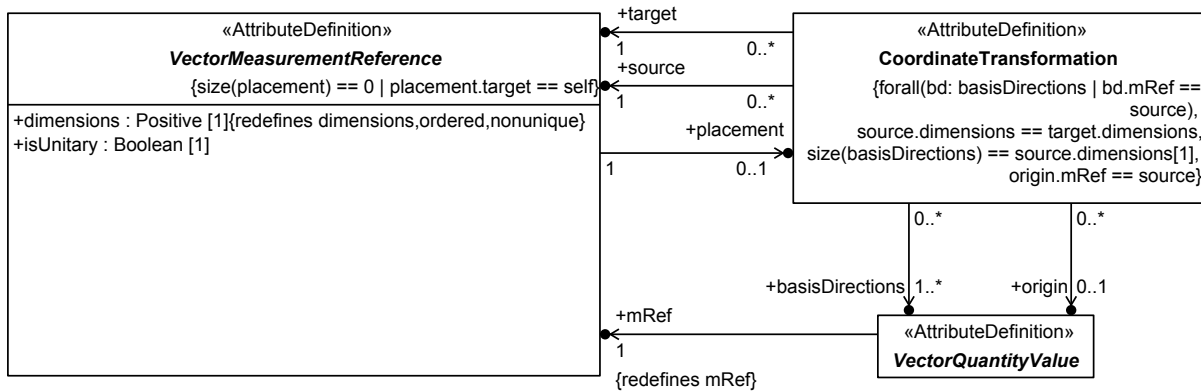


Figure 93. Coordinate Systems

8.2.3.2 Elements

8.2.3.2.1 ConversionByConvention <AttributeDefinition>

Description

ConversionByConvention is a UnitConversion that is defined according to some convention.

An example is the conversion relationship between "foot" (the owning MeasurementUnit) and "metre" (the referenceUnit MeasurementUnit), with conversionFactor 3048/10000, since 1 foot = 0.3048 metre, as defined in [NIST SP-811].

General Classes

UnitConversion

Attributes

No attributes.

Constraints

No constraints.

8.2.3.2.2 ConversionByPrefix <AttributeDefinition>

Description

ConversionByPrefix is a UnitConversion that is defined through reference to a named [ISO/IEC 80000-1] UnitPrefix, which represents a conversion factor that is a decimal or binary multiple or sub-multiple.

Example 1: "kilometre" (symbol "km") with the "kilo" UnitPrefix denoting conversion factor 1000 and referenceUnit "metre".

Example 2: "nanofarad" (symbol "nF") with the "nano" UnitPrefix denoting conversion factor 1E-9 and referenceUnit "farad".

Example 3: "mebibyte" (symbol "MiB" or alias "MiByte") with the "mebi" UnitPrefix denoting conversion factor 1024^2 (a binary multiple) and referenceUnit "byte".

General Classes

UnitConversion

Attributes

conversionFactor : Number {redefines conversionFactor}

Attribute `conversionFactor` is the Number value of the ratio between the quantity expressed in the owning `MeasurementUnit` over the quantity expressed in the `referenceUnit`.

prefix : UnitPrefix

Attribute `prefix` is a `UnitPrefix` that represents one of the named unit prefixes defined in [ISO/ICE-80000-1] as a decimal or binary multiple or sub-multiple.

Constraints

No constraints.

8.2.3.2.3 CoordinateTransformation <AttributeDefinition>

Description

A `CoordinateTransformation` is an `AttributeDefinition` that defines the transformation relationship between two coordinate systems, that are both represented by a `VectorMeasurementReference`.

The `basisDirections` specify the directions for each of the basis vector of the `target` coordinate system (`VectorMeasurementReference`), expressed in the coordinate system specified by the `source`.

If the `source` and the `target` `VectorMeasurementReferences` have `isBound` `false` then they span free vector spaces, and no translation of the origin is given. Otherwise, if both have `isBound` `true`, they span bound vector spaces and the `origin` defines the translation of the origin of the `target` w.r.t the `source` coordinate system. The `origin` may be the zero vector, establishing no origin translation.

General Classes

No general classes.

Attributes

basisDirections : VectorQuantityValue [1..*]

origin : VectorQuantityValue [0..1]

source : VectorMeasurementReference

target : VectorMeasurementReference

Constraints

basisDirectionsMRef

[no documentation]

```
forall(bd: basisDirections | bd.mRef == source)
```

```
matchingSourceAndTarget
```

```
[no documentation]
```

```
source.dimensions == target.dimensions
```

```
originMRef
```

```
[no documentation]
```

```
origin.mRef == source
```

```
numberOfBasisDirections
```

```
[no documentation]
```

```
size(basisDirections) == source.dimensions[1]
```

8.2.3.2.4 CyclicRatioScale <AttributeDefinition>

Description

CyclicRatioScale is a MeasurementScale that represents a ratio scale with a periodic cycle.

Example 1: "cyclic degree" (to express planar angular measures) with modulus 360 and unit "degree".

Example 2: "hour of day" with modulus 24 and unit "hour".

General Classes

MeasurementScale

Attributes

modulus : Number

Attribute modulus is a Number that defines the modulus, i.e. periodic cycle, of this CyclicRatioScale.

Constraints

No constraints.

8.2.3.2.5 DerivedUnit <AttributeDefinition>

Description

DerivedUnit is a MeasurementUnit that represents a measurement unit that depends on one or more powers of other measurement units.

General Classes

MeasurementUnit

Attributes

No attributes.

Constraints

No constraints.

8.2.3.2.6 IntervalScale <AttributeDefinition>

Description

IntervalScale is a MeasurementScale that represents a linear interval measurement scale, i.e. a scale on which only intervals between two values are meaningful and not their ratios.

Note: In order to enable quantity value conversion between an interval scale and another measurement scale, the offset (sometimes also called zero shift) between the source and target scale must be known. This offset is indirectly defined through a ScaleValueMapping, see `scaleValueMapping` of MeasurementScale.

General Classes

MeasurementScale

Attributes

isBound : Boolean {redefines isBound}

Constraints

No constraints.

8.2.3.2.7 LogarithmBaseKind

Description

General Classes

No general classes.

Literal Values

10

2

e

8.2.3.2.8 LogarithmScale <AttributeDefinition>

Description

LogarithmicScale is a MeasurementScale that represents a logarithmic measurement scale that is defined as follows. The numeric value v of a ratio quantity expressed on a logarithmic scale equivalent with a value x of the same quantity expressed on a ratio scale (i.e. only using a MeasurementUnit) is computed as follows:

$$v = f * \log_base((x / x_ref)^a)$$

where: f is a multiplication factor, \log_base is the log function for the given logarithm base, x is the actual quantity, x_ref is a reference quantity, a is an exponent.

General Classes

MeasurementScale

Attributes

exponent : Number

Attribute `exponent` is the exponent a in the logarithmic value expression.

factor : Number

Attribute `factor` is the multiplication factor f in the logarithmic value expression.

logarithmBase : LogarithmBaseKind

Attribute `logarithmicBase` is a Number that specifies the logarithmic base.

The `logarithmicBase` is typically 10, 2 or e (for the natural logarithm).

referenceQuantity : ScalarQuantityValue [0..1]

Attribute `referenceQuantity` is the reference quantity value (denominator) x_ref in the logarithmic value expression.

Constraints

No constraints.

8.2.3.2.9 MeasurementScale <AttributeDefinition>

Description

MeasurementScale is a MeasurementReference that represents a measurement scale.

Note: the majority of scalar quantities can be expressed by just using a MeasurementUnit directly as its MeasurementReference. This implies expression of a ScalarQuantityValue on a ratio scale. However, for full coverage of all quantity value expressions, additional explicit measurement scales with additional semantics are needed, such as ordinal scale, interval scale, ratio scale with additional limit values, cyclic ratio scale and logarithmic scale.

General Classes

ScalarMeasurementReference

Attributes

scaleValueMapping : ScaleValueMapping [0..*]

Attribute `scaleValueMapping` represents an optional ScaleValueMapping that specifies the relationship between this MeasurementScale and another MeasurementReference in terms of equivalent QuantityValues.

unit : MeasurementUnit

Attribute `unit` specifies the MeasurementUnit that defines an interval of one on this MeasurementScale.

Constraints

No constraints.

8.2.3.2.10 MeasurementUnit <AttributeDefinition>

Description

A MeasurementUnit is a ScalarMeasurementReference that represents a measurement unit. As defined in [VIM] a measurement unit is a "real scalar quantity, defined and adopted by convention, with which any other quantity of the same kind can be compared to express the ratio of the two quantities as a number".

Direct use of a MeasurementUnit as the `mRef` attribute of a ScalarQuantityValue, establishes expressing the ScalarQuantityValue on a ratio scale. Similarly, use of a MeasurementUnit as the `mRef` of a component of a tensor

General Classes

ScalarMeasurementReference

Attributes

unitConversion : UnitConversion [0..1]

Attribute `unitConversion` optionally specifies a UnitConversion that is a linear conversion factor relationship with another MeasurementUnit. This can be used to support automated unit conversion.

unitPowerFactor : UnitPowerFactor [1..*] {ordered}

Constraints

No constraints.

8.2.3.2.11 OrdinalScale <AttributeDefinition>

Description

An OrdinalScale is a MeasurementScale that represents an ordinal measurement scale, i.e. a scale on which only quantities value ordering is meaningful, not intervals between two values and neither their ratio.

General Classes

MeasurementScale

Attributes

isBound : Boolean {redefines isBound}

Constraints

No constraints.

8.2.3.2.12 ScalarMeasurementReference <AttributeDefinition>

Description

A ScalarMeasurementReference is a specialization of VectorMeasurementReference for scalar quantities that are typed by a ScalarQuantityValue and for components of tensor or vector quantities. Its order is zero. A ScalarMeasurementReference is also a generalization of MeasurementUnit and MeasurementScale. It establishes how to interpret the `num` numerical value of a ScalarQuantityValue or a component of a tensor or vector quantity value, and establishes its actual quantity dimension.

