

Date: August 2021



OMG Systems Modeling Language TM (SysML®)

*Version 2.0
Revised Submission*

OMG Document Number: ad/2021-08-03

Machine Readable Files:

SysML Metamodel (XMI) ad/2021-08-07
SysML Model Library (textual notation) ad/2021-08-08
SysML v1 to v2 Transformation (XMI) ad/2021-08-09

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

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

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, INCOSE
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 Kaiserslauten
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 Document Overview	69
6.2 Document Organization	70
6.3 Document Conventions.....	71
6.4 Acknowledgements	72
7 Language Description	75
7.1 Language Overview	75
7.2 Elements and Relationships	76
7.2.1 Overview	76
7.2.2 Abstract Syntax	77
7.2.3 Notation.....	77
7.3 Annotations	78
7.3.1 Overview	78
7.3.2 Abstract Syntax	79
7.3.3 Notation.....	80
7.4 Namespaces and Packages	85
7.4.1 Overview	85
7.4.2 Abstract Syntax	86
7.4.3 Notation.....	87
7.5 Dependencies	94
7.5.1 Overview	94
7.5.2 Abstract Syntax	94
7.5.3 Notation.....	94
7.6 Definition and Usage	95
7.6.1 Overview	95
7.6.2 Abstract Syntax	98
7.6.3 Notation.....	101
7.7 Attributes.....	105
7.7.1 Overview	105
7.7.2 Abstract Syntax	106
7.7.3 Notation.....	106
7.8 Enumerations	107
7.8.1 Overview	107
7.8.2 Abstract Syntax	108
7.8.3 Notation.....	108

7.9 Occurrences.....	109
7.9.1 Overview	109
7.9.2 Abstract Syntax	111
7.9.3 Notation.....	112
7.10 Items.....	115
7.10.1 Overview	115
7.10.2 Abstract Syntax	116
7.10.3 Notation.....	116
7.11 Parts.....	117
7.11.1 Overview	117
7.11.2 Abstract Syntax	118
7.11.3 Notation.....	118
7.12 Ports	120
7.12.1 Overview	120
7.12.2 Abstract Syntax	121
7.12.3 Notation.....	122
7.13 Connections.....	123
7.13.1 Overview	123
7.13.2 Abstract Syntax	125
7.13.3 Notation.....	126
7.14 Interfaces.....	129
7.14.1 Overview	129
7.14.2 Abstract Syntax	130
7.14.3 Notation.....	130
7.15 Allocations	131
7.15.1 Overview	131
7.15.2 Abstract Syntax	132
7.15.3 Notation.....	133
7.16 Actions	133
7.16.1 Overview	133
7.16.2 Abstract Syntax	136
7.16.3 Notation.....	138
7.17 States	141
7.17.1 Overview	141
7.17.2 Abstract Syntax	143
7.17.3 Notation.....	145
7.18 Calculations.....	147
7.18.1 Overview	147
7.18.2 Abstract Syntax	148
7.18.3 Notation.....	148
7.19 Constraints	149
7.19.1 Overview	149
7.19.2 Abstract Syntax	150
7.19.3 Notation.....	151
7.20 Requirements	152
7.20.1 Overview	152
7.20.2 Abstract Syntax	154
7.20.3 Notation.....	156
7.21 Cases	157
7.21.1 Overview	157
7.21.2 Abstract Syntax	158
7.21.3 Notation.....	159
7.22 Analysis Cases	159
7.22.1 Overview	159

7.22.2 Abstract Syntax	160
7.22.3 Notation	161
7.23 Verification Cases	161
7.23.1 Overview	161
7.23.2 Abstract Syntax	162
7.23.3 Notation	163
7.24 Use Cases	164
7.24.1 Overview	164
7.24.2 Abstract Syntax	165
7.24.3 Notation	166
7.25 Views and Viewpoints	166
7.25.1 Overview	166
7.25.2 Abstract Syntax	167
7.25.3 Notation	169
7.26 Language Extension	171
8 Metamodel	173
8.1 Metamodel Overview	173
8.2 Concrete Syntax	173
8.2.1 Concrete Syntax Overview	173
8.2.2 Textual Notation	173
8.2.2.1 Textual Notation Overview	173
8.2.2.1.1 Lexical Structure	173
8.2.2.1.2 EBNF Conventions	174
8.2.2.2 Elements Textual Notation	175
8.2.2.3 Annotations Textual Notation	175
8.2.2.3.1 Comments	175
8.2.2.3.2 Documentation	175
8.2.2.3.3 Textual Representation	176
8.2.2.3.4 Annotating Features	176
8.2.2.4 Packages Textual Notation	176
8.2.2.4.1 Packages	177
8.2.2.4.2 Package Elements	178
8.2.2.5 Dependencies Textual Notation	179
8.2.2.6 Definition and Usage Textual Notation	179
8.2.2.6.1 Definitions	179
8.2.2.6.2 Usages	180
8.2.2.6.3 Reference Usages	181
8.2.2.6.4 Body Elements	182
8.2.2.6.5 Specialization	183
8.2.2.6.6 Multiplicity	184
8.2.2.7 Attributes Textual Notation	184
8.2.2.8 Enumerations Textual Notation	185
8.2.2.9 Occurrences Textual Notation	185
8.2.2.9.1 Occurrence Definitions	185
8.2.2.9.2 Occurrence Usages	185
8.2.2.9.3 Occurrence Successions	186
8.2.2.10 Items Textual Notation	187
8.2.2.11 Parts Textual Notation	187
8.2.2.12 Ports Textual Notation	187
8.2.2.13 Connections Textual Notation	187
8.2.2.13.1 Connection Definition and Usage	187
8.2.2.13.2 Binding Connectors	188
8.2.2.13.3 Successions	188
8.2.2.13.4 Messages and Flow Connection Usages	188

8.2.2.14 Interfaces Textual Notation	190
8.2.2.14.1 Interface Definitions.....	190
8.2.2.14.2 Interface Usages	190
8.2.2.15 Allocations Textual Notation	191
8.2.2.16 Actions Textual Notation	191
8.2.2.16.1 Action Definitions	191
8.2.2.16.2 Action Usages.....	192
8.2.2.16.3 Action Parameters	193
8.2.2.16.4 Action Nodes.....	195
8.2.2.16.5 Action Successions.....	196
8.2.2.17 States Textual Notation	196
8.2.2.17.1 State Definitions.....	196
8.2.2.17.2 State Usages	198
8.2.2.17.3 Transition Usages	198
8.2.2.18 Calculations Textual Notation.....	199
8.2.2.18.1 Calculation Definitions.....	199
8.2.2.18.2 Calculation Usages	200
8.2.2.19 Constraints Textual Notation	200
8.2.2.20 Requirements Textual Notation	201
8.2.2.20.1 Requirement Definitions	201
8.2.2.20.2 Requirement Usages.....	202
8.2.2.20.3 Concerns	202
8.2.2.21 Cases Textual Notation	202
8.2.2.22 Analysis Cases Textual Notation	203
8.2.2.23 Verification Cases Textual Notation	203
8.2.2.24 Use Cases Textual Notation	204
8.2.2.25 Views and Viewpoints Textual Notation	204
8.2.2.25.1 View Definitions	204
8.2.2.25.2 View Usages.....	204
8.2.2.25.3 Viewpoints.....	205
8.2.2.25.4 Renderings.....	205
8.2.3 Graphical Notation	205
8.2.3.1 Graphical Notation Overview	205
8.2.3.2 Elements and Relationships Graphical Notation.....	206
8.2.3.3 Annotations Graphical Notation.....	206
8.2.3.4 Namespaces and Packages Graphical Notation	208
8.2.3.5 Dependencies Graphical Notation.....	210
8.2.3.6 Definition and Usage Graphical Notation.....	211
8.2.3.7 Attributes Graphical Notation	214
8.2.3.8 Enumerations Graphical Notation	214
8.2.3.9 Occurrences Graphical Notation	214
8.2.3.10 Items Graphical Notation	214
8.2.3.11 Parts Graphical Notation	214
8.2.3.12 Ports Graphical Notation.....	214
8.2.3.13 Connections Graphical Notation	214
8.2.3.14 Interfaces Graphical Notation	214
8.2.3.15 Allocations Graphical Notation	214
8.2.3.16 Actions Graphical Notation.....	214
8.2.3.17 States Graphical Notation.....	214
8.2.3.18 Calculations Graphical Notation	214
8.2.3.19 Constraints Graphical Notation	214
8.2.3.20 Requirements Graphical Notation	214
8.2.3.21 Cases Graphical Notation.....	214
8.2.3.22 Analysis Cases Graphical Notation	214

8.2.3.23 Verification Cases Graphical Notation	215
8.2.3.24 Use Cases Graphical Notation.....	215
8.2.3.25 Views and Viewpoints Graphical Notation.....	215
8.3 Abstract Syntax	215
8.3.1 Abstract Syntax Overview	215
8.3.2 Dependencies Abstract Syntax.....	215
8.3.2.1 Overview	215
8.3.2.2 Dependency	215
8.3.3 Definition and Usage Abstract Syntax	216
8.3.3.1 Overview	216
8.3.3.2 Definition	217
8.3.3.3 ReferenceUsage.....	220
8.3.3.4 Usage.....	221
8.3.3.5 VariantMembership.....	225
8.3.4 Attributes Abstract Syntax	225
8.3.4.1 Overview	225
8.3.4.2 AttributeUsage	225
8.3.4.3 AttributeDefinition	226
8.3.5 Enumerations Abstract Syntax	227
8.3.5.1 Overview	227
8.3.5.2 EnumerationDefinition	227
8.3.5.3 EnumerationUsage	228
8.3.6 Occurrences Abstract Syntax	228
8.3.6.1 Overview	229
8.3.6.2 EventOccurrenceUsage	229
8.3.6.3 LifeClass.....	230
8.3.6.4 OccurrenceDefinition	230
8.3.6.5 OccurrenceUsage	231
8.3.6.6 PortioningFeature	233
8.3.6.7 PortionKind	233
8.3.7 Items Abstract Syntax	234
8.3.7.1 Overview	234
8.3.7.2 ItemDefinition	234
8.3.7.3 ItemUsage.....	235
8.3.8 Parts Abstract Syntax	235
8.3.8.1 Overview	235
8.3.8.2 PartDefinition	236
8.3.8.3 PartUsage	236
8.3.9 Ports Abstract Syntax	236
8.3.9.1 Overview	237
8.3.9.2 ConjugatedPortDefinition	237
8.3.9.3 ConjugatedPortTyping	238
8.3.9.4 PortConjugation	239
8.3.9.5 PortDefinition.....	239
8.3.9.6 PortUsage	240
8.3.10 Connections Abstract Syntax	240
8.3.10.1 Overview	241
8.3.10.2 BindingConnectorAsUsage	242
8.3.10.3 ConnectionDefinition	242
8.3.10.4 ConnectionUsage	243
8.3.10.5 ConnectorAsUsage.....	243
8.3.10.6 FlowConnectionUsage	244
8.3.10.7 SuccessionAsUsage.....	244
8.3.10.8 SuccessionFlowConnectionUsage	245

8.3.11 Interfaces Abstract Syntax	245
8.3.11.1 Overview	246
8.3.11.2 InterfaceDefinition	246
8.3.11.3 InterfaceUsage.....	247
8.3.12 Allocations Abstract Syntax.....	247
8.3.12.1 Overview	248
8.3.12.2 AllocationDefinition.....	248
8.3.12.3 AllocationUsage	249
8.3.13 Actions Abstract Syntax.....	249
8.3.13.1 Overview	250
8.3.13.2 AcceptActionUsage.....	251
8.3.13.3 ActionDefinition.....	252
8.3.13.4 ActionUsage.....	252
8.3.13.5 ControlNode	253
8.3.13.6 DecisionNode	253
8.3.13.7 ForkNode.....	254
8.3.13.8 JoinNode.....	254
8.3.13.9 MergeNode.....	255
8.3.13.10 PerformActionUsage.....	255
8.3.13.11 SendActionUsage	256
8.3.13.12 TransferActionUsage	256
8.3.14 States Abstract Syntax.....	257
8.3.14.1 Overview	257
8.3.14.2 ExhibitStateUsage	258
8.3.14.3 StateSubactionKind	259
8.3.14.4 StateSubactionMembership	259
8.3.14.5 StateDefinition	260
8.3.14.6 StateUsage	261
8.3.14.7 TransitionFeatureKind	262
8.3.14.8 TransitionFeatureMembership	263
8.3.14.9 TransitionUsage	263
8.3.15 Calculations Abstract Syntax	264
8.3.15.1 Overview	265
8.3.15.2 CalculationDefinition	265
8.3.15.3 CalculationUsage	266
8.3.16 Constraints Abstract Syntax	266
8.3.16.1 Overview	266
8.3.16.2 AssertConstraintUsage	267
8.3.16.3 ConstraintDefinition.....	267
8.3.16.4 ConstraintUsage	268
8.3.17 Requirements Abstract Syntax	268
8.3.17.1 Overview	269
8.3.17.2 ActorMembership.....	271
8.3.17.3 ConcernDefinition	271
8.3.17.4 ConcernUsage	272
8.3.17.5 FramedConcernMembership	272
8.3.17.6 RequirementConstraintKind	273
8.3.17.7 RequirementConstraintMembership	273
8.3.17.8 RequirementDefinition	274
8.3.17.9 RequirementUsage	275
8.3.17.10 SatisfyRequirementUsage	277
8.3.17.11 SubjectMembership.....	277
8.3.17.12 StakeholderMembership.....	278

8.3.18 Cases Abstract Syntax	278
8.3.18.1 Overview	279
8.3.18.2 CaseDefinition.....	280
8.3.18.3 CaseUsage	280
8.3.18.4 ObjectiveMembership	281
8.3.19 Analysis Cases Abstract Syntax	282
8.3.19.1 Overview	282
8.3.19.2 AnalysisCaseDefinition.....	282
8.3.19.3 AnalysisCaseUsage	283
8.3.20 Verification Cases Abstract Syntax	284
8.3.20.1 Overview	284
8.3.20.2 RequirementVerificationMembership.....	285
8.3.20.3 VerificationCaseDefinition	286
8.3.20.4 VerificationCaseUsage	286
8.3.21 Use Cases Abstract Syntax.....	287
8.3.21.1 Overview	287
8.3.21.2 IncludeUseCaseUsage.....	288
8.3.21.3 UseCaseDefinition	288
8.3.21.4 UseCaseUsage	289
8.3.22 Views Abstract Syntax	289
8.3.22.1 Overview	289
8.3.22.2 Expose	292
8.3.22.3 RenderingDefinition.....	292
8.3.22.4 RenderingUsage	293
8.3.22.5 ViewDefinition.....	293
8.3.22.6 ViewpointDefinition.....	294
8.3.22.7 ViewpointUsage	295
8.3.22.8 ViewRenderingMembership	295
8.3.22.9 ViewUsage	296
8.4 Semantics	297
8.4.1 Semantics Overview.....	297
8.4.2 Definition and Usage Semantics	297
8.4.3 Attributes Semantics	297
8.4.4 Enumerations Semantics	297
8.4.5 Occurrences Semantics	297
8.4.6 Items Semantics	297
8.4.7 Parts Semantics	297
8.4.8 Ports Semantics	297
8.4.9 Connections Semantics	297
8.4.10 Interfaces Semantics.....	297
8.4.11 Allocations Semantics.....	297
8.4.12 Actions Semantics	297
8.4.13 States Semantics	297
8.4.14 Calculations Semantics	297
8.4.15 Constraints Semantics	297
8.4.16 Requirements Semantics	297
8.4.17 Cases Semantics	297
8.4.18 Analysis Cases Semantics	298
8.4.19 Verification Cases Semantics	298
8.4.20 Use Cases Semantics	298
8.4.21 View Semantics.....	298
9 Model Libraries.....	299
9.1 Model Libraries Overview	299

9.2 Systems Model Library	299
9.2.1 Overview	299
9.2.2 Attributes	299
9.2.2.1 Attributes Overview	299
9.2.2.2 Elements	299
9.2.2.2.1 attributeValues	300
9.2.2.2.2 AttributeValue	300
9.2.3 Items	300
9.2.3.1 Items Overview	300
9.2.3.2 Elements	300
9.2.3.2.1 Item	300
9.2.3.2.2 items	301
9.2.4 Parts	301
9.2.4.1 Parts Overview	301
9.2.4.2 Elements	301
9.2.4.2.1 Part	301
9.2.4.2.2 parts	302
9.2.5 Ports	302
9.2.5.1 Ports Overview	302
9.2.5.2 Elements	302
9.2.5.2.1 Port	302
9.2.5.2.2 ports	303
9.2.6 Connections	303
9.2.6.1 Connections Overview	303
9.2.6.2 Elements	303
9.2.6.2.1 Connection	303
9.2.6.2.2 connections	303
9.2.7 Interfaces	304
9.2.7.1 Interfaces Overview	304
9.2.7.2 Elements	304
9.2.7.2.1 Interface	304
9.2.7.2.2 interfaces	304
9.2.8 Allocations	305
9.2.8.1 Allocations Overview	305
9.2.8.2 Elements	305
9.2.8.2.1 Allocation	305
9.2.8.2.2 allocations	305
9.2.9 Actions	306
9.2.9.1 Actions Overview	306
9.2.9.2 Elements	306
9.2.9.2.1 AcceptAction	306
9.2.9.2.2 Action	306
9.2.9.2.3 actions	307
9.2.9.2.4 ControlAction	308
9.2.9.2.5 DecisionAction	308
9.2.9.2.6 ForkAction	308
9.2.9.2.7 JoinAction	309
9.2.9.2.8 MergeAction	309
9.2.9.2.9 SendAction	309
9.2.10 States	310
9.2.10.1 States Overview	310
9.2.10.2 Elements	310
9.2.10.2.1 StateAction	310
9.2.10.2.2 stateActions	311

9.2.10.2.3 TransitionAction	311
9.2.10.2.4 transitionActions	311
9.2.11 Calculations	312
9.2.11.1 Calculations Overview	312
9.2.11.2 Elements	312
9.2.11.2.1 Calculation	312
9.2.11.2.2 calculations	312
9.2.12 Constraints	313
9.2.12.1 Constraints Overview	313
9.2.12.2 Elements	313
9.2.12.2.1 ConstraintCheck	313
9.2.12.2.2 constraintChecks	313
9.2.13 Requirements	313
9.2.13.1 Requirements Overview	313
9.2.13.2 Elements	314
9.2.13.2.1 ConcernCheck	314
9.2.13.2.2 concernChecks	314
9.2.13.2.3 DesignConstraintCheck	314
9.2.13.2.4 FunctionalRequirementCheck	315
9.2.13.2.5 InterfaceRequirementCheck	315
9.2.13.2.6 PerformanceRequirementCheck	315
9.2.13.2.7 PhysicalRequirementCheck	316
9.2.13.2.8 RequirementCheck	316
9.2.13.2.9 requirementChecks	317
9.2.13.2.10 Stakeholder	317
9.2.13.2.11 stakeholders	317
9.2.14 Cases	318
9.2.14.1 Cases Overview	318
9.2.14.2 Elements	318
9.2.14.2.1 Case	318
9.2.14.2.2 cases	318
9.2.15 Analysis Cases	319
9.2.15.1 Analysis Cases Overview	319
9.2.15.2 Elements	319
9.2.15.2.1 AnalysisAction	319
9.2.15.2.2 AnalysisCase	319
9.2.15.2.3 analysisCases	320
9.2.16 Verification Cases	320
9.2.16.1 Verification Cases Overview	320
9.2.16.2 Elements	320
9.2.16.2.1 VerdictKind	320
9.2.16.2.2 VerificationCase	321
9.2.16.2.3 verificationCases	322
9.2.16.2.4 VerificationCheck	322
9.2.17 Use Cases	322
9.2.17.1 Use Cases Overview	322
9.2.17.2 Elements	322
9.2.17.2.1 UseCase	322
9.2.17.2.2 useCases	323
9.2.18 Views	323
9.2.18.1 Views Overview	323
9.2.18.2 Elements	323
9.2.18.2.1 asElementTable	323
9.2.18.2.2 asInterconnectionDiagram	324

9.2.18.2.3 asTextualNotation	324
9.2.18.2.4 asTreeDiagram	324
9.2.18.2.5 GraphicalRendering	325
9.2.18.2.6 Rendering	325
9.2.18.2.7 renderings	325
9.2.18.2.8 TabularRendering	326
9.2.18.2.9 TextualRendering	326
9.2.18.2.10 View	326
9.2.18.2.11 ViewpointCheck	327
9.2.18.2.12 viewpointChecks	327
9.2.18.2.13 viewpointConformance	328
9.2.18.2.14 views	328
9.3 Metadata Domain Library	328
9.3.1 Metadata Domain Library Overview	328
9.3.2 Modeling Metadata	328
9.3.3 Risk Metadata	328
9.4 Analysis Domain Library	328
9.4.1 Analysis Domain Library Overview	328
9.4.2 Analysis Tooling	329
9.4.3 Sampled Functions	329
9.4.4 State Space Representation	329
9.4.4.1 State Space Representation Overview	329
9.4.4.2 Elements	329
9.4.5 Trade Studies	330
9.4.5.1 Trade Studies Overview	330
9.4.5.2 Elements	330
9.4.5.2.1 MaximizeObjective	330
9.4.5.2.2 MinimizeObjective	330
9.4.5.2.3 ObjectiveFunction	330
9.4.5.2.4 TradeStudy	331
9.4.5.2.5 TradeStudyObjective	331
9.5 Quantities and Units Domain Library	331
9.5.1 Quantities and Units Domain Library Overview	331
9.5.2 Quantities	332
9.5.2.1 Quantities Overview	332
9.5.2.2 Elements	333
9.5.2.2.1 scalarQuantities	333
9.5.2.2.2 ScalarQuantityValue	333
9.5.2.2.3 tensorQuantities	334
9.5.2.2.4 TensorQuantityValue	334
9.5.2.2.5 vectorQuantities	336
9.5.2.2.6 VectorQuantityValue	336
9.5.3 Units and Scales	337
9.5.3.1 Units and Scales Overview	337
9.5.3.2 Elements	337
9.5.3.2.1 ConversionByConvention	337
9.5.3.2.2 ConversionByPrefix	337
9.5.3.2.3 CoordinateTransformation	338
9.5.3.2.4 CyclicRatioScale	339
9.5.3.2.5 DerivedUnit	339
9.5.3.2.6 IntervalScale	340
9.5.3.2.7 LogarithmBaseKind	340
9.5.3.2.8 LogarithmScale	341
9.5.3.2.9 MeasurementScale	341

9.5.3.2.10 MeasurementUnit	342
9.5.3.2.11 OrdinalScale	342
9.5.3.2.12 ScalarMeasurementReference	343
9.5.3.2.13 ScaleValueDefinition	344
9.5.3.2.14 ScaleValueMapping	344
9.5.3.2.15 SimpleUnit	345
9.5.3.2.16 TensorMeasurementReference	345
9.5.3.2.17 UnitConversion	346
9.5.3.2.18 UnitPowerFactor	346
9.5.3.2.19 UnitPrefix	347
9.5.3.2.20 VectorMeasurementReference	347
9.5.4 ISQ	348
9.5.4.1 ISQ Overview.....	348
9.5.4.2 Elements.....	349
9.5.5 SI Prefixes	349
9.5.5.1 SI Prefixes Overview	349
9.5.5.2 Elements.....	350
9.5.6 SI	350
9.5.6.1 SI Overview.....	350
9.5.6.2 Elements.....	350
9.5.7 US Customary Units	350
9.5.7.1 US Customary Units Overview.....	350
9.5.7.2 Elements	351
9.5.8 Time	351
9.5.8.1 Time Overview.....	351
9.5.8.2 Elements	351
9.5.8.2.1 DateTime	351
9.5.8.2.2 TimeOfDay.....	351
A Annex: Conformance Test Suite	353
B Annex: Example Model	355
B.1 Introduction	355
B.2 Model Organization.....	355
B.3 Definitions	356
B.4 Parts	360
B.5 Parts Interconnection.....	362
B.6 Actions	365
B.7 States	368
B.8 Requirements.....	371
B.9 Analysis.....	372
B.10 Verification.....	373
B.11 View and Viewpoint.....	375
B.12 Variability.....	377
B.13 Individuals	378
C Annex: SysML v1 to SysML v2 Transformation.....	381
C.1 General	381
C.1.1 Overview	381
C.1.2 Mapping Approach.....	381
C.2 Mappings	382
C.2.1 Overview	382
C.2.2 Generic Mappings	382
C.2.2.1 Overview	382
C.2.2.2 Generic Mappings to KerML	382
C.2.2.2.1 Overview.....	382

C.2.2.2.2 Mapping Specifications	383
C.2.2.2.2.1 GenericToAnnotatingElement_Mapping.....	383
C.2.2.2.2.2 GenericToAnnotation_Mapping.....	383
C.2.2.2.2.3 GenericToAssociation_Mapping.....	384
C.2.2.2.2.4 GenericToBehavior_Mapping.....	385
C.2.2.2.2.5 GenericToClassifier_Mapping.....	386
C.2.2.2.2.6 GenericToConjugation_Mapping.....	386
C.2.2.2.2.7 GenericToConnector_Mapping	387
C.2.2.2.2.8 GenericToElement_Mapping.....	388
C.2.2.2.2.9 GenericToExpression_Mapping	388
C.2.2.2.2.10 GenericToFeature_Mapping	389
C.2.2.2.2.11 GenericToFeatureMembership_Mapping.....	390
C.2.2.2.2.12 GenericToFeatureTyping_Mapping	391
C.2.2.2.2.13 GenericToFeatureValue_Mapping	392
C.2.2.2.2.14 GenericToFunction_Mapping.....	393
C.2.2.2.2.15 GenericToGeneralization_Mapping	393
C.2.2.2.2.16 GenericToImport_Mapping	394
C.2.2.2.2.17 GenericToMembership_Mapping.....	395
C.2.2.2.2.18 GenericToNamespace_Mapping	396
C.2.2.2.2.19 GenericToPackage_Mapping.....	396
C.2.2.2.2.20 GenericToParameterMembership_Mapping	397
C.2.2.2.2.21 GenericToPredicate_Mapping	398
C.2.2.2.2.22 GenericToRelationship_Mapping	398
C.2.2.2.2.23 GenericToReturnParameterMembership_Mapping.....	399
C.2.2.2.2.24 GenericToStep_Mapping	400
C.2.2.2.2.25 GenericToType_Mapping.....	401
C.2.2.3 Generic Mappings to Systems	401
C.2.2.3.1 Overview.....	401
C.2.2.3.2 Mapping Specifications	402
C.2.2.3.2.1 GenericToConjugatedPortDefinition_Mapping	402
C.2.2.3.2.2 GenericToConjugatedPortTyping_Mapping	403
C.2.2.3.2.3 GenericToConstraintDefinition_Mapping	403
C.2.2.3.2.4 GenericToDefinition_Mapping	404
C.2.2.3.2.5 GenericToItemDefinition_Mapping	405
C.2.2.3.2.6 GenericToPortConjugation_Mapping	406
C.2.2.3.2.7 GenericToPortDefinition_Mapping	406
C.2.2.3.2.8 GenericToUsage_Mapping.....	407
C.2.3 SysML v1.6	408
C.2.3.1 Overview	408
C.2.3.2 Activities.....	408
C.2.3.2.1 Overview.....	408
C.2.3.2.2 Mapping Specifications	408
C.2.3.2.2.1 Continuous_Mapping	408
C.2.3.2.2.2 Discrete_Mapping	409
C.2.3.2.2.3 Optional_Mapping	411
C.2.3.2.2.4 Rate_Mapping	412
C.2.3.2.3 Allocations	413
C.2.3.3.1 Overview.....	413
C.2.3.3.2 Mapping Specifications	413
C.2.3.3.2.1 Allocate_Mapping	413
C.2.3.2.4 Blocks	414
C.2.3.4.1 Overview.....	414
C.2.3.4.2 Mapping Specifications	415
C.2.3.4.2.1 AdjunctProperty_Mapping	415

C.2.3.4.2.2 BindingConnector_Mapping	416
C.2.3.4.2.3 Block_Mapping	417
C.2.3.4.2.4 BoundReference_Mapping	418
C.2.3.4.2.5 ClassifierBehaviorProperty_Mapping	419
C.2.3.4.2.6 ConnectorProperty_Mapping.....	420
C.2.3.4.2.7 EndPathMultiplicity_Mapping	421
C.2.3.4.2.8 Part_Mapping.....	423
C.2.3.4.2.9 ValueType_Mapping	424
C.2.3.5 Model Elements	425
C.2.3.5.1 Overview.....	425
C.2.3.5.2 Mapping Specifications	425
C.2.3.5.2.1 ElementGroup_Mapping	425
C.2.3.5.2.2 Problem_Mapping	426
C.2.3.5.2.3 Rationale_Mapping.....	427
C.2.3.5.2.4 Stakeholder_Mapping.....	427
C.2.3.5.2.5 StakeholderToConcernMapping.....	428
C.2.3.6 PortsAndFlows	429
C.2.3.6.1 Overview.....	429
C.2.3.6.2 Mapping Specifications	430
C.2.3.6.2.1 FullPort_Mapping.....	430
C.2.3.6.2.2 InterfaceBlock_Mapping	431
C.2.3.6.2.3 ProxyPort_Mapping	432
C.2.3.7 Requirements	433
C.2.3.7.1 Overview.....	433
C.2.3.7.2 Mapping Specifications	433
C.2.3.7.2.1 Requirement_Mapping	434
C.2.4 UML4SysML	434
C.2.4.1 Overview	434
C.2.4.2 Actions.....	434
C.2.4.2.1 Overview.....	435
C.2.4.2.2 Mapping Specifications	438
C.2.4.2.2.1 Action_Mapping	438
C.2.4.2.2.2 InputPin_Mapping	439
C.2.4.2.2.3 OpaqueAction_Mapping.....	440
C.2.4.2.2.4 OutputPin_Mapping	441
C.2.4.2.2.5 Pin_Mapping	443
C.2.4.2.2.6 ReadSelfAction_Mapping	444
C.2.4.2.2.7 ReadStructuralFeatureAction_Mapping	445
C.2.4.2.2.8 ValuePin_Mapping	446
C.2.4.3 Activities.....	447
C.2.4.3.1 Overview.....	447
C.2.4.3.2 Mapping Specifications	449
C.2.4.3.2.1 Activity_Mapping	449
C.2.4.3.2.2 ActivityFinalNode_Mapping	450
C.2.4.3.2.3 ActivityParameterNode_Mapping	451
C.2.4.3.2.4 ControlFlow_Mapping	452
C.2.4.3.2.5 ControlNode_Mapping	453
C.2.4.3.2.6 DecisionNode_Mapping	454
C.2.4.3.2.7 FlowFinalNode_Mapping	455
C.2.4.3.2.8 ForkNode_Mapping	456
C.2.4.3.2.9 JoinNode_Mapping.....	457
C.2.4.3.2.10 MergeNode_Mapping	458
C.2.4.4 Classification	459
C.2.4.4.1 Overview.....	460

C.2.4.4.2 Mapping Specifications	461
C.2.4.4.2.1 Classifier_Mapping.....	461
C.2.4.4.2.2 LowerBoundTyping_Mapping	462
C.2.4.4.2.3 MultiplicityBound_Mapping	463
C.2.4.4.2.4 MultiplicityBoundOwnership_Mapping.....	463
C.2.4.4.2.5 MultiplicityBoundTyping_Mapping.....	464
C.2.4.4.2.6 MultiplicityElement_Mapping.....	465
C.2.4.4.2.7 MultiplicityLowerBound_Mapping.....	466
C.2.4.4.2.8 MultiplicityMembership_Mapping.....	467
C.2.4.4.2.9 MultiplicityLowerBoundOwnership_Mapping	468
C.2.4.4.2.10 MultiplicityUpperBound_Mapping	469
C.2.4.4.2.11 MultiplicityUpperBoundOwnership_Mapping.....	470
C.2.4.4.2.12 StructuralFeature_Mapping	472
C.2.4.4.2.13 TypedElementToFeatureTyping_Mapping	472
C.2.4.4.2.14 UpperBoundTyping_Mapping.....	473
C.2.4.5 CommonBehavior.....	474
C.2.4.5.1 Overview.....	474
C.2.4.5.2 Mapping Specifications	475
C.2.4.5.2.1 Behavior	475
C.2.4.6 CommonStructure.....	476
C.2.4.6.1 Overview.....	476
C.2.4.6.2 Mapping Specifications	477
C.2.4.6.2.1 Abstraction_Mapping	478
C.2.4.6.2.2 Comment_Mapping	478
C.2.4.6.2.3 CommentToAnnotation_Mapping	479
C.2.4.6.2.4 Dependency_Mapping	480
C.2.4.6.2.5 DirectRelationship_Mapping	481
C.2.4.6.2.6 ElementMain_Mapping	482
C.2.4.6.2.7 ElementOwnership_Mapping	482
C.2.4.6.2.8 ElementOwningMembership_Mapping	483
C.2.4.6.2.9 Namespace_Mapping	484
C.2.4.6.2.10 Relationship_Mapping	485
C.2.4.7 InformationFlows	485
C.2.4.7.1 Overview.....	486
C.2.4.7.2 Mapping Specifications	486
C.2.4.8 Interactions	486
C.2.4.8.1 Overview.....	486
C.2.4.8.2 Mapping Specifications	487
C.2.4.8.2.1 Interaction_Mapping	487
C.2.4.9 Packages	488
C.2.4.9.1 Overview.....	488
C.2.4.9.2 Mapping Specifications	489
C.2.4.9.2.1 ElementImport_Mapping	489
C.2.4.9.2.2 Package_Mapping	490
C.2.4.9.2.3 PackageImport_Mapping	491
C.2.4.10 SimpleClassifiers	492
C.2.4.10.1 Overview.....	492
C.2.4.10.2 Mapping Specifications	492
C.2.4.10.2.1 Attribute_Mapping	492
C.2.4.10.2.2 BehavioredClassifier_Mapping	493
C.2.4.10.2.3 BehavioredClassifierToFeatureMembership_Mapping	494
C.2.4.10.2.4 BehavioredClassifierToFeatureTyping_Mapping	495
C.2.4.10.2.5 BehavioredClassifierToPerformActionUsage_Mapping	496
C.2.4.10.2.6 DataType_Mapping	497

C.2.4.10.2.7 Enumeration_Mapping	498
C.2.4.10.2.8 EnumerationLiteral_Mapping	499
C.2.4.10.2.9 Interface_Mapping	500
C.2.4.10.2.10 InterfaceConjugatedPortDefinition_Mapping	501
C.2.4.10.2.11 InterfaceConjugatedPortDefinitionMembership_Mapping	502
C.2.4.10.2.12 InterfacePortConjugation_Mapping	503
C.2.4.10.2.13 InterfaceRealization_Mapping	504
C.2.4.10.2.14 Reception_Mapping	505
C.2.4.10.2.15 Signal_Mapping	506
C.2.4.11 StructuredClassifiers	507
C.2.4.11.1 Overview	507
C.2.4.11.2 Mapping Specifications	507
C.2.4.11.2.1 Association_Mapping	507
C.2.4.11.2.2 AssociationClass_Mapping	508
C.2.4.11.2.3 AssociationToMetadata_Mapping	509
C.2.4.11.2.4 Class_Mapping	510
C.2.4.11.2.5 ConnectorMapping	511
C.2.4.11.2.6 Port_Mapping	512
C.2.4.12 UseCases	513
C.2.4.12.1 Overview	514
C.2.4.12.2 Mapping Specifications	514
C.2.4.12.2.1 UseCase_Mapping	514
C.2.4.13 Values	515
C.2.4.13.1 Overview	515
C.2.4.13.2 Mapping Specifications	516
C.2.4.13.2.1 LiteralBoolean_Mapping	516
C.2.4.13.2.2 LiteralInteger_Mapping	517
C.2.4.13.2.3 LiteralNull_Mapping	518
C.2.4.13.2.4 LiteralReal_Mapping	519
C.2.4.13.2.5 LiteralString_Mapping	521
C.2.4.13.2.6 LiteralUnlimitedToUnbounded_Mapping	522
C.2.4.13.2.7 LiteralUnlimitedToInteger_Mapping	523
C.2.4.13.2.8 ValueSpecification_Mapping	524

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. Standard Language Names	82
7. Annotations Notation - Representative Usage	84
8. Packages Notation - Representative Usage	92
9. Dependencies Notation - Representative Usage	94
10. Definition and Usage Notation - Representative Usage	101
11. Attributes Notation - Representative Usage	106
12. Enumerations Notation - Representative Usage	108
13. Occurrences Notation - Representative Usage	112
14. Items Notation - Representative Usage	116
15. Parts Notation - Representative Usage	118
16. Ports Notation - Representative Usage	122
17. Connections Notation - Representative Usage	126
18. Interfaces Notation - Representative Usage	130
19. Allocations Notation - Representative Usage	133
20. Actions Notation - Representative Usage	138
21. States Notation - Representative Usage	145
22. Calculations Notation - Representative Usage	148
23. Constraints Notation - Representative Usage	151
24. Requirements Notation - Representative Usage	156
25. Analysis Cases Notation - Representative Usage	161
26. Verification Cases Notation - Representative Usage	163
27. Use Cases Notation - Representative Usage	166
28. Views and Viewpoints Notation - Representative Usage	169
29. EBNF Notation Conventions	174
30. Abstract Syntax Synthesis Notation	174
31. Grammar Production Definitions	175
32. Graphical BNF Conventions	205
33. List of all Overview Mapping Specifications	382
34. Table GenericToAnnotatingElement_Mapping Rules	383
35. Table GenericToAnnotation_Mapping Rules	384
36. Table GenericToAssociation_Mapping Rules	384
37. Table GenericToBehavior_Mapping Rules	385
38. Table GenericToClassifier_Mapping Rules	386
39. Table GenericToConjugation_Mapping Rules	387
40. Table GenericToConnector_Mapping Rules	387
41. Table GenericToElement_Mapping Rules	388
42. Table GenericToExpression_Mapping Rules	389
43. Table GenericToFeature_Mapping Rules	390
44. Table GenericToFeatureMembership_Mapping Rules	390
45. Table GenericToFeatureTyping_Mapping Rules	391
46. Table GenericToFeatureValue_Mapping Rules	392
47. Table GenericToFunction_Mapping Rules	393
48. Table GenericToGeneralization_Mapping Rules	394
49. Table GenericToImport_Mapping Rules	394
50. Table GenericToMembership_Mapping Rules	395
51. Table GenericToNamespace_Mapping Rules	396