General Classes

VectorMeasurementReference

Attributes

`mRefs` : ScalarMeasurementReference {redefines mRefs, ordered, nonunique}

`negativeValueConnotation` : String [0..1]

Attribute `negativeValueConnotation` optionally specifies the connotation of negative quantity values for this MeasurementReference.

An example is "east" for positive values on the MeasurementReference (CyclicRatioScale) for "longitude" and "west" for negative values.

`order` : Natural {redefines order}

`positiveValueConnotation` : String [0..1]

Attribute `positiveValueConnotation` optionally specifies the connotation of positive quantity values for this MeasurementReference.

An example is "east" for positive values on the MeasurementReference (CyclicRatioScale) for "longitude" and "west" for negative values.

`scaleValueDefinition` : ScaleValueDefinition [0..*]

Attribute `scaleValueDefinition` specifies zero or more ScaleValueDefinition that represent particular essential values on a measurement unit or scale.

Constraints

No constraints.

8.2.3.2.13 ScaleValueDefinition <AttributeDefinition>

Description

ScaleValueDefinition is an AttributeDefinition that specifies a particular essential value of a MeasurementReference. Typically such a particular value is defined by convention.

Example: For the "kelvin" MeasurementUnit / ratio scale for thermodynamic temperature a ScaleValueDefinition with `num` is 273.16 and `description` is "absolute temperature of the triple point of pure water") can be specified.

General Classes

No general classes.

Attributes

`description` : String

`num` : Number

Constraints

No constraints.

8.2.3.2.14 ScaleValueMapping <AttributeDefinition>

Description

ScaleValueDefinition is an AttributeDefinition that represents the mapping of equivalent quantity values expressed on two different measurement scales.

Example: The mapping between the equivalent thermodynamic temperature quantity values of 273.16 K on the "kelvin" MeasurementUnit ratio scale and 0.01 degree Celsius on the "degree Celsius" IntervalScale would specify a `referenceScaleValue` being the ScaleValueDefinition with `num` is 273.16 and `description` is "absolute thermodynamic temperature of the triple point of water") of the "kelvin" ratio scale, as well as a `mappedScaleValue` being the ScaleValueDefinition with `num` is 0.01 and `description` is "absolute thermodynamic temperature of the triple point of water" of the "degree Celsius" IntervalScale. From this ScaleValueMapping the offset (or zero shift) of 271.15 K between the two scales can be derived.

General Classes

No general classes.

Attributes

`equivalentScaleValue` : ScaleValueDefinition

Attribute `mappedScaleValue` is a ScaleValueDefinition defined on the owning MeasurementScale that is equivalent to the `referenceScaleValue`.

`referencedScaleValue` : ScaleValueDefinition

Attribute `referenceScaleValue` is a ScaleValueDefinition defined on a reference MeasurementReference.

Constraints

No constraints.

8.2.3.2.15 SimpleUnit <AttributeDefinition>

Description

SimpleUnit is a MeasurementUnit that does not depend on any other measurement unit.

Note: As a consequence the `unitPowerFactor` of a SimpleUnit references itself with an exponent of one.

General Classes

MeasurementUnit

Attributes

No attributes.

Constraints

exponentIsOne

[no documentation]

```
this.unitPowerFactor1.exponent == 1
```

ExponentIsOne

[no documentation]

```
self.unitPowerFactor.exponent = 1
```

8.2.3.2.16 TensorMeasurementReference <AttributeDefinition>

Description

A MeasurementReference is an AttributeDefinition that represents the [VIM] concept *measurement reference*, but generalized for tensor, vector and scalar quantities.

[VIM] defines measurement reference as a measurement unit, a measurement procedure, a reference material, or a combination of such. In this generalized definition, the measurement references for all components in all dimensions of the QuantityValue are specified through the `mRefs` attribute, which are all ScalarMeasurementReferences.

The `longName` of a MeasurementReference is the spelled out human readable name of the measurement reference. For example for typical measurement units for the speed quantity the `longName` would be "metre per second", "kilometre per hour" and "mile per hour".

General Classes

Array

Attributes

isBound : Boolean

longName : String

mRefs : ScalarMeasurementReference [0..*] {redefines elements, ordered, nonunique}

order : Natural {redefines rank}

Constraints

No constraints.

8.2.3.2.17 UnitConversion <AttributeDefinition>

Description

A UnitConversion is an AttributeDefinition that represents a linear conversion relationship between one measurement unit and another measurement unit, that acts as a reference.

General Classes

No general classes.

Attributes

conversionFactor : Number

Attribute `conversionFactor` is the Number value of the ratio between the quantity expressed in the owning MeasurementUnit over the quantity expressed in the `referenceUnit`.

referenceUnit : MeasurementUnit

Attribute `referenceUnit` establishes the reference MeasurementUnit with respect to which this UnitConversion is defined.

Constraints

No constraints.

8.2.3.2.18 UnitPowerFactor <AttributeDefinition>

Description

A UnitPowerFactor is an AttributeDefinition that represents a power factor of a MeasurementUnit and an exponent.

Note: A collection of UnitPowerFactors defines a unit power product.

General Classes

No general classes.

Attributes

exponent : Number

Attribute `exponent` is a Number that specifies the exponent of this UnitPowerFactor.

quantity : MeasurementUnit

Attribute `unit` is the MeasurementUnit of this UnitPowerFactor.

Constraints

No constraints.

8.2.3.2.19 UnitPrefix <AttributeDefinition>

Description

UnitPrefix is an AttributeDefinition that represents a named multiple or sub-multiple measurement unit prefix as defined in ISO/IEC 80000-1.

General Classes

No general classes.

Attributes

conversionFactor : Integer

Attribute `conversionFactor` is an Integer that specifies the value of multiple or sub-multiple of this UnitPrefix.

symbol : String

Attribute `symbol` represents the short symbolic name of this UnitPrefix.

Examples are: "k" for "kilo", "m" for "milli", "MeBi" for "mega binary".

Constraints

No constraints.

8.2.3.2.20 VectorMeasurementReference <AttributeDefinition>

Description

A VectorMeasurementReference is a specialization of MeasurementReference for vector quantities that are typed by a VectorQuantityValue. Its order is one. Implicitly it defines a vector space of dimension $N = \text{dimensions}[1]$. The N basis unit vectors that span the vector space are defined by the `mRefs` which each are a ScalarMeasurementReference, typically a MeasurementUnit or an IntervalScale.

It is possible to specify purely symbolic vector spaces, without committing to particular measurement units or scales by setting the measurement references for all dimensions to unit one and quantity of dimension one, thereby basically reverting to the representation of a purely mathematical vector space.

A VectorMeasurementReference can be used to represent a coordinate system for a vector space. The directions of its basis vectors can be defined for with respect to another coordinate system (represented by a different VectorMeasurementReference) via a CoordinateTransformation.

It is also possible to define nested chains of coordinate transformations, from a top level coordinate system that is posited per some convention (described in text). The subsequent coordinate systems are then placed (translated and oriented) via a chain of VectorMeasurementReferences with `placement` attribute specifications for each level of decomposition. The `source` of each `placement` is the reference coordinate system for each transformation. A top level VectorMeasurementReference that represents a coordinate system will not have a `placement`.

The attribute `longName` may include the conventional, textual definition of the datum (origin and orientation) of a top level coordinate system.

The attribute `isUnitary` indicates whether the inner products of the basis vectors of the vector space specified by `VectorMeasurementReference` are unitary or not. This can be regarded as a generalization of being orthogonal for a real vector space. Unitarity extends the concept to complex number and quaternion vector spaces.

General Classes

`TensorMeasurementReference`

Attributes

`dimensions` : Positive {redefines dimensions, ordered, nonunique}

`isUnitary` : Boolean

`placement` : `CoordinateTransformation` [0..1]

Constraints

`placementCheck`

[no documentation]

```
size(placement) == 0 | placement.target == self
```

8.2.4 ISQ

8.2.4.1 ISQ Overview

8.2.4.2 Elements

8.2.5 SI Prefixes

8.2.5.1 SI Prefixes Overview

8.2.5.2 Elements

8.2.6 SI

8.2.6.1 SI Overview

8.2.6.2 Elements

8.2.7 US Customary Units

8.2.7.1 US Customary Units Overview

8.2.7.2 Elements

8.2.8 Date and Time

8.2.8.1 Date and Time Overview

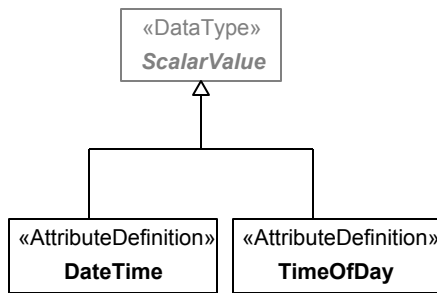


Figure 94. Calendar Dates and Times

8.2.8.2 Elements

8.2.8.2.1 DateTime

Description

Special type to represent ISO 8601 date-time as a convenience because it is so pervasive. Conversion functions can be created that convert ISO 8601 date-time to and from time instants defined using quantities, units and scales, i.e. time instants expressed on an IntervalScale that defines the (proleptic) Gregorian Calendar.

Attributes

No attributes.

8.2.8.2.2 TimeOfDay

Description

ISO 8601 time instant within an ISO 8601 24 hour day.

Attributes

No attributes.

8.3 Analysis Domain Library

8.3.1 Overview

The Analysis Domain Library provides library models supporting the modeling of analysis cases (see [7.20](#)).

8.3.2 Trade Studies

The Trade Studies model library provides a simple framework for defining trade-off study analysis cases.

8.3.2.1 Trade Studies Overview

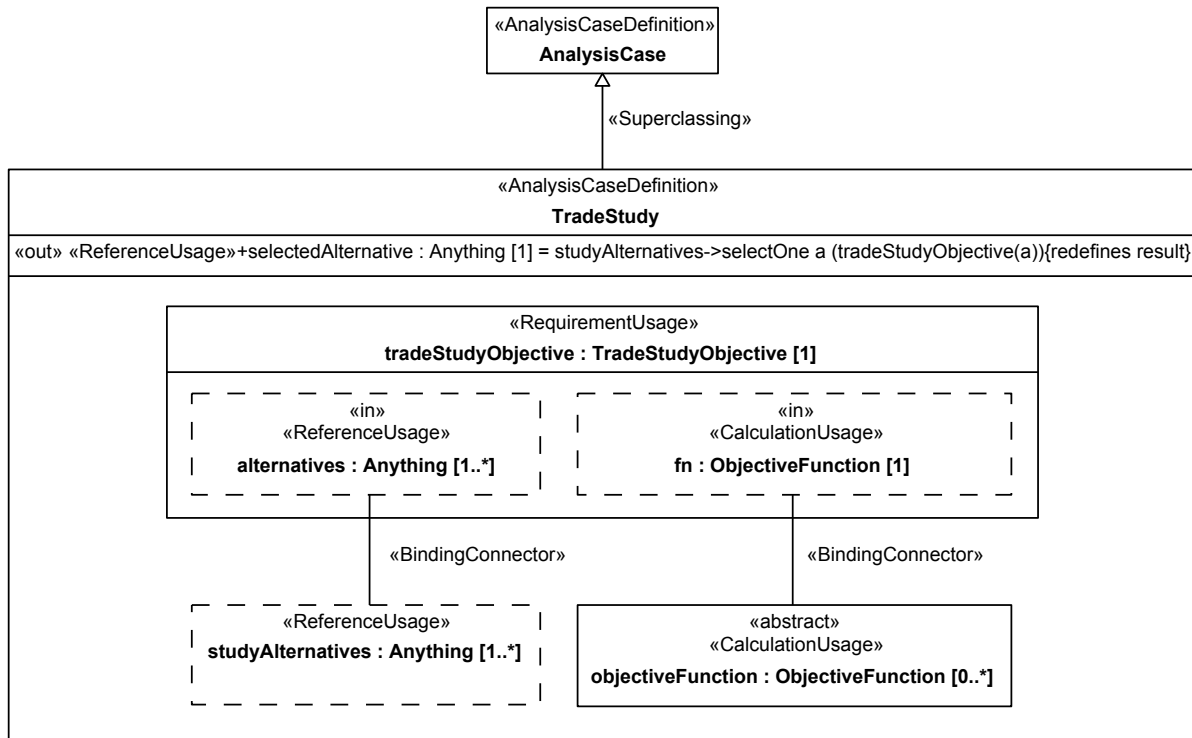


Figure 95. Trade Studies Domain Model

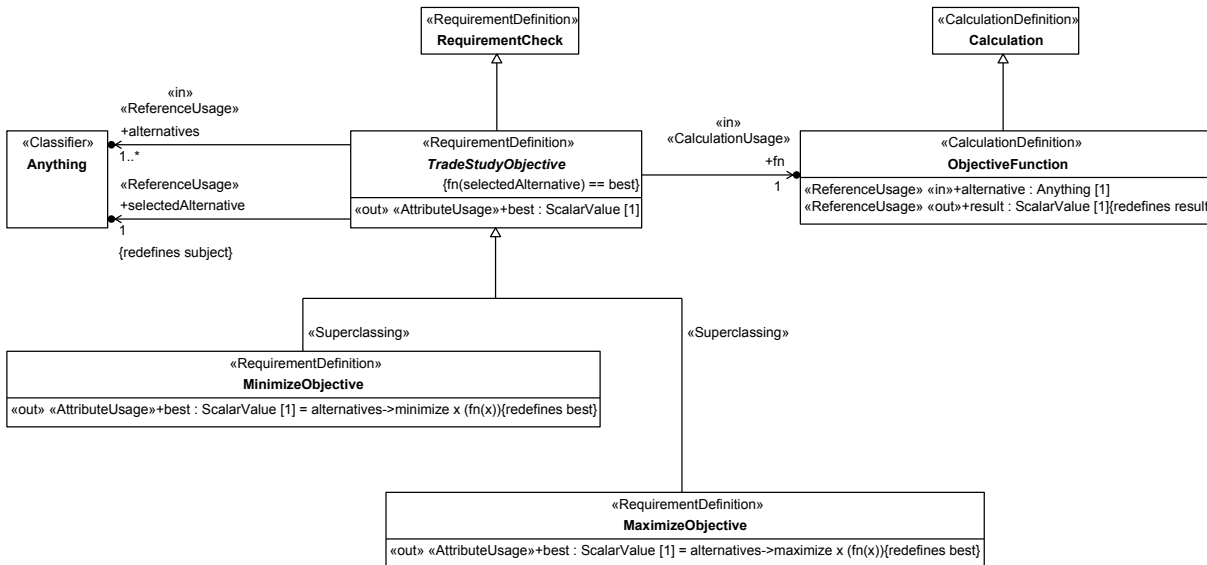


Figure 96. Trade Study Objectives

8.3.2.2 Elements

8.3.2.2.1 MaximizeObjective

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

8.3.2.2 MinimizeObjective

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

8.3.2.3 ObjectiveFunction

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

8.3.2.4 TradeStudy

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

8.3.2.2.5 TradeStudyObjective

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

A Annex: Conformance Test Suite

Submission Note. A conformance test suite will be provided in the revised submission.

B Annex: Example Model

Release Note. The following example is provided as a sample of the SysML textual notation as it has been defined so far. A graphical representation will be provided in a future release.

```
package VehicleModel {
  import Definitions::*;
  package Definitions{
    // These imports eliminate the need to import into each nested package
    import PartDefinitions::*;
    import PortDefinitions::*;
    import ItemDefinitions::*;
    import SignalDefinitions::*;
    import InterfaceDefinitions::*;
    import ActionDefinitions::*;
    import StateDefinitions::*;
    import RequirementDefinitions::*;
    import AttributeDefinitions::*;
    import IndividualDefinitions::*;
    package PartDefinitions{
      part def Vehicle {
        attribute mass :> ISQ::mass;
        attribute dryMass:>ISQ::mass;
        attribute cargoMass:>ISQ::mass;
        attribute position:>ISQ::length;
        attribute velocity:>ISQ::speed;
        attribute acceleration:>ISQ::acceleration;
        attribute electricalPower:>ISQ::power;
        attribute Tmax:>ISQ::temperature;
        attribute maintenanceTime: Time::DateTime;
        attribute brakePedalDepressed: Boolean;
        port fuelCmdPort:FuelCmdPort;
        port vehicleToRoadPort:VehicleToRoadPort;
        perform action providePower;
        perform action performSelfTest;
        perform action applyParkingBrake;
        perform action senseTemperature;
        exhibit state vehicleStates;
      }
      part def Engine;
      part def Cylinder;
      part def Transmission;
      part def Driveshaft;
      part def AxleAssembly;
      part def Axle{
        attribute mass:>ISQ::mass;
      }
      part def FrontAxle:>Axle{
        attribute steeringAngle:ScalarValues::Real;
      }
      part def HalfAxle;
      part def Differential;
      part def Wheel {
```

```

        attribute diameter:LengthValue;
    }
    abstract part def Software;
    part def VehicleSoftware:>Software;
    part def VehicleController:>Software {
        exhibit state controllerStates: ControllerStates;
    }
    part def FuelTank{
        attribute mass :> ISQ::mass;
        ref item fuel:Fuel{
            attribute redefines fuelMass;
        }
        attribute fuelMassMax:>ISQ::mass;
        assert constraint {fuel::fuelMass<=fuelMassMax}
    }
    part def Road{
        attribute incline:Real;
        attribute friction:Real;
    }
    part def VehicleRoadContext{
        attribute time:TimeValue;
    }

    // Used for Specifying Context for Individuals
    part def SpatialTemporalReference;

    // Used for Defining Variants for Superset Model
    part def Engine4Cyl;
    part def Engine6Cyl;
    part def TransmissionChoices;
    part def TransmissionAutomatic;
    part def TransmissionManual;
    part def Sunroof;

}
package PortDefinitions{
    port def FuelCmdPort{
        in item fuelCmd:FuelCmd;
    }
    port def DrivePwrPort{
        out engineTorque:Torque;
    }
    port def ClutchPort;
    port def ShaftPort_a;
    port def ShaftPort_b;
    port def ShaftPort_c;
    port def ShaftPort_d;
    port def DiffPort;
    port def AxlePort;
    port def AxleToWheelPort;
    port def WheelToAxlePort;
    port def WheelToRoadPort;
    port def VehicleToRoadPort;
}

```

```

package ItemDefinitions{
    item def Fuel{
        attribute fuelMass:>ISQ::mass;
    }
    item def FuelCmd;
}
package SignalDefinitions{
    attribute def VehicleStartSignal;
    attribute def VehicleOnSignal;
    attribute def VehicleOffSignal;
    attribute def StartSignal;
    attribute def OffSignal;
    attribute def OverTemp;
    attribute def ReturnToNormal;

    // The following are work arounds until time events and
    // change events are available
    attribute def 'at(vehicle::maintenanceTime)';
    attribute def 'when(temp>vehicle::Tmax)';
}
package InterfaceDefinitions{
    interface def EngineToTransmissionInterface{
        end p1:DrivePwrPort;
        end p2:ClutchPort;
    }
}
package ActionDefinitions{
    action def ProvidePower {
        in item fuelCmd:FuelCmd;
        out wheelToRoadTorque:Torque[2];
    }
    action def GenerateTorque {
        in item fuelCmd:FuelCmd;
        out engineTorque:Torque;
    }
    action def AmplifyTorque {
        in engineTorque:Torque;
        out transmissionTorque:Torque;
    }
    action def TransferTorque {
        in transmissionTorque:Torque;
        out driveshaftTorque:Torque;
    }
    action def DistributeTorque {
        in driveshaftTorque:Torque;
        out wheelToRoadTorque:Torque[2];
    }
    action def PerformSelfTest;
    action def ApplyParkingBrake;
    action def SenseTemperature{
        out temp: ISQ::TemperatureValue;
    }
}
package StateDefinitions {