52. Table GenericToPackage_Mapping Rules	396
53. Table GenericToParameterMembership_Mapping Rules	397
54. Table GenericToPredicate_Mapping Rules.....	398
55. Table GenericToRelationship_Mapping Rules.....	399
56. Table GenericToReturnParameterMembership_Mapping Rules	399
57. Table GenericToStep_Mapping Rules.....	400
58. Table GenericToType_Mapping Rules.....	401
59. List of all Overview Mapping Specfications	401
60. Table GenericToConjugatedPortDefinition_Mapping Rules	402
61. Table GenericToConjugatedPortTyping_Mapping Rules	403
62. Table GenericToConstraintDefinition_Mapping Rules.....	404
63. Table GenericToDefinition_Mapping Rules	404
64. Table GenericToItemDefinition_Mapping Rules	405
65. Table GenericToPortConjugation_Mapping Rules	406
66. Table GenericToPortDefinition_Mapping Rules.....	407
67. Table GenericToUsage_Mapping Rules.....	407
68. List of all Overview Mapping Specfications	408
69. Table Continuous_Mapping Rules	409
70. Table Discrete_Mapping Rules.....	410
71. Table Optional_Mapping Rules	411
72. Table Rate_Mapping Rules.....	412
73. List of all Overview Mapping Specfications	413
74. Table Allocate_Mapping Rules	413
75. List of all Overview Mapping Specfications	414
76. Table AdjunctProperty_Mapping Rules	415
77. Table BindingConnector_Mapping Rules	416
78. Table Block_Mapping Rules	418
79. Table BoundReference_Mapping Rules	419
80. Table ClassifierBehaviorProperty_Mapping Rules	420
81. Table ConnectorProperty_Mapping Rules	421
82. Table EndPathMultiplicity_Mapping Rules	422
83. Table Part_Mapping Rules	423
84. Table ValueType_Mapping Rules	424
85. List of all Overview Mapping Specfications	425
86. Table ElementGroup_Mapping Rules	426
87. Table Problem_Mapping Rules	426
88. Table Rationale_Mapping Rules.....	427
89. Table Stakeholder_Mapping Rules.....	428
90. Table StakeholderToConcernMapping Rules	428
91. List of all Overview Mapping Specfications	429
92. Table FullPort_Mapping Rules.....	430
93. Table InterfaceBlock_Mapping Rules	431
94. Table ProxyPort_Mapping Rules.....	432
95. List of all Overview Mapping Specfications	433
96. Table Requirement_Mapping Rules	434
97. List of all Overview Mapping Specfications	435
98. Table Action_Mapping Rules	438
99. Table InputPin_Mapping Rules	440
100. Table OpaqueAction_Mapping Rules.....	441
101. Table OutputPin_Mapping Rules	442
102. Table Pin_Mapping Rules.....	443
103. Table ReadSelfAction_Mapping Rules	444
104. Table ReadStructuralFeatureAction_Mapping Rules	445
105. Table ValuePin_Mapping Rules	447

106. List of all Overview Mapping Specifications	447
107. Table Activity_Mapping Rules.....	449
108. Table ActivityFinalNode_Mapping Rules.....	450
109. Table ActivityParameterNode_Mapping Rules.....	451
110. Table ControlFlow_Mapping Rules	452
111. Table ControlNode_Mapping Rules.....	453
112. Table DecisionNode_Mapping Rules	455
113. Table FlowFinalNode_Mapping Rules.....	456
114. Table ForkNode_Mapping Rules.....	456
115. Table JoinNode_Mapping Rules.....	458
116. Table MergeNode_Mapping Rules.....	459
117. List of all Overview Mapping Specifications	460
118. Table Classifier_Mapping Rules.....	461
119. Table LowerBoundTyping_Mapping Rules	462
120. Table MultiplicityBound_Mapping Rules	463
121. Table MultiplicityBoundOwnership_Mapping Rules.....	464
122. Table MultiplicityBoundTyping_Mapping Rules.....	465
123. Table MultiplicityElement_Mapping Rules	466
124. Table MultiplicityLowerBound_Mapping Rules.....	467
125. Table MultiplicityMembership_Mapping Rules.....	468
126. Table MultiplicityLowerBoundOwnership_Mapping Rules	469
127. Table MultiplicityUpperBound_Mapping Rules	470
128. Table MultiplicityUpperBoundOwnership_Mapping Rules.....	471
129. Table StructuralFeature_Mapping Rules	472
130. Table TypedElementToFeatureTyping_Mapping Rules	473
131. Table UpperBoundTyping_Mapping Rules.....	474
132. List of all Overview Mapping Specficiations	474
133. Table Behavior Rules.....	476
134. List of all Overview Mapping Specficiations	476
135. Table Abstraction Mapping Rules	478
136. Table Comment_Mapping Rules	479
137. Table CommentToAnnotation_Mapping Rules	480
138. Table Dependency_Mapping Rules	480
139. Table DirectRelationship_Mapping Rules.....	481
140. Table ElementMain_Mapping Rules	482
141. Table ElementOwnership_Mapping Rules	483
142. Table ElementOwningMembership_Mapping Rules	483
143. Table Namespace_Mapping Rules	484
144. Table Relationship_Mapping Rules.....	485
145. List of all Overview Mapping Specficiations	486
146. List of all Overview Mapping Specficiations	486
147. Table Interaction_Mapping Rules.....	487
148. List of all Overview Mapping Specficiations	488
149. Table ElementImport_Mapping Rules.....	489
150. Table Package_Mapping Rules.....	490
151. Table PackageImport_Mapping Rules	491
152. List of all Overview Mapping Specficiations	492
153. Table Attribute_Mapping Rules	493
154. Table BehavioredClassifier_Mapping Rules	494
155. Table BehavioredClassifierToFeatureMembership_Mapping Rules	495
156. Table BehavioredClassifierToFeatureTyping_Mapping Rules	496
157. Table BehavioredClassifierToPerformActionUsage_Mapping Rules	496
158. Table DataType_Mapping Rules	497
159. Table Enumeration_Mapping Rules	498

160. Table EnumerationLiteral_Mapping Rules.....	499
161. Table Interface_Mapping Rules.....	501
162. Table InterfaceConjugatedPortDefinition_Mapping Rules	502
163. Table InterfaceConjugatedPortDefinitionMembership_Mapping Rules	503
164. Table InterfacePortConjugation_Mapping Rules	503
165. Table InterfaceRealization_Mapping Rules	504
166. Table Reception_Mapping Rules.....	505
167. Table Signal_Mapping Rules.....	506
168. List of all Overview Mapping Specfications	507
169. Table Association_Mapping Rules	508
170. Table AssociationClass_Mapping Rules	509
171. Table AssociationToMetadata_Mapping Rules.....	510
172. Table Class_Mapping Rules	511
173. Table ConnectorMapping Rules	512
174. Table Port_Mapping Rules	513
175. List of all Overview Mapping Specfics	514
176. Table UseCase_Mapping Rules.....	514
177. List of all Overview Mapping Specfics	515
178. Table LiteralBoolean_Mapping Rules	517
179. Table LiteralInteger_Mapping Rules	518
180. Table LiteralNull_Mapping Rules	519
181. Table LiteralReal_Mapping Rules	520
182. Table LiteralString_Mapping Rules	521
183. Table LiteralUnlimitedToUnbounded_Mapping Rules.....	522
184. Table LiteralUnlimitedToInteger_Mapping Rules	523
185. Table ValueSpecification_Mapping Rules	524

List of Figures

1. SysML Language Architecture	70
2. Elements.....	77
3. Annotation.....	79
4. Comments	79
5. Textual Representation	80
6. Metadata Annotation.....	80
7. Namespaces.....	86
8. Packages.....	87
9. Dependencies	94
10. Definition and Usage	98
11. Multiplicities	99
12. Classifiers.....	99
13. Subsetting.....	100
14. Variant Membership	100
15. Attribute Definition and Usage	106
16. Enumeration Definition and Usage	108
17. Occurrence Definition and Usage	111
18. Event Occurrences	111
19. Item Definition and Usage	116
20. Part Definition and Usage	118
21. Port Definition and Usage.....	121
22. Port Conjugation	121
23. Connectors as Usages	125
24. Connection Definition and Usage	125
25. Flow Connections	126
26. Feature Values	126
27. Interface Definition and Usage	130
28. Allocation Definition and Usage	132
29. Action Definition and Usage	136
30. Control Nodes	136
31. Action Performance	137
32. Send and Accept Actions	137
33. State Definition and Usage	143
34. State Membership	143
35. State Exhibition.....	144
36. Transition Usage	144
37. Calculation Definition and Usage	148
38. Constraint Definition and Usage.....	150
39. Constraint Assertion.....	150
40. Requirement Definition and Usage	154
41. Requirement Satisfaction	155
42. Concern Definition and Usage.....	155
43. Requirement Constraint Membership	156
44. Requirement Parameter Memberships	156
45. Case Definition and Usage	158
46. Case Membership.....	159
47. Analysis Case Definition and Usage	160
48. Verification Case Definition and Usage	162
49. Verification Membership	163
50. Use Case Definition and Usage	165
51. Use Case Inclusion.....	165

52. View Definition and Usage.....	167
53. Viewpoint Definition and Usage	168
54. Rendering Definition and Usage.....	168
55. Expose Relationship.....	169
56. View Rendering Membership	169
57. Dependencies	215
58. Definition and Usage	216
59. Variant Membership	217
60. Attribute Definition and Usage.....	225
61. Enumeration Definition and Usage.....	227
62. Occurrence Definition and Usage	229
63. Event Occurrences	229
64. Item Definition and Usage.....	234
65. Part Definition and Usage.....	235
66. Port Definition and Usage.....	237
67. Port Conjugation	237
68. Connectors as Usages	241
69. Connection Definition and Usage	241
70. Flow Connections	242
71. Interface Definition and Usage	246
72. Allocation Definition and Usage	248
73. Action Definition and Usage	250
74. Control Nodes	250
75. Action Performance	251
76. Send and Accept Actions	251
77. State Definition and Usage	257
78. State Membership	257
79. State Exhibition.....	258
80. Transition Usage	258
81. Calculation Definition and Usage.....	265
82. Constraint Definition and Usage.....	266
83. Constraint Assertion.....	267
84. Requirement Definition and Usage.....	269
85. Requirement Satisfaction	269
86. Concern Definition and Usage	270
87. Requirement Constraint Membership	270
88. Requirement Parameter Memberships.....	271
89. Case Definition and Usage.....	279
90. Case Membership.....	279
91. Analysis Case Definition and Usage.....	282
92. Verification Case Definition and Usage	284
93. Verification Membership	285
94. Use Case Definition and Usage	287
95. Use Case Inclusion.....	287
96. View Definition and Usage.....	290
97. Viewpoint Definition and Usage	290
98. Rendering Definition and Usage	291
99. Expose Relationship.....	291
100. View Rendering Membership	292
101. State Space Representation action definitions	329
102. Model Organization for SimpleVehicleModel	355
103. Part Definition for Vehicle.....	356
104. Part Definition for FuelTank Referencing Fuel it Stores	358
105. Axle and its Subclass FrontAxe	358

106. Example Definition Elements	359
107. Part Usage for vehicle_b	360
108. Parts Tree for vehicle_b	361
109. Variant engine4Cyl	362
110. Parts Interconnection for vehicle_b	363
111. Action providePower	365
112. Action flow for providePower	366
113. Action flow for transportPassenger	367
114. Vehicle States.....	368
115. Requirement Definition MassRequirement	371
116. Requirements Group vehicleSpecification	372
117. Analysis Case fuelEconomyAnalysis	373
118. Vehicle Mass Verification Test	374
119. Vehicle Safety View	376
120. Rendering of view vehiclePartsTree_Safety	377
121. Variability Model for vehicleFamily	377
122. Vehicle Individuals and Snapshots	379

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 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.

0.2 Submission Submitters

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

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

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. This is now publicly available under an open source license at <https://github.com/Systems-Modeling>. The latest release is the 2021-06 version, based on the KerML and SysML metamodels baselined in June 2021 and used for this submission. However, since the conformance test suite has not been developed as of the time of this 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 2021-06 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 planning to continue the incremental development of the pilot implementation and complete it for the final submission.

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 built 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 this 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.

The SST developed an initial UML profile for a portion of SysML v2, and determined that the profile was difficult to use and implement. As a result, the SST has decided not to propose a more extensive UML profile for SysML v2.

0.4 Language Requirements Tables

0.4.1 Mandatory Language Requirements Table

Table 1. Mandatory Language Requirements Table

Reqt. ID	Reqt. 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.

Reqt. ID	Reqt. 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	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. Supporting Information: This is intended to be a specialization of Case.
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	Proposals for SysML v2 shall include the capability to specify an analysis model. 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.
ANL 1.11	Analysis Result	Proposals for SysML v2 shall include the capability to relate the results of executing analysis models to the analysis. 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.

Reqt. ID	Reqt. 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	Proposals for SysML v2 shall include a capability to represent a decision as one or more selections among alternatives. Supporting Information: This Decision and Rationale can be related through an Explanation relationship. The Rationale can refer to the supporting analysis.
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.

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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	<p>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).</p>
CNF 1.2	Profile Conformance	<p>Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 profile specification.</p>

Reqt. ID	Reqt. 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	Proposals for SysML v2 shall include a capability to represent a Model (aka system model) that contains a set of uniquely identifiable model elements. Supporting Information: This is intended to be a kind of Container or Namespace.
CRC 1.1.2	Model Library	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. Supporting Information: This is intended to be a kind of Container or Namespace.
CRC 1.1.3	Container	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. Supporting Information: This provides a way to organize the model. Containers can contain other containers.
CRC 1.2	Model Element Group	
CRC 1.2.2	Unique Identifier	Proposals for SysML v2 shall include a capability to represent a single universally unique identifier for each model element that cannot be changed. Supporting Information: The unique identifier should enable assignment of URIs.

Reqt. ID	Reqt. 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	Proposals for SysML v2 shall include a capability to represent one or more definitions and/or descriptions for each model element.
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	The requirements in this group include additional concepts that can be associated with any model element.

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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.

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. Name	Text
CRC 1.6	Metadata Group	The requirements in this group identify metadata as 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	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. Supporting Information: This can include markings such as ITAR, proprietary or security classifications
INF 1	Interface Requirements Group	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. Supporting Information: An interface should support the following: <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 A Port is also used to refer to an Interface End.

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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	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.
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>

Reqt. ID	Reqt. 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. 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	

Reqt. ID	Reqt. 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	Proposals for SysML v2 shall be specified using a metamodel that includes abstract syntax, concrete syntax, semantics, and the relationships between them.
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	

Reqt. ID	Reqt. 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.

Reqt. ID	Reqt. 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> <ul 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 it 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.

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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.

Reqt. ID	Reqt. 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	<html> <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.

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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	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. 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.
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	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. Supporting Information: This is intended to be a specialization of the Conform Relationship.
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	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. Supporting Information: This relationship groups requirements into a shared context.

Reqt. ID	Reqt. 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, whereas 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>

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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 effectiveness 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	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.
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	The verification case shall include verification objectives to be implemented by the verification activities.

Reqt. ID	Reqt. 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

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

Reqt. ID	Reqt. 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).</p> <p>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>

Reqt. ID	Reqt. 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	

Reqt. ID	Reqt. 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.

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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>

Reqt. ID	Reqt. 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	<pre data-bbox="458 1474 556 1510"><html></pre> <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>

Reqt. ID	Reqt. 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	

Reqt. ID	Reqt. 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	

Reqt. ID	Reqt. 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	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).

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 speicfy analysis model
ANL 1.03	Parameters of Interest	No	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	Yes		
ANL 1.11	Analysis Result	Partial	AnalysisCases	
ANL 1.13	Analysis Metadata	Yes		
ANL 1.14	Decision Group			
ANL 1.14.2	Alternative	Yes	TradeStudies	
ANL 1.14.4	Decision	Yes	TradeStudies	
ANL 1.14.5	Criteria	Yes	TradeStudies	
ANL 1.14.6	Rationale	Yes		
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	Yes	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	Partial	Behaviors	
BHV 1.06	Events	No	Actions AcceptActionUsage States TransitionUsage	Signal events are supported, but not time or change events.
BHV 1.07	Control Nodes	No	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	Yes	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	Partial		
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	Not Planned		
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	Yes	Namespaces	Can use package, but no explicitly identified model construct
CRC 1.1.2	Model Library	Yes	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	Partial		
CRC 1.2.7	Additional Cross-Cutting Concepts Group			
CRC 1.2.7.1	Problem	Yes		
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	No		
CRC 1.3.03	Dependency Relationship	Yes	Dependencies	
CRC 1.3.04	Cause-Effect Relationship	No		
CRC 1.3.05	Explanation Relationship	Partial		
CRC 1.3.06	Conform Relationship	Not Planned		
CRC 1.3.07	Refine Relationship	No		
CRC 1.3.08	Allocation Relationship	Yes	Allocations	

ID	Name	Satisfied?	Satisfied-by	Comment
CRC 1.3.09	Element Group Relationship	Partial		
CRC 1.3.10	Navigation Relationship	Not Planned		This is addressed by API external 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	Not Planned		
CRC 1.5	View and Viewpoint Group			
CRC 1.5.1	View Definition	Partial	Views	
CRC 1.5.2	Viewpoint	Partial	Views	
CRC 1.6	Metadata Group		Annotations	
CRC 1.6.1	Version	Not Planned	API_Model	Refer to API versioning service
CRC 1.6.2	Time Stamp	Not Planned	API_Model	Refer to API versioning service
CRC 1.6.3	Data Protection Controls	Partial		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	Yes	Items	need support for update rate
INF 1.07	Interface Agreement Group			
INF 1.07.1	Item Exchange Constraints	Yes		This will be addressed by conartaining transfers.
INF 1.07.2	Property Constraints	Partial		
INF 1.08	Interface Medium	Not Planned		
INF 1.09	Interface Layers	Yes		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	Yes	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	Partial		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	Partial		
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 Planned		
PRP 1	Properties, Values and Expressions Requirements Group			
PRP 1.01	Unified Representation of Values	Partial	Attributes Expressions	need to identify 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	Partial		
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	Yes		

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	Partial		
RQT 1.2.4	Requirement Usage Ordering	Partial		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	Partial	Requirements	
RQT 1.3.3	Requirement Verification	Yes	VerificationCases	
RQT 1.3.4	Requirement Derivation	No		
RQT 1.3.5	Requirement Group Relationship	Partial		
RQT 1.3.6	Relationships to a Requirement Group	Partial		
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	Occurrences	
STC 1.11	Usage Specific Localized Type	Yes	General	
VRF 1	Verification and Validation Requirements Group			
VRF 1.1	Verification Context	Yes		
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	Partial		
VRF 1.2.4	Verification Methods	Yes		
VRF 1.3	Verification System	Yes		
VRF 1.4	Verification Relationships Group			
VRF 1.4.1	Verification Objectives to Verification Cases	Yes	VerificationCases	
VRF 1.4.2	Validate Relationship	Not Planned		

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	Yes		
ANL 1.14.3	Decision Expression	Yes		
BHV 1	Behavior Requirements Group			
BHV 1.03	Function-based Behavior Group			
BHV 1.03.2	Composite Input and Output	Partial		
BHV 1.03.5	Behavior Library	No		
BHV 1.09	State History	Yes	Occurrences	
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	Yes		
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	Need if, then else
PRP 1.09	Automated Quantity Value Conversion	Yes	Quantities and Units	
PRP 1.13	Discretely Sampled Function Value Type	Yes		
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	Partial		
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	Partial		
STC 1	Structure Requirements Group			

ID	Name	Satisfied?	Satisfied-by	Comment
STC 1.09	Structure Resolved to a Single Variant	Yes		
VRF 1	Verification and Validation Requirements Group			
VRF 1.2	Verification Case Group			
VRF 1.2.5	Verification Activity	Yes		
VRF 1.2.6	Verification Evaluation Activity	Yes		

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.

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, and behavior; the constraints on its system properties to support engineering analysis; and requirements verification. 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.

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 8](#). 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 [8.2](#), including normative constraints in the Notation subclauses of [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 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 [8.3](#).)
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 [8.2](#) and the Notation subclauses of [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 [8.4](#) and [9.2](#). 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 of the domain model libraries specified in [Clause 9](#).
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 a valid model interchange format 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 [9.5](#)):

[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 Document Overview

The Systems Modeling Language (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. This document provides the standard specification for Version 2 of SysML. SysML Version 2 is intended to enhance the precision, expressiveness, interoperability, and the consistency and integration of the language relative to SysML Versions 1.0 to 1.7 [SysMLv1].

SysML v1 was specified as a profile of the Unified Modeling Language v2 [UML]. SysML v2, on the other hand, is specified as a metamodel that extends a formal kernel metamodel [KerML], which is greatly simplified compared to the UML metamodel. In order to facilitate the transition from SysML v1 to SysML v2, this standard also specifies a formal transformation from UML models using the SysML v1.7 profile to models using the SysML v2 metamodel (see [Annex C](#)).

SysML v2 provides a complete textual notation (see [8.2.2](#)) in addition to a graphical notation (see [8.2.3](#)). These notations provide the concrete syntax representation of the SysML abstract syntax (see [8.3](#)), which extends the Kernel abstract syntax, providing specialized constructs for modeling systems (as shown in [Fig. 1](#)). Further, the SysML Systems Model Library (see [9.2](#)) extends the Kernel Model Library to provide the semantic specification for SysML (see [8.4](#); see also [KerML, 7.1] on the use of model libraries for semantic specification). Finally, SysML provides an additional set of Domain Libraries (see [9.4](#) and following) to provide a rich set of reference models in various domains important to systems modeling (such as Analysis and Quantities and Units).

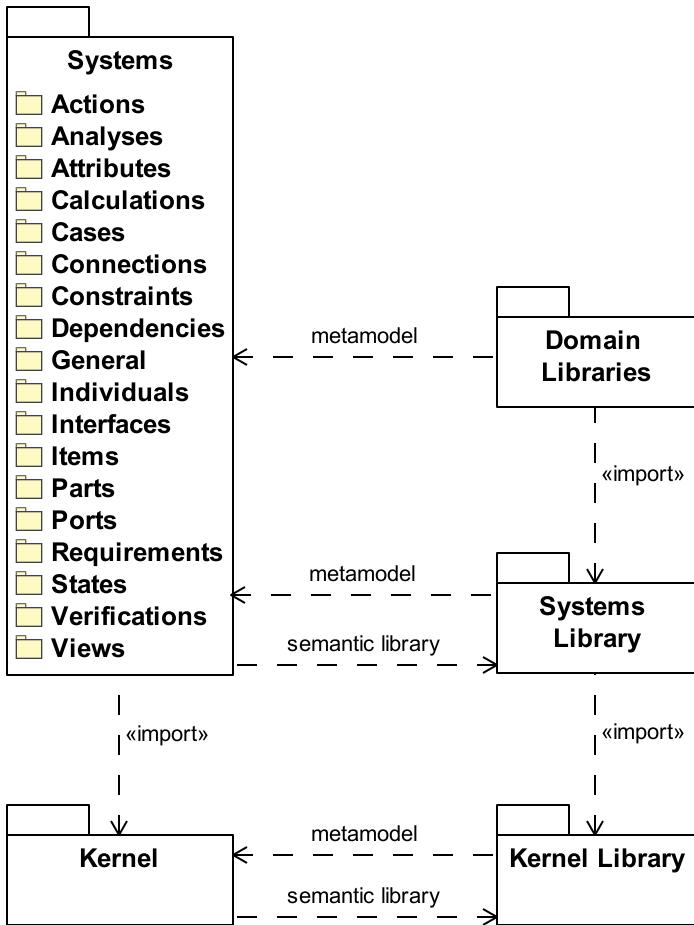


Figure 1. SysML Language Architecture

6.2 Document Organization

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

- [Clause 7](#) describes SysML from a user point of view. Its subclauses describe all of the modeling constructs in SysML, including for each a general overview, related abstract syntax diagrams and a description of the textual and graphical notation. The overviews in this clause should be considered informative. The abstract syntax and notation subclauses, however, are normative, including descriptions of the processing of the textual notation and its relationship to the graphical notation and the abstract syntax.
- [Clause 8](#) provides the normative specification of the metamodel that defines the SysML language. This includes the concrete syntax (textual and graphical notations), the abstract syntax and the semantics for the language. The SysML abstract syntax and semantics are formally extensions of the Kernel abstract syntax and semantics provided by KerML (as discussed in [6.1](#)). However, this clause does not cover details of the Kernel metamodel, which are included by normative reference to the KerML specification [KerML].
- [Clause 9](#) specifies a set of model libraries defined in SysML itself. The Systems Library extends the Kernel Library from [KerML] in order to provide systems-modeling-specific semantics to SysML language constructs. The Domain Libraries 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 that covers 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 an example model using the SysML language as defined in this specification to illustrate how the language features can be used to model a system.
- [Annex C](#) defines 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.

6.3 Document Conventions

The following stylistic conventions are applied in the Notation subclauses in [Clause 7](#) (Language Description), in the abstract syntax, concrete syntax and semantics descriptions of [Clause 8](#) (Metamodel) and in the element descriptions of [Clause 9](#) (Model Library) when model elements are referenced by name in body text paragraphs. However, they are *not* used in Overview subclauses of [Clause 7](#), 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 appear exactly as in the abstract syntax, including capitalization. When used as English common nouns, e.g., "PartDefinition, "ActionUsage", they refer to instances of the metaclass (in models), e.g., "A PartUsage must be defined by a PartDefinition" refers to instances of the metaclasses PartUsage and PartDefinition that reside in models. This can be modified using the term "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 "code" font. When used as English common nouns, e.g., "an ownedPart of a Definition", "multiple ownedParts of a Definition", they refer to values of the properties. This can be modified using the term "metaproPERTY" as necessary to refer to the metaproPERTY itself instead of its values, e.g., "The ownedParts metaproPERTY is contained in the General package."
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, and always in a "code" font. 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 Notation 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, without being italic or bold, but still in "code" font.
4. Longer samples of textual notation are written in separate paragraphs, indented relative to body paragraphs.

The grammar of the textual notation and its mapping to the abstract syntax is specified using a specialized *extended Backus-Naur form* (EBNF) notation, with conventions described in [8.2.2.1.2](#). For the graphical notation, this EBNF notation is further extended to allow the use of graphical symbols within productions (see [8.2.3.1](#)).

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 this submission of the proposed specification, or changes to the specification that are expected before the final

submission of the proposal. Such notes will be removed in the final submission as the issues they address are resolved.

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

6.4 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
- Thematix

However, work on the specification was also supported by over 170 people in over 70 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.openmbee.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 Language Description

7.1 Language Overview

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

SysML language extends the Kernel Modeling Language (KerML) as specified in the KerML specification [KerML]. SysML directly uses some elements of KerML, but most SysML elements are specializations of KerML elements. This clause describes all these language concepts in their context of use in SysML.

SysML directly uses the following concepts from KerML:

- *Elements* and *relationships* that define the basic graph structure of a model (see [7.2](#)).
- *Annotations* for attaching metadata to a model, including comments, textual representations and modeler-defined metadata features (see [7.3](#)).
- *Namespaces* that contain and name elements, and, particularly, *packages* used to organize the elements in a model (see [7.4](#)).
- *Specialization* of elements that specify types, including subclassification, subsetting, redefinition and feature typing (see [7.6](#)).
- *Expressions* can be used to specify calculations, case results, constraints and formal requirements. The full KerML expression sub-language is available in SysML, as described in the KerML specification. The description of this sub-language is not repeated in the SysML specification document.

The modeling constructs specific to SysML, as specified in subclauses [7.5](#) through [7.25](#), 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 [7.5](#)).
 - The general pattern of *definition* and *usage*, which is applied to many of the SysML language constructs (see [7.6](#)). The pattern of definition and usage elements facilitates model reuse, such that a concept can be defined once and then used in many different contexts. A usage element can also be further specialized for its specific context.
 - 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 [7.6](#)).
- The modeling of attributive information about things, including:
 - *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 [7.7](#)).
 - *Enumerations* that are attributes restricted to a specified set of enumerated values (see [7.8](#)).
- The modeling of *occurrences* with identity, which can be represented at specific points in time, over a duration in time, or over an entire lifetime, including modeling of *individuals* with specific identities (see [7.9](#)).
- The modeling of *structure* to represent how parts are decomposed, interconnected and classified, and includes:
 - *Items* that may flow through a process or system or be stored by a system (see [7.10](#)).
 - *Parts* that are the foundational units of structure, which can be composed and interconnected, forming composite parts and entire systems (see [7.11](#)).
 - *Ports* that define connection points on parts that enable interactions between parts (see [7.12](#)).

- *Connections* (see [7.13](#)) and *interfaces* (see [7.14](#)) that define how parts are interconnected.
- *Allocations* that assign responsibility for realizing the features of one element by another element (see [7.15](#)).
- 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 [7.16](#)).
 - *States* exhibited by a part, the allowable *transitions* between those states, and the actions enabled in a state or during a transition (see [7.17](#)).
- The modeling of *calculations* that are parameterized expressions that can be evaluated to produce specific results (see [7.18](#)).
- 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 or false (see [7.19](#)).
- 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 [7.20](#)).
- The modeling of *cases*, which define the steps required to produce a desired result relative to a *subject* (see [7.21](#)), to achieve a specific *objective*, including:
 - *Analysis cases*, whose steps are the *actions* necessary to analyze a subject (see [7.22](#)).
 - *Verification cases*, whose objective is to verify how a requirement is satisfied by the subject (see [7.23](#)).
 - *Use cases*, that specify required behavior of the subject with the objective of providing a measurable benefit to one or more external *actors* (see [7.24](#)).
- 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 [7.25](#)).

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 [7.26](#).)

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 simply as a part (see [7.11](#)). The language provides straightforward extension mechanisms to specify terminology that is appropriate for the domain of interest.

7.2 Elements and Relationships

7.2.1 Overview

A model in SysML is represented as a collection of *elements*, some of which are *relationships* that relate other elements. Every element has a unique identifier, and an element can have a name and any number of aliases (see [7.4](#)).

A relationship is a kind of element that relates two or more other elements. Some relationships are constrained to have exactly two related elements (i.e., *binary* relationships) while others may have more. The related elements of relationships are ordered. A relationship may designate certain of its related elements as *sources* with the rest being *targets*. In this case, the relationship is said to be *directed* from the sources to the targets. An *undirected* relationship simply designates all its related elements to be targets, with no source elements.

7.2.2 Abstract Syntax

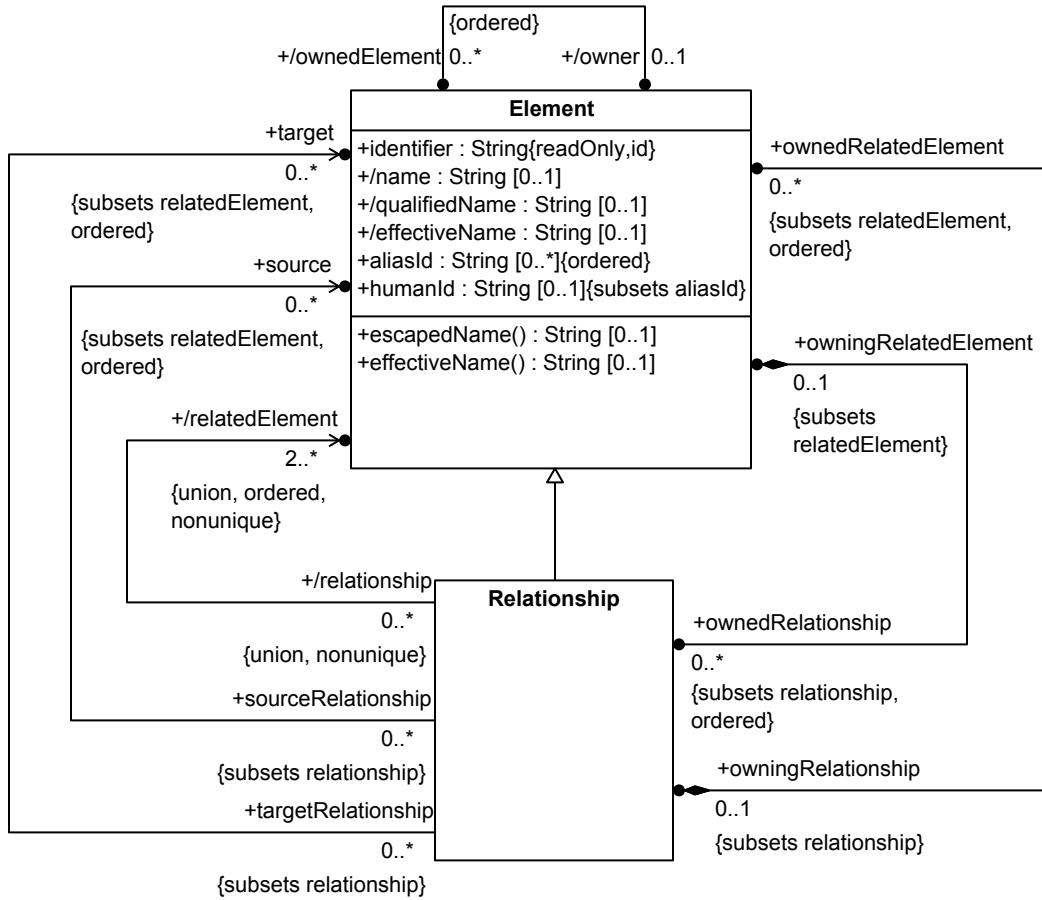


Figure 2. Elements

7.2.3 Notation

While Element and Relationship metaclasses are not abstract in the KerML abstract syntax metamodel, they shall not be instantiated in any SysML model. Notations for the various specific kinds of model elements in SysML are described in subsequent subclauses. However, there are certain notations for identification that apply to all model elements.

Every Element has an **identifier** that shall be a Universally Unique Identifier (UUID) (as specified in [UUID]). Generally, the properties of an Element can change over its lifetime, but the **identifier** shall not change after the Element is created. An Element may also have additional identifiers, its **aliasIds**, which may be assigned for tool-specific purposes.

While a tool may display the **identifier** and any **aliasIds**, these should not be entered by a modeler but, rather, managed by the underlying modeling tooling. However, the modeler-entered declaration of an Element may specify a **humanId** for it, as a lexical name preceded by the keyword **id**.

If the Element is an **ownedMember** of a Namespace, 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 Namespace (see [7.4](#)).

```
part id '1.2.4' MyName;
```

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.

Graphically, non-Relationship Elements are generally represented using a box-like shape or other icon, while Relationships are shown using lines connecting the symbols for the `relatedElements`. However, in some cases additional shapes may be attached to Relationship lines in order to present additional information. The specific conventions for such graphical notations are covered in subsequent subclauses.

7.3 Annotations

7.3.1 Overview

An *annotating element* is an element that is used to provide additional information about other elements. An annotation is a relationship between an annotating element and an element that is being described. An annotating element can annotate multiple annotated elements, and each element can have multiple annotations.

A *comment* is one kind of annotating element that is used to provide textual descriptions about other elements. Comments can be members of namespaces and, therefore, can be named. Such member comments may be about the namespace that owns them, or they may be about different elements.

Documentation is a special kind of annotation relationship that identifies a distinguished comment used to document the annotated element. Documentation comments are always owned by the documented element via the documentation relationship. Such comments are never namespace members, so they cannot be named (though they can be given an identifier).

A *textual representation* is an annotating element whose textual body provides a representation of the annotated element in a specifically named language. This representation may be in the SysML textual notation or it may be in another language. 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 *annotating feature* is a kind of annotating element 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 of 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 an attribute usage that is defined by a single attribute definition (see [7.7](#)). If the definition has no nested features itself, then the annotating feature simply acts as a user-defined syntactic tag on the annotated element. If the definition does have features, then the annotating feature must provide value bindings for all of them, specifying attributive metadata for the annotated element.

7.3.2 Abstract Syntax

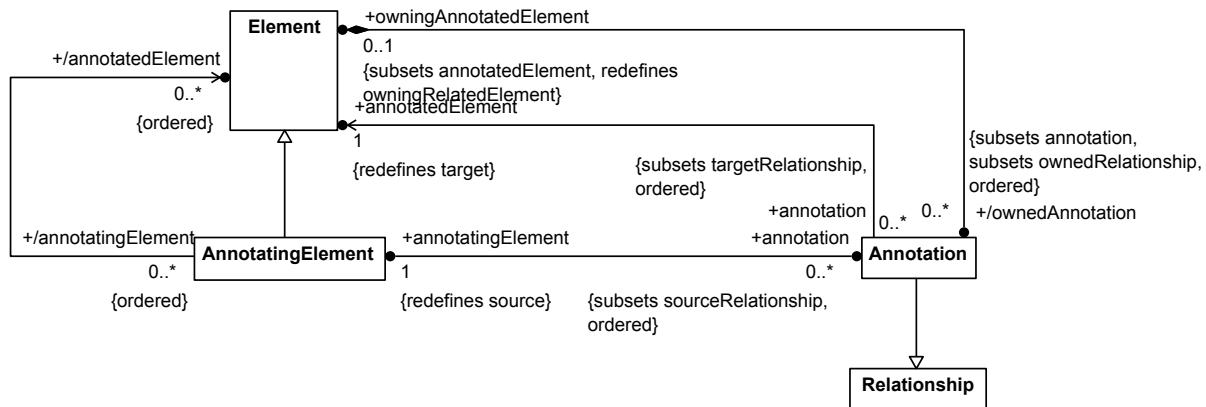


Figure 3. Annotation

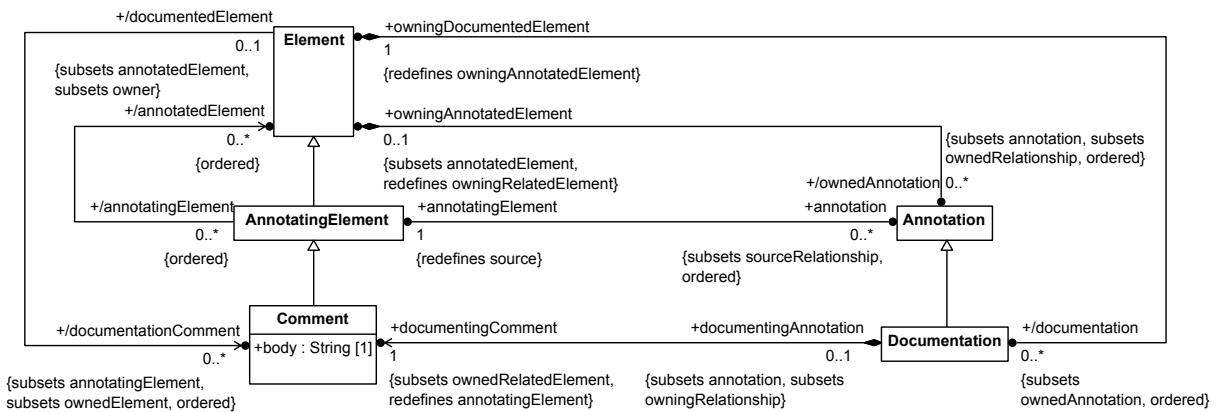


Figure 4. Comments

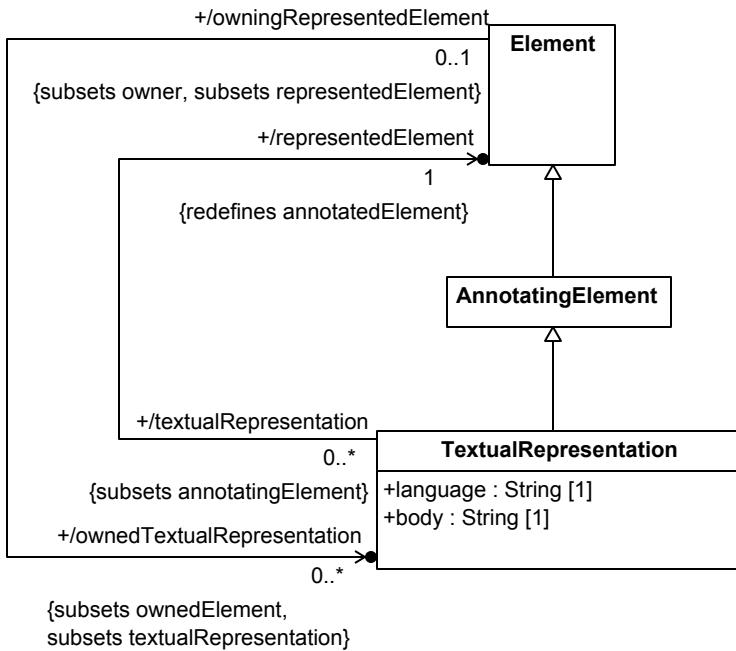


Figure 5. Textual Representation

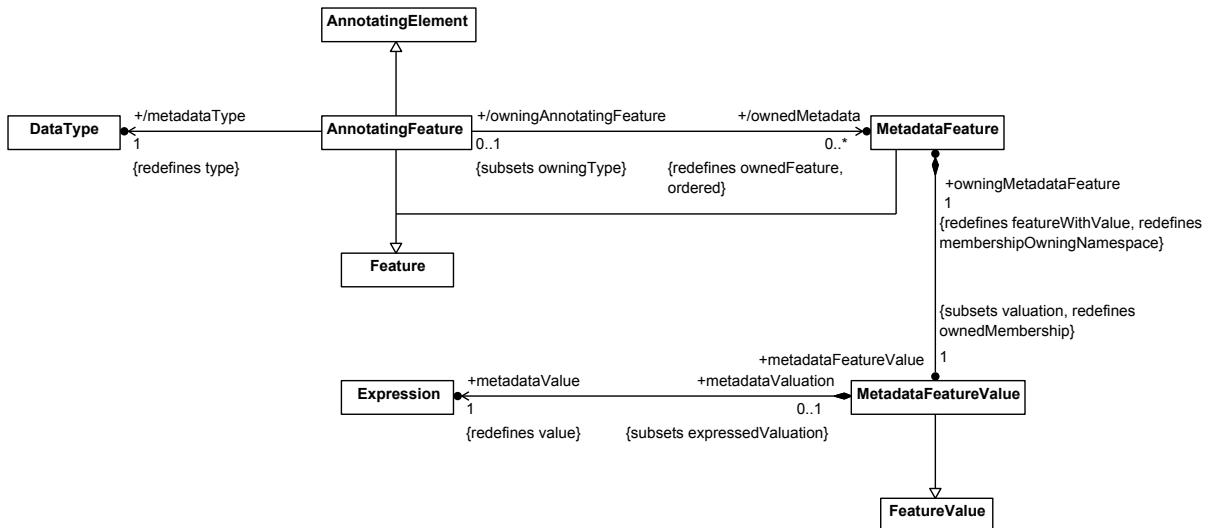


Figure 6. Metadata Annotation

7.3.3 Notation

Comments

The full declaration of a Comment begins with the keyword **comment**, optionally followed by a **humanId** and/or **name** (see [7.2](#)). One or more qualified names of **annotatedElements** for the Comment, separated by commas, are then given after the keyword **about**, indicating that the Comment has Annotation Relationships to each of the identified Elements. The body of the Comment is written lexically as regular comment text between **/*** and ***/** delimiters.

```

item A;
part B;
comment Comment1 about A, B
    /* This is the comment body text. */

```

If the Comment is an `ownedMember` of a Namespace (see [7.4](#)), then the explicit identification of `annotatedElements` can be omitted, in which case the `annotatedElement` shall be implicitly the containing Namespace. Further, in this case, if no `humanId` or name is given for the Comment, then the `comment` keyword can also be omitted.

```

package P {
    comment C /* This is a comment about P. */

    /* This is also a comment about P. */
}

```

When a Comment is written in the textual notation, the actual `body` text of the Comment shall be extracted from the lexical comment body according to the rules given in the KerML specification [KerML, 7.2.3.2.1]. The body text of a Comment can include markup information (such as HTML), and a conforming tool may display such text as rendered according to the markup. However, marked up "rich text" for a Comment written using the textual notation shall be stored in the Comment body in plain text including all mark up text, with all line terminators and white space included as entered, other than what is removed according to the rules above.

Documentation

A documentation Comment is notated similarly to a regular Comment, but using the keyword `doc` rather than `comment`. Since a documentation Comment is always an `ownedElement` of its `annotatedElement`, the notation of a documentation Comment is always nested within the notation of its owning Element, so there is no need to explicitly identify the `annotatedElement`. Further, since a documentation Comment is always owned via its Documentation Relationship, it cannot be an `ownedMember` of a Namespace and therefore cannot have a name specified for it. However, it can optionally be given a `humanId` (or separately given an alias via a non-owning Membership Relationship—see [7.4](#)).

```

part X {
    doc id X_Comment
        /* This is a documentation comment about X. */
    doc /* This is more documentation about X. */
}

```

If the Element being documented is the member of a Namespace, then a special notation can be used in which lexical documentation comment text is placed immediately *before* the notation of the Element being documented. In this case, the initial `/*` delimiter of the comment is replaced with `/**`. If no `humanId` is specified for a documentation Comment of this form, then the keyword `doc` can also be omitted.

```

package P {

    /** This is a documentation comment about Q. */
    package Q;

}

```

Documentation comment text shall also be processed according to the rules given in the KerML specification [KerML, 7.2.3.2.2].

Textual Representation

A TextualRepresentation is notated similarly to a regular Comment, but with the keyword **rep** used instead of **comment**. Similarly to a Comment, the **about** keyword can be used to specify the **representedElement** for the TextualRepresentation. However, only one **representedElement** may be identified for a TextualRepresentation. If the TextualRepresentation is an **ownedMember** of a Namespace (see [7.4](#)), then, if the **representedElement** is not identified explicitly, it shall by default be the containing Namespace. A TextualRepresentation declaration must also specify the **language** as a literal string following the keyword **language**. If the TextualRepresentation has no **humanId**, name or explicit **representedElement**, then the **rep** keyword can also be omitted.

```
part def P {
    attribute x: Real;
    assert x_constraint;
    rep inOCL about x_constraint language "ocl"
        /* self.x > 0.0 */
}
action def setX(c : C, newX : Real) {
    language "alf"
    /* c.x = newX;
     * WriteLine("Set new x");
     */
}
```

The lexical comment text given for a TextualRepresentation shall be processed as for regular comment text, and it is the result after such processing that is the TextualRepresentation **body** expected to conform to the named language.

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

The interpretation of the named language string in a TextualRepresentation shall be case insensitive. If the named language string matches one of the language names shown in [Table 6](#) (without regard to case), then the body text shall be syntactically legal according to the specification shown in the table (this is the same set of standard language names as in [KerML], with the addition of "sysml"). Other specifications may define specific language strings, other than those shown in [Table 6](#), to be used to indicate the use of languages from those specifications in SysML TextualRepresentations.

If the **language** of a TextualRepresentation is "sysml", then the **body** text shall be a legal representation of the **representedElement** in the SysML textual notation as defined in this specification. A conforming tool can use such a TextualRepresentation Annotation to record the original SysML textual notation text from which an Element was parsed. In this case, it is a tool responsibility to ensure that the **body** of the TextualRepresentation remains correct (or the Annotation is removed) if the annotated Element changes other than by re-parsing the **body** text.

Table 6. Standard Language Names

Language Name	Specification
sysml	Systems Modeling Language (this specification)
kerml	Kernel Modeling Language [KerML]
ocl	Object Constraint Language [OCL]
alf	Action Language for fUML [Alf]

Metadata

An AnnotatingFeature is declared using the keyword **metadata** (or the symbol @), optionally followed by a nameId and/or name, followed by the keyword **defined by** (or the symbol :) and the qualified name of exactly one KerML DataType (see [KerML]) or SysML AttributeDefinition (see [7.7](#)). If no nameId or name is given, then the keyword **defined by** (or the symbol :) may also be omitted. One or more annotatedElements are then identified for the AnnotatingFeature after the keyword **about**, indicating that the AnnotatingFeature has Annotation Relationships to each of the identified Elements.

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

If the specified DataType or AttributeDefinition has features, then a body must be given for the AnnotatingFeature that declares MetadataFeatures that redefine each of the features of the DataType or AttributeDefinition and binds them to the result of model-level evaluable Expressions (see [KerML, 7.4.8]). The MetadataFeatures of an AnnotatingFeature must always have the same names as the names of the features of the metadata definition, so the shorthand prefix redefines notation (see [7.6](#)) is always used.

```
attribute def ApprovalAnnotation {  
    attribute approved : Boolean;  
    attribute approver : String;  
}  
  
metadata ApprovalAnnotation about Design {  
    attribute redefines approved = true;  
    attribute redefines approver = "John Smith";  
}
```

The keyword **attribute** and/or **redefines** (or the equivalent symbol :>>) may be omitted in the declaration of a MetadataFeature.

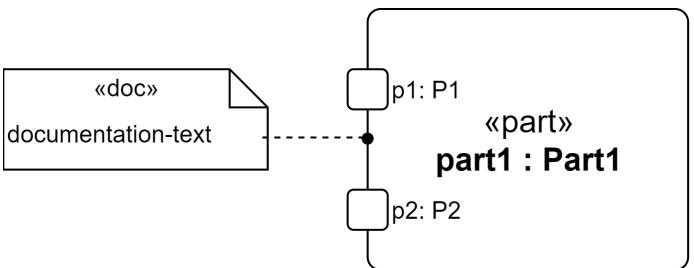
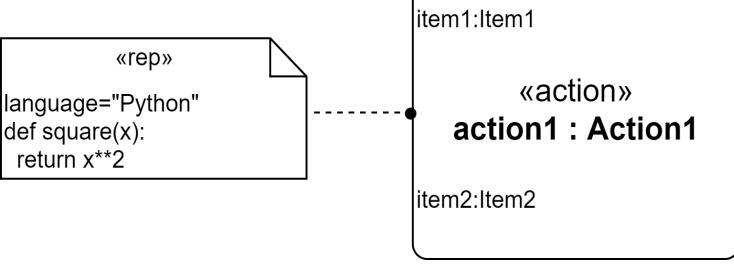
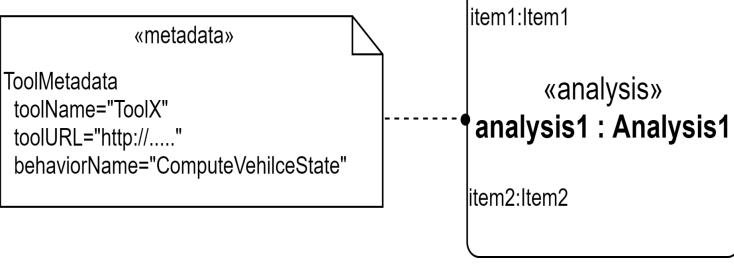
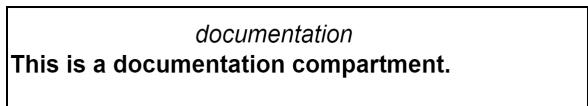
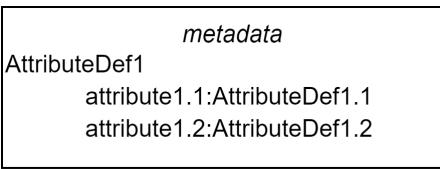
```
metadata ApprovalAnnotation about Design {  
    approved = true;  
    approver = "John Smith";  
}
```

If the AnnotatingFeature is an ownedMember of a Namespace (see [7.4](#)), then the explicit identification of annotatedElements can be omitted, in which case the annotatedElement shall be implicitly the containing Namespace.

```
part def Design {  
    // This AnnotatingFeature is implicitly about the part def Design.  
    @ApprovalAnnotation {  
        approved = true;  
        approver = "John Smith";  
    }  
}
```

Table 7. Annotations Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Comment		<code>/* This is a comment */</code>
Comment		<code>comment Comment1 /* This is a comment. */</code>
Documentation		<code>doc /* This is documentation. */</code>
Documentation		<code>doc /*This is documentation.*/</code>
Textual Representation		
Metadata		<code>metadata AttributeDef1{ attribute1=value1; attribute2="value2"; attribute3="http://...." } or @ AttributeDef1{ attribute1=value1; ... }</code>
Annotation		<code>Comment1 about attribute1 /* The annotated element * is attribute1. */</code>

Element	Graphical Notation	Textual Notation
Annotation-Documentation	 <p>A diagram showing a documentation-text compartment (represented by a rectangle with a triangular tail) connected to a part compartment (represented by a rounded rectangle) via a dashed line with a dot. The part compartment contains a membership relationship labeled 'p1: P1' and another unlabeled relationship 'p2: P2'.</p>	
Annotation-Textual Representation	 <p>A diagram showing a textual representation compartment (rectangle with tail) connected to an action compartment (rounded rectangle) via a dashed line with a dot. The action compartment contains two items: 'item1:item1' and 'item2:item2'. The textual representation contains Python code: 'language="Python"', 'def square(x):', and 'return x**2'.</p>	
Annotation-Metadata	 <p>A diagram showing a metadata compartment (rectangle with tail) connected to an analysis compartment (rounded rectangle) via a dashed line with a dot. The analysis compartment contains two items: 'item1:item1' and 'item2:item2'. The metadata compartment contains 'ToolMetadata', 'toolName="ToolX"', 'toolURL="http://....."', and 'behaviorName="ComputeVehicleState"'.</p>	
Documentation	 <p>A diagram showing a documentation compartment (rectangle) containing the text: 'documentation' and 'This is a documentation compartment.'</p>	
Metadata Compartment	 <p>A diagram showing a metadata compartment (rectangle) containing 'AttributeDef1', 'attribute1.1:AttributeDef1.1', and 'attribute1.2:AttributeDef1.2'.</p>	

7.4 Namespaces and Packages

7.4.1 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. A membership relationship can specify the name by which its member element is known relative to the containing namespace and whether the element membership is visible outside the namespace or not.

An element may be *owned* via its membership in a namespace. When a namespace is deleted, all such owned members shall also be deleted. An element may also have a membership in a namespace without being owned by the namespace. In this case, the membership may introduce an *alias* name for the element relative to the namespace. Note that it is possible for an element to have both owning and non-owning memberships with the same namespace, but it shall have at most one owning membership across all namespaces.

An *import* relationship allows one namespace to import memberships from another namespace. The member elements from imported memberships become (unowned) members of the importing namespace in addition to being members of the imported namespace. In particular, this allows members of the imported namespace to be referenced in textual notations within the scope of the importing namespace without having to qualify the member names with the name of the imported namespace. An import can also be *recursive*, which means that, in addition to importing members of the referenced namespace itself, all namespaces that are owned members of the imported package are also recursively imported.

A *package* is a kind of namespace that is used solely as a container for other elements to organize the model. In addition, a package has the capability to *filter* imported elements based on certain conditions defined in terms of the metadata provided by annotating features of those elements (see [7.3](#)). 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.

7.4.2 Abstract Syntax

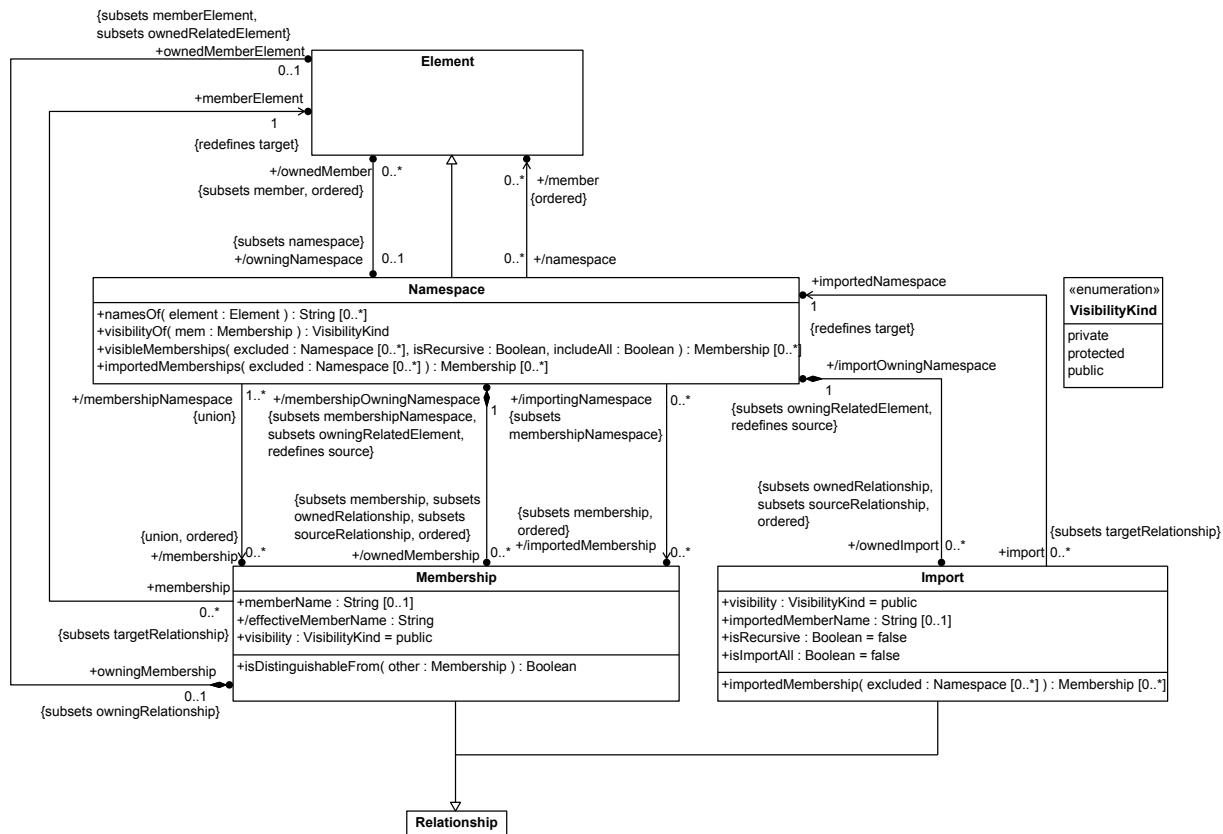


Figure 7. Namespaces

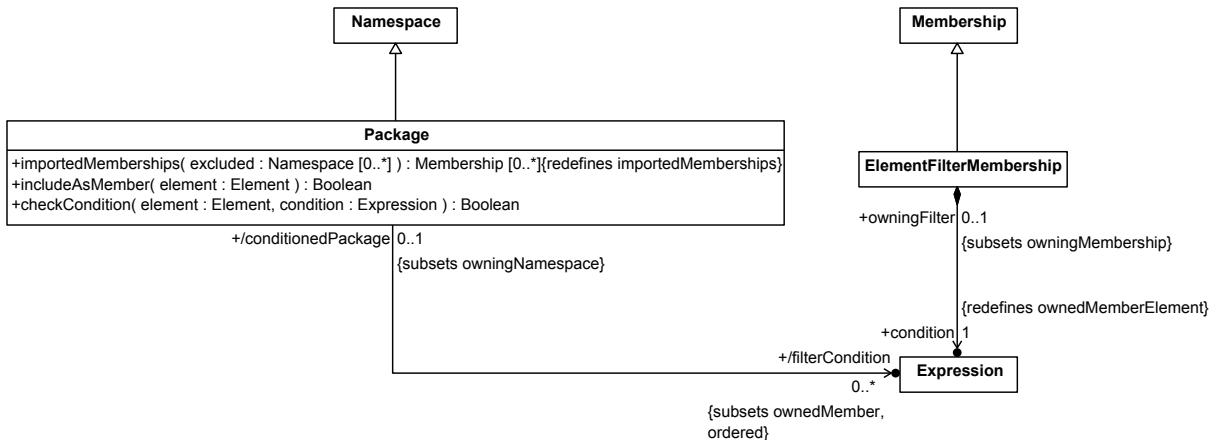


Figure 8. Packages

7.4.3 Notation

Declarations and Bodies

In addition to Packages, all kinds of SysML Definitions and Usages are also Namespaces (see [7.6](#) and following subclauses). Therefore, all rules discussed below generically for Namespaces shall apply generally to Packages, Definitions and Usages (though the examples in this subclause are all given in terms of Packages). In addition, the KerML rules for name resolution shall also apply for the SysML textual concrete syntax (see [[KerML, 7.2.4.2.4](#)]).

In general, the *declaration* of a Namespace gives its identification, while the *body* of a Namespace specifies its contents. The body of a Namespace is notated as a list of representations of the content of the Namespace delimited between curly braces `{ ... }`. If the Namespace is empty, then the body may be omitted and the declaration ended instead with a semicolon.

Root Namespaces

A *root Namespace* is a Namespace that has no owner. The `ownedElements` of a root Namespace are known as *top-level Elements*. Any Element that is not a root Namespace shall have an `owner` and, therefore, must be in the ownership tree of a top-level Element of some root Namespace.

Note. The set of all Elements owned directly or indirectly by a root Namespace may be considered to be the representation of a single "model", though this term is not formally defined within SysML.

The declaration of a root Namespace is implicit and no identification of it is provided in the SysML 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. For the textual notation, this will typically be contained in a single textual document. In a model repository, a single modeling "project" may contain one or more root Namespaces.

```

doc /* This is a model notated in SysML textual notation. */
item def I;
attribute def A;
item i: I;
package P;
  
```

Owned Members and Aliases

Declaring an Element within the body of a Namespace denotes that the Element is an `ownedMember` of the Namespace—that is, that there is an `ownedMembership` of the Namespace with the Element as its `ownedMemberElement`. The name given for the Element (if any) becomes the `memberName` of the Membership. The visibility of the Membership can also be specified by placing the keyword `public` or `private` before the Element declaration. If no visibility is specified, the default is `public`.

```
package P {  
    public part def A;  
    private attribute def B;  
    part a : A; // public by default  
}
```

An alias for an Element is declared using the keyword `alias` followed by the alias name, with a qualified name (see below) identifying the Element given after the keyword `for`. This denotes an `ownedMembership` of the containing Namespace, with the identified Element as an unowned `memberElement`. The visibility of the Membership can be specified as for an `ownedMember`.

Imports

An `ownedImport` of a Namespace is denoted using the keyword `import` followed by a qualified name. This specifies an Import whose `importedNamespace` is the qualification part of the qualification and whose `importedMemberName` is given by the the unqualified name. If the name given for the `import` is unqualified, then the `importedNamespace` shall be null and the given name shall be resolved in the scope of the Namespace owning the Import.

Such an Import results in the Membership of the `importedNamespace` whose `memberName` is the given `importedMemberName` becoming an `importedMembership` of the Namespace owning the Import. That is, the `memberElement` of this Membership becomes an imported member of the importing Namespace. Note that the `importedMemberName` may be an alias of the imported Element in the `importedNamespace`, in which case the Element is still imported with that name.

```
package P1 {  
    item A;  
    item B;  
    alias C for B;  
}  
package P2 {  
    import P1::A;  
    import P1::C; // Imported with name "C".  
    package Q {  
        import C; // "C" is re-imported from P2 into Q.  
    }  
}
```

If the qualified name in an import is follow followed by ::*, then the entire qualified name shall identify the `importedNamespace` and the `importedMemberName` shall be null. In this case, all visible Memberships of the `importedNamespace` of the Import shall become `importedMemberships` of the importing Namespace.

```
package P3 {  
    // Memberships A, B and C are all imported from P1.
```

```

import P1::*;

}

```

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

```

package P4 {
    item A;
    item B;
    package Q {
        item C;
    }
}

package P5 {
    import P4::**;
    // The above recursive import is equivalent to all
    // of the following taken together:
    //     import P4;
    //     import P4::*;
    //     import P4::Q::*;
}

package P6 {
    import P4::*::*;
    // The above recursive import is equivalent to all
    // of the following taken together:
    //     import P4::*;
    //     import P4::Q::*;
    // (Note that P4 itself is not imported.)
}

```

The visibility of the Import can be specified by placing the keyword **public** or **private** before the Import declaration. If no visibility is specified, the default is **public**.

```

package P7 {
    // The imported membership is visible outside P7.
    public import P1::A;

    // None of the imported memberships are visible outside of P7.
    private import P4::*;
}

```

An Import may also be declared with one or more `filterConditions`, given as model-level evaluable Boolean Expressions (see [KerML, 7.4.8]), listed at the end of the **import**, each surrounded by square brackets [...]. For such a filtered Import, Memberships shall be imported from the `importedNamespace` if and only if they satisfy all the given `filterConditions`.

```

package P8 {
    import Annotations::*;

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

```

Comments in Namespaces

A regular Comment (see [7.3](#)) declared within a Namespace body also becomes an `ownedMember` of the Namespace. If no `annotatedElements` are specified for the Comment, then, by default, the Comment is considered to be about the containing Namespace.

```
package P9 {
    item A;
    comment Comment1 about A
        /* This is a comment about item A. */

    comment Comment 2
        /* This is a comment about package P9. */

    /* This is also a comment about package P9. */
}
```

A Documentation Comment (see [7.3](#)) declared within a Namespace body, however, is *not* an `ownedMember` of the Namespace. Instead, if it is a regular Comment, then it is owned via a Documentation Relationship by the containing Namespace. If it is a prefix Comment, then it is owned via a Documentation Relationship with the next Membership or Import declared in the Namespace lexically after the Comment.

```
package P10 {
    doc id P10_Doc
        /* This is documentation about package P10. */

    /** This is documentation about member B. */
    /** This is more documentation about member B. */
    private item B;

    /** This is documentation about alias B1. */
    public alias B1 for B;

    /** This is documentation about the import of P4. */
    import P4::*;
}
```

Packages

A Package is declared using the keyword **package**, optionally followed by a `humanId` and/or name (see [7.2](#)).

```
package Configurations {
    attribute def ConfigEntry {
        attribute key: String;
        attribute value: String;
    }
    item ConfigData {
        attribute entries[*]: ConfigEntry;
    }
}
```

In addition, a Package body may contain one or more members that give `filterConditions` for the Package. These are notated using the keyword **filter** followed by a Boolean-valued, model-level evaluable Expression.

```

package Annotations {
    attribute def ApprovalAnnotation {
        attribute approved : Boolean;
        attribute approver : String;
        attribute level : Natural;
    }
    ...
}

package DesignModel {
    import Annotations::.*;
    part System {
        @ApprovalAnnotation {
            approved = true;
            approver = "John Smith";
            level = 2;
        }
    }
    ...
}

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

```

Note that a `filterCondition` in a Package will filter *all* imports of that Package. That is why full qualification is used for `Annotations::ApprovalAnnotation` above, since an imported elements of the `Annotations` Package would be filtered out by the very `filterCondition` in which the elements are intended to be used. This may be avoided by combining one or more `filterConditions` with a specific import, using the filtered Import notation described above.

```

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

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

```

The *SysML* library package contains a complete model of the SysML abstract syntax represented in SysML itself, and it publicly imports the *KerML* package from the Kernel Library containing the KerML abstract syntax model . When a `filterCondition` is evaluated on an Element, abstract syntax metadata for the Element can be tested as if the Element had an implicit `AnnotatingFeature` defined by the definition from the *SysML* package corresponding to the metaclass of the Element.

```

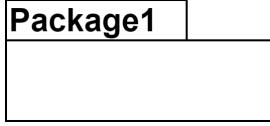
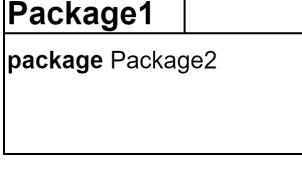
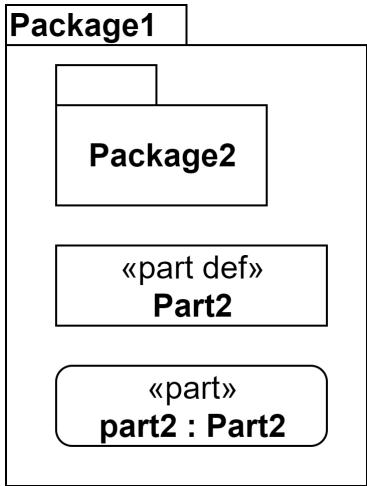
package PackageApprovals {
    import Annotations::.*;
    import SysML::*;

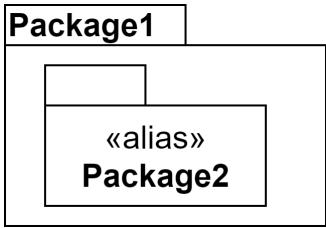
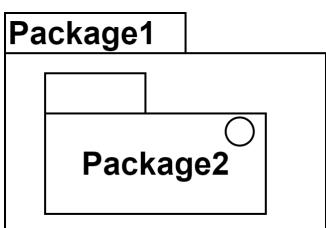
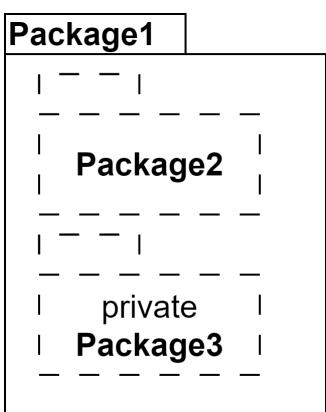
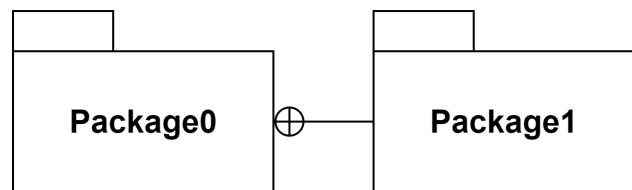
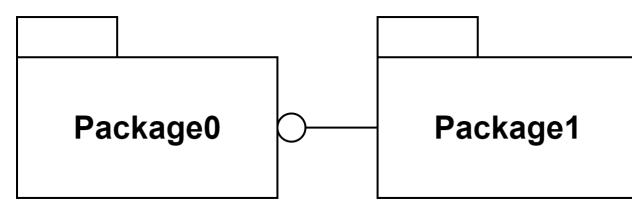
    // This imports all part definitions from the DesignModel that have
    // at least one owned part usage and have been marked as approved.
    import DesignModel::**[@PartDefinition and
                           @PartDefinition::ownedPart != null &&
                           @ApprovalAnnotation and
                           ApprovalAnnotation::approved];
}

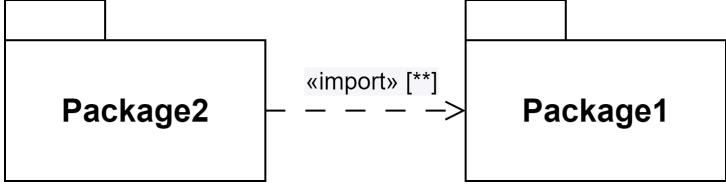
```

Note. Namespaces other than Packages cannot have filterConditions (except for their special use in ViewDefinitions and ViewUsages – see [7.25](#)). However, any kind of Namespaces may have filtered imports.

Table 8. Packages Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Package (name in body)		package Package1;
Package (name in tab)		package Package1;
Package with owned package		package Package1 { package Package 2; }
Package with owned members		package Package1 { package Package 2; }

Element	Graphical Notation	Textual Notation
Package with alias member		
Package with unowned member		
Package with imported package (nested notation)		<pre>package Package1 { import Package2; }</pre>
Membership (owned member)		<pre>package Package1{ package Package2; }</pre>
Membership (unowned member)		

Element	Graphical Notation	Textual Notation
Import		

7.5 Dependencies

7.5.1 Overview

A *dependency* is a kind of relationship between any number of client (source) and supplier (target) elements. This implies that a change to a supplier element may result in a change to a client element.

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.

7.5.2 Abstract Syntax

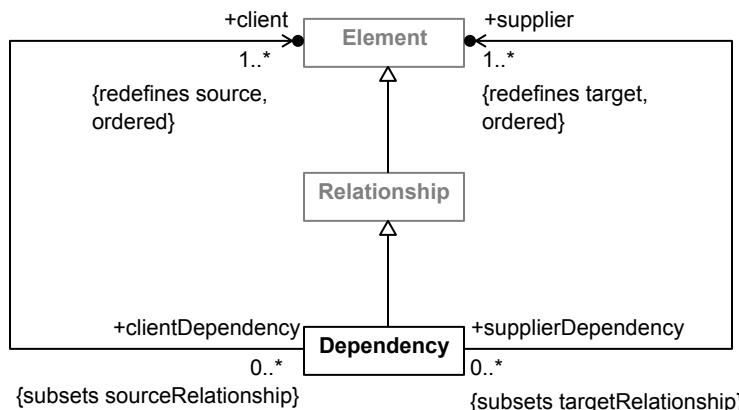
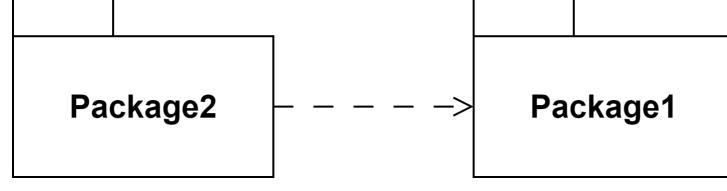


Figure 9. Dependencies

7.5.3 Notation

Table 9. Dependencies Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Dependency		

Element	Graphical Notation	Textual Notation
Dependency - nary	<pre> graph LR P1[Package1] --> P3[Package3] P12[Package12] --> P4[Package4] </pre>	

7.6 Definition and Usage

7.6.1 Overview

Definitions and Usages

The modeling capabilities of SysML facilitate reuse in different contexts. Definition and usage elements provide a consistent foundation for many SysML language constructs to provide this capability, including attributes, occurrences, items, parts, ports, connections, interfaces, allocations, actions, states, calculations, constraints, requirements, analysis cases, verification cases, use cases, viewpoints and renderings.

In general, a *definition* element classifies a certain kind of element (e.g., a classification of attributes, parts, actions, etc.). A *usage* element is a usage of a definition element in a certain context. A usage 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 of the appropriate kind from the Systems Library (see [9.2](#)). For example, a part usage is implicitly defined by the most general part definition *Part* from the model library package *Parts*.

Features

A definition may have owned usage elements nested in it, referred to as its *features*. A usage may also have nested usage elements as features. In this case, the context for the nested usages is the containing usage. A simple example is illustrated by a parts tree that 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 feature relates instances of its featuring definition or usage to instances of its definition. For example, a *mass* feature with definition *MassValue*, featured by the definition *Vehicle*, relates each specific instance of *Vehicle* to the specific *MassValue* for that vehicle, known as the *value* of the *mass* feature of the vehicle.

A usage can also be contained directly in an owning package. In this case, the usage element is considered to be an implicit feature of the most general kernel type *Anything*. That is, a package-level usage is essentially a generic feature that can be applied in any context, or further specialized in specific contexts (as described under Specialization below).

A usage may have a *multiplicity* that constrains its cardinality, that is, the allowed number of values it may have for any instance of its featuring definition or usage. 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 bounds

must be natural numbers. The lower bound must be finite, but the upper bound may also have the infinite 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. For example, a *Vehicle* definition could include a usage element called *wheels* with multiplicity 4, meaning each *Vehicle* has exactly four *wheels*. A less restrictive constraint, such as a multiplicity of 4 .. 8, means each *Vehicle* can have 4 to 8 wheels.

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

A usage may be *referential* or *composite*. A referential usage represents a simple reference between a featuring instance and one or more values. A composite usage, on the other hand, indicates that the related instance is integral to the structure of the containing instance. As such, if the containing instance is destroyed, then any instances related to it by composite usages are also destroyed. For example, a *Vehicle* would have a composite usage of its *wheels*, but only a referential usage of the *road* on which it is driving.

Note. The concept of composition only applies to occurrences that exist over time and can be created and destroyed (see [7.9](#)). Attribute usages are always referential and any nested features of attributes definitions and usages are also always referential (see [7.7](#)).

Specialization

Definition and usage elements can be specialized using several different kinds of *specialization* relationships.

A definition is specialized using the *subclassification* relationship. The specialized definition inherits the features of 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*. An inherited feature can be subsetted or redefined as described below. The *Truck* definition can also add its own features such as *cargoSize*.

A definition can specialize more than one other definition, in which case the definition inherits the features from each of the definitions it specializes. All inherited features must have names that are distinct from each other and any owned features of the specializing definition. Name conflicts can be resolved by redefining one or more of the otherwise conflicting inherited features (see below).

A usage inherits the features from its definition in the same way that a specialized definition inherits from 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*. In some cases, a usage may have more than one definition element, in which case the usage inherits the features from each of its definition elements, with the same rules for conflicting names as described above for subclassification. A usage can also add its own features, and subset or redefine its inherited features. This enables each usage to be modified for its context.

A usage can be specialized using the *subsetting* relationship. A subsetting usage has a subset of the values of the subsetted usage. The subsetting usage may further constrain its definition and multiplicity. For the example above, *Truck* inherits the feature *wheels* with multiplicity 4 .. 8 from *Vehicle*. The part usage *truck* further inherits *wheels* with multiplicity 4 .. 8 from *Truck*. The part usage *truck* can subset *wheels* by defining *frontLeftWheel*, *frontRightWheel*, *rearLeftwheel1*, and *rearRightWheel1*, each with multiplicity 1..1, together giving the minimum total multiplicity of 4. The *truck* usage can then define additional subsets of *wheels*, such as *rearLeftwheel2*, and *rearRightwheel2*, with multiplicity 0..1, indicating they are optional.

Redefinition is a kind of subsetting. While, in general, a subsetting usage is an additional feature to the subsetted usage, a redefining usage *replaces* the redefined usage in the context of redefining usage. For the example above, *Vehicle* contains a feature called *fuel* that is defined by *Fuel*. *Truck* inherits *fuel* from *Vehicle*. The part usage *truck* would then normally inherit *fuel* as defined by *Fuel* from *Truck*. However, *truck* can instead redefine *fuel* to restrict its definition to *DieselFuel*, a subclassification of *Fuel*. In this case, the new redefining feature replaces the *fuel* feature that would otherwise be inherited, meaning that the *fuel* of the *truck* part must be *DieselFuel*.