```

```

state def VehicleStates;
state def ControllerStates;
}
package RequirementDefinitions{
  requirement def MassRequirement{
    doc /*The actual mass shall be less than the required mass*/
    attribute massRequired:>ISQ::mass;
    attribute massActual:>ISQ::mass;
    require constraint {massActual<=massRequired}
  }
  requirement def ReliabilityRequirement{
    doc /*The actual reliability shall be greater than the required reliabilit
    attribute reliabilityRequired:Real;
    attribute reliabilityActual:Real;
    require constraint {reliabilityActual>=reliabilityRequired}
  }
  requirement def TorqueGenerationRequirement {
    subject engine:Engine;
    doc /* The engine shall generate torque as a function of RPM as shown in
       * Table 1.*/
  }
  requirement def DrivePowerInterfaceRequirement {
    subject engine:Engine;
    doc /* The engine shall transfer its generated torque to the transmission
       * the clutch interface.*/
  }
  requirement def FuelEconomyRequirement {
    doc /* The vehicle shall maintain an average fuel economy of at least
       * x miles per gallon for the nominal driving scenario */
    attribute actualFuelEconomy : DistancePerVolumeValue;
    attribute requiredFuelEconomy : DistancePerVolumeValue;
    require constraint {actualFuelEconomy >= requiredFuelEconomy}
  }
}
package AttributeDefinitions{
  import ScalarValues::*;
  // Scalar Functions provides Sum expression
  import ScalarFunctions::*;
  import ISQ::*;
  import SI::*;
  alias ISQ::TorqueValue as Torque;

  //quantity used in analysis
  attribute def DistancePerVolumeValue:>Quantities::QuantityValue;
}
package IndividualDefinitions{
  individual def SpatialTemporalReference_1:>SpatialTemporalReference;
  individual def VehicleRoadContext_1:>VehicleRoadContext;
  individual def Vehicle_1:>Vehicle;
  individual def FrontAxleAssembly_1:>AxleAssembly;
  individual def FrontAxle_1:>FrontAxle;
  individual def Wheel_1:>Wheel;
  individual def Wheel_2:>Wheel;
}

```

```

        individual def RearAxleAssembly_1:>AxleAssembly;
        individual def Road_1:>Road;
    }
}
package VehicleConfigurations{
    package VehicleConfiguration_a{
        package PartsTree{
            part vehicle_a:Vehicle{
                attribute mass redefines Vehicle::mass=
                    dryMass+cargoMass+fuelTank::fuel::fuelMass;
                attribute dryMass redefines Vehicle::dryMass=sum(partMasses);
                attribute redefines Vehicle::cargoMass=0;
                attribute partMasses=
                    (fuelTank::mass,frontAxleAssembly::mass,rearAxleAssembly::mass);
            part fuelTank:FuelTank{
                attribute redefines mass=75@[kg];
                ref item redefines fuel{
                    attribute redefines fuelMass=50@[kg];
                }
            }
            part frontAxleAssembly:AxleAssembly{
                attribute mass :> ISQ::mass=800@[kg];
                part frontAxle:Axle;
                part frontWheels:Wheel[2];
            }
            part rearAxleAssembly:AxleAssembly{
                attribute mass :> ISQ::mass=875@[kg];
                attribute driveTrainEfficiency:Real = 0.6;
                part rearAxle:Axle;
                part rearWheels:Wheel[2]{
                    attribute redefines diameter;
                }
            }
        }
    }
}
package ActionTree{
}
}
package VehicleConfiguration_b{
    package PartsTree{
        part vehicle_b : Vehicle{
            attribute mass redefines Vehicle::mass=
                dryMass+cargoMass+fuelTank::fuel::fuelMass;
            attribute dryMass redefines Vehicle::dryMass=sum(partMasses);
            attribute redefines Vehicle::cargoMass=0;
            attribute partMasses=
                (fuelTank::mass,frontAxleAssembly::mass,rearAxleAssembly::mass,
                    engine::mass,transmission::mass,driveshaft::mass);
            port redefines fuelCmdPort {
                in item redefines fuelCmd;
            }
            port vehicleToRoadPort redefines vehicleToRoadPort{
                port wheelToRoadPort1:WheelToRoadPort;
            }
        }
    }
}

```

```

    port wheelToRoadPort2:WheelToRoadPort;
}
perform ActionTree::providePower redefines providePower;
perform ActionTree::performSelfTest redefines performSelfTest;
perform ActionTree::applyParkingBrake redefines applyParkingBrake;
perform ActionTree::senseTemperature redefines senseTemperature;
exhibit States::vehicleStates redefines vehicleStates {
    ref vehicle redefines vehicle = vehicle_b;
}
}
part fuelTank:FuelTank{
    attribute redefines mass=75@[kg];
    ref item redefines fuel{
        attribute redefines fuelMass=60@[kg];
    }
    attribute redefines fuelMassMax=60;
}
part frontAxleAssembly:AxleAssembly{
    attribute mass :> ISQ::mass=800@[kg];
    port shaftPort_d:ShaftPort_d;
    part frontAxle:FrontAxle;
    part frontWheels:Wheel[2];
}
part rearAxleAssembly:AxleAssembly{
    attribute mass :> ISQ::mass=875@[kg];
    attribute driveTrainEfficiency:Real = 0.6;
    port shaftPort_d:ShaftPort_d;
    perform ActionTree::providePower::distributeTorque;
    part rearWheel1:Wheel{
        attribute redefines diameter;
        port wheelToAxlePort:WheelToAxlePort;
        port wheelToRoadPort:WheelToRoadPort;
    }
    part rearWheel2:Wheel{
        attribute redefines diameter;
        port wheelToRoadPort:WheelToRoadPort;
        port wheelToAxlePort:WheelToAxlePort;
    }
    part differential:Differential{
        port shaftPort_d:ShaftPort_d;
        port leftDiffPort:DiffPort;
        port rightDiffPort:DiffPort;
    }
    part rearAxle{
        part leftHalfAxle:HalfAxle{
            port leftAxleToDiffPort:AxlePort;
            port leftAxleToWheelPort:AxlePort;
        }
        part rightHalfAxle:HalfAxle{
            port rightAxleToDiffPort:AxlePort;
            port rightAxleToWheelPort:AxlePort;
        }
    }
}

bind shaftPort_d=differential::shaftPort_d;

```

```

        connect differential::leftDiffPort
            to rearAxle::leftHalfAxle::leftAxleToDiffPort;
        connect differential::rightDiffPort
            to rearAxle::rightHalfAxle::rightAxleToDiffPort;
    }
    part engine:Engine{
        attribute mass :> ISQ::mass=200@[kg];
        attribute peakHorsePower:PowerValue = 200;
        port fuelCmdPort:FuelCmdPort{
            in item redefines fuelCmd;
        }
        port drivePwrPort:DrivePwrPort{
            out redefines engineTorque;
        }
        perform ActionTree::providePower::generateTorque;
        part cylinders:Cylinder[4..6];
    }
    part transmission:Transmission{
        attribute mass :> ISQ::mass=100@[kg];
        //conjugate notation ~
        port clutchPort:~DrivePwrPort;
        port shaftPort_a:ShaftPort_a;
        perform ActionTree::providePower::amplifyTorque;
    }
    part driveshaft:Driveshaft{
        attribute mass :> ISQ::mass=100@[kg];
        port shaftPort_b:ShaftPort_b;
        port shaftPort_c:ShaftPort_c;
        perform ActionTree::providePower::transferTorque;
    }
    part vehicleSoftware:VehicleSoftware{
        part vehicleController: VehicleController {
            exhibit States::controllerStates redefines controllerStates{
                ref controller = vehicleController;
            }
        }
    }

    bind engine::fuelCmdPort=fuelCmdPort;
    interface engineToTransmissionInterface:EngineToTransmissionInterface
        connect engine::drivePwrPort to transmission::clutchPort{
            ref action generateToAmplify :> ActionTree::providePower::gene
        }
    connect transmission::shaftPort_a to driveshaft::shaftPort_b;
    connect driveshaft::shaftPort_c to rearAxleAssembly::shaftPort_d;
    bind rearAxleAssembly::rearWheel1::wheelToRoadPort=vehicleToRoadPort::
    bind rearAxleAssembly::rearWheel2::wheelToRoadPort=vehicleToRoadPort::

    }
}
package ActionTree{
    action providePower:ProvidePower{
        in item redefines fuelCmd;
        out wheelToRoadTorque redefines wheelToRoadTorque [2] =
            distributeTorque::wheelToRoadTorque;
    }
}

```

```

// No successions (control flows) between these actions, because the
// flows between them are continuous streams.
action generateTorque:GenerateTorque {
    in item = providePower::fuelCmd;
    out engineTorque redefines engineTorque;
}
action amplifyTorque:AmplifyTorque {
    in engineTorque redefines engineTorque;
    out transmissionTorque redefines transmissionTorque;
}
action transferTorque:TransferTorque {
    // This is a shorthand for the stream commented out below.
    in transmissionTorque redefines transmissionTorque;
    out driveshaftTorque redefines driveshaftTorque;
}
action distributeTorque:DistributeTorque{
    in driveshaftTorque redefines driveshaftTorque;
    out wheelToRoadTorque redefines wheelToRoadTorque [2];
}

stream generateToAmplify from generateTorque::engineTorque
    to amplifyTorque::engineTorque;
stream amplifyTorque::transmissionTorque
    to transferTorque::transmissionTorque;
stream transferTorque::driveshaftTorque
    to distributeTorque::driveshaftTorque;
}
action performSelfTest: PerformSelfTest;
action applyParkingBrake: ApplyParkingBrake;
action senseTemperature: SenseTemperature;
}
package States{
    state vehicleStates: VehicleStates {
        ref vehicle : Vehicle;
        ref controller : VehicleController;

        state operatingStates {
            entry action initial;

            state off;
            state starting;
            state on {
                entry vehicle::performSelfTest;
                // was vehicle::providePower;
                do ActionTree::providePower;
                exit vehicle::applyParkingBrake;
                constraint {vehicle::electricalPower<=500}
            }

            transition initial then off;

            transition 'off-starting'
                first off