A usage, particularly one with nested usages, can be reused by subsetting it. For example, subsetting the part usage *vehicle* is analogous to specializing the part definition *Vehicle*. Suppose *vehicle1* is a part usage that subsets *vehicle*, with the parts-tree decomposition described above. This enables *vehicle1* to inherit the features and structure of *vehicle*. The part usage *vehicle1* can be further specialized by adding other part usages to it, such as a *body* and *chassis*, and it can redefine 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 a unique design configuration extending that of *vehicle*. Other part usages, such as *vehicle2*, could be created in a similar way to represent other design configurations.

Note. 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, then 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.

Variability

Variation and *variant* are used to model variability typically associated with a family of design configurations. A variation (sometimes referred to as a *variation point*) identifies an element in a 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* feature is designated as a point of variation, the design choices are a *4-cylinder engine* variant or a *6-cylinder engine* variant.

Variation can apply to any kind of definition or usage in the model (except for enumeration, see [7.8](#)). The variation element then specifies all possible variants (i.e., choices) for that variation point. For example, the specified variants for the *engine* variation are the *4-cylinder engine* and the *6-cylinder engine*.

Variants are usage elements. If the containing variation is a definition, then each of its variants is implicitly defined by the variation definition. If the containing variation is a usage, then each of its variants implicitly subsets the variation usage. For example, the *4-cylinder engine* and the *6-cylinder engine* are subsets of all possible *engines*.

Variations can be nested within other variations, to any level of nesting. For example, the *6-cylinder engine* variant may in turn contain a variation for *bore diameter* that includes variants for *small-bore diameter* and *large-bore diameter*. Alternatively, the *bore diameter* variation could 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 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*.

Variations 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 provides 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.

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

7.6.2 Abstract Syntax

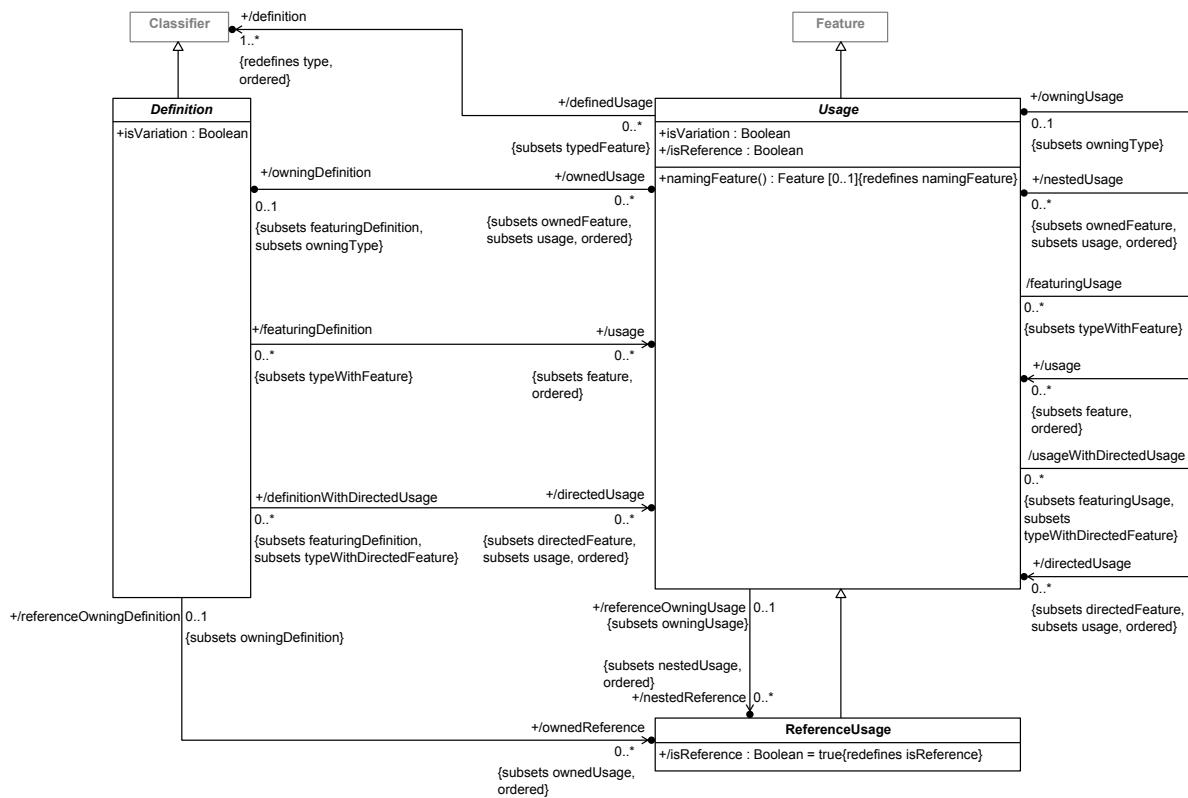


Figure 10. Definition and Usage

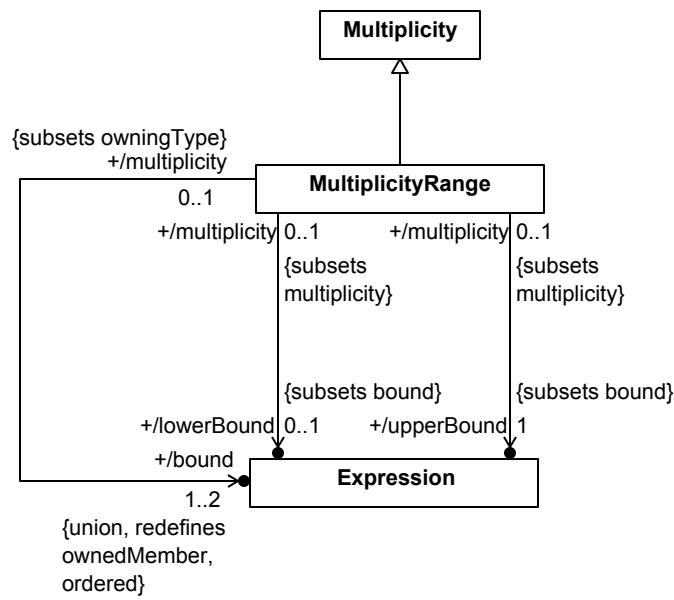


Figure 11. Multiplicities

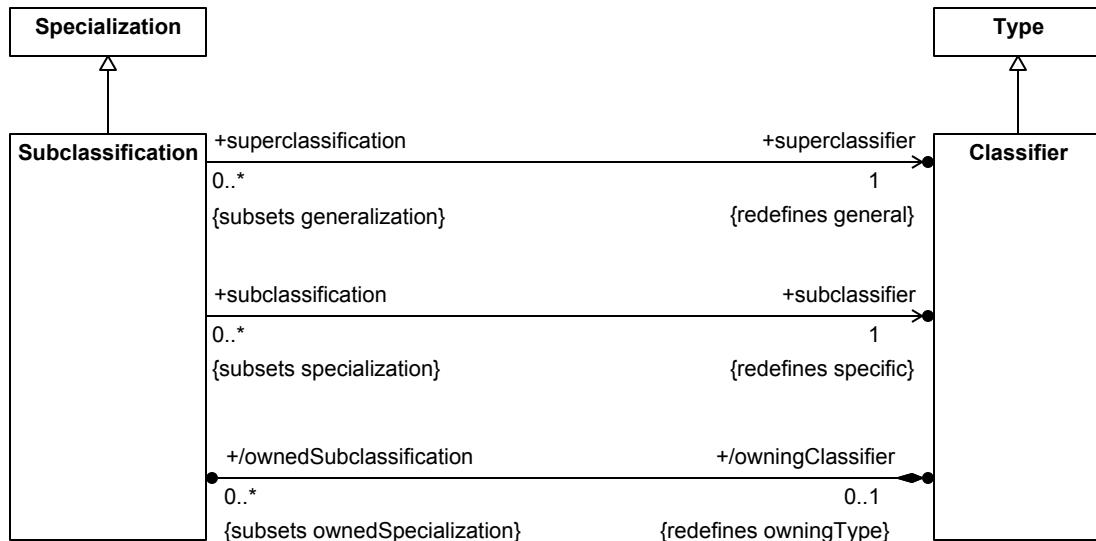


Figure 12. Classifiers

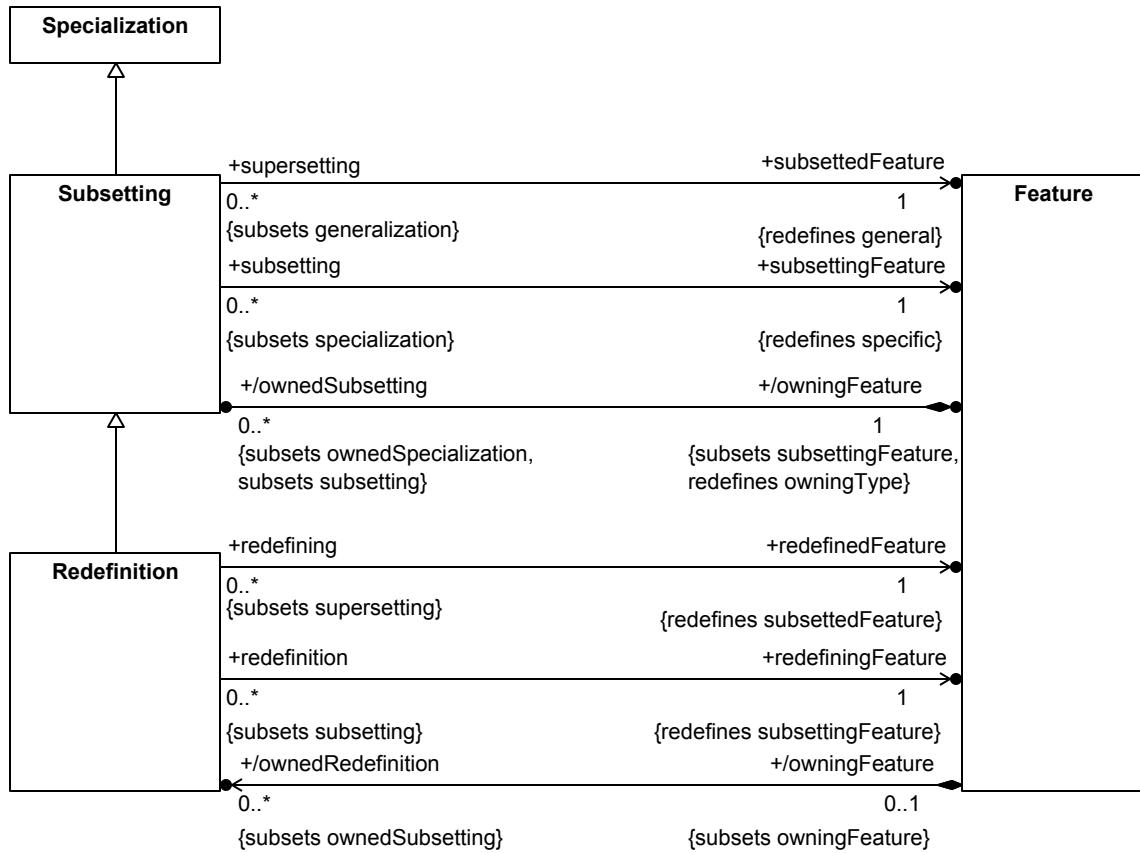


Figure 13. Subsetting

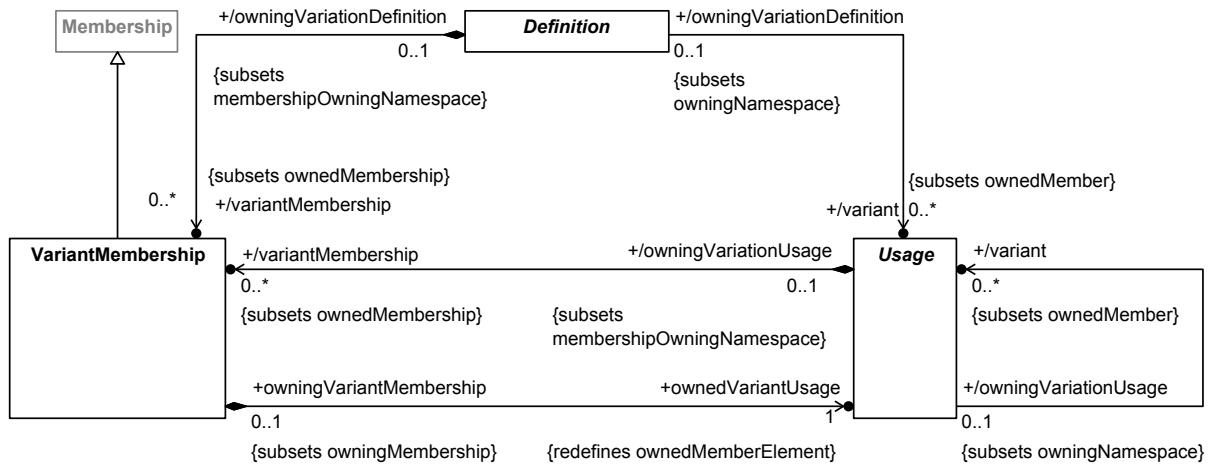
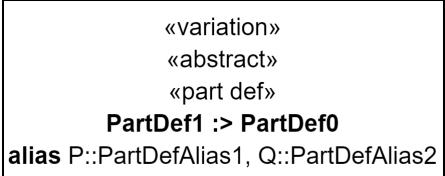
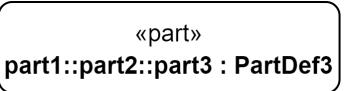
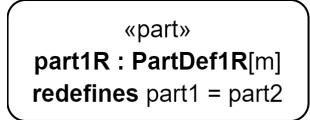
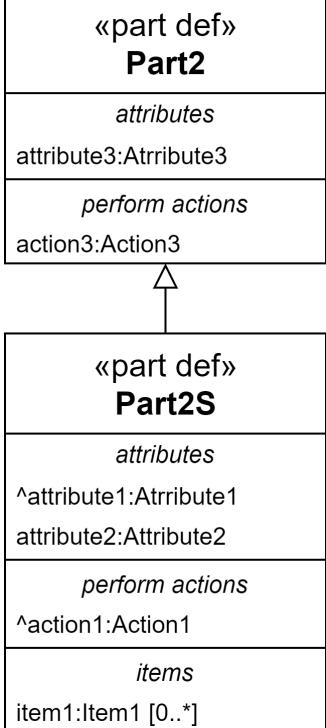
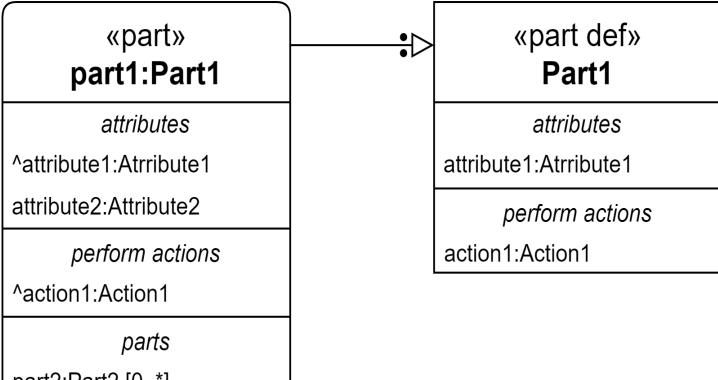


Figure 14. Variant Membership

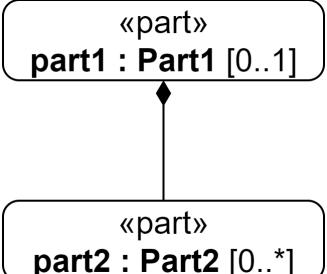
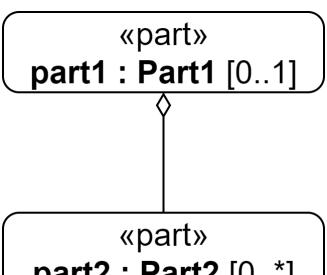
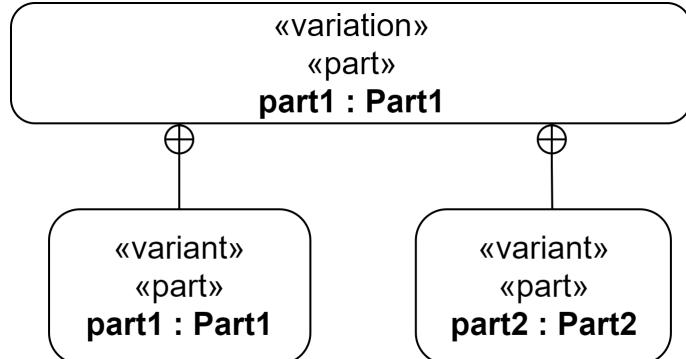
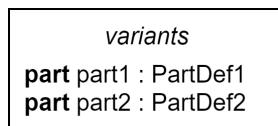
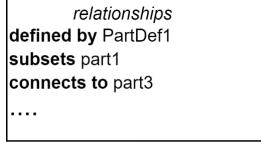
7.6.3 Notation

Table 10. Definition and Usage Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Name Compartment - Definition		part def PartDef1;
Name Compartment - Definition		part def PartDef1 :> PartDef0; alias PartDefAlias1 for PartDef1; alias PartDefAlias2 for PartDef1;
Name Compartment - Definition (qualified name)		
Name Compartment - Definition (abstract)		abstract part def PartDef1;
Name Compartment - Usage		part part1:PartDef1;
Name Compartment - Usage		
Name Compartment - Usage		variation part part1:PartDef1 alias PartDefAlias1 for PartDef1; alias PartDefAlias2 for PartDef1;
Name Compartment - Usage		part ^part1:PartDef1;
Name Compartment - Usage		part part1S : PartDef1S [m] subsets part1;

Element	Graphical Notation	Textual Notation
Name Compartment - Usage		part part1R : PartDef1R [m] redefines part1 = part2;
Subclassification		:<> or specializes
Definition		: or defined by

Element	Graphical Notation	Textual Notation
Subsetting	<pre> <<part>> part1:Part1 attributes ^attribute1:Attribute1 attribute2:Attribute2 perform actions ^action1:Action1 parts part2:Part2 [0..*] <<part>> part1S:Part1 attributes ^attribute1:Attribute1 ^attribute2:Attribute2 attribute2:Attribute3 perform actions ^action1:Action1 parts ^part2:Part2 [0..*] part3:Part3 [0..*] </pre>	:> or subsets
Redefinition	<pre> <<part>> part1 : Part1 [0..*] <<part>> part1S : Part1S [1] --> <<part>> part2 : Part2 [0..*] <<part>> part2R : Part2R <--> </pre>	:>> or redefines
Feature Membership (isComposite=true)	<pre> <<part def>> PartDef1 <<part>> part2 [0..*] 0..1 </pre>	part def PartDef1 { part part2:Part2 [0..*]; }

Element	Graphical Notation	Textual Notation
Feature Membership (isComposite=true)	 <pre data-bbox="1192 354 1481 466"> part part1:Part1 { part part2:Part2 [0..*]; } </pre>	
Feature Membership (isComposite=false)	 <pre data-bbox="1192 671 1491 783"> part part1:Part1 { ref part part2:Part2 [0..*]; } </pre>	
Variant Membership		
Variants Compartment	 <pre data-bbox="677 1353 905 1425"> variants part part1 : PartDef1 part part2 : PartDef2 </pre>	
Variant Parts Compartment	 <pre data-bbox="677 1522 873 1615"> variant parts part1 : PartDef1 part2 : PartDef2 </pre>	
Relationships Compartment	 <pre data-bbox="677 1700 840 1826"> relationships defined by PartDef1 subsets part1 connects to part3 </pre>	

7.7 Attributes

7.7.1 Overview

An *attribute definition* defines a set of data values, such as numbers, quantitative values with units, qualitative values such as text strings, or data structures of such values. An *attribute usage* is a usage of an attribute definition. An attribute usage shall always be referential and any nested features of an attribute definition or usage shall also be referential (see also [7.6](#) on referential and composite usages).

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 [7.19](#)). 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 [7.8](#)).

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.

Attribute usages representing quantities with units are defined using the SysML Quantities and Units Domain Library or extensions of the elements in this library (see [9.5](#)). The library provides base attribute definitions for scalar, vector and tensor quantity values, 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., *kg*, *m*, *s*, 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.).

The values of an attribute usage are data values, whether primitive values like integers or structured values with nested attributes, that do not themselves change over time. However, the owner of an attribute usage may be an occurrence definition or usage (or a specialized kind of occurrence, such as an item, part or action). In this case, the value of the attribute usage may vary over the lifetime of its owning occurrence, in the sense that it can have different values at different points in time, reflecting the changing condition of the occurrence over time. (See also the discussion in [7.9](#).)

7.7.2 Abstract Syntax

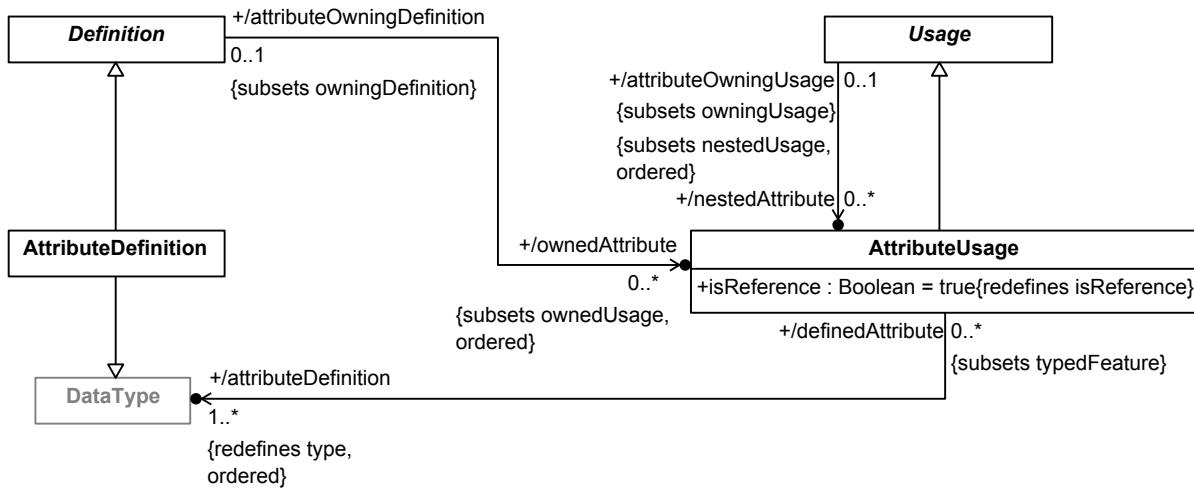
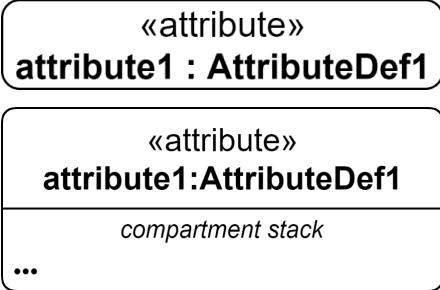


Figure 15. Attribute Definition and Usage

7.7.3 Notation

Table 11. Attributes Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Attribute Definition (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>allocations</i> , <i>attributes</i> , <i>constraints</i> , <i>documentation</i> , <i>metadata</i> , <i>namespaces</i> , <i>relationships</i> , <i>variants</i> , <i>variant element usages</i>)	<div style="border: 1px solid black; padding: 5px; text-align: center;"> «attribute def» AttributeDef1 </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> «attribute def» AttributeDef1 </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> <i>compartment stack</i> ... </div>	<pre> attribute def AttributeDef1; or attribute def AttributeDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Attribute (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>allocations</i> , <i>attributes</i> , <i>constraints</i> , <i>documentation</i> , <i>metadata</i> , <i>namespaces</i> , <i>relationships</i> , <i>variants</i> , <i>variant elementusages</i>)		<pre data-bbox="1036 340 1395 403">attribute attribute1 : AttributeDef1;</pre> <p data-bbox="1036 439 1068 466">or</p> <pre data-bbox="1036 494 1395 614">attribute attribute1 : AttributeDef1 { /* members */ }</pre>
Attributes Compartment	<pre data-bbox="448 741 1003 1094"> attributes attribute1:AttributeDef1 [1..*] ordered nonunique ^attribute2:AttributeDef2 attribute3R:AttributeDef3R redefines attribute3 attribute4R:AttributeDef4R:>>attribute5 :>>attribute5 attribute6S:AttributeDef6S [m] subsets attribute6 [n] attribute7S:AttributeDef7S [m] :> attribute7 [n] attribute8 :AttributeDef8 = expression1 attribute10 :AttributeDef10 := expression2 ... </pre>	<pre data-bbox="1036 813 1428 1030"> { attribute attribute1:AttributeDef1 [1..*] ordered unique; /* ... */ } </pre>

7.8 Enumerations

7.8.1 Overview

An enumeration definition is a kind of attribute definition (see [7.7](#)) whose instances are limited to specific set of *enumerated values*. An *enumeration usage* is an attribute usage that is required to have a single definition that is an enumeration definition.

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 (see also [7.13](#) on binding connections). In this case, the results of all the expressions shall be distinct (typically they will just be literals).

For example, an enumeration definition *DiameterChoices* may specialize the attribute definition *LengthValue*. *DiameterChoices* may include literals that are equal to 60 mm, 80 mm, and 100 mm. An

attribute called `cylinderDiameter` can be defined by `DiameterChoices`, and its value can equal one of the three enumerated values.

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 specialization require that the set of 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 specialization.

Submission Note. It is expected that the restriction of enumerations to just attribute definitions will be removed in the final submission.

7.8.2 Abstract Syntax

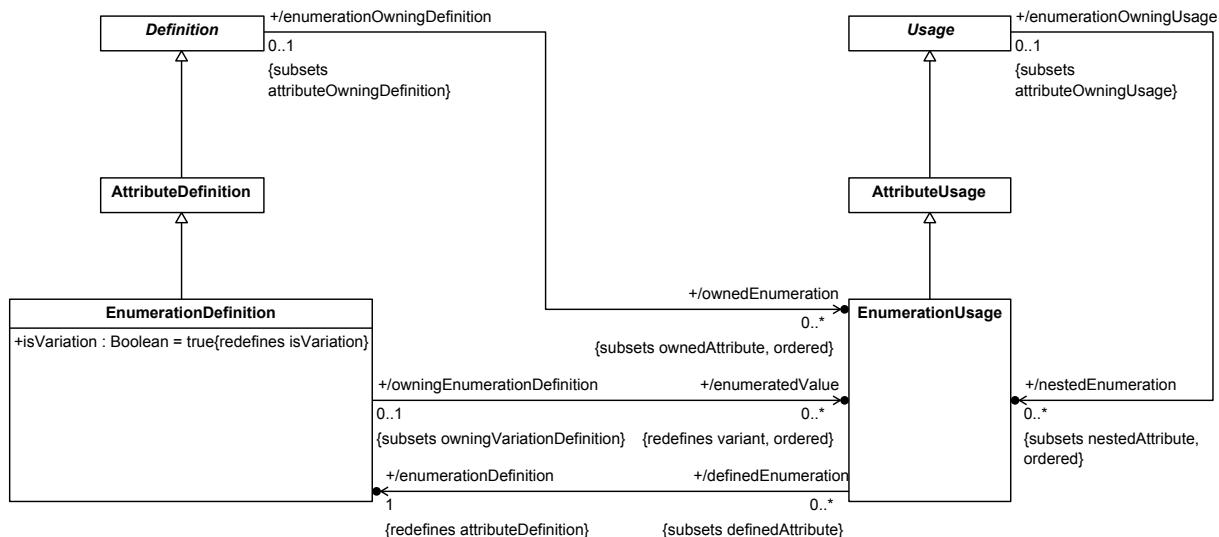


Figure 16. Enumeration Definition and Usage

7.8.3 Notation

Table 12. Enumerations Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Enumeration Definition (can include any of the following compartments in the compartment stack: <i>allocations, attributes, constraints, documentation, enums, metadata, namespaces, relationships, variants, variant elementusages</i>)	<div style="border: 1px solid black; padding: 5px; text-align: center;"> «enumeration def» EnumerationDef1 </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> «enumeration def» EnumerationDef1 </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> <i>compartment stack</i> </div> <div style="border: 1px solid black; padding: 2px;">...</div>	enumeration def EnumerationDef1; or enumeration def EnumerationDef1 { <i> /* members */</i> }

Element	Graphical Notation	Textual Notation
Enums Compartment	<pre> enums enum1 enum2 </pre>	
Enums Compartment	<pre> enums = value1 [unit1] = value2 [unit2] </pre>	

7.9 Occurrences

7.9.1 Overview

Occurrences

An *occurrence definition* is a definition of a class of occurrences that have an extent in time and may have spatial extent. An *occurrence usage* is a usage of an occurrence definition.

The extent of an occurrence in time is known as its *lifetime*, which covers the period in time from the occurrence's creation to its destruction. An occurrence maintains its identity over its lifetime, while the values of its features may change over time. The lifetime of an occurrence begins when the identity of the occurrence is established, and the lifetime ends when the occurrence loses its identity. For example, the lifetime of a car could begin when it leaves the production-line, or when a vehicle identification number is assigned to the car. Similarly, the lifetime of a car could end when the car is disassembled or demolished.

The performance of a behavior is also an occurrence that takes place over time. The lifetime of a behavior performance begins at the start of the performance and ends when the performance is completed. Structural and behavioral occurrences are often related. For example, a machine on an automobile assembly line, during its lifetime, will repeatedly perform a behavioral task, each performance of which has its own respective lifetime, which then affects the construction of some car being assembled on the line.

If an occurrence definition or usage has nested composite features, then those features must also be usages of occurrence definitions (including the various specialized kinds of occurrences, such as parts, items and actions). If an occurrence has values for any composite features at the end of its lifetime, then the lifetime of those composite values must also end. However, if a composite suboccurrence is removed from its containing occurrence before the end of the lifetime of the containing occurrence, then the former suboccurrence can continue to exist. For example, if a wheel is attached to a car when the car is destroyed, then the wheel is also destroyed. However, if the wheel is removed before the car is destroyed, then the wheel can continue to exist even after the car is destroyed. (See also [7.6](#) on referential and composite usages.)

Time Slices and Snapshots

The lifetime of an occurrence can be partitioned into *time slices* which correspond to some duration of time. These time slices represent periods or phases of a lifetime, such as the deployment or operational phase. Time slices can be further partitioned into other time slices. For example, the lifetime of a car might be divided into time slices corresponding to its assembly, being in inventory before being sold, and then sequential periods of ownership with different owners.

A time slice with zero time is a *snapshot*. Start, end and intermediate snapshots can be defined for any time slice to represent particular instants of time in an occurrence's lifetime. For example, the start snapshot of each ownership time slice of a car corresponds to the sale of the car to a new owner.

Individuals

Any kind of occurrence definition can be restricted to define a class that represents an *individual*, that is, a single real or perceived object with a unique identity. For example, consider the part definition *Car* modeling the class of all cars (parts are kinds of items which are kinds of occurrences, see [7.11](#)). An individual car called *Car1* with a unique vehicle identification number can then be modeled as an individual part definition that is a subclassification of the general part definition *Car*. As such, *Car1* inherits all the features of *Car* (such as its parts *engine*, *transmission*, *chassis*, *wheels*, etc.), but has individual values for each of those features (individual *engine*, individual *transmission*, individual *chassis*, four individual *wheels*, etc.), each of which is itself a uniquely identifiable subclassification of a more general part definition (*Engine*, *Transmission*, *Chassis*, *Wheel1*, etc.).

An occurrence usage can also be restricted to be the usage of a single individual. In this case, exactly one of the definitions of the occurrence usage must be an occurrence definition for that individual. Such an individual usage can be used to model a role that an individual plays for some period of time. For example, the individual part 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. Let *car1InService* be the usage of *Car1* 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. The four wheels of *Car1* are individual *Wheel* usages defined by the individual definitions *Wheel1*, *Wheel2*, *Wheel3*, and *Wheel4*. Each of these individual definitions is a subclassification of *Wheel*. When *car1InService* enters the shop, the *front-left wheel* individual usage is initially defined by the individual definition *Wheel1*, but after the tires are rotated, the *front-left wheel* is defined by the individual definition *Wheel2*.

The lifetime of an individual 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.

At a given point in time, the condition of an individual (some times called its "state", which should not be confused with a behavioral state usage, as described in [7.17](#)) can be specified by the values of its attributes. As an example, the condition of *car1inOperation* at different points in time can be specified in terms of its *acceleration*, *velocity*, and *position*. In addition, its finite (i.e., discrete) state (in the sense that can be modeled with state definitions and usages, see [7.17](#)) can be specified at different points in time as *off* or *on*, and any nested state such as *forward* or *reverse*. The condition of the car can continue to change over its lifetime, and can be specified as a function of discrete and/or continuous time.

Events

An *event* is a reference from one occurrence to another occurrence that represents some relevant happening during the lifetime of the first occurrence. Such an event model makes no commitment as to how the event actually happens. Different specializations of the containing occurrence may realize the modeled event in different ways.

In particular, occurrences may collaborate by transferring information between each other via *messages* (see [7.13](#)). The sending of such a message is an event in the lifetime of the sending occurrence, while the receipt of such a message is an event in the lifetime of the receiving occurrence. These events can possibly be realized as the performance of a send action and corresponding accept action, respectively, with the message being the resulting

transfer between them (see [7.16](#)). However, it could also be realized as the start and end of an explicitly modeled flow connection (see also [7.13](#) on flow connections and messages).

7.9.2 Abstract Syntax

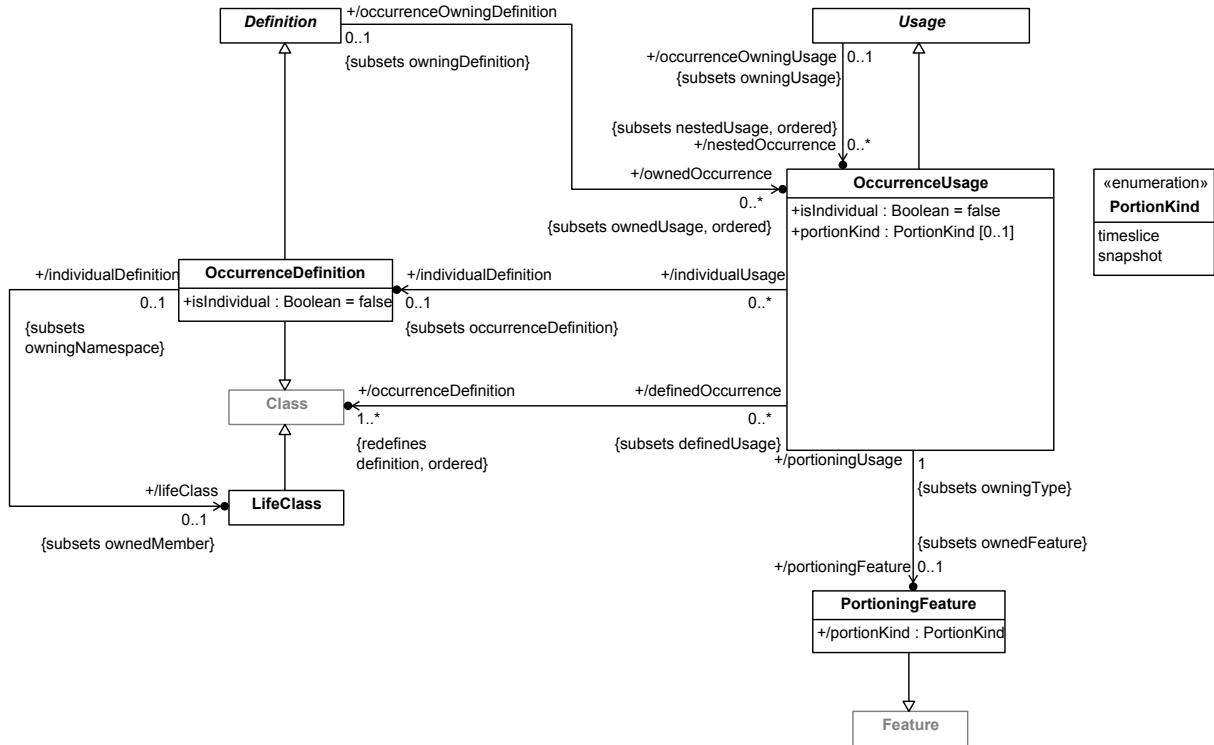


Figure 17. Occurrence Definition and Usage

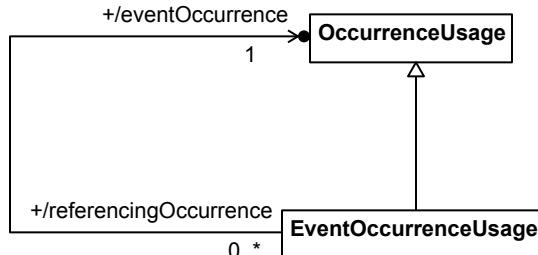
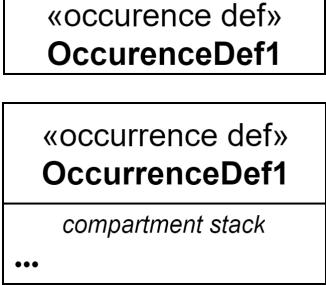


Figure 18. Event Occurrences

7.9.3 Notation

Table 13. Occurrences Notation - Representative Usage

Element	Graphical Notation	Textual Notation
<p>Occurrence Definition (can include any of the following compartments in the compartment stack: <i>actions</i>, <i>allocations</i>, <i>attributes</i>, <i>calcs</i>, <i>connections</i>, <i>constraints</i>, <i>directed features</i>, <i>documentation</i>, <i>individuals</i>, <i>interfaces</i>, <i>items</i>, <i>item flows</i>, <i>metadata</i>, <i>namespaces</i>, <i>occurrences</i>, <i>parameters</i>, <i>parts</i>, <i>ports</i>, <i>relationships</i>, <i>satisfy requirements</i>, <i>shapes</i>, <i>snapshots</i>, <i>states</i>, <i>exhibit states</i>, <i>timeslices</i>, <i>variants</i>, <i>variant elementusages</i>)</p>		<pre data-bbox="918 783 1408 931">occurrence def OccurrenceDef1; or occurrence def OccurrenceDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
Occurrence (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>allocations</i> , <i>attributes</i> , <i>calcs</i> , <i>connections</i> , <i>constraints</i> , <i>directed</i> <i>features</i> , <i>documentation</i> , <i>individuals</i> , <i>interfaces</i> , <i>items</i> , <i>item</i> <i>flows</i> , <i>metadata</i> , <i>namespaces</i> , <i>occurrences</i> , <i>parameters</i> , <i>parts</i> , <i>ports</i> , <i>relationships</i> , <i>satisfy</i> <i>requirements</i> , <i>shapes</i> , <i>snapshots</i> , <i>states</i> , <i>exhibit</i> <i>states</i> , <i>timeslices</i> , <i>variants</i> , <i>variant</i> <i>elementusages</i>)	<div style="border: 1px solid black; padding: 10px;"> <p>«occurrence»</p> <p>occurrence1 : OccurrenceDef1</p> <p>«occurrence»</p> <p>occurrence1:OccurrenceDef1</p> <p style="text-align: center;">compartment stack</p> <p>...</p> </div>	<pre>occurrence occurrence1 : OccurrenceDef1; or occurrence occurrence1 : OccurrenceDef1 { /* members */ }</pre>
Individual Definition	<div style="border: 1px solid black; padding: 5px; text-align: center;"> <p>«individual def»</p> <p>«occurrence def»</p> <p>OccurrenceDef1-1</p> </div>	<pre>individual def 'OccurrenceDef1-1';</pre>
Individual Definition		<pre>individual def 'OccurrenceDef1-1' { }</pre>
Individual	<div style="border: 1px solid black; padding: 5px; text-align: center;"> <p>«individual»</p> <p>«occurrence»</p> <p>occurrence1 : OccurrenceDef1-1</p> </div>	<pre>individual occurrence occurrence1:'OccurrenceDef1-1';</pre>
Individual		<pre>Individual occurrence occurrence1:'OccurrenceDef1-1'{ }</pre>

Element	Graphical Notation	Textual Notation
Timeslice	<p>«timeslice»</p>	timeslice timeslice11:OccurrenceDef1;
Timeslice		timeslice timeslice11:OccurrenceDef1{ }
Snapshot	<p>«snapshot»</p>	snapshot snapshot1:OccurrenceDef1;
Snapshot		snapshot snapshot1:OccurrenceDef1{ }
Portion Membership		
Occurrences Compartment	<pre> occurrences occur1:OccrDef1 [1..*] ordered nonunique ^occur2:OccurDef2 occur3R:OccurDef3R redefines occur3 occur4R:OccurDef4R:>>occur4 :>>part5 occur6S:OccurDef6S [m] subsets occur6 [n] occur7S:OccurDef7S [m] :> occur7 [n] occur8R = occur8 ref occur9:OccurDef9 occur 10 occur 10.1 occur 10.2 </pre>	
Individuals Compartment	<pre> individual parts part1:PartDef1_1 [1..*] ordered nonunique ^part2:PartDef2_1 part3R:PartDef3R_1 redefines part3 part4R:PartDef4R:_1>>part4 :>>part5 part6S:PartDef6S_1 [m] subsets part6 [n] part7S:PartDef7S_1 [m] :> part7 [n] part8R_1 = part8 ref part9:PartDef9_1 part10 part10.1 part10.2 </pre>	

Element	Graphical Notation	Textual Notation
Timeslices Compartment	<pre> timeslices attribute attribute1 =: x state state1 part1_timeslice1:PartDef1 [1..*] ordered nonunique ^part2_timeslice2:PartDef2 part3R_timeslice3R:PartDef3R redefines part3 part4R_timeslice4R:PartDef4R:>>part4 :>>part5 part6S_timeslice6S:PartDef6S [m] subsets part6 [n] part7S_timelice7S:PartDef7S [m] >> part7 [n] part8R_timeslice8R = part8 ref part9_timelice9:PartDef9 part10_timslice_10 part10.1_timeslice10 part10.2_timeslice10 </pre>	
Snapshots Compartment	<pre> snapshots attribute attribute1 =: x state state1 part1_snapshot1:PartDef1 [1..*] ordered nonunique ^part2_snapshot2:PartDef2 part3R_snahpshot3R:PartDef3R redefines part3 part4R_snapshot4R:PartDef4R:>>part4 :>>part5 part6S_snapshot6S:PartDef6S [m] subsets part6 [n] part7S_snapshot7S:PartDef7S [m] >> part7 [n] part8R_snapshot8R = part8 ref part9_snapshot9:PartDef9 part10_snaphsot10 part10.1_snapshot10 part10.2_snapshot10 </pre>	

7.10 Items

7.10.1 Overview

An *item definition* is a kind of occurrence definition (see [7.9](#)) that defines a class of identifiable objects that may be acted on over time, but which do not necessarily perform actions themselves. An *item usage* is a usage of one or more item definitions.

Item usages can be used to represent inputs and outputs to actions such as water, fuel, electrical signals and data. Item usages, such as fuel, may flow through a system and be stored by a system. An item may have attributes (see [7.7](#)), states (see [7.17](#)), and nested item usages.

An item that performs actions is normally modeled as a part (see [7.11](#)). All parts are items, but not all items are necessarily parts. An item may also be considered to be a part during some time slices of its lifetime but not others. For instance, an engine being assembled can be modeled as an item moving along an assembly line. However, once the engine assembly is completed, the engine can be considered to be a part that may be installed into a car, where it is expected to perform the action providing power to propel the car. But later it may be removed from the car and again be considered only an inactive item in a junkyard.

7.10.2 Abstract Syntax

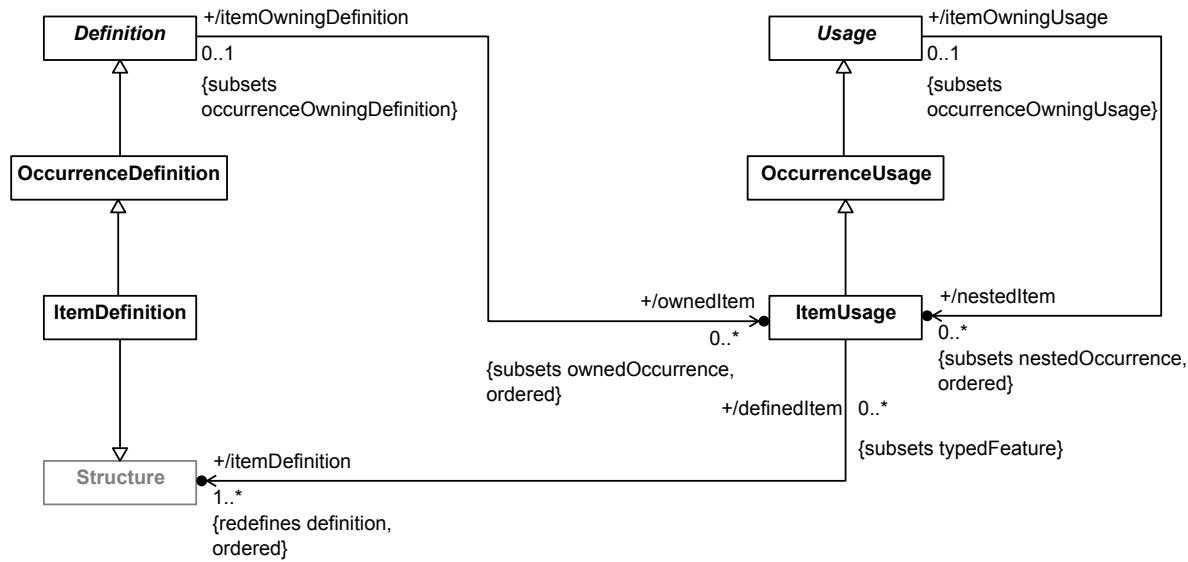
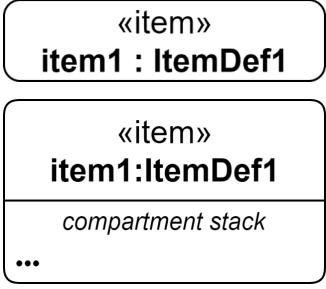


Figure 19. Item Definition and Usage

7.10.3 Notation

Table 14. Items Notation - Representative Usage

Element	Graphical Notation	Textual Notation
<p>Item Definition (can include any of the following compartments in the compartment stack: <i>allocations, attributes, connections, constraints, documentation, individuals, items, metadata, namespaces, relationships, snapshots, states, exhibit states, timeslices, variants, variant elementusages</i>)</p>	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«item def»</p> <p>ItemDef1</p> </div> <div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«item def»</p> <p>ItemDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>item def ItemDef1; or item def ItemDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
<p>Item (can include any of the following compartments in the compartment stack: <i>allocations</i>, <i>attributes</i>, <i>connections</i>, <i>constraints</i>, <i>documentation</i>, <i>individuals</i>, <i>items</i>, <i>metadata</i>, <i>namespaces</i>, <i>relationships</i>, <i>snapshots</i>, <i>states</i>, <i>exhibit states</i>, <i>timeslices</i>, <i>variants</i>, <i>variant element usages</i>)</p>		<pre>item item1 : ItemDef1; or item item1 : ItemDef1 { /* members */ }</pre>
<p>Items Compartment</p>	<pre>items item1:ItemDef1 [1..*] ordered nonunique ^item2:ItemDef2 item3R:ItemDef3R redefines item3 item4R:ItemDef4R:>>item4 :>>item5 item6S:ItemDef6S [m] subsets item6 [n] item7S:ItemDef7S [m] :> item7 [n] item8R = item8 ref item9:ItemDef9 item10 item10.1 item10.2</pre>	

7.11 Parts

7.11.1 Overview

A *part definition* 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 part definition is a kind of item definition (see [7.10](#)) and, as such, defines a class of part objects that are occurrences with temporal (and possibly spatial) extent. A part usage is a kind of item usage that is a usage of one or more part definitions, but may also be a usage of item definitions that are not part 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 [7.7](#)) that represent different performance, physical and other quality characteristics. The parts may have ports (see [7.12](#)) that define the points at which those parts may be interconnected (see [7.13](#) and [7.14](#)). Parts may also *perform* actions (see [7.16](#)) resulting in items flowing across the connections between them, and *exhibit* states (see [7.17](#)) 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 hardware components, software components, facilities, organizations, or users of a system.

7.11.2 Abstract Syntax

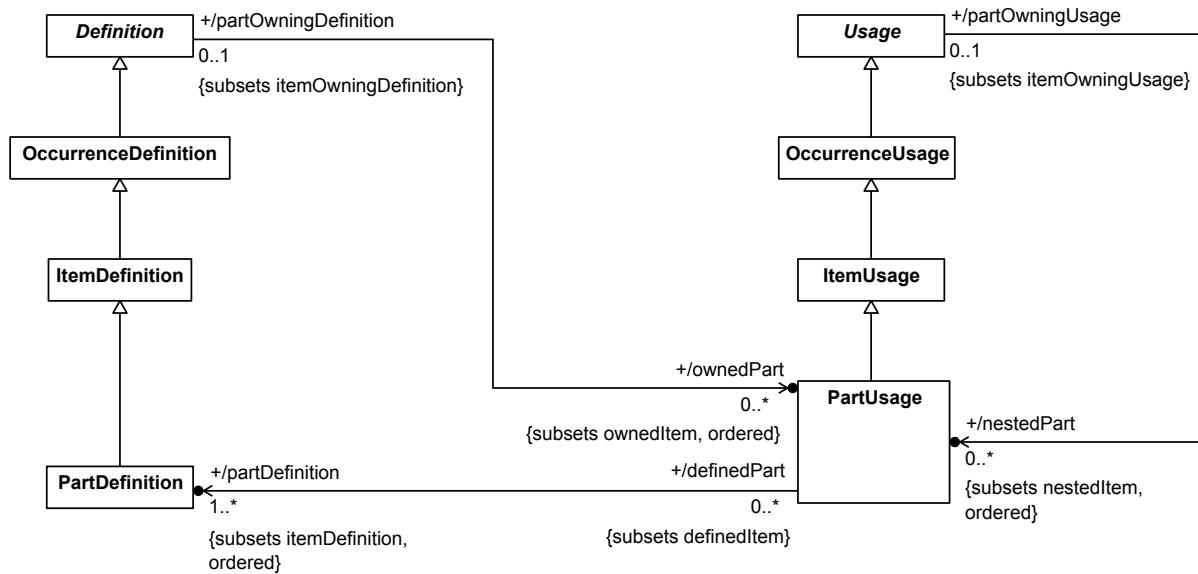
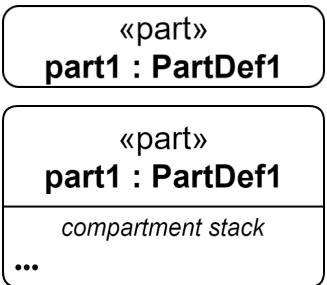
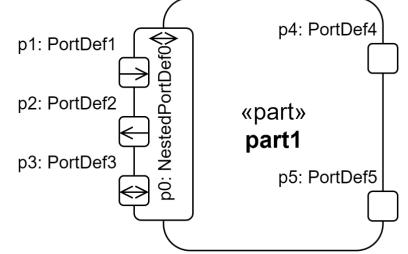
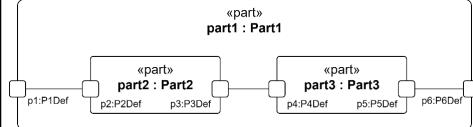
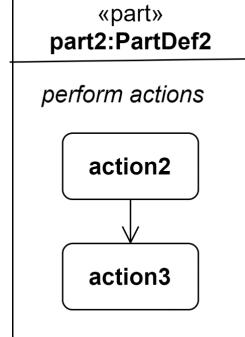


Figure 20. Part Definition and Usage

7.11.3 Notation

Table 15. Parts Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Part Definition (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>perform actions</i> , <i>allocations</i> , <i>attributes</i> , <i>connections</i> , <i>constraints</i> , <i>assert constraints</i> , <i>directed features</i> , <i>documentation</i> , <i>individuals</i> , <i>interfaces</i> , <i>items</i> , <i>metadata</i> , <i>namespaces</i> , <i>parts</i> , <i>ports</i> , <i>relationships</i> , <i>satisfy requirements</i> , <i>shapes</i> , <i>snapshots</i> , <i>states</i> , <i>exhibit states</i> , <i>timeslices</i> , <i>variants</i> , <i>variant elementusages</i>)	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> «part def» PartDef1 </div> <div style="border: 1px solid black; padding: 10px; width: fit-content;"> «part def» PartDef1 <div style="border: 1px solid black; padding: 5px; margin-top: 5px;">compartment stack</div> ... </div>	<pre> part def PartDef1; or part def PartDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Part (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>perform actions</i> , <i>allocations</i> , <i>attributes</i> , <i>connections</i> , <i>constraints</i> , <i>assert constraints</i> , <i>directed features</i> , <i>documentation</i> , <i>individuals</i> , <i>interfaces</i> , <i>items</i> , <i>metadata</i> , <i>namespaces</i> , <i>parts</i> , <i>ports</i> , <i>relationships</i> , <i>satisfy requirements</i> , <i>shapes</i> , <i>snapshots</i> , <i>states</i> , <i>exhibit states</i> , <i>timeslices</i> , <i>variants</i> , <i>variant element usages</i>)		<pre>part part1 : PartDef1; or part part1 : PartDef1 { /* members */ }</pre>
Part with Ports		<pre>part part1:PartDef1{ port p0:NestedPortDef0 }</pre>
Part with Parts Compartment - Graphical		<pre>part part1:PartDef1{ port p1:P1Def; port p2:P2Def; part part2:Part2; part part3:Part3}</pre>
Part with Perform Actions Compartment		

Element	Graphical Notation	Textual Notation
Parts Compartment	<pre data-bbox="652 264 1085 601"> parts part1:PartDef1 [1..*] ordered nonunique ^part2:PartDef2 part3R:PartDef3R redefines part3 part4R:PartDef4R:>>part4 :>>part5 part6S:PartDef6S [m] subsets part6 [n] part7S:PartDef7S [m] :> part7 [n] part8R = part8 ref part9:PartDef9 part10 part10.1 part10.2 </pre>	
Shapes Compartment	<pre data-bbox="758 644 971 707"> shapes shape1:ShapeDef1 </pre>	

7.12 Ports

7.12.1 Overview

A *port definition* is a kind of occurrence definition (see [7.9](#)) that defines a connection point for interactions between occurrences (most commonly parts). A *port usage* is a kind of occurrence usage definition that is a usage of a port definition.

A port usage may be connected to one or more other port usages (see [7.14](#)). Such connections enable interactions between the occurrences that own the ports. The features of the port usages (whether inherited from its definition or declared locally for the usage) specify what can be exchanged in such interactions. Since ports are themselves kinds of occurrences, port definitions and usages can contain nested port usages.

A feature of a port may have one of the *directions* **in**, **out**, or **inout**. A feature with direction is called a *directed feature*. Connected ports must *conform*: each feature of a port at one end of a connection must have a matching feature on a port at the other end of the connection. Two features match if they have conforming definitions and either both have no direction or they have conjugate directions. The *conjugate* of direction **in** is **out** and vice versa, while direction **inout** is its own conjugate. A transfer can occur from the **out** features of one port usage to the matching **in** features of connected port usages. Transfers can occur in both directions between matching **inout** features.

Each port definition has a *conjugated* port definition whose directed features are conjugate to those of the original port definition. A conjugate port usage is a usage of a conjugated port definition. A conjugate port usage automatically conforms to a usage of the corresponding original port definition.

7.12.2 Abstract Syntax

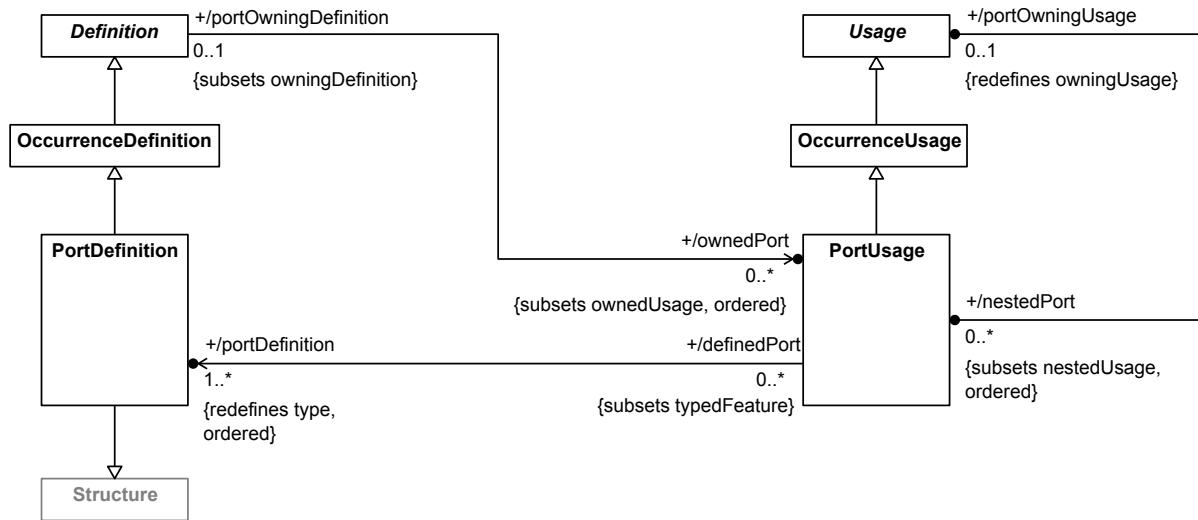


Figure 21. Port Definition and Usage

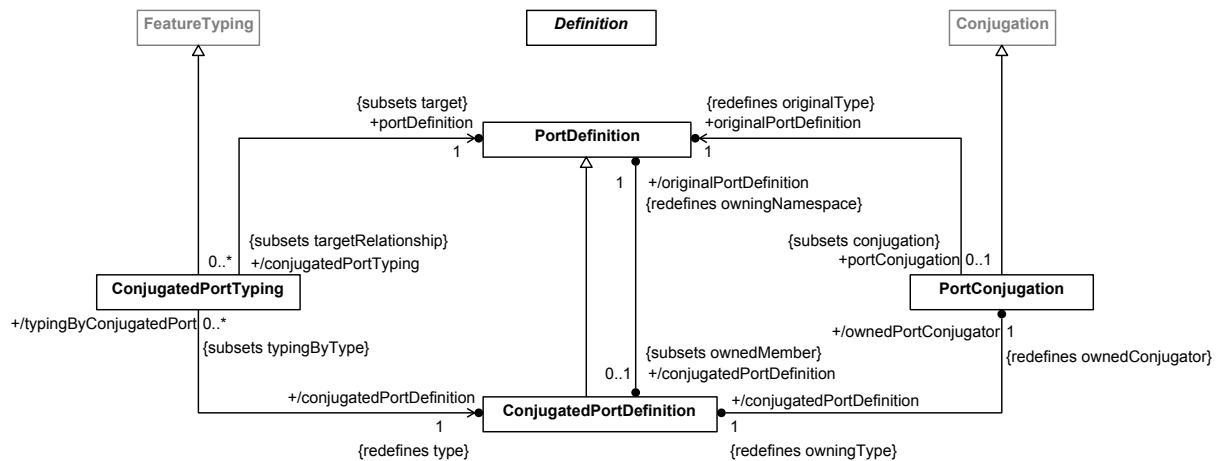
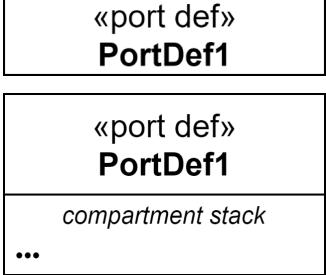
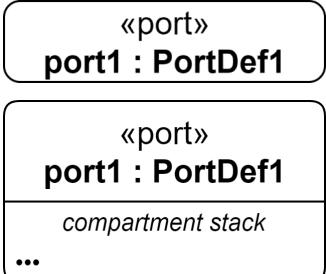


Figure 22. Port Conjugation

7.12.3 Notation

Table 16. Ports Notation - Representative Usage

Element	Graphical Notation	Textual Notation
<p>Port Definition (can include any of the following compartments in the compartment stack: <i>allocations</i>, <i>attributes</i>, <i>constraints</i>, <i>documentation</i>, <i>individuals</i>, <i>metadata</i>, <i>namespaces</i>, <i>ports</i>, <i>relationships</i>, <i>snapshots</i>, <i>timeslices</i>, <i>variants</i>, <i>variant element usages</i>)</p>		<pre>port def PortDef1; or port def PortDef1 { /* members */ }</pre>
<p>Port (can include any of the following compartments in the compartment stack: <i>allocations</i>, <i>attributes</i>, <i>constraints</i>, <i>documentation</i>, <i>individuals</i>, <i>metadata</i>, <i>namespaces</i>, <i>ports</i>, <i>relationships</i>, <i>snapshots</i>, <i>timeslices</i>, <i>variants</i>, <i>variant element usages</i>)</p>		<pre>port port1 : PortDef1; or port port1 : PortDef1 { /* members */ }</pre>
Ports Compartment	<pre>ports port1:PortDef1 [1..*] ordered nonunique ^port2:PortDef2 port3R:PortDef3R redefines port3 port4R:PortDef4R:>>port4 :>>port5 port6S:PortDef6S [m] subsets port6 [n] port7S:PortDef7S [m] > port7 [n] port8R = port8 ref port9:PortDef9 port10 port10.1 port10.2 port11:-PortDef11</pre>	

Element	Graphical Notation	Textual Notation
Directed Features Compartment	<pre> <i>directed features</i> in attribute1:AttributeDef1 out attribute2:AttributeDef2 in-out attribute3:AttributeDef3 in item1:ItemDef1 out item2:ItemDef2 in-out item3:ItemDef3 reqd action1:ActionDef1 (to be confirmed) prov action2:ActionDef2 (to be confirmed) </pre>	

7.13 Connections

7.13.1 Overview

Connection Definition and Usage

A *connection definition* is both a relationship and a kind of part definition (see [7.11](#)) that classifies connections between related things, such as items and parts. At least two of the owned features of a connection definition must be *connection ends*, which specify the things that are related by the connection definition. Connection definitions with exactly two connection ends are called *binary connection definitions*, and they classify *binary connections*.

The features of a connection definition that are not connection ends characterize connections separately from the connected things. Since a connection is a part, values of these non-end features can potentially change over the lifetime of the connection. However, the values of connection ends (i.e., the things that are actually connected) do not change over time (though they can potentially be occurrences that themselves have features whose values change over time).

A connection usage is a part usage (see [7.11](#)) that is a usage of a connection definition, connecting usage elements such as item and part usages. A connection usage inherits connection ends from its definition and associates those ends with the specific usage elements that are to be connected. For example, a connection definition could have connection ends that are part usages defined by part definitions *Pump* and *Tank*. A usage of this connection definition would then associate corresponding connection ends with specific *pump* and *tank* part usages.

Supposing that the *pump* and *tank* part usages have multiplicity 1, then this means that the single value of the *pump* usage is to be connected to the single value of the *tank* usage.

A connection usage that connects parts is often a logical connection that abstracts away details of how the parts 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. As a part itself, a connection can contain the plumbing either as a composite feature, 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 into a physical connection.

Bindings and Successions

Bindings and successions are special kinds of connection usages that are not part usages. Unlike regular connection usages, the connections specified by bindings and successions are not occurrences and are not created and destroyed. Rather, they assert specific relationships between the features that they connect, which must be true at all times.

A *binding* is a binary connection that requires its two related usages to have the same values. A binding can also be used to bind a referential feature in one context to a composite feature in another context to assert they are the same thing. For example, the steering wheel in a car may be considered part of the interior of the car, while at the same time it is considered part of the 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 to assert that they are the same part.

A *feature value* is a shorthand for binding a usage to the result of an *expression* (see [7.18](#)) as part of the declaration of the usage. There are two types of feature value.

- A *fixed* feature value establishes the binding of the usage to the result of evaluating the given expression at the point of declaration of the usage. Such a binding cannot be overridden in a redefinition of the usage because, once asserted, a binding must be true at all times for all instances of the usage.
- A *default* feature value also includes an expression, but it does not immediately establish the binding of the usage. Instead, the evaluation of the expression and the binding of the usage is delayed until the instantiation of a definition or usage that features the original usage. Unlike a fixed feature value, a default feature value can be overridden in a redefinition of its original feature with a new feature value (fixed or default). In this case, the new overriding feature value is used instead of the original feature value for binding the redefining usage.

A *succession* is a binary connection that requires its two related usages to have values that are occurrences that happen completely separated in time, with the first occurrence happening before the second. Successions can be used to assert the ordering of any kind of occurrences in time, but are particularly useful for event occurrences (see [7.9](#)) and performances of actions (see [7.16](#)).

Flow Connection Usages

A *flow connection usage* is a connection usage that also represents the performance of a *transfer* of some *payload* between the values of connected usages, which must be occurrences. The transferred payload can be anything (attribute, item, part, etc.). The transfer is directed from the first connector end (the *source*) to the second connector end (the *target*). There are three kinds of flow connections.

1. A *message* is modeled as a flow connection usage that specifies that some transfer happens between the source and target ends, and can define the *payload* that is to be transferred. However, a message does not specify how the payload is to be obtained from the source or delivered to the target.
2. A *streaming flow connection* is modeled as a flow connection usage that not only specifies the source and target of a transfer (and, optionally, the payload), but also identifies the *source output* feature of the source usage from which the payload is obtained and the *target input* feature of the target usage to which the payload is to be delivered.
3. A *succession flow connection* is modeled as a *succession flow connection usage*, which is both a connection usage and a succession. A succession flow connection is specified in the same way as a streaming flow connection, but it adds the further constraint that the transfer source must complete before the transfer starts, and the transfer must complete before the target can start.

Messages are typically used to model abstract logical interaction between part usages in a certain context, which may be realized in a more detailed model using streaming or succession flow connections.

7.13.2 Abstract Syntax

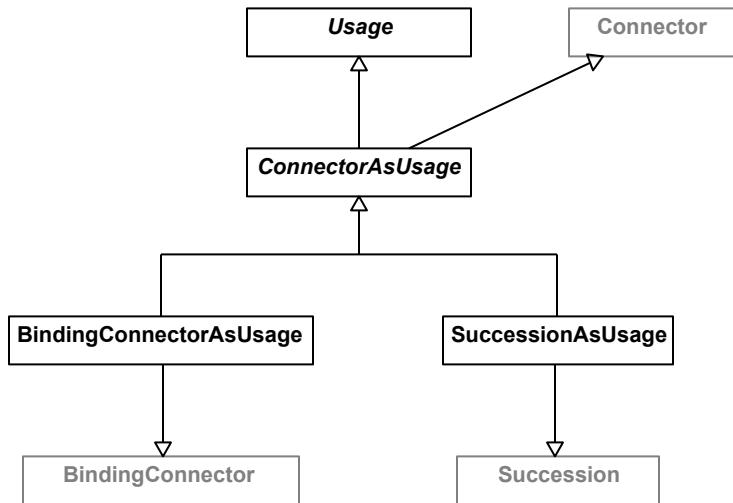


Figure 23. Connectors as Usages

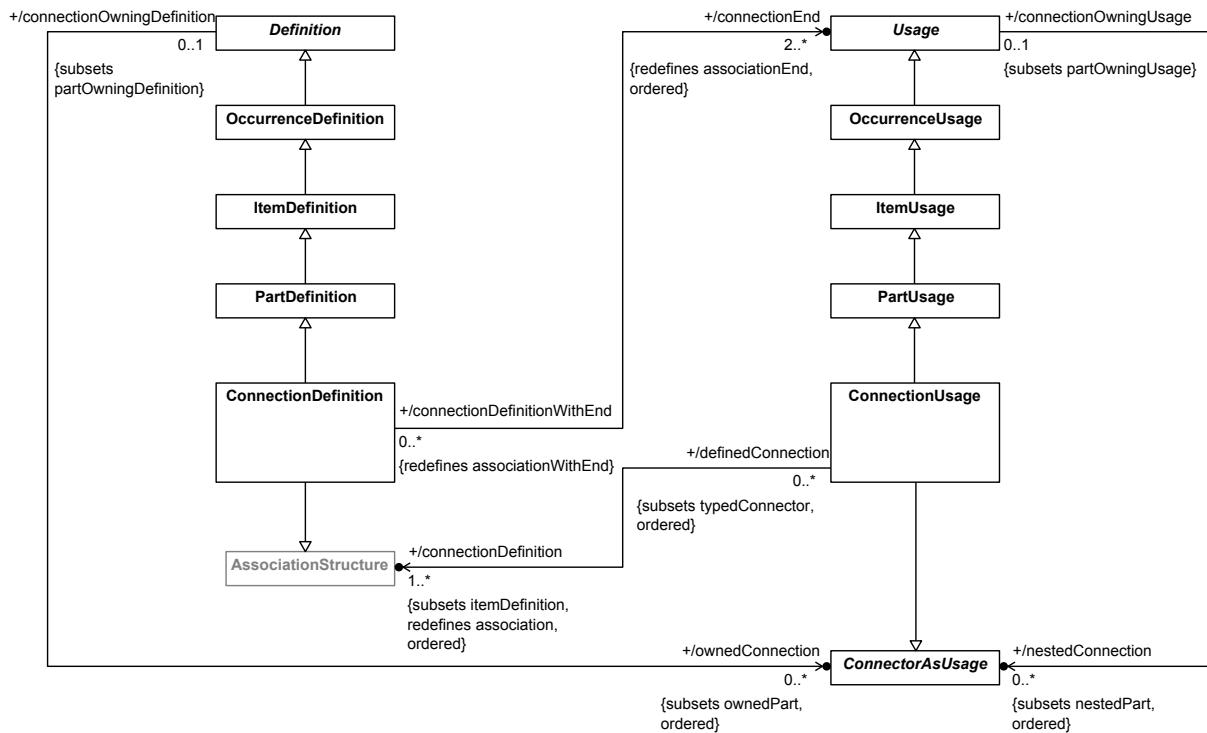


Figure 24. Connection Definition and Usage

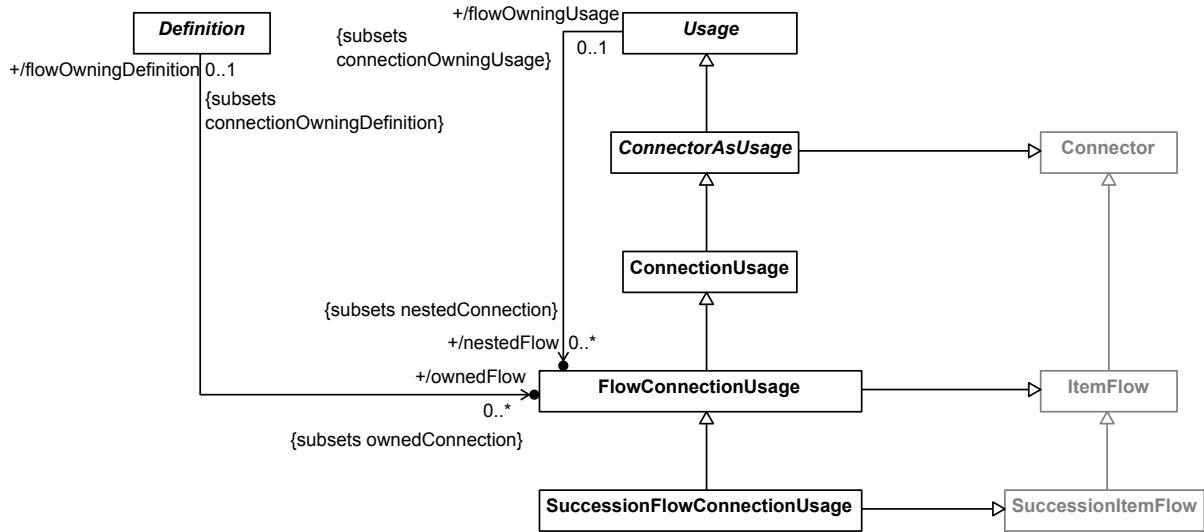


Figure 25. Flow Connections

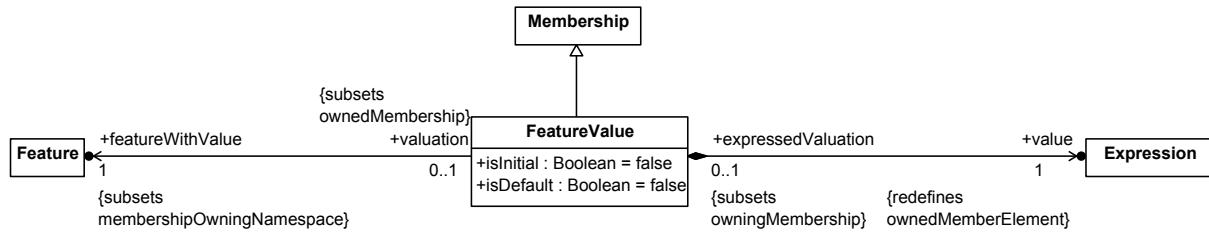


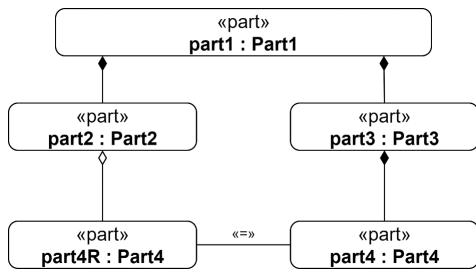
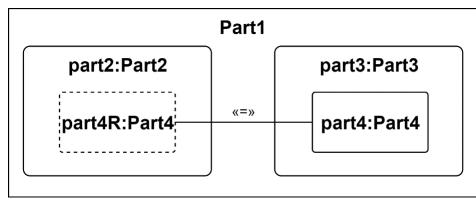
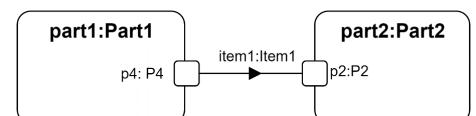
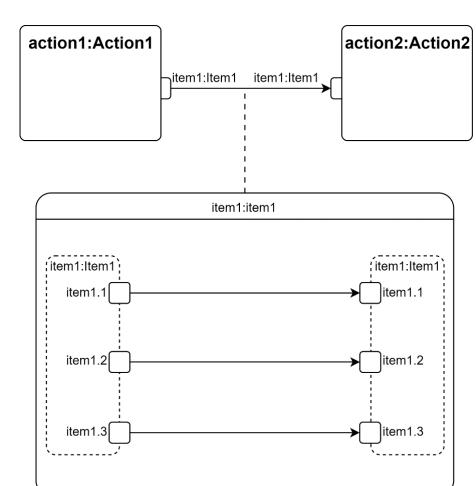
Figure 26. Feature Values

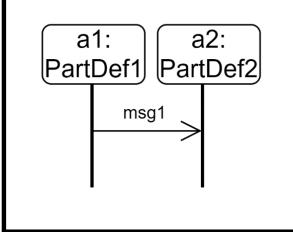
7.13.3 Notation

Table 17. Connections Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Connection Definition (can include any of the following compartments in the compartment stack: <i>actions, perform actions, allocations, attributes, connections, constraints, documentation, ends, individuals, metadata, namespaces, relationships, snapshots, timeslices, variants, variant elementusages</i>)	<pre>«connection def» ConnectionDef1 «connection def» ConnectionDef1 compartment stack ...</pre>	<pre>connection def ConnectionDef1; or connection def ConnectionDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
Connection (can include any of the following compartments in the compartment stack: <i>actions, perform actions, allocations, attributes, connections, constraints, documentation, ends, individuals, metadata, namespaces, relationships, snapshots, timeslices, variants, variant elementusages</i>)	<pre> <<connection>> connection1 : ConnectionDef1 <<connection>> connection1 : ConnectionDef1 compartment stack ... </pre>	connection connection1 : ConnectionDef1; or connection connection1 : ConnectionDef1 { /* members */ }
Connections Compartment		
Connection	<pre> part1:Part1 --- connection1: Connection1 --- part2:Part2 </pre>	connection connection1 : Connection1 connect part1 to part2;
Connection Definition	<pre> part def Part1 part1 0..1 Connection1 part2 1..* part def Part2 </pre>	
Connection Definition	<pre> part def Part1 part2R 0..1 {redefines part2} part def Part2R part2R 0..* Connection1 </pre>	
Connection Definition	<pre> part def Part1 part2 0..1 {redefines part2} part def Part2R part2 0..* Connection1 </pre>	

Element	Graphical Notation	Textual Notation
Binding Connection	 <pre> graph TD part1["part1 : Part1"] --> part2["part2 : Part2"] part1 --> part3["part3 : Part3"] part2 --> part4R["part4R : Part4"] part3 --> part4["part4 : Part4"] part4R <--> part4 </pre>	
Binding Connection	 <pre> graph TD subgraph Part1 [Part1] part2["part2:Part2"] part3["part3:Part3"] part4["part4:Part4"] part4R["part4R:Part4"] part4R -.-> part4 end </pre>	bind part4R = part4;
Item Flow	 <pre> graph LR action1["action1:Action1"] -- item1:item1 --> action2["action2:Action2"] </pre>	
Item Flow	 <pre> graph LR part1["part1:Part1"] -- item1:item1 --> part2["part2:Part2"] part1 --- p4[P4] part2 --- p2[P2] </pre>	
Item Flow as Node	 <pre> graph TD action1["action1:Action1"] -- item1:item1 --> action2["action2:Action2"] subgraph item1 [item1:item1] item1_1["item1.1"] item1_2["item1.2"] item1_3["item1.3"] item1_1 -- item1:item1 --> item1_2 item1_2 -- item1:item1 --> item1_3 end </pre>	
Item Flows Compartment		

Element	Graphical Notation	Textual Notation
Message		

7.14 Interfaces

7.14.1 Overview

An *interface definition* is a kind of connection definition (see [7.13](#)) whose ends are restricted to be port definitions (see [7.12](#)). An *interface usage* is a kind of connection that is usage of an interface definition. The ends of an interface usage are restricted to be port usages.