```



```

        accept VehicleStartSignal
        if vehicle::brakePedalDepressed
        do send StartSignal() to controller
        then starting;

    transition 'starting-on'
        first starting
        accept VehicleOnSignal
        then on;

    transition 'on-off'
        first on
        accept VehicleOffSignal
        then off;
}

state healthStates {
    entry action initial;
    do vehicle::senseTemperature (out temp);

    state normal;
    state maintenance;
    state degraded;

    transition initial then normal;

    transition 'normal-maintenance'
        first normal
        accept 'at(vehicle::maintenanceTime)'
        then maintenance;

    transition 'normal-degraded'
        first normal
        accept 'when(temp>vehicle::Tmax)'
        do send OverTemp() to controller
        then degraded;

    transition 'maintenance-normal'
        first maintenance
        accept ReturnToNormal
        then normal;

    transition 'degraded-normal'
        first degraded
        accept ReturnToNormal
        then normal;
}

state controllerStates: ControllerStates {
    state operatingStates {
        entry action initial;
        state off;
        state on;

```

```

        transition initial then off;
        transition 'off-on'
            first off
            //why not a usage of StartSignal?
            accept StartSignal
            then on;
        transition 'on-off'
            first on
            accept OffSignal
            then off;
    }
}

package Requirements{
    import Definitions::PartDefinitions::*;
    import VehicleConfiguration_b::*;
    import VehicleConfiguration_b::PartsTree::*;
    import vehicle_b_SpecificationContext::*;
    item marketSurvey;
    dependency from vehicleSpecification to marketSurvey;
    part vehicle_b_SpecificationContext{
        // The subject of the specification is vehicle:Vehicle, which is a
        // usage of the black box specification. Other vehicles are further
        // specializations of this black box specification.
        // Redefine vehicle_b within the vehicle_b_Specification Context so
        // satisfies can be added.
        part redefines vehicle_b;
        requirement vehicleSpecification{
            subject vehicle:Vehicle;
            requirement id '1' vehicleMassRequirement: MassRequirement {
                doc /* The total mass of a vehicle shall be less than or equal
                    * to the required mass.*/
                /* Assume total mass includes a full tank of gas*/
                attribute redefines massRequired=2000;
                attribute redefines massActual = vehicle::mass;
            }
            requirement id '2' vehicleFuelEconomyRequirements{
                doc /* fuel economy requirements group */
                attribute assumedCargoMass:>ISQ::mass;
                requirement id '2_1' cityFuelEconomyRequirement:
                    FuelEconomyRequirement{
                        redefines requiredFuelEconomy=25; //@[mpg];
                        assume constraint {assumedCargoMass>=500 @[kg]}
                    }
                requirement id '2_2' highwayFuelEconomyRequirement:
                    FuelEconomyRequirement{
                        redefines requiredFuelEconomy=30; //@[mpg]
                        attribute assumedCargoMass:>ISQ::mass;
                        assume constraint {assumedCargoMass>=500 @[kg]}
                    }
            }
        }
    }
    satisfy vehicleSpecification::vehicleMassRequirement
        by vehicle_b;
}

```

```

        satisfy vehicleSpecification::vehicleFuelEconomyRequirements
            by vehicle_b;
        //this should not pass the requirement since vehicle_b::mass=2010
        //and the massRequired=2000

        requirement engineSpecification {
            doc /* Engine power requirements group */
            subject engine: Engine;

            requirement torqueGeneration : TorqueGenerationRequirement;

            requirement drivePowerInterface : DrivePowerInterfaceRequirement;
        }
        satisfy engineSpecification::torqueGeneration by vehicle_b::engine;
    }
}

package VehicleAnalysis{
    import Definitions::AttributeDefinitions::*;
    // the following is a general vehicle dynamics analysis model that is not bound to
    package DynamicsEquations{
        attribute p:PowerValue; // engine power
        attribute m:MassValue;
        attribute v:SpeedValue;
        attribute a:AccelerationValue;

        attribute dt:TimeValue;
        attribute x0:LengthValue;
        attribute x_f:LengthValue;
        attribute v0:SpeedValue;
        attribute v_f:SpeedValue;

        constraint def StraightLineDynamicsEquations{
            attribute v_avg:SpeedValue = (v0 + v_f)/2;
            a == p/(m*v) &
            v_f == v0 +a*dt &
            x_f == x0+v*dt
        }
    }
}

package FuelEconomyAnalysisModel{
    import VehicleConfigurations::VehicleConfiguration_b::PartsTree::*;
    import VehicleConfigurations::VehicleConfiguration_b::Requirements::*;
    import NonScalarValues::SampledFunctionValue;

    attribute def NominalScenario :> SampledFunctionValue;

    analysis fuelEconomyAnalysis {
        return attribute calculatedFuelEconomy:DistancePerVolumeValue;
        in attribute scenario: NominalScenario;
        subject = vehicle_b;

        objective fuelEconomyAnalysisObjective {
            doc /* the objective of this analysis is to determine whether

```

```

        * the vehicle design configuration can
        * satisfy the fuel economy requirements */
require vehicleSpecification::vehicleFuelEconomyRequirements;
}

action straightLineDynamics {
    in power : PowerValue=vehicle_b::engine::peakHorsePower;
    in mass : MassValue=vehicle_b::mass;
    in delta_t : TimeValue;
    in x_in : LengthValue;
    in v_in : SpeedValue;
    out x_out : LengthValue;
    out v_out : SpeedValue;
    out a_out : AccelerationValue;

    assert constraint dynamics {
        attribute v_avg:SpeedValue = (v_in + v_out)/2;
        a_out == power/(mass*v_avg) &
        v_out == v_in +a_out*delta_t &
        x_out == x_in+v_avg*delta_t
    }
}

// perform the fuel consumption analysis based on the outputs from
// the dynamics analysis (e.g., power vs time)

action fuelConsumptionAnalysis {
}

}

package ElectricalPowerAnalysis{
}

package ReliabilityAnalysis{
}

}

package VehicleVerification{
    import Definitions::AttributeDefinitions::*;
    import VehicleConfigurations::VehicleConfiguration_b::*;
    import VehicleConfigurations::VehicleConfiguration_b::PartsTree::*;
    import VehicleConfigurations::VehicleConfiguration_b::Requirements::vehicle_b_Spec
    import VerificationCaseDefinitions::*;
    import VerificationCases::*;
    package VerificationCaseDefinitions{
        verification def MassTest;
        verification def AccelerationTest;
        verification def ReliabilityTest;
    }
    package VerificationCases{
        verification massTests:MassTest {
            subject = vehicle_b;
            objective {
                verify vehicleSpecification::vehicleMassRequirement{
                    redefines massActual=weighVehicle::massMeasured;
                }
            }
        }
    }
}

```

```

        action weighVehicle (
            out massMeasured:>ISQ::mass);
    }
}
package VerificationSystem{
    part massVerificationSystem{
        perform massTests;
        part scale{
            perform massTests::weighVehicle;
        }
        part operator;
    }
}
package VehicleIndividuals{
    individual a:SpatialTemporalReference_1{
        timeslice t0_t2:VehicleRoadContext_1{
            snapshot start redefines start{:>>time=0;}
            snapshot done redefines done {:>>time=2;}
        }
        snapshot t0:VehicleRoadContext_1{
            attribute t0 redefines time=0;
            snapshot t0_r:Road_1{
                :>>Road::incline =0;
                :>>Road::friction=.1;
            }
            snapshot t0_v:Vehicle_1{
                :>>Vehicle::position=0;
                :>>Vehicle::velocity=0;
                :>>Vehicle::acceleration=1.96;
                // .2 g where 1 g = 9.8 meters/sec^2
                // how do you represent state=on;
                snapshot t0_fa:FrontAxleAssembly_1{
                    snapshot t0_leftFront:Wheel_1;
                    snapshot t0_rightFront:Wheel_2;
                }
            }
        }
        snapshot t1:VehicleRoadContext_1{
            attribute t1 redefines time=1;
            snapshot t1_r:Road_1{
                :>>Road::incline =0;
                :>>Road::friction=.1;
            }
            snapshot t1_v:Vehicle_1{
                :>>Vehicle::position=.98;
                :>>Vehicle::velocity=1.96;
                :>>Vehicle::acceleration=1.96;
                // .2 g where 1 g = 9.8 meters/sec^2
                // how do you represent state=on;
                snapshot t1_fa:FrontAxleAssembly_1{
                    snapshot t1_leftFront:Wheel_1;
                    snapshot t1_rightFront:Wheel_2;
                }
            }
        }
    }
}

```

```

    }
  }
  snapshot t2:VehicleRoadContext_1{
    attribute t2 redefines time=2;
    snapshot t2_r:Road_1{
      :>>Road::incline =0;
      :>>Road::friction=.1;
    }
    snapshot t2_v:Vehicle_1{
      :>>Vehicle::position=3.92;
      :>>Vehicle::velocity=3.92;
      :>>Vehicle::acceleration=1.96;
      // .2 g where 1 g = 9.8 meters/sec^2
      // how do you represent state=on;
      snapshot t2_fa:FrontAxleAssembly_1{
        snapshot t2_leftFront:Wheel_1;
        snapshot t2_rightFront:Wheel_2;
      }
    }
  }
}

package VehicleSuperSetModel{
  import VehicleConfigurations::*;
  import VehicleConfigurations::VehicleConfiguration_a::PartsTree::*;
  import VehicleConfigurations::VehicleConfiguration_a::ActionTree::*;
  // Make vehicle_b a specific vehicle configuration from this product family
  // instead of a subset of vehicle_a. This requires that we add all
  // of vehicle_b into the superset model.
  package VariationPointDefinitions {
    variation part def TransmissionChoices:>Transmission {
      variant part transmissionAutomatic:TransmissionAutomatic;
      variant part transmissionManual:TransmissionManual;
    }
  }
  package VehiclePartsTree{
    import VariationPointDefinitions::*;
    abstract part vehicleFamily:>vehicle_a{
      // variation with nested variation
      variation part engine:Engine{
        variant part engine4Cyl:Engine4Cyl;
        variant part engine6Cyl:Engine6Cyl{
          part cylinder:Cylinder [6]{
            variation attribute diameter:LengthValue{
              variant attribute smallDiameter:LengthValue;
              variant attribute largeDiagmeter:LengthValue;
            }
          }
        }
      }
    }
    // variation point based on variation of part definition
    variation part transmissionChoices:TransmissionChoices;
    // optional variation point
    variation part sunroof:Sunroof;
  }
}