An interface is simply a connection all of whose ends are ports. As such, an interface facilitates the specification and reuse of compatible connections between parts. A logical interface may be defined that can then be specialized to represent more specific physical interfaces. For example, consider a *Power* interface definition between an *Appliance* and *Wall Power*. The *power* port on one end of the interface represents the *Appliance* connection point, and the *outlet* port on the other end represents the *Wall Power* connection point. This interface can then be specialized as necessary and used for connecting many different appliances to wall power.

When modeling physical interactions, an interface definition or usage can contain constraints (see [7.19](#)) to constrain the values of the features of the ports on its ends. For example, such features may be *across* and *through* variables, which are constrained by conservation laws across the interface (e.g., Kirchhoff's Laws). When specifying an interface between electrical components, the across and through variables are port features defined as *Voltage* and *Current* quantities, respectively. The feature values on either port are constrained such that the voltages must be equal, and the sum of the currents must equal zero.

7.14.2 Abstract Syntax

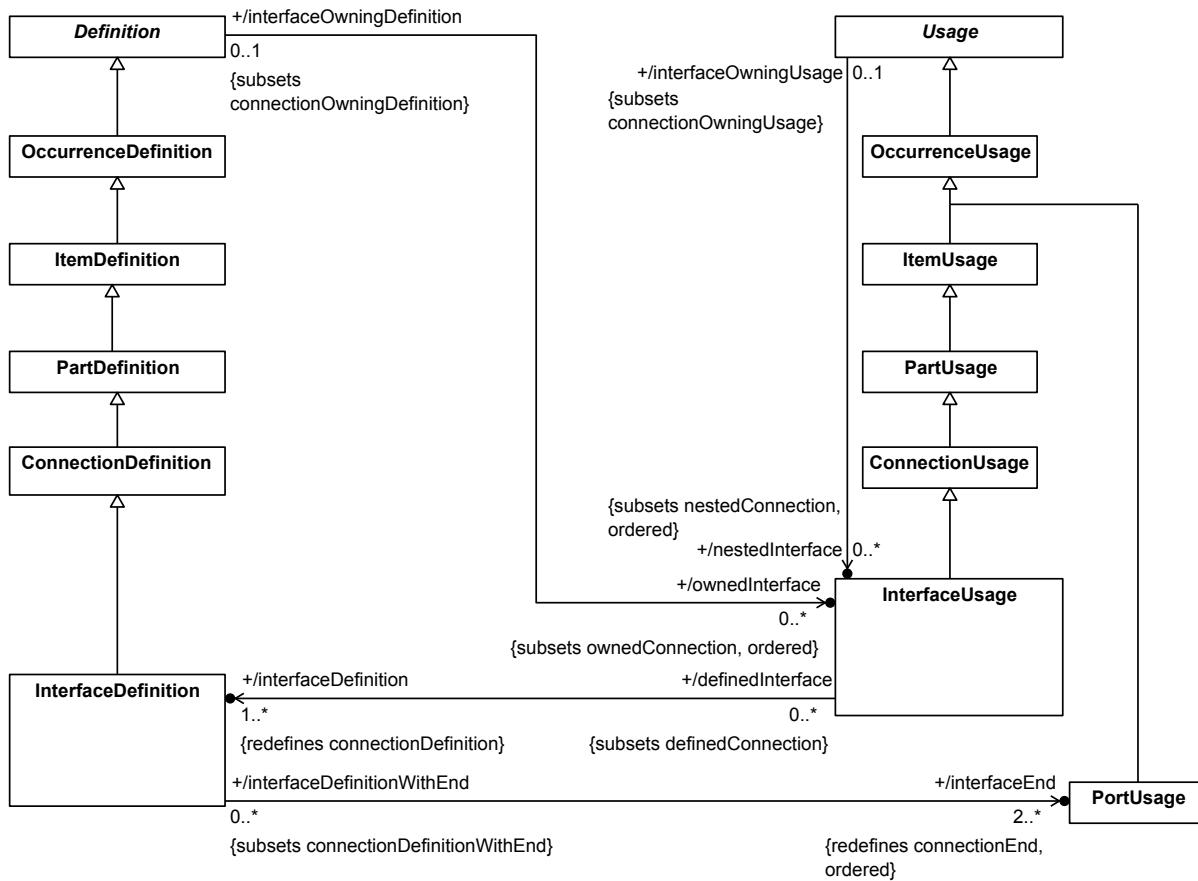
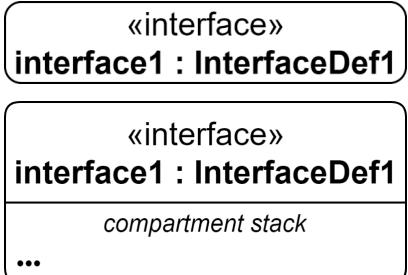
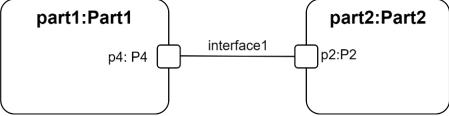
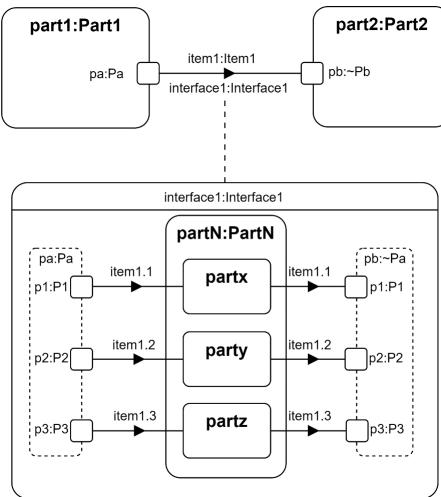


Figure 27. Interface Definition and Usage

7.14.3 Notation

Table 18. Interfaces Notation - Representative Usage

Element	Graphical Notation	Textual Notation
<p>Interface Definition (can include any of the following compartments in the compartment stack: <i>actions, allocations, attributes, constraints, documentation, individuals, interfaces, metadata, namespaces, relationships, snapshots, states, timeslices, variants, variant element usages</i>)</p>	<div style="border: 1px solid black; padding: 5px;"> «interface def» InterfaceDef1 </div> <div style="border: 1px solid black; padding: 5px;"> «interface def» InterfaceDef1 </div> <div style="border: 1px solid black; padding: 5px;"> <i>compartment stack</i> <i>...</i> </div>	<pre>interface def InterfaceDef1; or interface def InterfaceDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
Interface (can include any of the following compartments in the compartment stack: <i>actions, allocations, attributes, constraints, documentation, individuals, interfaces, metadata, namespaces, relationships, snapshots, states, timeslices, variants, variant elementusages</i>)		<pre> interface interface1 : InterfaceDef1; or interface interface1 : InterfaceDef1 { /* members */ } </pre>
Interfaces Compartment		
Interface		
Interface as Node (with item flow)		

7.15 Allocations

7.15.1 Overview

An *allocation definition* is a *connection definition* (see 7.13) that specifies that a target element is responsible for realizing some or all of the intent of the source element. An allocation is a usage of one or more allocation definitions. An allocation definition or usage can be further refined using nested allocation usages that provide 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 (see [7.20](#)), performing actions (see [7.16](#)) and exhibiting states (see [7.17](#)). Modelers can also create specialized allocation definitions to reflect conventions for allocation on specific projects or within certain system models.

Submission 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 the final submission 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.

7.15.2 Abstract Syntax

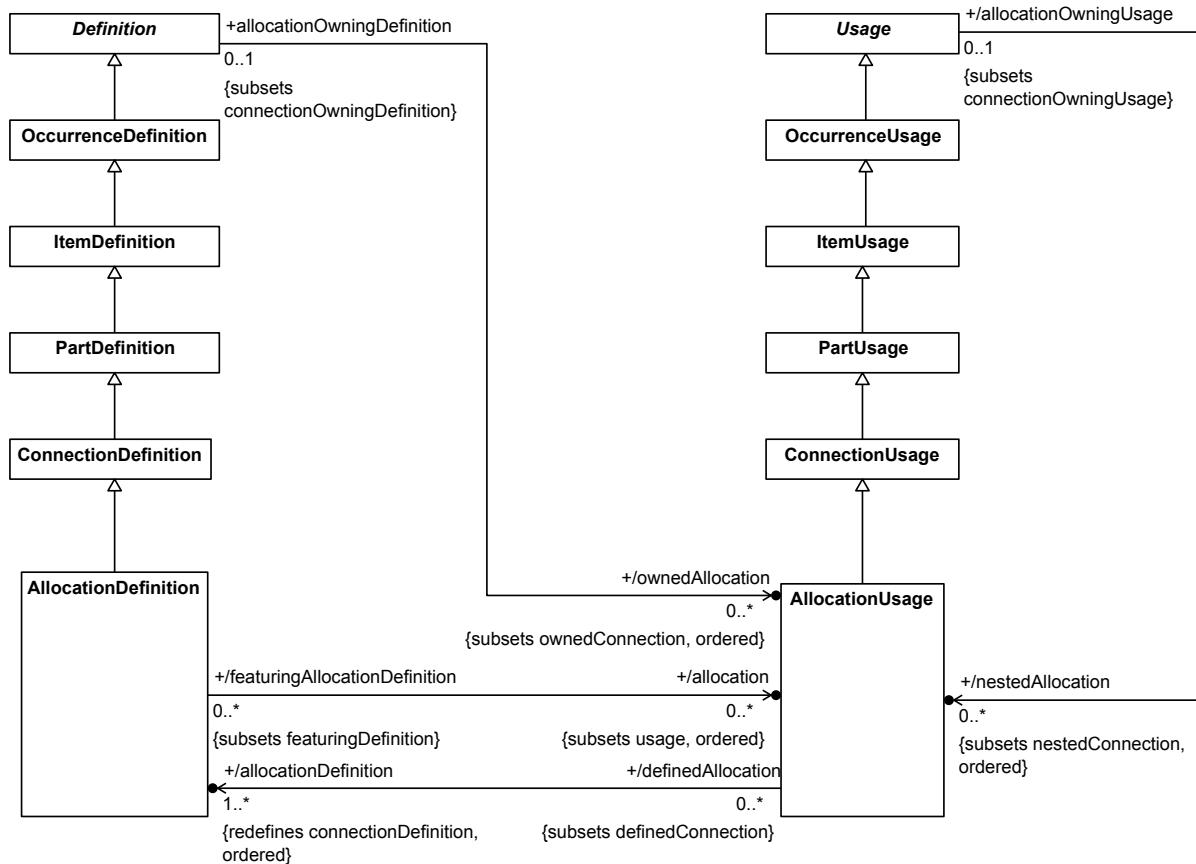


Figure 28. Allocation Definition and Usage

7.15.3 Notation

Table 19. Allocations Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Allocation Definition (can include any of the following compartments in the compartment stack: <i>allocations, attributes, constraints, documentation, individuals, metadata, namespaces, relationships, snapshots, timeslices, variants, variant elementusages</i>)		allocation def AllocationDef1; or allocation def AllocationDef1 { /* members */ }
Allocation (can include any of the following compartments in the compartment stack: <i>allocations, attributes, constraints, documentation, individuals, metadata, namespaces, relationships, snapshots, timeslices, variants, variant elementusages</i>)		allocation allocation1: AllocationDef1; or allocation allocation1: AllocationDef1 { /* members */ }
Allocations Compartment		
Allocation		
Allocation (with sub allocation)		

7.16 Actions

7.16.1 Overview

Action Definition and Usage

An *action definition* is a kind of occurrence definition (see 7.9) that classifies action performances. An *action usage* is a kind of occurrence usage that is a usage or one or more action definitions and, so, has action performances as its values.

An action definition may have features with directions **in**, **out** or **inout** that act as the *parameters* of the action. Features with direction **in** or **inout** are input parameters, and features with direction **out** or **inout** are output parameters. An action usage inherits the parameters of its definitions, if any, and it can also define its own parameters to augment or redefine those of its definitions.

Actions are occurrences over time that can coordinate the performance of other actions and generate effects on items and parts involved in the performance (including those items' existence and relation to other things). The features of an action definition or usage that are themselves action usages specify the performance of the action in terms of the performances of each of the subactions. If an action has parameters, then it may also transform the values of its input parameters into values of its output parameters.

Action definitions and usages follow the same patterns that apply to structural elements (see [7.6](#)). 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 subclassified, and an action usage can be subsetted or redefined. This provides enhanced flexibility to modify a hierarchy of action usages to adapt to its context.

Performed Actions

A *perform action usage* is an action usage that specifies that an action is performed by the owner of the performed action usage. A perform action usage is referential, which allows the performed action behavior to be defined in a different context than that of the performer (perhaps by an action usage in a functionally decomposed action tree). However, if the owner of the perform action usage is an occurrence, then the referenced action performance must be carried out entirely within the lifetime of the performing occurrence.

In particular, a perform action usage can be a feature of a part definition or usage, specifying that the referenced action is performed by the containing part. The values of a perform action usage are then references to the performances of the action that are carried out by the part during its lifetime.

A perform action usage can also be a feature of an action definition or usage. In this case, the perform action usage represents a "call" from the containing action to the performed action.

Sequencing of Actions

Since action usages are kinds of occurrence usages, their ordering can be specified using successions (see [7.13](#)). However, a succession between action usages may, additionally, have a *guard condition*, represented as a Boolean expression (see [7.18](#)). If the succession has a guard, then the time ordering of the source and target of the succession is only asserted when 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 successions (with or without guards). The kinds of control nodes include the following.

- A *fork node* has one incoming succession and one or more outgoing successions. The actions connected to the outgoing successions cannot start until the action connected to the incoming succession has completed.
- A *join node* has one or more incoming successions and one outgoing succession. The action connected to the outgoing succession cannot start until all the actions connected to the incoming successions have completed.
- A *decision node* has one incoming succession and one or more outgoing successions. Exactly one of the actions connected to an outgoing succession can start after the action connected to the incoming succession has completed. Which of the downstream actions is performed can be controlled by placing guards on the outgoing successions.

- A *merge node* has one or more incoming successions and one outgoing succession. The action connected to the outgoing succession cannot start until any one of the actions connected to an incoming succession has completed.

Flows Between Actions

An output parameter of one action usage may be *bound* to the input parameter of another action usage (see [7.13](#) on binding). Such a binding indicates that the values of the target input parameter will always be the same as the values of the source output parameter. If the two actions are performed concurrently, then this equivalence will be maintained over time throughout their performances.

The binding of action parameters, however, does not model the case when there is an actual *transfer* of items between the actions that may itself take time or have other modeled properties. Such a transfer can be more properly modeled using a flow connection between the two action usages (see [7.13](#)), in which the transfer source output is an output parameter of the source action usage and the transfer target input is the input parameter of the target action usage. A streaming flow connection represents a flow in which the transfer can be ongoing while both the source and target action are being performed. A succession flow connection represents a flow that imposes the additional succession constraint that the transfer cannot begin until the source action completes and the target action cannot start until the transfer has completed.

Transfers can also be performed using *send and accept action usages*. 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 usage contained in some source part or action, while the target is given by an accept action usage in some destination part or action (which may be the same as or different than the source). A send action usage 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 [7.12](#) and [7.14](#) on interfaces between ports). An accept action usage specifies the type of values that can be received by the action. When a send action performed in the source is matched with a compatible accept action performed in the destination, then the transfer of values from the origin to the destination can be completed.

7.16.2 Abstract Syntax

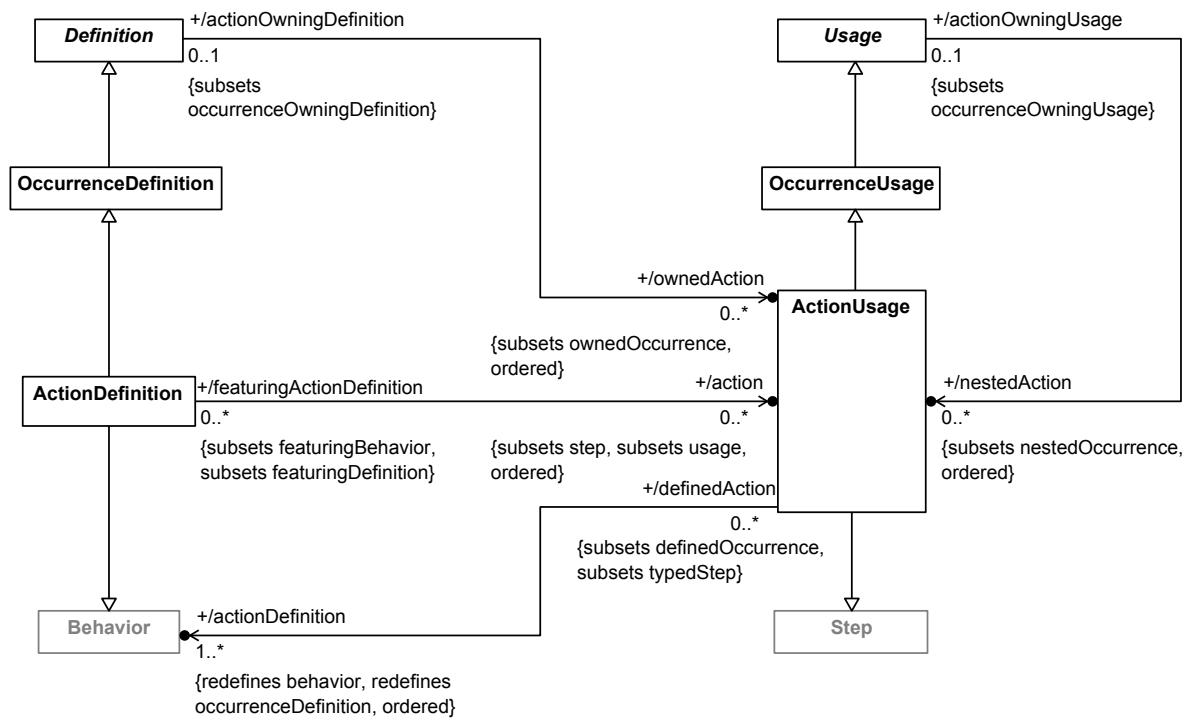


Figure 29. Action Definition and Usage

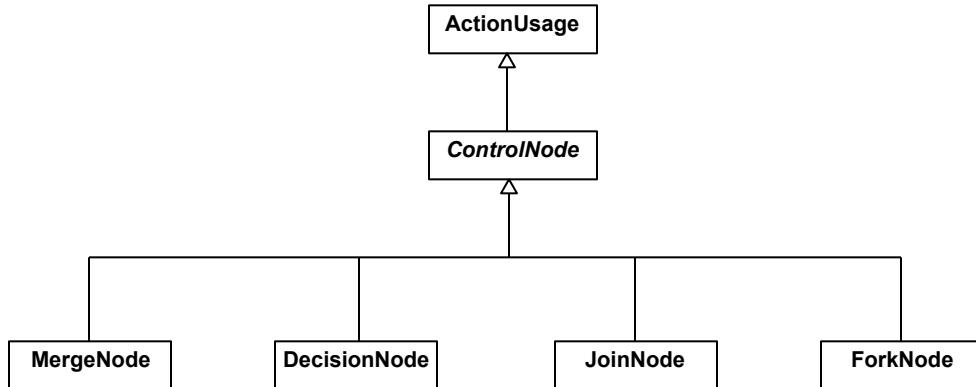


Figure 30. Control Nodes

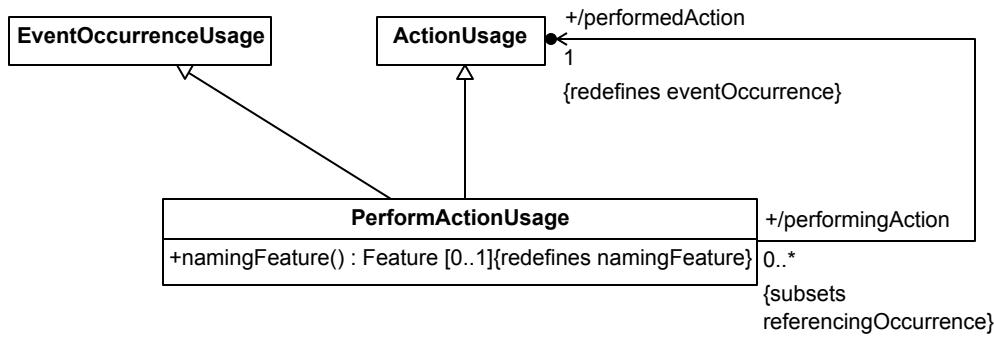


Figure 31. Action Performance

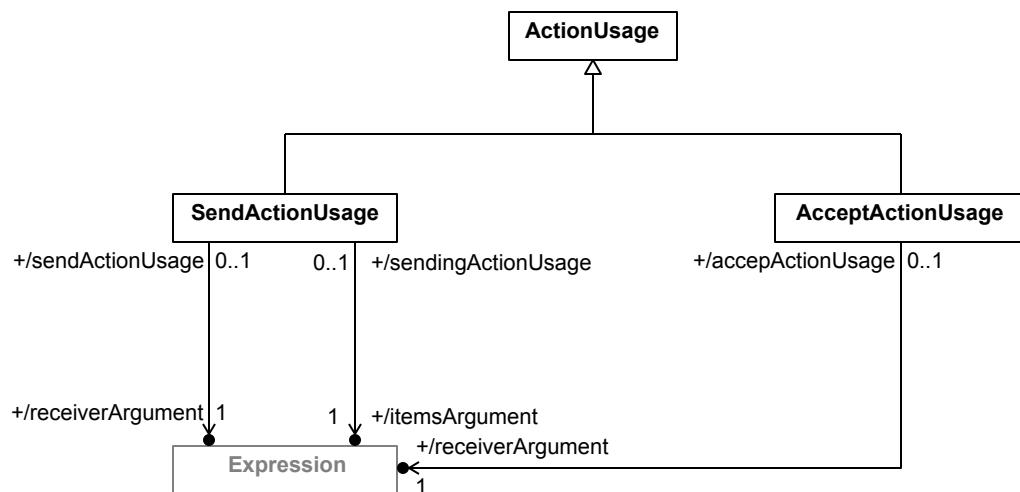
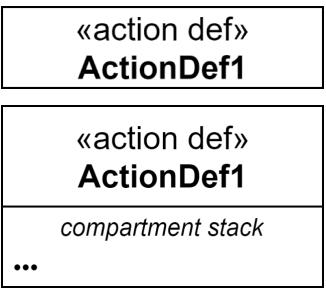
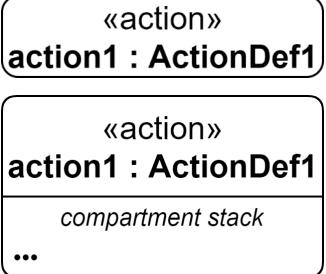
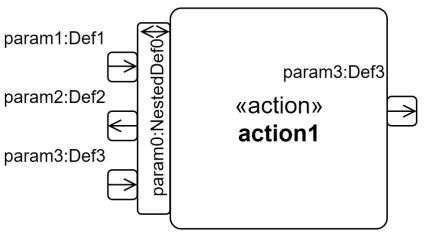
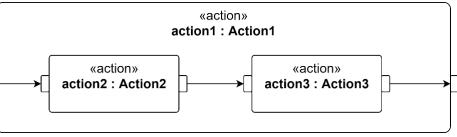


Figure 32. Send and Accept Actions

7.16.3 Notation

Table 20. Actions Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Action Definition (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>include actions</i> , <i>allocations</i> , <i>analyses</i> , <i>attributes</i> , <i>body</i> , <i>calcs</i> , <i>constraints</i> , <i>assert</i> <i>constraints</i> , <i>documentation</i> , <i>individuals</i> , <i>items</i> , <i>metadata</i> , <i>namespaces</i> , <i>parameters</i> , <i>relationships</i> , <i>satisfy</i> <i>requirements</i> , <i>snapshots</i> , <i>timeslices</i> , <i>variants</i> , <i>variant</i> <i>elementusages</i>)		<pre data-bbox="902 650 1269 861">action def ActionDef1; or action def ActionDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
<p>Action (can include any of the following compartments in the compartment stack: <i>actions</i>, <i>include actions</i>, <i>allocations</i>, <i>analyses</i>, <i>attributes</i>, <i>body</i>, <i>calcs</i>, <i>constraints</i>, <i>assert</i>, <i>constraints</i>, <i>documentation</i>, <i>individuals</i>, <i>items</i>, <i>metadata</i>, <i>namespaces</i>, <i>parameters</i>, <i>performed by</i> (<i>Usage Only</i>), <i>relationships</i>, <i>satisfy requirements</i>, <i>snapshots</i>, <i>timeslices</i>, <i>variants</i>, <i>variant element usages</i>)</p>		<pre data-bbox="902 587 1367 804">action action1 : ActionDef1; or action action1 : ActionDef1 { /* members */ }</pre>
Action with Parameters		
Action with Action Compartment-Graphical		

Element	Graphical Notation	Textual Notation
Actions Compartment	<pre> <i>actions</i> action1:ActionDef1 [1..*] ordered nonunique ^action2:ActionDef2 (<i>in</i> :ParamDef1, <i>out</i> :ParamDef2) action3R:ActionDef3R redefines action3 action4R:ActionDef4R:>>action4 :>>action5 action6S:ActionDef6S [m] subsets action6 [n] action7S:ActionDef7S [m] > action7 [n] action8R = action8 ref action9:ActionDef9 perform action10 action11 action11.1 action11.2 </pre>	
Perform Actions Compartment	<pre> <i>perform actions</i> action1:ActionDef1 [1..*] ordered nonunique ^action2:ActionDef2 (<i>in</i> :ParamDef1, <i>out</i> :ParamDef2) action3R:ActionDef3R redefines action3 action4R:ActionDef4R:>>action4 :>>action5 action6S:ActionDef6S [m] subsets action6 [n] action7S:ActionDef7S [m] > action7 [n] action8R = action8 action11 action11.1 action11.2 </pre>	
Parameters Compartment	<pre> <i>parameters</i> <i>in</i> param1:ParamDef1 [1..*] ordered nonunique <i>out</i> param2:ParamDef2 <i>in-out</i> param3:ParamDef3 <i>return</i> param4:ParamDef4 <i>in</i> ^param5:ParamDef5 <i>in</i> param6R:ParamDef6R redefines param6 <i>in</i> param7R:ParamDef7R:>>param7 <i>in</i> :>>param8 <i>in</i> param9S:ParamDef9S [m] subsets param9 [n] <i>in</i> param10S:ParamDef10S [m] > param10 [n] <i>in</i> param11 :ParamDef11 = expression1 </pre>	
Succession		then
Perform		part part1 { perform action1; }

Element	Graphical Notation	Textual Notation
Perform Action	<pre> graph TD part1["<<part>> part1 : Part1"] part1 -- "<<perform>>" --> action1["<<action>> action1 : Action1"] </pre>	part def Part1 { perform action action1; }
Succession	<pre> graph LR action1["action1:Action1"] --> action2["action2:Action2"] </pre>	succession action1 then action2; or action1; then action2;
Performed By Compartment		

7.17 States

7.17.1 Overview

States

A *state definition* is a kind of action definition (see [7.16](#)) that defines the conditions under which other actions can execute. A state usage 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 definition or usage can contain specially identified action usages that are only performed while the state is activated.

- An *entry action* starts when the state is activated.
- A *do action* starts after the entry action completes and continues while the state is active.
- An *exit action* starts when the state is exited, and the state becomes inactive once the exit action is completed.

State definitions and usages follow the same patterns that apply to structural elements (see [7.6](#)). 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 subersetted and redefined. This provides enhanced flexibility to modify a state hierarchy to adapt to its context.

Exhibited States

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.

An *exhibit state usage* is a state usage that specifies that a state is exhibited by the owner of the exhibit state usage. An exhibit state usage is referential, which allows the exhibited state behavior to be defined in a different context than that of the exhibitor (perhaps by a state usage in a state decomposition hierarchy). However, if the owner of the

exhibit state usage is an occurrence, then the referenced state performance must be carried out entirely within the lifetime of the performing occurrence.

In particular, an exhibit state usage can be a feature of a part definition or usage, specifying that the referenced state is exhibited by the containing part. Typically, the exhibited state and its substates will reflect conditions of the exhibiting part, such as the operating states of a vehicle. The values of the exhibit state usage are then references to occurrences of the state when the exhibiting part is "in" that state.

Transitions

State usages can be connected by *transition usages*, which can activate and deactivate the state usages. 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. The trigger of a transition usage is an accept action usage (see [7.16](#)), which accepts an incoming transfer. The transition usage can contain a *guard condition*, which is a Boolean expression (see [7.18](#)) that must evaluate to `true` for the transition to occur. In addition, a transition usage may specify an *effect action usage* that starts if the transition is triggered, after the source state is deactivated, and must complete before the target state is activated. If the triggering transfer of a transition has a payload, then this payload is available for use in the guard condition and effect action of the transition, and after the transition completes.

Submission Note. The language currently only supports triggering a transition on receipt of a transfer sent with a send action usage. Support for triggering transitions by change and time events will be included in the final submission. Consideration will also be given to allowing the declaration of transition definitions.

Parallel States

A *parallel state* is one whose substates are performed concurrently. As such, no transitions are allowed between the substates of a parallel state. In contrast, if a non-parallel state has substates then, exactly one of the substates shall be active at any point in time in the lifetime of the containing state after completion of the entry action (if any).

7.17.2 Abstract Syntax

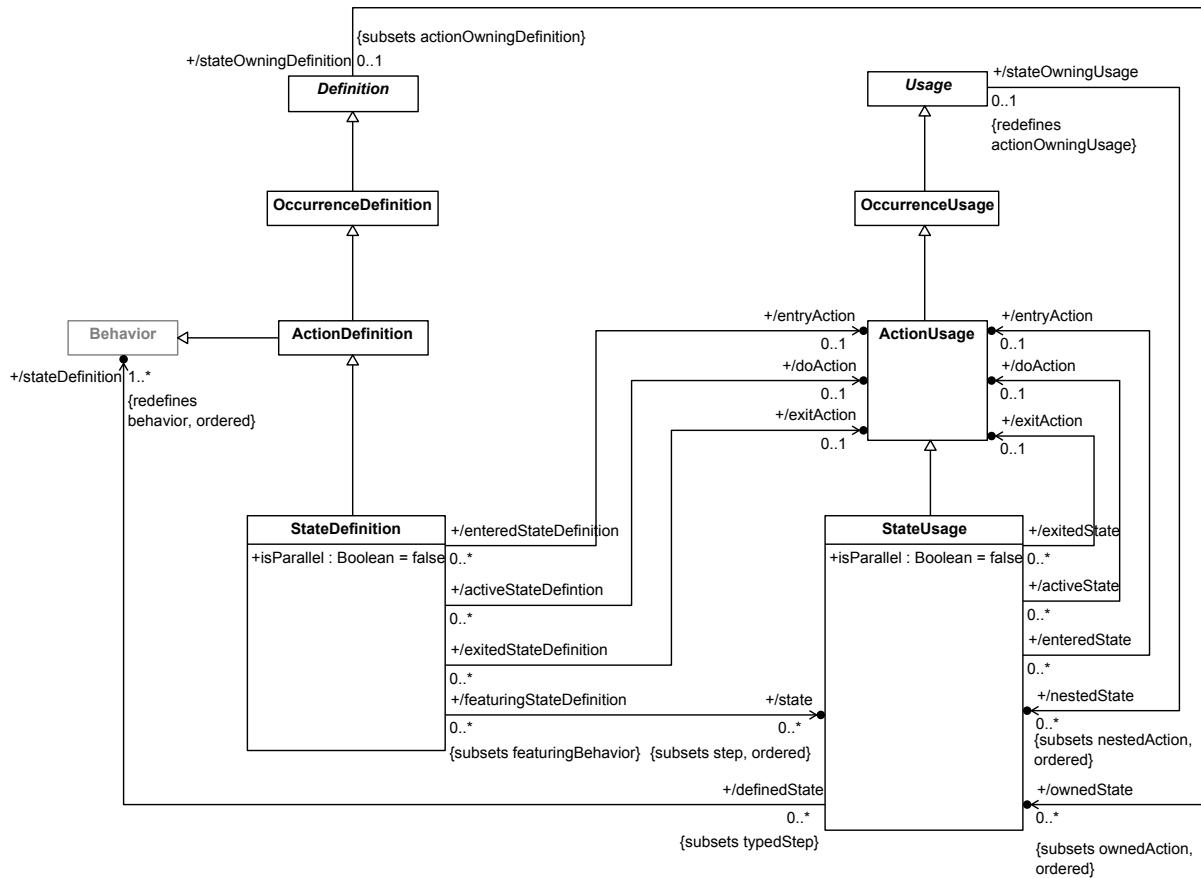


Figure 33. State Definition and Usage

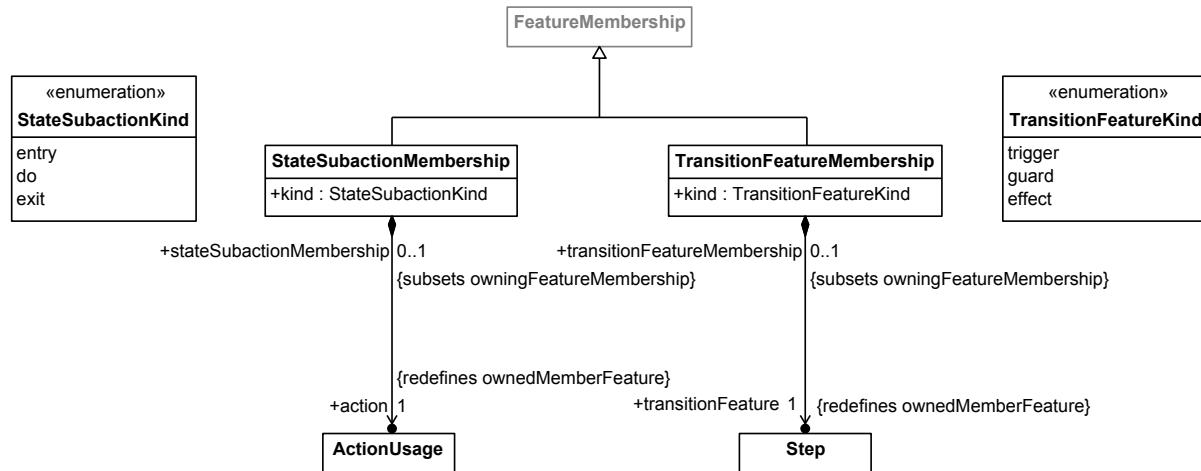


Figure 34. State Membership

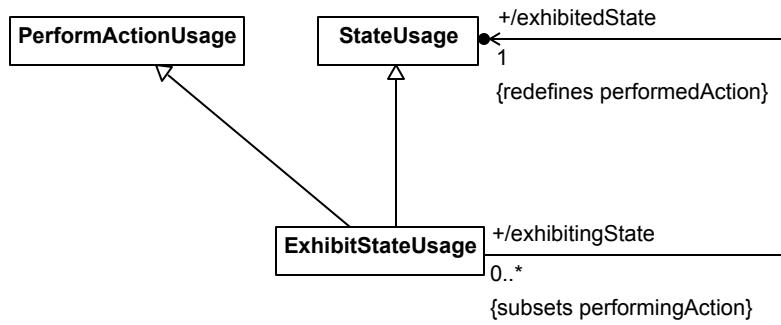


Figure 35. State Exhibition

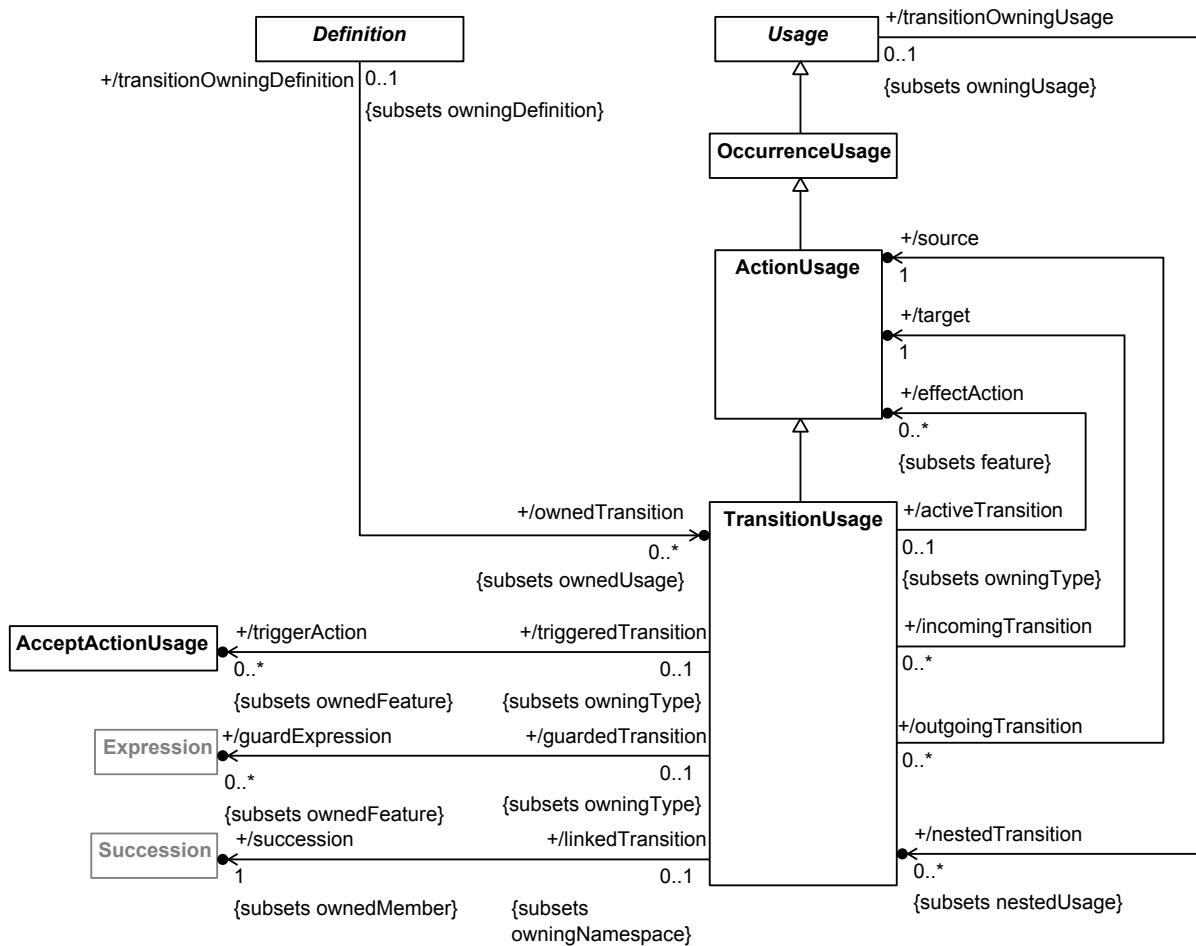
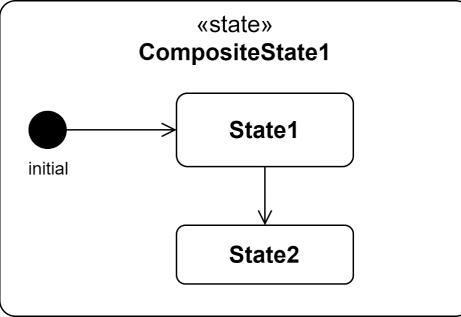


Figure 36. Transition Usage

7.17.3 Notation

Table 21. States Notation - Representative Usage

Element	Graphical Notation	Textual Notation
<p>State Definition (can include any of the following compartments in the compartment stack: <i>actions</i>, <i>allocations</i>, <i>attributes</i>, <i>constraints</i>, <i>assert constraints</i>, <i>documentation</i>, <i>exhibited by</i>, <i>individuals</i>, <i>metadata</i>, <i>namespaces</i>, <i>parameters</i>, <i>relationships</i>, <i>satisfy requirements</i>, <i>snapshots</i>, <i>states</i>, <i>timeslices</i>, <i>variants</i>, <i>variant elementusages</i>)</p>	 	<pre>state def StateDef1; or state def StateDef1 { /* members */ }</pre>
<p>State (can include any of the following compartments in the compartment stack: <i>actions</i>, <i>allocations</i>, <i>attributes</i>, <i>constraints</i>, <i>assert constraints</i>, <i>documentation</i>, <i>exhibited by</i>, <i>individuals</i>, <i>metadata</i>, <i>namespaces</i>, <i>parameters</i>, <i>relationships</i>, <i>satisfy requirements</i>, <i>snapshots</i>, <i>states</i>, <i>exhibited by (Usage Only)</i>, <i>timeslices</i>, <i>variants</i>, <i>variant elementusages</i>)</p>	 	<pre>state state1 : StateDef1; or state def state1 : StateDef1 { /* members */ }</pre>
State		
State with Sequential States Compartment-Graphical		

Element	Graphical Notation	Textual Notation
State with Parallel States Compartment-Graphical	<pre> «state» CompositeState2 isParallel initial --> State1 initial --> State2 </pre>	
Transition	<pre> state1 : State1 trigger1[guard1] / action1 state2 : State2 </pre>	
Exhibit	<pre> part1 : Part1 «exhibit» state1 : State1 </pre>	<pre> part part1 { exhibit state1; } </pre>
Exhibit State	<pre> part1 : Part1 «exhibit» state1 : State1 </pre>	<pre> part def Part1 { exhibit state state1; } </pre>
States Compartment	<pre> states state1:StateDef1 [1..*] ordered nonunique ^state2:StateDef2 state3R:StateDef3R redefines state3 state4R:StateDef4R:>>state4 :>>state5 state6S:StateDef6S [m] subsets state6 [n] state7S:StateDef7S [m] :> state7 [n] state8R = state8 ref state9:StateDef9 exhibit state10 state11 state11.1 state11.2 </pre>	

Element	Graphical Notation	Textual Notation
Exhibit States Compartment	<pre> exhibit states state1:StateDef1 [1..*] ordered nonunique ^state2:StateDef2 state3R:StateDef3R redefines state3 state4R:StateDef4R:>>state4 :>>state5 state6S:StateDef6S [m] subsets state6 [n] state7S:StateDef7S [m] >> state7 [n] state8R = state8 state11 state11.1 state11.2 </pre>	
Exhibited By Compartment		

7.18 Calculations

7.18.1 Overview

A *calculation definition* is a kind of action definition (see [7.16](#)) all of whose parameters have direction in, except for one with direction out called the *result parameter*. A calculation definition specifies a reusable computation that returns a result in the result parameter. A *calculation usage* is an action usage that is a usage of a calculation definition.

In addition to its parameters, a calculation definition or usage may have features that are calculation or action usages that carry out steps in the computation of the result of the calculation. The calculation may also have other features that are used to record intermediate results in the computation. The final result is specified as an *expression* written in terms of the input parameters of the calculation and any intermediate results.

KerML includes extensive syntax for constructing expressions, including traditional operator notations for functions in the Kernel Model Library, which is adopted in its entirety into SysML. In addition a calculation is also a KerML function and a calculation usage is itself a KerML expression. This allows a calculation definition or usage to also be invoked using the notation of a KerML invocation expression. (See the KerML Specification [KerML] for a complete description of the KerML expression sublanguage.)

Calculation definitions are often used to define mathematical functions, in which case the defined computation should be *pure*. A pure calculation has the following properties:

1. Two invocations of the calculation definition with the same values for the input parameters always produce the same values for the result parameter.
2. The performance of the calculation does not produce any side effects (that is, it does not effect any occurrence that is not a composite part of its performance or that of a subaction or subcalculation).

Any subcalculations or subactions of a pure calculation must also be pure, including the final expression computing the result. Further, the inputs of a pure calculation should either be attributes or the calculation should not rely on features of input occurrences that may change from one invocation of the calculation definition to another.

7.18.2 Abstract Syntax

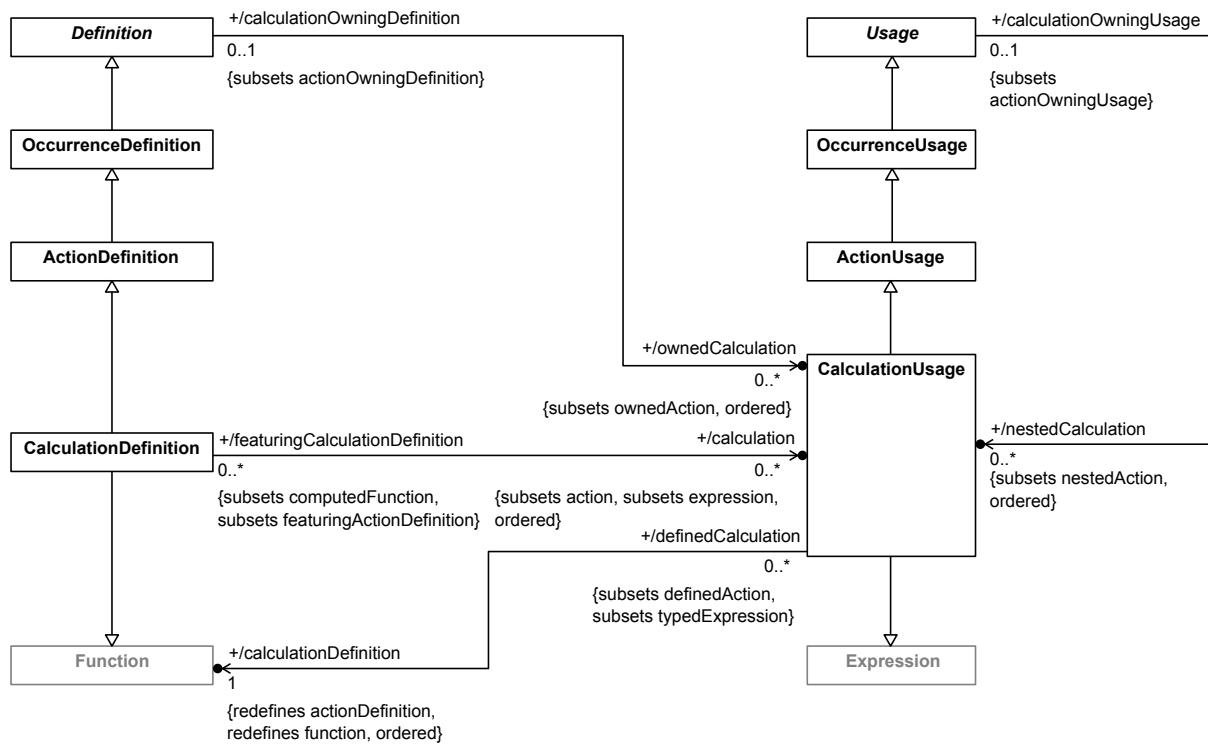
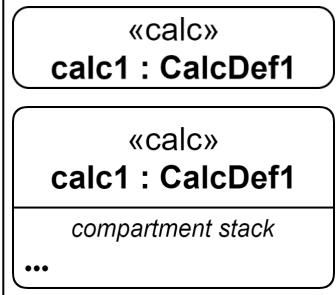


Figure 37. Calculation Definition and Usage

7.18.3 Notation

Table 22. Calculations Notation - Representative Usage

Element	Graphical Notation	Textual Notation
<p>Calc Definition (can include any of the following compartments in the compartment stack: <i>actions</i>, <i>allocations</i>, <i>analyses</i>, <i>attributes</i>, <i>body</i>, <i>calcs</i>, <i>constraints</i>, <i>documentation</i>, <i>individuals</i>, <i>metadata</i>, <i>namespaces</i>, <i>parameters</i>, <i>relationships</i>, <i>snapshots</i>, <i>timeslices</i>, <i>variants</i>, <i>variant element usages</i>)</p>	<div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> «calc def» CalcDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> «calc def» CalcDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> <i>compartment stack</i> ... </div>	<pre> calc def CalcDef1; or calc def CalcDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
<p>Calc (can include any of the following compartments in the compartment stack: <i>actions</i>, <i>allocations</i>, <i>analyses</i>, <i>attributes</i>, <i>body</i>, <i>calcs</i>, <i>constraints</i>, <i>documentation</i>, <i>individuals</i>, <i>metadata</i>, <i>namespaces</i>, <i>parameters</i>, <i>performed by</i> (<i>Usage Only</i>), <i>relationships</i>, <i>snapshots</i>, <i>timeslices</i>, <i>variants</i>, <i>variant elementusages</i>)</p>		<pre>calc calc1 : CalcDef1; or calc calc1 : CalcDef1 { /* members */ }</pre>
<p>Body Compartment</p>		

7.19 Constraints

7.19.1 Overview

Constraint Definition and Usage

A *constraint definition* is a kind of occurrence definition (see [7.9](#)) that defines a logical predicate. Similarly to a calculation definition (see [7.18](#)), a constraint definition may have parameters with direction **in**. A calculation always has an implicit Boolean-value result parameter with direction **out**. A constraint usage is an occurrence usage that is the usage of a constraint definition.

Also similarly to a calculation, a constraint definition or usage may have features that are calculation or action usages that carry out steps in the computation of the result of the calculation. The constraint may also have other features that are used to record intermediate results in the computation. The final result is specified as an expression written in terms of the input parameters of the calculation and any intermediate results. In addition a constraint definition is also a KerML predicate and a constraint usage is a KerML Boolean expression, which allows a constraint definition or usage to also be invoked using the notation of a KerML invocation expression.

For a given set of input parameter values, a constraint usage is *satisfied* if its expression evaluates to `true` and is *violated* otherwise. 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`. E.g., if `x` is 3 and `y` is 5, then the expression `x < y` evaluates to `true`, and the constraint is satisfied. 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 constraint usage that is a feature of another definition or usage may also directly reference features of its containing context, in which case it may be used to effectively constrain the values of those features. In a context

with the features `bolt diameter` and `hole diameter`, a constraint usage may be defined directly without parameters using the expression `{'bolt diameter' < 'hole diameter'}`.

Asserted Constraints

In general, a constraint may be satisfied sometimes and violated other times. However, an *assert constraint usage* asserts that the result of a given constraint must be `true`, which requires that the asserted 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 `{'fuel level' > 0}` may be violated if the `fuel level` equals zero, but the model is still valid.

An assert constraint usage can also be *negated*, which means that the given constraint is asserted to be `false` rather than `true`. A negated assert constraint usage can be used to assert that some condition must never happen for the model to be valid.

7.19.2 Abstract Syntax

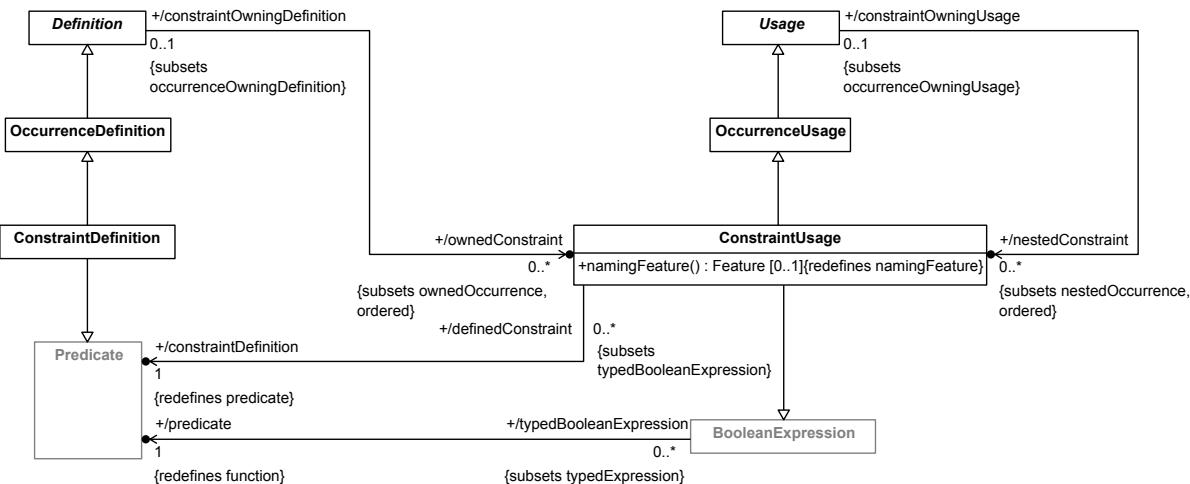


Figure 38. Constraint Definition and Usage

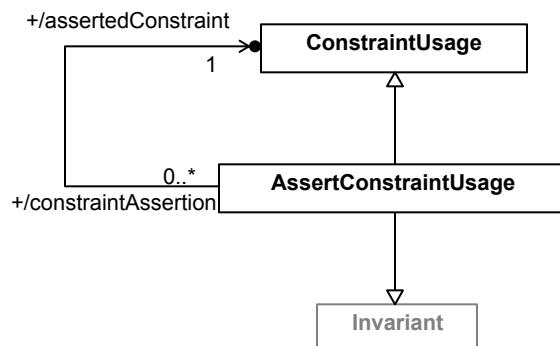


Figure 39. Constraint Assertion

7.19.3 Notation

Table 23. Constraints Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Constraint Definition (can include any of the following compartments in the compartment stack: <i>allocations, attributes, body, constraints, documentation, individuals, metadata, namespaces, parameters, relationships, snapshots, timeslices, variants, variant elementusages</i>)	<div style="border: 1px solid black; padding: 10px;"> <p>«constraint def»</p> <p>ConstraintDef1</p> </div> <div style="border: 1px solid black; padding: 10px; margin-top: 10px;"> <p>«constraint def»</p> <p>ConstraintDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>constraint def ConstraintDef1; or constraint def ConstraintDef1 { /* members */ }</pre>
Constraint (can include any of the following compartments in the compartment stack: <i>allocations, attributes, body, constraints, documentation, individuals, metadata, namespaces, parameters, performed by (Usage Only), relationships, snapshots, timeslices, variants, variant elementusages</i>)	<div style="border: 1px solid black; padding: 10px;"> <p>«constraint»</p> <p>constraint1 : ConstraintDef1</p> </div> <div style="border: 1px solid black; padding: 10px; margin-top: 10px;"> <p>«constraint»</p> <p>constraint1 : ConstraintDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>constraint constraint1 : ConstraintDef1; or constraint constraint1 : ConstraintDef1 { /* members */ }</pre>
Constraints Compartment	<div style="border: 1px solid black; padding: 10px;"> <pre> constraints constraint1:ConstraintDef1 [1..*] ordered nonunique ^constraint2:ConstraintDef2 constraint3R:ConstraintDef3R redefines constraint3 constraint4R:ConstraintDef4R:>>constraint4 :>>constraint5 constraint6S:ConstraintDef6S [m] subsets constraint6 [n] constraint7S:ConstraintDef7S [m] > constraint7 [n] constraint8R = constraint8 ref constraint9:ConstraintDef9 assert constraint10 assert {constraint expression1} constraint11 constraint11.1 constraint11.2 require constraint 12 assume constraint 13 precondition {constraint expression2} postcondition {constraint expression3} </pre> </div>	

Element	Graphical Notation	Textual Notation
Assert Constraints Compartment	<pre> assert constraints constraint1:ConstraintDef1 [1..*] ordered nonunique ^constraint2:ConstraintDef2 constraint3R:ConstraintDef3R redefines constraint3 constraint4R:ConstraintDef4R:>>constraint4 :>>constraint5 constraint6S:ConstraintDef6S [m] subsets constraint6 [n] constraint7S:ConstraintDef7S [m] > constraint7 [n] constraint8R = constraint8 ref constraint9:ConstraintDef9 {constraint expression} constraint11 constraint11.1 constraint11.2 precondition {constraint expression2} postcondition {constraint expression3} </pre>	

7.20 Requirements

7.20.1 Overview

Requirements

A *requirement definition* is a kind of constraint definition (see [7.19](#)) that specifies stakeholder-imposed constraints that a design solution must satisfy to be a valid solution. A requirement definition contains one or more features that are constraint usages designated as the *required constraints*. These may be specified informally using text statements (commonly known as "shall" statements) or more formally using constraint expressions. A requirement definition may also optionally include *assumed constraints*. The required constraints of a requirement only apply if all the assumed constraints are satisfied.

A *requirement usage* is a kind of constraint usage (see [7.19](#)) that is a usage of a requirement definition in some context. The context for multiple requirements can be provided by a package (see [7.4](#)), a part (see [7.11](#)) or another requirement. A design solution must satisfy the requirement and all of its member requirements and constraints to be a valid solution.

A requirement definition or usage may be decomposed into nested requirement usages, which may themselves be further decomposed. Since a requirement usage is a kind of constraint usage, any nested composite requirement usage is automatically considered to be a required constraint of the containing requirement definition or usage. A requirement definition or usage may also reference another requirement usage as a required constraint. For the overall requirement to then be satisfied, all such composite or referenced requirements must be satisfied.

Like any usage element, the features of a requirement usage can redefine the features of its requirement definition. For example, a requirement definition *MaximumMass* may include the require constraint `{massActual <= massRequired}`, written in terms of the attribute usages *massActual* and *massRequired*. A requirement usage *maximumVehicleMass* defined by *MaximumMass* could restrict the subject of the requirement to be a *Vehicle*, redefine the *massActual* attribute to be the mass of the subject *Vehicle*, and redefine the *massRequired* attribute and bind it to 2000 kilograms. In this way, the requirement definition serves as a requirement template that can be reused and tailored to each context of use.

Subjects

A requirement definition or usage always has a *subject*, which is a distinguished parameter that identifies the entity on which the requirement is being specified. A requirement usage can only be satisfied by an entity that conforms to the definition of its subject. For example, if the subject of a requirement is defined to be a *Vehicle*, then a

standard vehicle model or sports vehicle model can satisfy the requirement, as long as these usages are defined by *Vehicle* or a specialization of it. The subject can also be restricted to be a certain kind of definition element, if it is desired to constrain what kind of entity can satisfy the requirement. For example, the subject can be restricted to be an action, if it is desired to constrain the requirement to be satisfied only by action usages.

Constraining the subject of a requirement definition or usage is also useful to allow features of the subject definition to be used in formal expressions for the assumed and required constraints of the requirement. However, this may not be necessary if the requirement is specified more informally, or in terms of parameters or other features to be bound later. In this case, it is not necessary to explicitly specify the subject of a requirement, in which case it the subject is implicitly assumed to be defined as *Anything*.

Note. Cases also have subjects (see [7.21](#)).

Actors, Stakeholders and Concerns

Actors and stakeholders are additional distinguished parameters that may be specified for a requirement definition or usage. Actor and stakeholder parameters are part usages whose definitions represent entities that play special roles relative to the requirement definition or usage. A requirement may have multiple actors and stakeholders, some of which may have the same definition, representing the same kind of entity playing different roles relative to the requirement.

An *actor parameter* represents a role played by an entity external to the subject of the requirement but necessary for the satisfaction of the requirement. For example, a requirement whose subject is a *Vehicle* may also specify an actor that is the *Driving Environment*. Features of this actor may be used in, for example, the assumed constraints of the requirement, to constrain the environment in which the required constraints apply. The satisfaction of the requirement by a specific subject entity is then relative to the specific environment entity filling the actor parameter.

Note. Actor parameters may also be specified for cases (see [7.21](#)) and, in particular, use cases (see [7.24](#)).

A *stakeholder parameter* represents a role played by an entity (usually a person, organization or other group) having concerns related to the containing requirement. Stakeholder concerns may also be explicitly modeled as special kinds of requirements. A *concern definition* is a kind of requirement definition that represents a stakeholder concern. A *concern usage* is a kind of requirement usage that is a usage of a concern definition. The stakeholder parameters of a concern definition or usage then delineate the stakeholders that have a certain concern.

Rather than explicitly referencing specific stakeholders, a requirement definition or usage can be specified as *framing* the modeled concerns of relevant stakeholders. All the framed concerns of a requirement must then be *addressed* for the requirement to be satisfied.

Note. Stakeholder and concern modeling is frequently used in the context of view and viewpoint modeling (see [7.25](#)). A viewpoint is a kind of requirement that frames certain stakeholder concerns to be addressed by one or more views satisfying the viewpoint.

Requirement Satisfaction

Since a requirement is a kind of constraint, a requirement can be evaluated to be `true` or `false`. A requirement is *satisfied* when it evaluates to `true`.

A *satisfy requirement usage* is a kind of assert constraint usage (see [7.19](#)) that asserts that a requirement is satisfied when a given feature is bound to the subject parameter of the requirement. Other parameters or features of the requirement may also be bound in the body of the satisfy requirement usage. For example, the *maximumVehicleMass* requirement above could be asserted to be satisfied by a specific *vehicle c1* usage,

which means that the required constraint `{massActual <= massRequired}` must be true when `massActual` is bound to the mass of `vehicle c1`.

Similarly to an assert constraint usage, a satisfy requirement usage can also be *negated*. A negated satisfy requirement usage asserts that some entity does *not* satisfy the given requirement.

7.20.2 Abstract Syntax

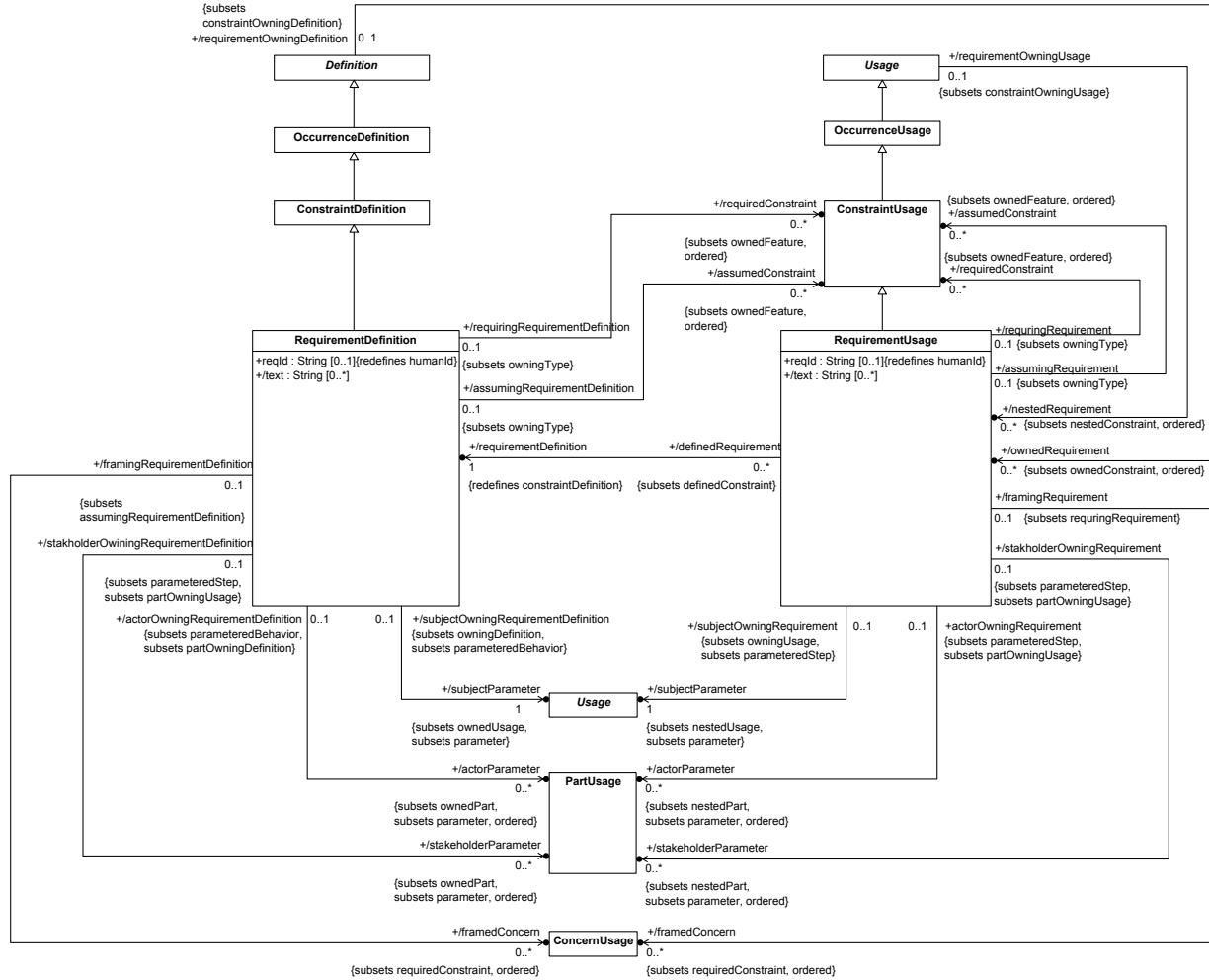


Figure 40. Requirement Definition and Usage

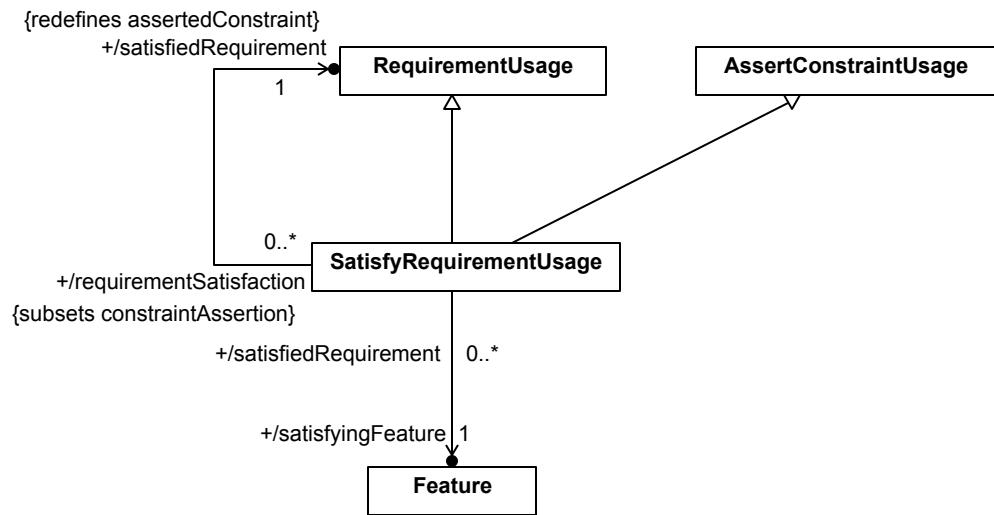


Figure 41. Requirement Satisfaction

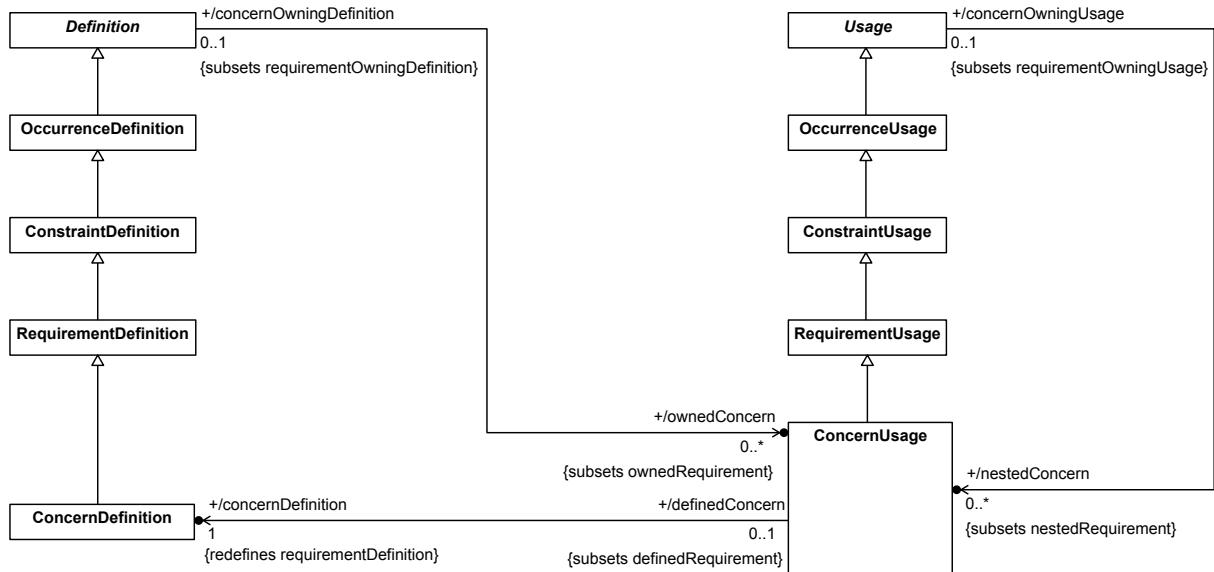


Figure 42. Concern Definition and Usage

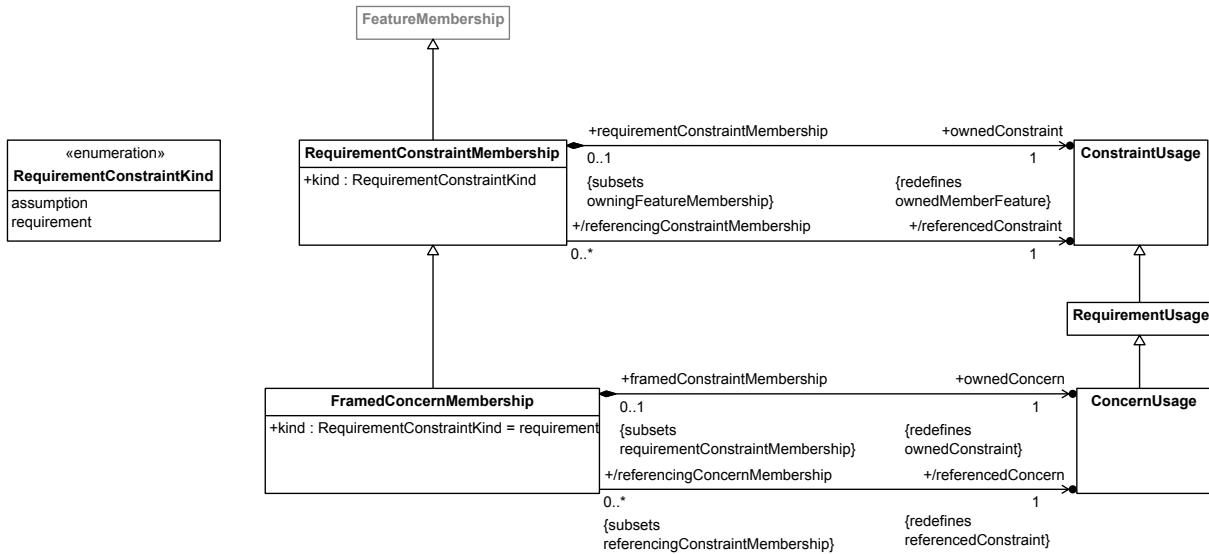


Figure 43. Requirement Constraint Membership

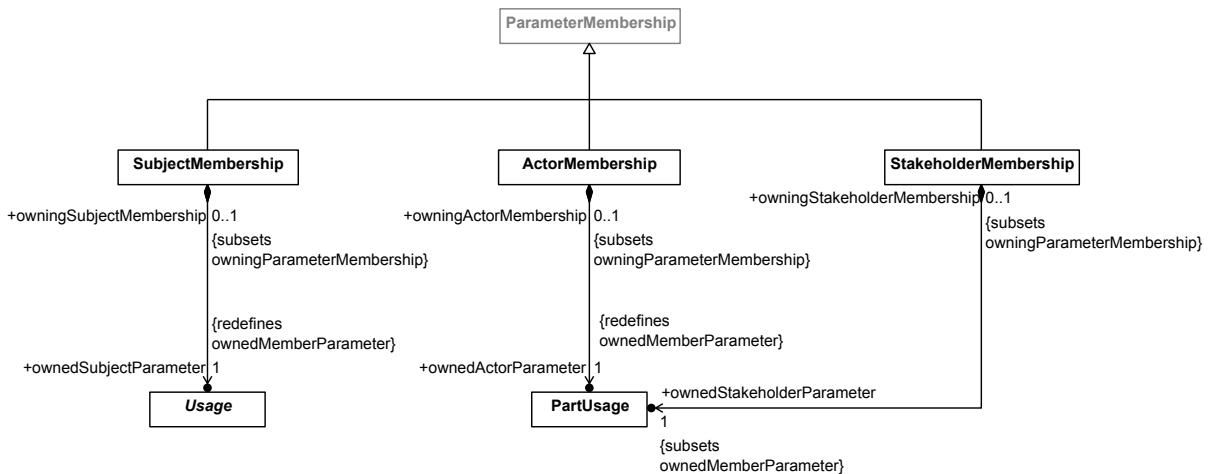


Figure 44. Requirement Parameter Memberships

7.20.3 Notation

Table 24. Requirements Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Requirement Definition (can include any of the following compartments in the compartment stack: <i>actors</i> , <i>allocations</i> , <i>attributes</i> , <i>constraints</i> , <i>documentation</i> , <i>individuals</i> , <i>metadata</i> , <i>namespaces</i> , <i>relationships</i> , <i>requirements</i> , <i>snapshots</i> , <i>stakeholders</i> , <i>timeslices</i> , <i>variants</i> , <i>variant element usages</i>)	<div style="border: 1px solid black; padding: 5px; text-align: center;"> «requirement def» RequirementDef1 </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> «requirement def» RequirementDef1 </div> <div style="border: 1px solid black; padding: 5px; text-align: center;"> compartment stack ... </div>	<pre> requirement def RequirementDef1; or requirement def RequirementDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Requirement (can include any of the following compartments in the compartment stack: <i>actors</i> , <i>allocations</i> , <i>attributes</i> , <i>constraints</i> , <i>documentation</i> , <i>individuals</i> , <i>metadata</i> , <i>namespaces</i> , <i>relationships</i> , <i>requirements</i> , <i>snapshots</i> , <i>stakeholders</i> , <i>timeslices</i> , <i>variants</i> , <i>variant elementusages</i>)	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«requirement»</p> <p>requirement1 : RequirementDef1</p> <p>«requirement»</p> <p>requirement1 : RequirementDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>requirement requirement1 : RequirementDef1; or requirement requirement1 : RequirementDef1 { /* members */ }</pre>
Requirements Compartment	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p><i>requirements</i></p> <p>satisfy requirement11</p> <p>requirement11.1</p> <p>requirement11.2</p> </div>	
Satisfy Requirements Compartment	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p><i>satisfy requirements</i></p> <pre>requirement1:RequirementDef1 [1..*] ordered nonunique ^requirement2:RequirementDef2 requirement3R:RequirementDef3R redefines requirement3 requirement4R:RequirementDef4R:>>requirement4 :>>requirement5 requirement6S:RequirementDef6S [m] subsets requirement6 [n] requirement7S:RequirementDef7S [m] :> requirement7 [n] requirement8R = requirement8 ref requirement9:RequirementDef9 requirement11 requirement11.1 requirement11.2</pre> </div>	
Verify Requirements Compartment	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p><i>verify requirements</i></p> <p>requirement1:RequirementDef1</p> </div>	
Satisfy	<div style="display: flex; align-items: center; justify-content: center;"> <div style="border: 1px solid black; padding: 2px 10px; margin-right: 10px;">«part»</div> <div style="border: 1px solid black; padding: 2px 10px; border-radius: 10px; background-color: #f0f0f0;">part1 : Part1</div> — <div style="border: 1px solid black; padding: 2px 10px; border-radius: 10px; background-color: #f0f0f0;">«satisfy»</div> -> <div style="border: 1px solid black; padding: 2px 10px; border-radius: 10px; background-color: #f0f0f0;">«requirement»</div> <div style="border: 1px solid black; padding: 2px 10px; border-radius: 10px; background-color: #f0f0f0;">requirement1 : Requirement1</div> </div>	

7.21 Cases

7.21.1 Overview

A *case definition* is a kind of calculation definition (see [7.18](#)) that produces a result intended to achieve a specific objective regarding a given subject. A *case usage* is a kind of calculation usage that is a usage of a case definition. A case is a general concept that may be used in its own right, but also provides the basis for more specific kinds of cases, including analysis cases (see [7.22](#)), verification cases (see [7.23](#)), and use cases (see [7.24](#)).

The *subject* of a case is modeled as a distinguished parameter, similarly to the subject of a requirement (see [7.20](#)). The *objective* of a case is modeled as a requirement usage to be satisfied by the performance of the case. Depending on the kind of case, the subject of the objective may be the same as the subject of the case (such as for a verification case or a use case) or it may be the result of the case (such as for an analysis case).

A case definition or usage may also have one or more *actor parameters* that represent roles played by an entity external to the subject of the case but necessary to the specification of the case. An actor parameter is a part usage whose definition represents an entity that plays a designated actor role for the case. A case may have multiple actors, some of which may have the same definition, representing the same kind of entity playing different roles relative to the case.