```

```

        // selection constraint
        assert constraint {
            (engine==engine::engine4Cyl &
             transmissionChoices==TransmissionChoices::transmissionManual) ^
            (engine==engine::engine6Cyl &
             transmissionChoices==TransmissionChoices::transmissionAutomatic)
        }
    }
}

package Views_Viewpoints{
    package ViewpointDefinitions{
        viewpoint def StructureViewpoint;
        viewpoint def BehaviorViewpoint;
    }
    package RenderingDefinitions{
        rendering def Table;
    }
    package ViewDefinitions{
        view def PartsTree;
    }
    package Views{
        import ViewpointDefinitions::*;
        import RenderingDefinitions::*;
        import ViewDefinitions::*;
        import VehicleConfigurations::VehicleConfiguration_b::*;
        view vehiclePartsTree:PartsTree{
            viewpoint treeDiagram:StructureViewpoint;
            expose PartsTree::*;
            rendering partsTreeTable:Table;
        }
    }
}

```


C Annex: SysML v1 to SysML v2 Transformation

C.1 General

C.1.1 Overview

This annex describes a transformation that specifies a translation from SysML v1 [SysMLv1] to SysML v2 on a semantic basis in a precise way. (In this annex, "SysML v1" refers to SysML v1.7, the last version of SysML prior to v2.0, and "SysML v2" refers to SysML as defined in this specification.)

The main intent is to provide the rules on which automated conversions of SysML v1 models to the SysML v2 standard can be developed. In addition, this annex can be considered an educational document that provides useful information for people who would like to compare using SysML v2 to using SysML v1.

More sophisticated applications of this transformation can also be envisaged. For instance, a SysML v1 conformant tool could use this transformation to implement a limited subset of the SysML v2 API that will provided "SysMLv2-like" read-only access to its SysMLv1 models for external applications.

Submission Note: For this initial submission the transformation specification will cover a restricted scope only, which will be extended in the revised submission. For the initial submission, we focus on the metaclasses of UML4SYSML that represent structural concepts. As of this submission, the latest completed version of SysML is v1.6, with v1.7 still in preparation, but the subset covered in this initial submission is not expected to change for SysML 1.7.

C.1.2 Mapping Approach

The SysML v1 to v2 transformation is specified by directional mappings between UML metaclasses (i.e., the UML4SYSML subset) and stereotypes that are part of the SysML v1 specification and set of the metaclasses included in KerML and the SysMLv2 libraries.

Each mapping is a directed relationship that reifies a semantic link between a concept belonging to the SysMLv1 scope on the source side and one concept belonging to the SysMLv2 scope on the target side. As a set, all those mappings specify a formal transformation that describes how the information encoded by the SysMLv1 concepts can be reliably represented using constructs of SysMLv2 metaclasses instances.

In this approach, a mapping is represented by a UML class that has a pair of associations. One provides the "from" end that designates the source SysML v1 concept while the other provides the "to" end that designates the target SysML v2 metaclass.

In addition to those associations, a Mapping class provides a set of operations defining how the attribute values of the target metaclass instance have to be computed based on attribute values of the reachable from the source object. The computation algorithm is provided by the body condition of those operations and expressed using OCL code.

Note that the values assigned to attributes of the target object shall be instances of the target (i.e., SysMLv2) metamodel, coming themselves from transformations of SysMLv1 objects to SysMLv2 objects. The `getMapped` and `getMappedUnique` operations are provided for this purpose. The first one returns a (possibly empty) set of values while the second returns a (possibly null) value, based on a mapping type and a target type.

Each mapping specification enables the transformation of any object that has the type specified by the "from" role to an object of the type specified by the "to" role, as long as it is not overloaded by a more specific mapping defined. In other words, assume a mapping is specified as the class "A" (i.e., that has A typing the its "from" property) it will apply to any instance of a class B if B is a subclass of A and if there is no specialization of that mapping class specified for B (i.e., that has B typing the its "from" property).

Some mapping classes have one or more qualifiers for their "from" attribute. In such a case, each of those qualifiers reflect the specific attribute of the source type (i.e. the type of the "from" attribute) that has the same name and the same type. For those specific mappings, it is expected to get one instance of the target class (as specified by the type of the "to" attribute) for each combination of value of those attributes per instance of object of the source type, assuming they pass the applicability filter as described below.

Indeed, it is also possible to restrict the applicability of a mapping specification to a specific subset of objects. This is achieved by the "filter" operation that is evaluated against each candidate object. Only objects for which this "filter" operation returns "true" shall be translated according to that mapping rules. By default, the filter operation always returns "true".

C.2 Mappings

C.2.1 Generic Mappings

C.2.1.1 Overview

Generic mappings are partial definitions of transformation rules that are intended to factorize reusable algorithms for making the global specification more compact and easier to read and maintain. Basically, they provide a default value for all the non-derived attributes of their target metaclass wherever possible, or declare an abstract operation for them otherwise. All of them have "UML::Element" defined as their source type. The operations provided by the generic mappings are redefined by their specialization, as appropriate according to the source type specified by the redefinition of their "from" attribute.

All of those generic mappings are abstract, except one (the "GenericToElement_Mapping") that is the root of all other mappings class.

C.2.1.2 Generic Mappings to KerML

Table 6. List of all Generic Mappings to KerML Mapping Specifications

Mapping Class	SysML v2 Concept
GenericToAnnotatingElement_Mapping	from
GenericToAnnotation_Mapping	from
GenericToAssociation_Mapping	from
GenericToBehavior_Mapping	from
GenericToClassifier_Mapping	from
GenericToConjugation_Mapping	from
GenericToConnector_Mapping	from
GenericToElement_Mapping	from
GenericToExpression_Mapping	from
GenericToFeature_Mapping	from
GenericToFeatureMembership_Mapping	from
GenericToFeatureTyping_Mapping	from
GenericToFeatureValue_Mapping	from
GenericToFunction_Mapping	from

Mapping Class	SysML v2 Concept
GenericToGeneralization_Mapping	from
GenericToImport_Mapping	from
GenericToMembership_Mapping	from
GenericToNamespace_Mapping	from
GenericToPackage_Mapping	from
GenericToParameterMembership_Mapping	from
GenericToRelationship_Mapping	from
GenericToReturnParameterMembership_Mapping	from
GenericToStep_Mapping	from
GenericToType_Mapping	from

C.2.1.3 Generic Mappings to Systems

Table 7. List of all Generic Mappings to Systems Mapping Specifications

Mapping Class	SysML v2 Concept
GenericToConjugatedPortDefinition_Mapping	from
GenericToConjugatedPortTyping_Mapping	from
GenericToConstraintDefinition_Mapping	from
GenericToDefinition_Mapping	from
GenericToItemDefinition_Mapping	from
GenericToPortConjugation_Mapping	from
GenericToPortDefinition_Mapping	from
GenericToUsage_Mapping	from

C.2.2 UML4SysML

C.2.2.1 Overview

C.2.2.2 Classification

C.2.2.2.1 Overview

Table 8. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
Classifier	Classifier	Classifier_Mapping
MultiplicityElement	MultiplicityElement	LowerBoundTyping_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityBound_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityBoundOwnership_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityBoundTyping_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityElement_Mapping

SysML v1 Concept	SysML v2 Concept	Mapping Class
MultiplicityElement	MultiplicityElement	MultiplicityLowerBound_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityLowerBoundOwnership_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityMembership_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityUpperBound_Mapping
MultiplicityElement	MultiplicityElement	MultiplicityUpperBoundOwnership_Mapping
StructuralFeature	StructuralFeature	StructuralFeature_Mapping
TypedElement	TypedElement	TypedElementToFeatureTyping_Mapping
MultiplicityElement	MultiplicityElement	UpperBoundTyping_Mapping

C.2.2.2.2 Mapping Specifications

C.2.2.2.2.1 Classifier_Mapping

General Mappings

GenericToClassifier_Mapping
Namespace_Mapping

Mapping Source

Classifier

Mapping Target

Classifier

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 9. Table Classifier_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set {}
Element::documentation	result = Set {}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Classifier::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set {}
Type::ownedFeatureMembership	result = Set {}
Namespace::ownedImport	result = Set {}
Namespace::ownedMembership	result = Set {}

Target Property	Target Value
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)

C.2.2.2.2.2 LowerBoundTyping_Mapping

General Mappings

MultiplicityBoundTyping_Mapping

Mapping Source

MultiplicityElement

Mapping Target

FeatureTyping

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 10. Table LowerBoundTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Generalization::general	<i>(abstract rule)</i>
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Generalization::specific	<i>(abstract rule)</i>
Relationship::target	result = Set{}
FeatureTyping::type	<i>(abstract rule)</i>
FeatureTyping::typedFeature	result = self.lowerBound.to
FeatureTyping::typedFeature	<i>(abstract rule)</i>

C.2.2.2.2.3 MultiplicityBound_Mapping

General Mappings

GenericToFeature_Mapping
FromElement_Mapping

Mapping Source

MultiplicityElement

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 11. Table MultiplicityBound_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Feature::ownedMembership	result = Set{}
Namespace::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.2.2.2.4 MultiplicityBoundOwnership_Mapping

General Mappings

GenericToFeatureMembership_Mapping
FromElement_Mapping

Mapping Source

MultiplicityElement

Mapping Target

FeatureMembership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 12. Table MultiplicityBoundOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
FeatureMembership::isComposite	result = true
Membership::memberElement	<i>(abstract rule)</i>
FeatureMembership::memberFeature	<i>(abstract rule)</i>
Membership::memberName	result = null
Membership::membershipOwningPackage	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = null
FeatureMembership::ownedMemberFeature	result = self.memberFeature()
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
FeatureMembership::owningType	result = MultiplicityElement_Mapping.getMapped(from)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.2.2.2.5 MultiplicityBoundTyping_Mapping

General Mappings

GenericToFeatureTyping_Mapping
FromElement_Mapping

Mapping Source

MultiplicityElement

Mapping Target

FeatureTyping

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 13. Table MultiplicityBoundTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Generalization::general	<i>(abstract rule)</i>
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Generalization::specific	<i>(abstract rule)</i>
Relationship::target	result = Set{}
FeatureTyping::type	result = Helper.getScalarValueTypeByName('Integer')
FeatureTyping::typedFeature	<i>(abstract rule)</i>

C.2.2.2.2.6 MultiplicityElement_Mapping

General Mappings

GenericToFeature_Mapping
FromElement_Mapping

Mapping Source

MultiplicityElement

Mapping Target

MultiplicityRange

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 14. Table MultiplicityElement_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>

Target Property	Target Value
Type::isAbstract	result = false
Type::isSufficient	result = false
MultiplicityRange::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = Set{}
MultiplicityRange::ownedMembership	result = Set{IBoundOwnership.to, uBoundOwnership.to}
Element::ownedRelationship	result = Set{}

C.2.2.2.2.7 MultiplicityLowerBound_Mapping

General Mappings

MultiplicityBound_Mapping

Mapping Source

MultiplicityElement

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 15. Table MultiplicityLowerBound_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isOrdered	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}

Target Property	Target Value
Namespace::ownedImport	result = Set{}
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = Set{self.lowerBoundTyping.to}

C.2.2.2.2.8 MultiplicityMembership_Mapping

General Mappings

GenericToFeatureMembership_Mapping
FromElement_Mapping

Mapping Source

MultiplicityElement

Mapping Target

FeatureMembership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 16. Table MultiplicityMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureMembership::isComposite	result = true
Membership::memberElement	(abstract rule)
FeatureMembership::memberFeature	result = self.multiplicityElement.to
Membership::memberName	result = null
Membership::membershipOwningPackage	(abstract rule)
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = null
FeatureMembership::ownedMemberFeature	result = self.memberFeature()
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
FeatureMembership::owningType	result = StructuralFeature_Mapping.getMapped(from)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.2.2.2.9 MultiplicityLowerBoundOwnership_Mapping

General Mappings

MultiplicityBoundOwnership_Mapping

Mapping Source

MultiplicityElement

Mapping Target

FeatureMembership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 17. Table MultiplicityLowerBoundOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
FeatureMembership::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureMembership::isComposite	result = false
FeatureMembership::isDerived	result = false
FeatureMembership::isPort	result = false
FeatureMembership::isPortion	result = false
FeatureMembership::isReadOnly	result = false
FeatureMembership::memberFeature	(abstract rule)
FeatureMembership::memberFeature	self.lowerBound.to
FeatureMembership::memberName	result = 'lowerBound'
Element::ownedAnnotation	result = Set{}
FeatureMembership::ownedMemberFeature	result = self.memberFeature()
FeatureMembership::ownedMemberFeature	result = null
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
FeatureMembership::owningType	(abstract rule)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.2.2.2.10 MultiplicityUpperBound_Mapping

General Mappings

MultiplicityBound_Mapping

Mapping Source

MultiplicityElement

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 18. Table MultiplicityUpperBound_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	result = false
Feature::isOrdered	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = Set{self.upperBoundTyping.to}

C.2.2.2.2.11 MultiplicityUpperBoundOwnership_Mapping

General Mappings

MultiplicityBoundOwnership_Mapping

Mapping Source

MultiplicityElement

Mapping Target

FeatureMembership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 19. Table MultiplicityUpperBoundOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
FeatureMembership::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureMembership::isComposite	result = false
FeatureMembership::isDerived	result = false
FeatureMembership::isPort	result = false
FeatureMembership::isPortion	result = false
FeatureMembership::isReadOnly	result = false
FeatureMembership::memberFeature	(abstract rule)
FeatureMembership::memberFeature	result = self.upperBound.to
FeatureMembership::memberName	result = 'upperBound'
Element::ownedAnnotation	result = Set{}
FeatureMembership::ownedMemberFeature	result = null
FeatureMembership::ownedMemberFeature	result = self.memberFeature()
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
FeatureMembership::owningType	(abstract rule)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.2.2.2.12 StructuralFeature_Mapping

General Mappings

GenericToFeature_Mapping
ElementMain_Mapping

Mapping Source

StructuralFeature

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 20. Table StructuralFeature_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set {}
Element::documentation	result = Set {}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Feature::isAbstract	result = false
Feature::isOrdered	result = from.isOrdered
Type::isSufficient	result = false
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set {}
Type::ownedFeatureMembership	result = Set {}
Namespace::ownedImport	result = Set {}
Namespace::ownedMembership	result = Set {}
Feature::ownedMembership	Set {self.multiplicityMembership.to}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.ocllsUndefined() then Set {} else Set {typing} endif
Element::ownedRelationship	result = Set {}

C.2.2.2.13 TypedElementToFeatureTyping_Mapping

General Mappings

GenericToFeatureTyping_Mapping

Mapping Source

TypedElement

Mapping Target

FeatureTyping

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
result = not from.type.ocllsUndefined() and not from.ocllsKindOf(UML::ValueSpecification)

Table 21. Table TypedElementToFeatureTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Generalization::general	<i>(abstract rule)</i>
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Generalization::specific	<i>(abstract rule)</i>
Relationship::target	result = Set{}
FeatureTyping::type	result = if from.type.ocIsKindOf(UML::PrimitiveType) then Helper.getScalarValueType(from.type) else Classifier_Mapping.getMapped(from.type) endif
FeatureTyping::typedFeature	result = StructuralFeature_Mapping.getMapped(from)

C.2.2.2.2.14 UpperBoundTyping_Mapping**General Mappings**

MultiplicityBoundTyping_Mapping

Mapping Source

MultiplicityElement

Mapping Target

FeatureTyping

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 22. Table UpperBoundTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Generalization::general	<i>(abstract rule)</i>
Element::humanId	result = null

Target Property	Target Value
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Generalization::specific	<i>(abstract rule)</i>
Relationship::target	result = Set{}
FeatureTyping::type	result = Helper.getScalarValueTypeByName('UnlimitedNatural')
FeatureTyping::type	<i>(abstract rule)</i>
FeatureTyping::typedFeature	<i>(abstract rule)</i>
FeatureTyping::typedFeature	result = self.upperBound.to

C.2.2.3 CommonBehavior

C.2.2.3.1 Overview

Table 23. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
Behavior	Behavior	Behavior_Mapping

C.2.2.3.2 Mapping Specifications

C.2.2.3.2.1 Behavior

General Mappings

GenericToBehavior_Mapping
Class_Mapping

Mapping Source

Behavior

Mapping Target

Behavior

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 24. Table Behavior Rules

Target Property	Target Value
Element::aliasId	result = Set{}

Target Property	Target Value
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Classifier::ownedFeatureMembership	result = if from.classifierBehavior.ocIsUndefined() then Set{} else (OrderedSet{from.classifierBehavior})->collect(e thisModule.resolve_BehavoredClassifierToFeatureMembership_Mapping(from)) endif
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet{}

C.2.2.4 CommonStructure

C.2.2.4.1 Overview

Table 25. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
Abstraction	Abstraction	Abstraction_Mapping
Comment	Comment	Comment_Mapping
Comment	Comment	CommentToAnnotation_Mapping
Dependency	Dependency	Dependency_Mapping
DirectedRelationship	DirectedRelationship	DirectedRelationship_Mapping
Element	Element	ElementMain_Mapping
Element	Element	ElementOwnership_Mapping
Element	Element	ElementOwningMembership_Mapping
Element	Element	FromElement_Mapping
Namespace	Namespace	Namespace_Mapping
Relationship	Relationship	Relationship_Mapping

C.2.2.4.2 Mapping Specifications

C.2.2.4.2.1 Abstraction Mapping

General Mappings

Dependency_Mapping

Mapping Source

Abstraction

Mapping Target

Dependency

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 26. Table Abstraction Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Relationship::source	result = from.source->collect(e ElementMain_Mapping.getMapped(e))
Relationship::target	result = from.target->collect(e ElementMain_Mapping.getMapped(e))

C.2.2.4.2.2 Comment_Mapping

General Mappings

ElementMain_Mapping
GenericToAnnotatingElement_Mapping

Mapping Source

Comment

Mapping Target

Comment

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 27. Table Comment_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Comment::annotation	result = from.annotatedElement->collect(e CommentToAnnotation_Mapping.getMapped(from, e))
Comment::body	result = if from.body->isEmpty() then " else from.body endif
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Comment::ownedRelationship	result = self.annotation()
Element::ownedRelationship	result = Set{}

C.2.2.4.2.3 CommentToAnnotation_Mapping**General Mappings**

GenericToAnnotation_Mapping
FromElement_Mapping

Mapping Source

Comment

Mapping Target

Annotation with qualifier: annotatedElement:Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 28. Table CommentToAnnotation_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Annotation::annotatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Annotation::annotatingElement	result = Comment_Mapping.getMapped(from)
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}

Target Property	Target Value
Annotation::owningAnnotatedElement	result = null
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.2.4.2.4 Dependency_Mapping

General Mappings

DirectedRelationship_Mapping

Mapping Source

Dependency

Mapping Target

Dependency

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 29. Table Dependency_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Dependency::client	result = from.source->collect(e ElementMain_Mapping.getMapped(e))
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Relationship::source	result = Set{}
Dependency::supplier	result = from.target->collect(e ElementMain_Mapping.getMapped(e))
Relationship::target	result = Set{}

C.2.2.4.2.5 DirectRelationship_Mapping

General Mappings

Relationship_Mapping

Mapping Source

DirectedRelationship

Mapping Target

Relationship

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 30. Table DirectRelationship_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)
Relationship::owningRelatedElement	result = null
Relationship::source	result = from.source->collect(e ElementMain_Mapping.getMapped(e))
Relationship::target	result = from.target->collect(e ElementMain_Mapping.getMapped(e))

C.2.2.4.2.6 ElementMain_Mapping

General Mappings

GenericToElement_Mapping

Mapping Source

Element

Mapping Target

Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 31. Table ElementMain_Mapping Rules

Target Property	Target Value
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)

C.2.2.4.2.7 ElementOwnership_Mapping

General Mappings

GenericToRelationship_Mapping

Mapping Source

Element

Mapping Target

Relationship with qualifier: owner:Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 32. Table ElementOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = self.target()
Element::ownedRelationship	result = Set{}
Relationship::source	result = OrderedSet{ElementMain_Mapping.getMapped(from.owner)}
Relationship::target	result = OrderedSet{ElementMain_Mapping.getMapped(from)}

C.2.2.4.2.8 ElementOwningMembership_Mapping

General Mappings

ElementOwnership_Mapping

GenericToMembership_Mapping

Mapping Source

Element

Mapping Target

Membership with qualifier: owner:Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 33. Table ElementOwningMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Membership::memberElement	result = self.target()->at(1)
Membership::memberName	result = if (from.oclIsKindOf(UML::NamedElement)) then from.oclAsType(UML::NamedElement).name else null endif
Membership::membershipOwningPackage	result = Namespace_Mapping.getMapped(from.owner)
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = self.memberElement()
Relationship::ownedRelatedElement	result = Set{}
Membership::ownedRelatedElement	result = OrderedSet{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}
Membership::visibility	result = if (from.oclIsKindOf(UML::NamedElement)) then from.oclAsType(UML::NamedElement).visibility else KerML::VisibilityKind::public endif

C.2.2.4.2.9 FromElement_Mapping

General Mappings

GenericToElement_Mapping

Mapping Source

Element

Mapping Target

Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

C.2.2.4.2.10 Namespace_Mapping

General Mappings

GenericToNamespace_Mapping
ElementMain_Mapping

Mapping Source

Namespace

Mapping Target

Namespace

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 35. Table Namespace_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set {}
Element::documentation	result = Set {}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set {}
Namespace::ownedImport	result = Set {}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet {}
Element::ownedRelationship	result = Set {}

C.2.2.4.2.11 Relationship_Mapping

General Mappings

GenericToRelationship_Mapping
ElementMain_Mapping

Mapping Source

Relationship

Mapping Target

Relationship

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 36. Table Relationship_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)

C.2.2.5 Packages

C.2.2.5.1 Overview

Table 37. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
ElementImport	ElementImport	ElementImport_Mapping
undefined	undefined	getMappingElement
Package	Package	Package_Mapping
PackageImport	PackageImport	PackageImport_Mapping

C.2.2.5.2 Mapping Specifications

C.2.2.5.2.1 ElementImport_Mapping

General Mappings

GenericToMembership_Mapping
DirectedRelationship_Mapping

Mapping Source

ElementImport

Mapping Target

Membership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 38. Table ElementImport_Mapping Rules

Target Property	Target Value
Membership::aliases	result = from.alias->asSet()
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Membership::memberElement	result = ElementMain_Mapping.getMapped(from.importedElement)
Membership::memberName	result = from.importedElement.name
Membership::membershipOwningPackage	result = Namespace_Mapping.getMapped(from.importingNamespace)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Relationship::source	result = Set{}
Relationship::target	result = Set{}
Membership::visibility	result = Helper.getKerMLVisibilityKind(from.visibility)

C.2.2.5.2.2 Package_Mapping**General Mappings**

Namespace_Mapping

Mapping Source

Package

Mapping Target

Package

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 39. Table Package_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}

Target Property	Target Value
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)

C.2.2.5.2.3 PackageImport_Mapping

General Mappings

DirectedRelationship_Mapping

Mapping Source

PackageImport

Mapping Target

Import

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 40. Table PackageImport_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Import::importedPackage	result = Namespace_Mapping.getMapped(from.importedPackage)
Import::importOwningPackage	result = Namespace_Mapping.getMapped(from.importingNamespace)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Relationship::source	result = Set{}
Relationship::target	result = Set{}

Target Property	Target Value
Import::visibility	result = Helper.getKerMLVisibilityKind(from.visibility)

C.2.2.6 SimpleClassifiers

C.2.2.6.1 Overview

Table 41. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
BehavioredClassifier	BehavioredClassifier	BehavioredClassifier_Mapping
BehavioredClassifier	BehavioredClassifier	BehavioredClassifierToFeatureMembership_Mapping
BehavioredClassifier	BehavioredClassifier	BehavioredClassifierToPerformActionUsage_Mapping
DataType	DataType	DataType_Mapping
Enumeration	Enumeration	Enumeration_Mapping
EnumerationLiteral	EnumerationLiteral	EnumerationLiteral_Mapping
Signal	Signal	Signal_Mapping

C.2.2.6.2 Mapping Specifications

C.2.2.6.2.1 BehavioredClassifier_Mapping

General Mappings

Classifier_Mapping

Mapping Source

BehavioredClassifier

Mapping Target

Classifier

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 42. Table BehavioredClassifier_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	result = false
Type::isSufficient	result = false

Target Property	Target Value
Element::ownedAnnotation	result = Set{}
Classifier::ownedFeatureMembership	result = if from.classifierBehavior.ocllsUndefined() then Set{} else (OrderedSet{from.classifierBehavior}->collect(e thisModule.resolve_BehavoredClassifierToFeatureMembership_Mapping(from)) endif
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet{}

C.2.2.6.2.2 DataType_Mapping

General Mappings

Classifier_Mapping

Mapping Source

DataType

Mapping Target

DataType

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 43. Table DataType_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet{}

C.2.2.6.2.3 Enumeration_Mapping

General Mappings

DataType_Mapping

Mapping Source

Enumeration

Mapping Target

EnumerationDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 44. Table Enumeration_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Classifier::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet{}

C.2.2.6.2.4 EnumerationLiteral_Mapping

General Mappings

ElementMain_Mapping

Mapping Source

EnumerationLiteral

Mapping Target

EnumerationUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 45. Table EnumerationLiteral_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.2.6.2.5 Signal

General Mappings

DataType_Mapping

Mapping Source

Signal

Mapping Target

AttributeDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 46. Table Signal Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Classifier::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))

Target Property	Target Value
Namespace::ownedRelationship	result = OrderedSet{}

C.2.2.7 StructuredClassifiers

C.2.2.7.1 Overview

Table 47. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
Association	Association	Association_Mapping
AssociationClass	AssociationClass	AssociationClass_Mapping
Association	Association	AssociationToMetadata_Mapping
Class	Class	Class_Mapping
Connector	Connector	Connector_Mapping
Classifier	Classifier	EncapsulatedClassifier_Mapping
Port	Port	Port_Mapping
Classifier	Classifier	StructuredClassifier_Mapping

C.2.2.7.2 Mapping Specifications

C.2.2.7.2.1 Association_Mapping

General Mappings

Classifier_Mapping
Relationship_Mapping

Mapping Source

Association

Mapping Target

Association

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 48. Table Association_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false

Target Property	Target Value
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Relationship::ownedRelatedElement	result = Set{}
Association::ownedRelatedElement	result = from.relatedElement->asSet()->intersection(from.ownedElement->asSet()->collect(e ElementMain_Mapping.getMapped(e)))
Namespace::ownedRelationship	result = OrderedSet{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.2.7.2.2 AssociationClass_Mapping

General Mappings

Association_Mapping

Class_Mapping

Mapping Source

AssociationClass

Mapping Target

Association

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 49. Table AssociationClass_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Classifier::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}

Target Property	Target Value
Classifier::ownedFeatureMembership	result = if from.classifierBehavior.ocllIsUndefined() then Set{} else (OrderedSet{from.classifierBehavior}->collect(e thisModule.resolve_BehavioiredClassifierToFeatureMembership_Mapping(from)) endif
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Relationship::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))
Namespace::ownedRelationship	result = OrderedSet{}
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.2.7.2.3 AssociationToMetadata_Mapping

General Mappings

GenericToAnnotatingElement_Mapping

GenericToFeature_Mapping

Mapping Source

Association

Mapping Target

AnnotatingFeature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 50. Table AssociationToMetadata_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}

Target Property	Target Value
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.2.7.2.4 Class_Mapping

General Mappings

EncapsulatedClassifier_Mapping

BehavioredClassifier_Mapping

Mapping Source

Class

Mapping Target

PartDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 51. Table Class_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
PartDefinition::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet{}

C.2.2.7.2.5 ConnectorMapping

General Mappings

GenericToConnector_Mapping

Mapping Source

Connector

Mapping Target

Connector

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 52. Table ConnectorMapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isOrdered	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Feature::ownedMembership	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.2.7.2.6 EncapsulatedClassifier_Mapping

General Mappings

StructuredClassifier_Mapping

Mapping Source

Classifier

Mapping Target

Classifier

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 53. Table EncapsulatedClassifier_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set {}
Element::documentation	result = Set {}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Classifier::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set {}
Type::ownedFeatureMembership	result = Set {}
Namespace::ownedImport	result = Set {}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet {}

C.2.2.7.2.7 Port_Mapping

General Mappings

StructuralFeature_Mapping

Mapping Source

Port

Mapping Target

PortUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 54. Table Port_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set {}
Element::documentation	result = Set {}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>

Target Property	Target Value
Type::isAbstract	result = false
Feature::isOrdered	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set {}
Type::ownedFeatureMembership	result = Set {}
Namespace::ownedImport	result = Set {}
Feature::ownedMembership	result = Set {}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement, from)
PortUsage::ownedRelationship	result = if thisModule.TypedElementToFeatureTyping_Mapping((src)).oclIsUndefined() then Set {} else Set {thisModule.TypedElementToFeatureTyping_Mapping((src))} endif

C.2.2.7.2.8 StructuredClassifier_Mapping

General Mappings

Classifier_Mapping

Mapping Source

Classifier

Mapping Target

Classifier

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 55. Table StructuredClassifier_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set {}
Element::documentation	result = Set {}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set {}

Target Property	Target Value
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e, from))
Namespace::ownedRelationship	result = OrderedSet{}

C.2.3 SysML v1.6

C.2.3.1 Overview