Note. Actor parameters may also be specified for any kind of case, but they are used, in particular, in the specification of use cases (see [7.24](#)). Requirements may also have actor parameters (see [7.20](#)).

The body of a case can be specified using subactions and subcalculations needed to achieve the case objective. This generally includes some combination of collecting information about the subject, evaluating it, and then producing a result.

7.21.2 Abstract Syntax

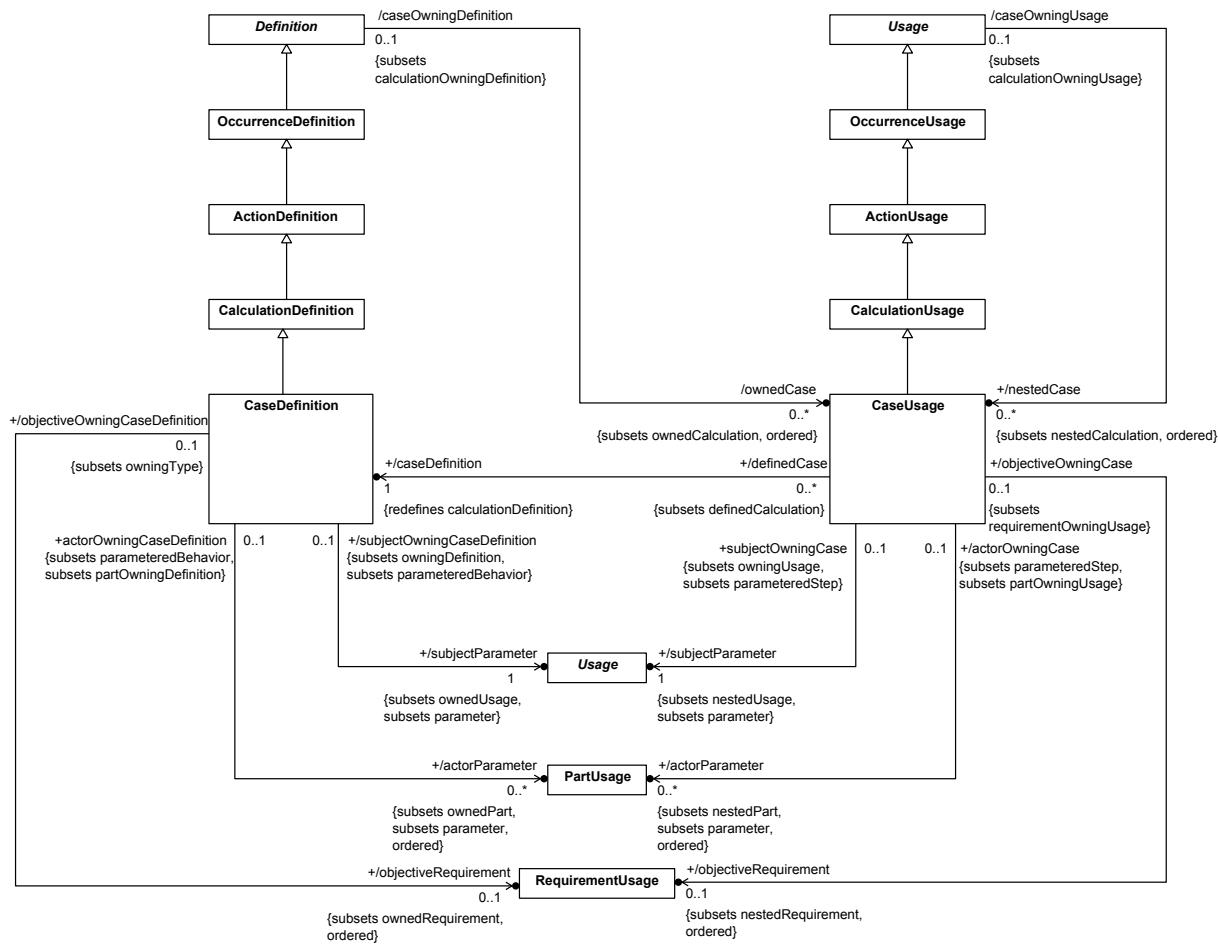


Figure 45. Case Definition and Usage

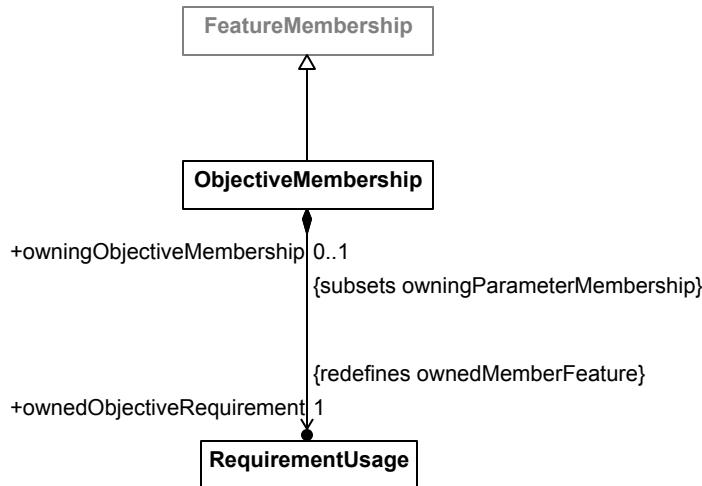


Figure 46. Case Membership

7.21.3 Notation

7.22 Analysis Cases

7.22.1 Overview

Analysis Case Definition and Usage

An *analysis case definition* is a kind of case definition (see [7.21](#)) whose objective is to carry out an analysis on the subject of the case. An *analysis case usage* is a kind of case usage that is a usage of an analysis case definition.

The subject of an analysis case identifies what is being analyzed. The subject can often be kept quite general in an analysis case definition and then made more specific in usages of that definition. Performing an analysis case returns a 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 performance of an analysis case can be specified in a number of different ways.

- The analysis case can include a set of *analysis actions*, each of which can specify calculations that return results. For 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.
- An analysis can be specified in SysML and solved by external solvers. In this case, the analysis case specifies the analysis to be performed, but does not define how the analysis is actually executed. For example, the analysis case could specify that the analysis result is obtained by integrating a differential equation, without detailing what integration algorithm is to be used to do this.
- An analysis case can also specify a set of simultaneous equations to be solved. This can be done defining one or more constraint usages (see [7.19](#)) 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.

Submission Note. Providing a further library model of specific kinds of analysis actions will be considered for the final submission.

Trade-Off Analysis

A *trade-off analysis* is a special kind of analysis used to evaluate and compare alternatives. Such an analysis can be modeled by a usage of the *TradeStudy* analysis case definition from the Trade Studies library model found in the Analysis Domain Library (see [9.4.5](#)).

The subject of a *TradeStudy* analysis case is the collection of alternatives to be analyzed. An *objective function* is then provided that is used to evaluate each alternative, in order to find the alternative that meets the objective of the analysis case. Common *TradeStudy* objectives are to maximize or minimize the value of the objective function.

An example of a trade-off analysis is an analysis that evaluates and compares multiple vehicle design alternatives in terms of various criteria, such as acceleration, reliability, and fuel economy. 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.

7.22.2 Abstract Syntax

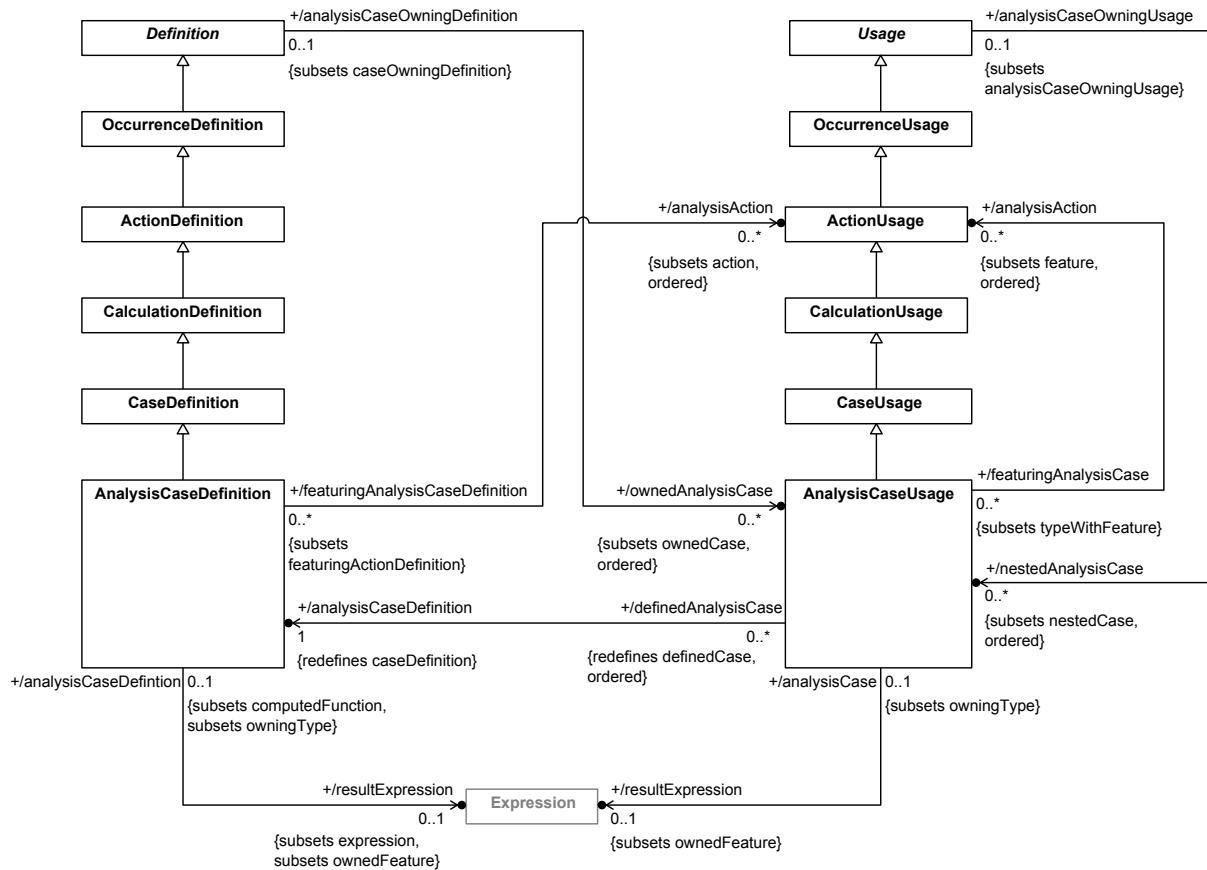


Figure 47. Analysis Case Definition and Usage

7.22.3 Notation

Table 25. Analysis Cases Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Analysis Case Definition (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>actors</i> , <i>allocations</i> , <i>analyses</i> , <i>attributes</i> , <i>body</i> , <i>calcs</i> , <i>constraints</i> , <i>documentation</i> , <i>individuals</i> , <i>metadata</i> , <i>namespaces</i> , <i>parameters</i> , <i>relationships</i> , <i>snapshots</i> , <i>timeslices</i> , <i>variants</i> , <i>variant elementusages</i>)	<pre>«analysis def» AnalysisDef1 compartment stack ...</pre>	analysis def AnalysisDef1; or analysis def AnalysisDef1 { /* members } }
Analysis Case (can include any of the following compartments in the compartment stack: <i>actions</i> , <i>actors</i> , <i>allocations</i> , <i>analyses</i> , <i>attributes</i> , <i>body</i> , <i>calcs</i> , <i>constraints</i> , <i>documentation</i> , <i>individuals</i> , <i>metadata</i> , <i>namespaces</i> , <i>parameters</i> , <i>performed by</i> (<i>Usage Only</i>), <i>relationships</i> , <i>snapshots</i> , <i>timeslices</i> , <i>variants</i> , <i>variant elementusages</i>)	<pre>«analysis» analysis1 : AnalysisDef1 compartment stack ...</pre>	analysis analysis1 : AnalysisDef1; or analysis analysis1 : AnalysisDef1 { /* members } }

7.23 Verification Cases

7.23.1 Overview

A *verification case definition* is a kind of case definition (see [7.21](#)) whose result is a verdict on whether the subject of the case satisfies certain requirements. A *verification case usage* is a case usage that is a usage of a verification case definition.

The subject of a verification case is an input parameter that identifies the system or other entity that is being evaluated as to whether it satisfies certain requirements (often referred to as the "unit under test" or "unit under verification"). The subject may be kept general in a verification case definition and then made more specific in usages of that definition. The objective of a verification case is to verify that the verification subject satisfies one or more specific requirements. The result of the validation case is a *verdict*, which is one of the following:

- *Pass* indicates that the subject has been determined to satisfy the requirements to be verified.
- *Fail* indicates that the subject has been determined *not* to satisfy the requirements to be verified.
- *Inconclusive* indicates that a determination could not be made as to whether the subject satisfies the requirements to be verified.
- *Error* indicates that an error occurred during the performance of the verification.

A typical verification case includes a set of *verification actions* that perform the following steps.

1. *Collect data* about the subject as needed to support the verification objective, which is typically done using *verification methods* such as analysis, inspection, demonstration, and test.

2. *Analyze 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.
3. *Evaluate the results of the analysis* and produce a verdict.

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 more specifically, using actor parameters on the verification case.

7.23.2 Abstract Syntax

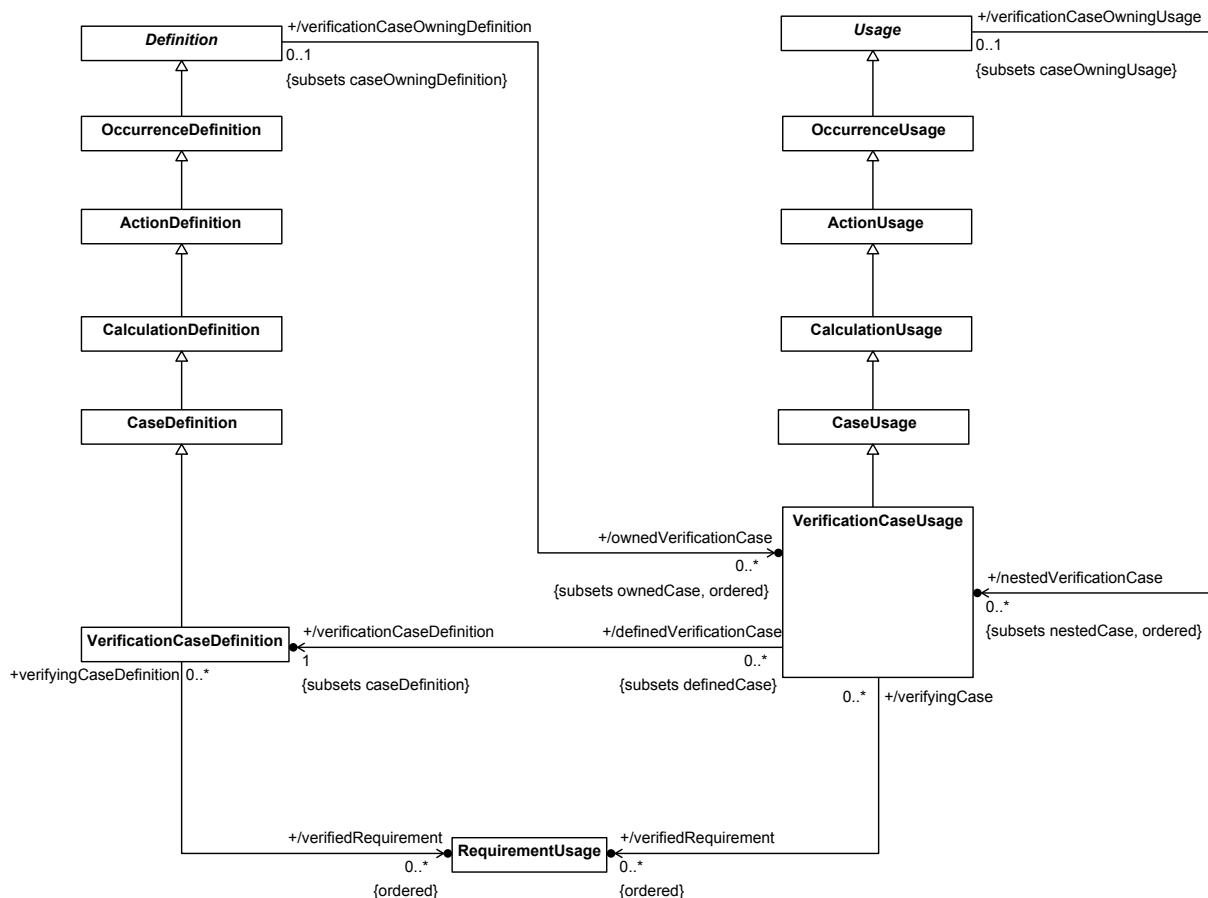


Figure 48. Verification Case Definition and Usage

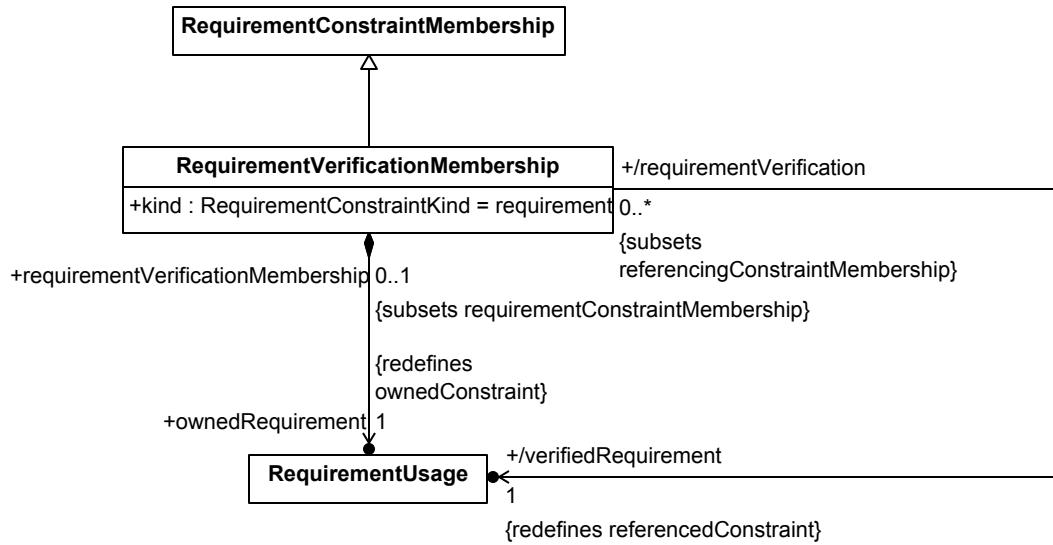


Figure 49. Verification Membership

7.23.3 Notation

Table 26. Verification Cases Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Verification Case Definition (can include any of the following compartments in the compartment stack: <i>actions, actors, allocations, attributes, body, calcs, constraints, documentation, individuals, metadata, namespaces, parameters, relationships, verify requirements, snapshots, timeslices, variants, variant elementusages, verifications, verification methods</i>)	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> «verification def» VerificationDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> «verification def» VerificationDef1 <i>compartment stack</i> <i>...</i> </div>	verification def VerificationDef1; or verification def VerificationDef1 { /* members */ }
Verification Case (can include any of the following compartments in the compartment stack: <i>actions, actors, allocations, attributes, body, calcs, constraints, documentation, individuals, metadata, namespaces, parameters, performed by (Usage Only), relationships, verify requirements, snapshots, timeslices, variants, variant elementusages, verifications, verification methods</i>)	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> «verification» verification1 : VerificationDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> «verification» verification1 : VerificationDef1 <i>compartment stack</i> <i>...</i> </div>	verification verification1 : VerificationDef1; or verification verification1 : VerificationDef1 { /* members */ }

Element	Graphical Notation	Textual Notation
Verified By Compartment		
Verification Methods Compartment		

7.24 Use Cases

7.24.1 Overview

A *use case definition* is a kind of case definition (see [7.21](#)) that specifies the required behavior of its subject relative to one or more external actors. The objective of the use case is to provide an observable result of value to one or more of its actors. A *use case usage* is a case usage that is a usage of a use case.

A use case is typically specified as a sequence of interactions between the subject and the various actors, which are all modeled as part usages. Each interaction can be modeled as a *message* (see [7.13](#)) that delivers some payload or signal from an actor to the system or vice versa. The sources and target ends of these messages can either be modeled simply as abstract events within the lifetime of the subject and actor occurrences (see [7.9](#)), or more concretely as actions performed to carry out the interaction (see [7.16](#)).

An *include use case usage* is a use case usage that is also a kind of perform action usage (see [7.16](#)). A use case definition or usage may contain an include use case usage to specify that the behavior of the containing use case includes the behavior of the included use case. The subject of the included use case is the same as the subject of the containing use case, so the subject parameter of the included use case must have a definition that is compatible with the definition of the containing use case. Actor parameters of the included use case may be bound to corresponding actor parameters of the containing use case as necessary (see also [7.16](#) on parameter binding and [7.13](#) on binding in general).

Submission Note. The language currently does not include a construct that corresponds to the traditional use case "extend" relationship. This is expected to be addressed in the final submission.

As a behavior, a use case can be performed with specific values for its subject and actor parameters. If a given subject also has a design model that decomposes its internal structure, then it should be possible to construct an interaction of the internal parts of the subject, consistent with the design model, that can be shown to be a specialization of the behavior specified by the performance of the use case for that subject. This is known as a *realization* of the use case relative to the design model. A system is properly designed to provide the behavior required by a set of use cases if there is a legal realization of each use case relative to the design of the system.

7.24.2 Abstract Syntax

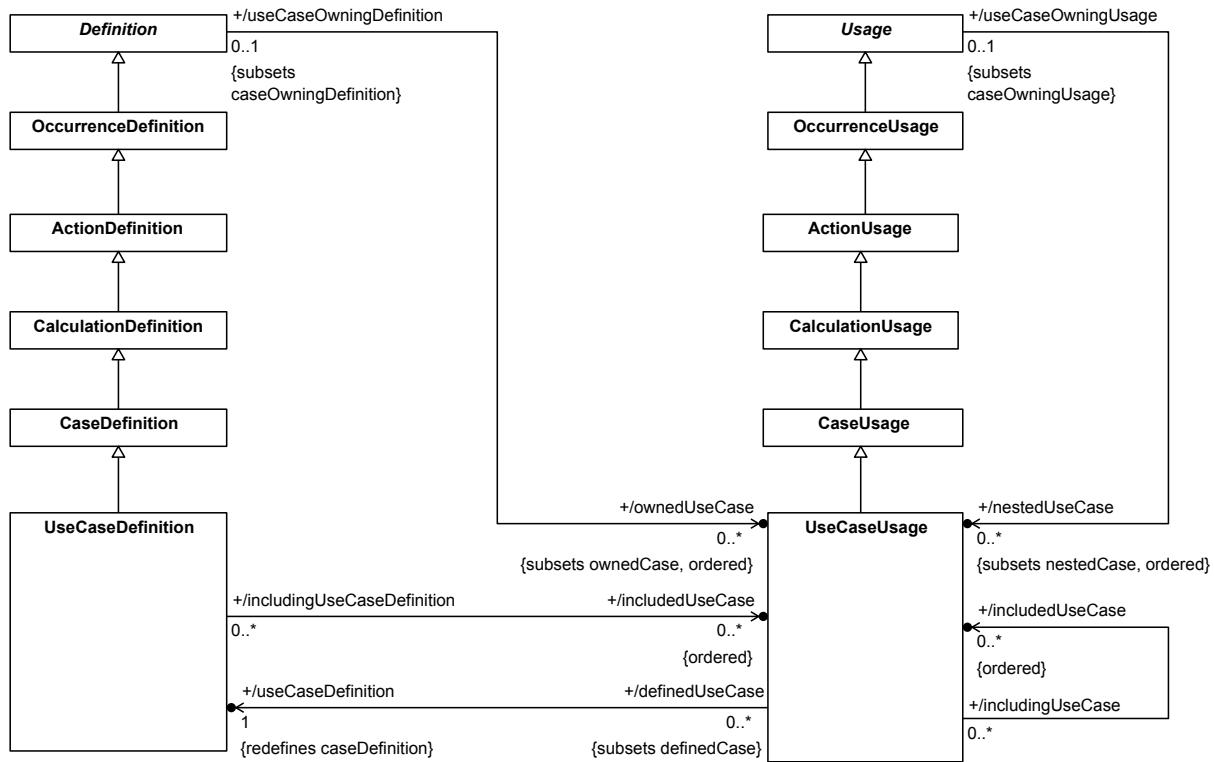


Figure 50. Use Case Definition and Usage

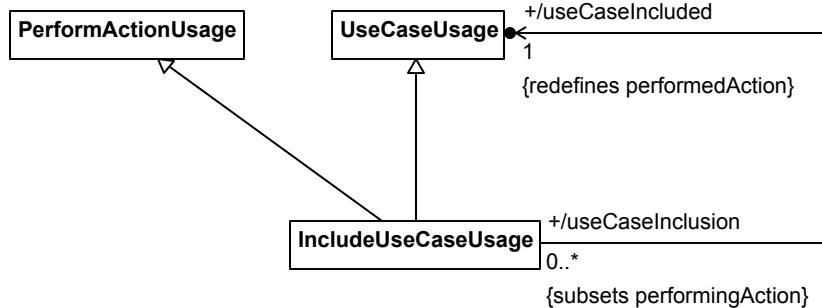
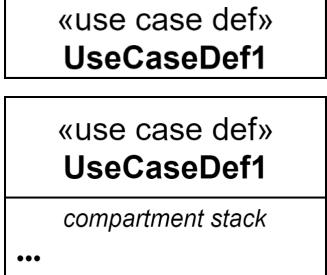
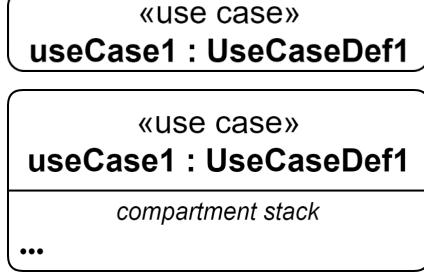
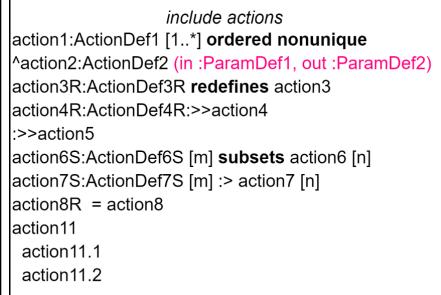


Figure 51. Use Case Inclusion

7.24.3 Notation

Table 27. Use Cases Notation - Representative Usage

Element	Graphical Notation	Textual Notation
Use Case Definition (can include any of the following compartments in the compartment stack: <i>include actions, actors, allocations, attributes, constraints, documentation, individuals, metadata, namespaces, relationships, snapshots, stakeholders, timeslices, variants, variant element usages</i>)	 <pre> <<use case def>> UseCaseDef1 <<use case def>> UseCaseDef1 compartment stack ... </pre>	<pre> use case def UseCaseDef1; or use case def UseCaseDef1 { /* members */ } </pre>
Use Case (can include any of the following compartments in the compartment stack: <i>include actions, actors, allocations, attributes, constraints, documentation, individuals, metadata, namespaces, performed by (Usage Only), relationships, snapshots, stakeholders, timeslices, variants, variant element usages</i>)	 <pre> <<use case>> useCase1 : UseCaseDef1 <<use case>> useCase1 : UseCaseDef1 compartment stack ... </pre>	<pre> use case useCase1 : UseCaseDef1; or use case useCase1 : UseCaseDef1 { /* members */ } </pre>
Include Actions Compartment	 <pre> <i>include actions</i> action1:ActionDef1 [1..*] ordered nonunique ^action2:ActionDef2 (<i>in</i> :ParamDef1, <i>out</i> :ParamDef2) action3R:ActionDef3R redefines action3 action4R:ActionDef4R:>>action4 :>>action5 action6S:ActionDef6S [m] subsets action6 [n] action7S:ActionDef7S [m] > action7 [n] action8R = action8 action11 action11.1 action11.2 </pre>	

7.25 Views and Viewpoints

7.25.1 Overview

A *viewpoint definition* is a kind of requirement definition (see [7.20](#)) that frames the concerns of one or more stakeholders regarding information about a modeled system or domain of interest. A *viewpoint usage* is a requirement usage that is a usage of a viewpoint definition. The subject of a viewpoint is a *view* that is required to address the stakeholder concerns.

A *view definition* is a kind of part definition (see [7.11](#)) that specifies how to create a view artifact to satisfy one or more viewpoints. A view artifact is a rendering of information that addresses some aspect of a system or domain of interest of concern to one or more stakeholders. A view definition can include *view conditions* to extract the relevant model content, and a *rendering* that specifies how the model content should be rendered in a view artifact. A view condition is specified using metadata, in the same way as for a filter condition on a package (see [7.4](#)).

A view usage is a kind of part usage (see [7.11](#)) that is a usage of a view definition. A view usage *exposes* a portion of a model, which is a kind of import (see [7.4](#)) without regard to visibility that provides the scope of application of the view conditions. The view rendering can then be applied to those exposed elements that meet all the view conditions are then to produce the view artifact. A view usage can add further view conditions to those inherited from its view definition, and it can specify a view rendering if one is not provided by its definition.

View usages can be nested and ordered within a composite view to generate composite view artifacts. The view usage also can contain further rendering specifications on the symbolic representation, style, and layout for a particular view. For example, a complex view definition with deeply nested structures can be rendered as a document, where each nested view usage corresponds to a section of a document, and the ordering represents the order of the sections within the document. Within each section of the document, the nested view usages can then specify the information that is rendered as a combination of text, graphical, and tabular information.

A *rendering definition* is a kind of part definition (see [7.11](#)) that specifies how a view artifact is to be rendered. A *rendering usage* is a kind of part usage that is a usage of a rendering definition. A rendering usage is used in a view definition or usage to specify the view rendering.

Submission Note. Currently, the Systems Library only includes a limited set of standard kinds of rendering for text, diagrams and tables. It is expected that, in the final submission, the library will also include models of more comprehensive standard views that correspond to standard diagram kinds.

7.25.2 Abstract Syntax

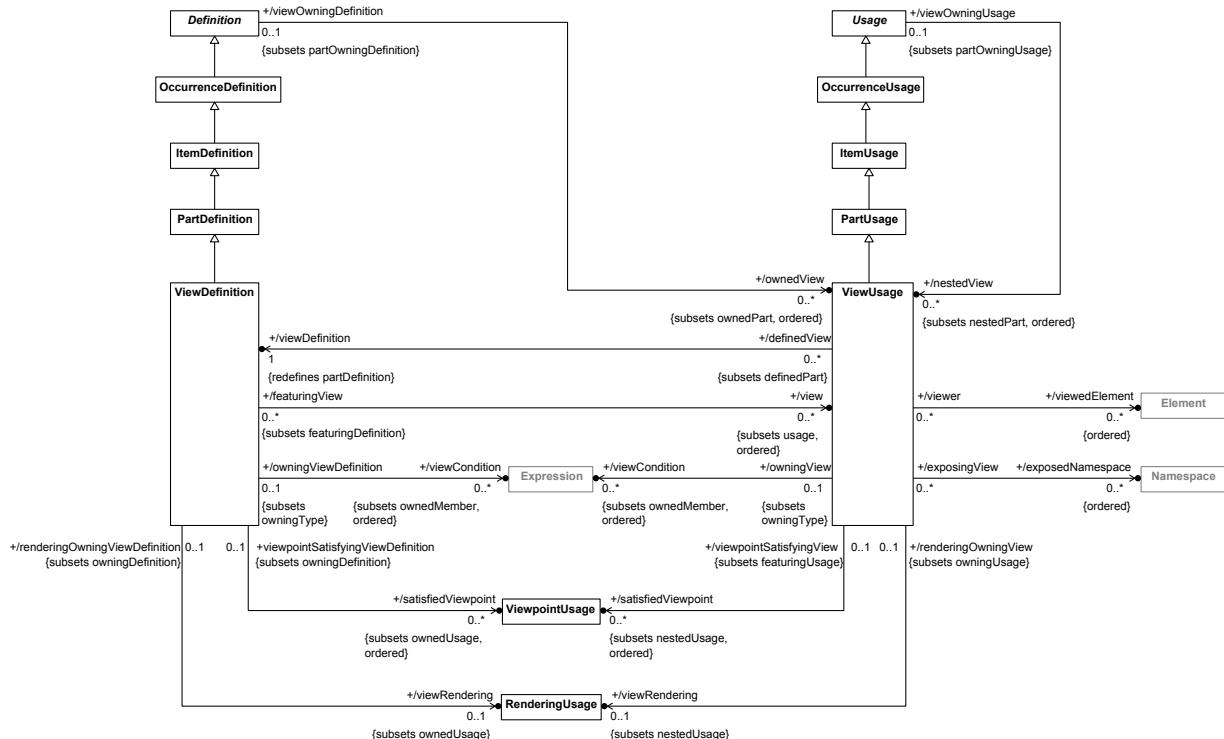


Figure 52. View Definition and Usage

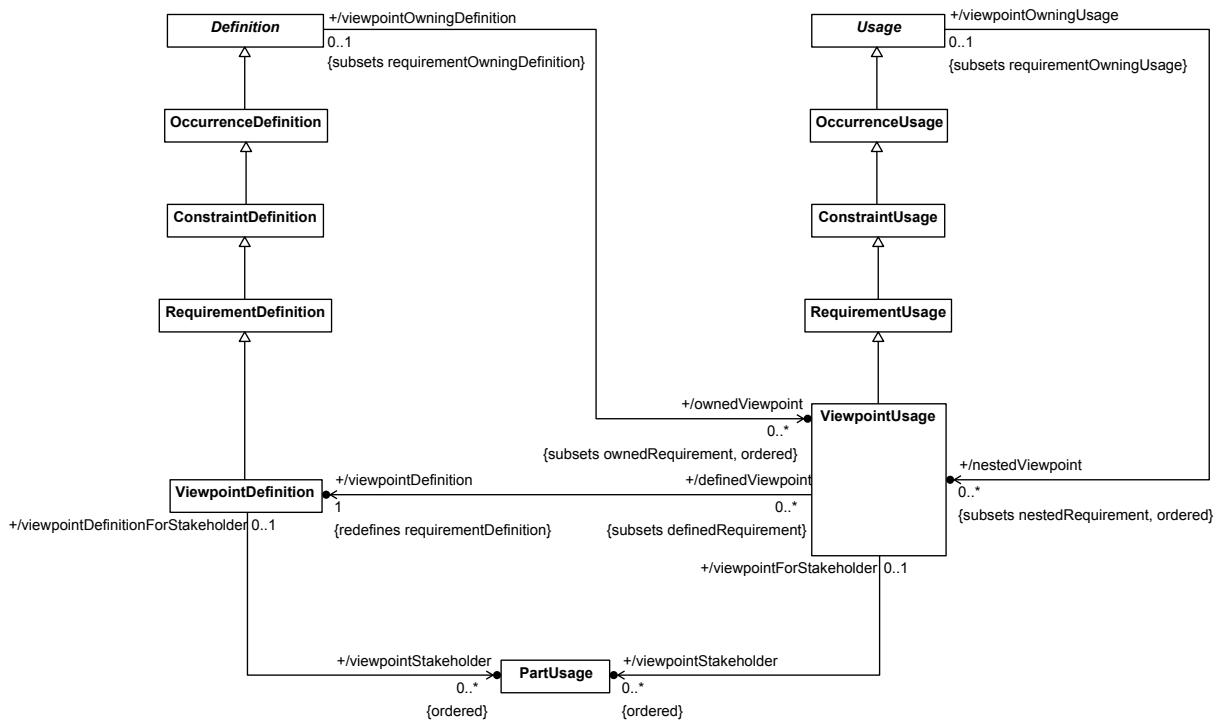


Figure 53. Viewpoint Definition and Usage

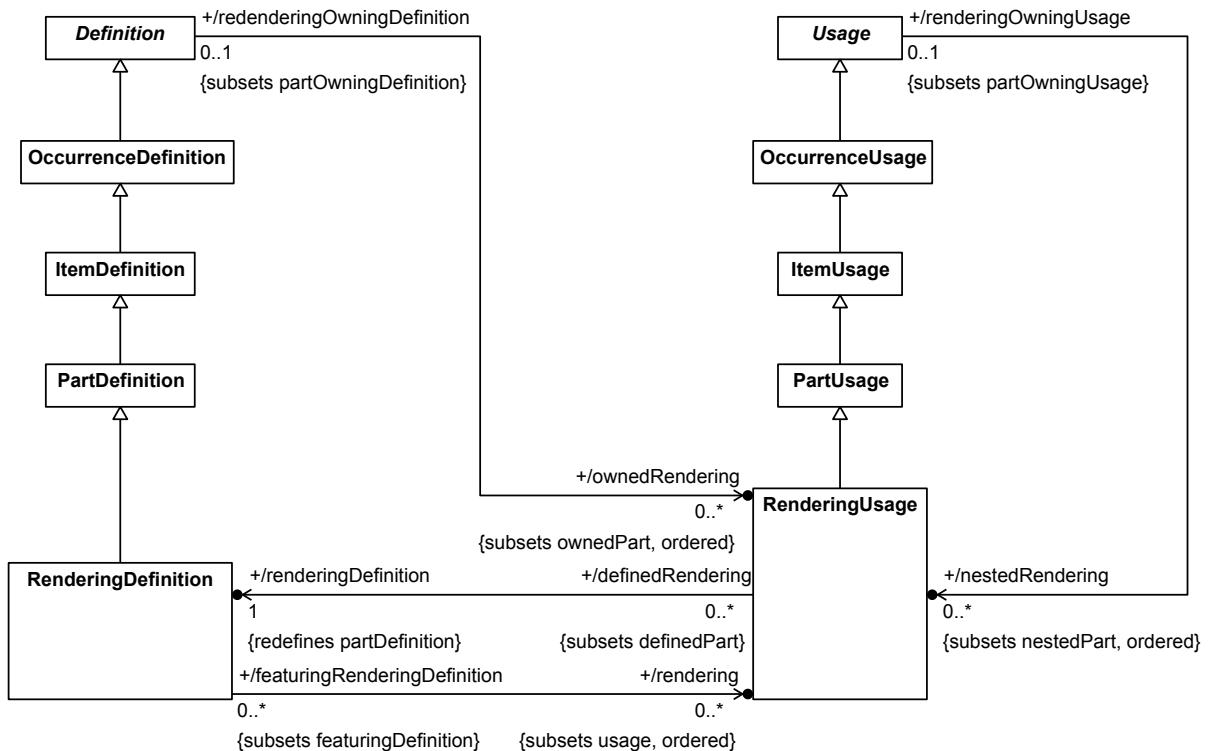


Figure 54. Rendering Definition and Usage

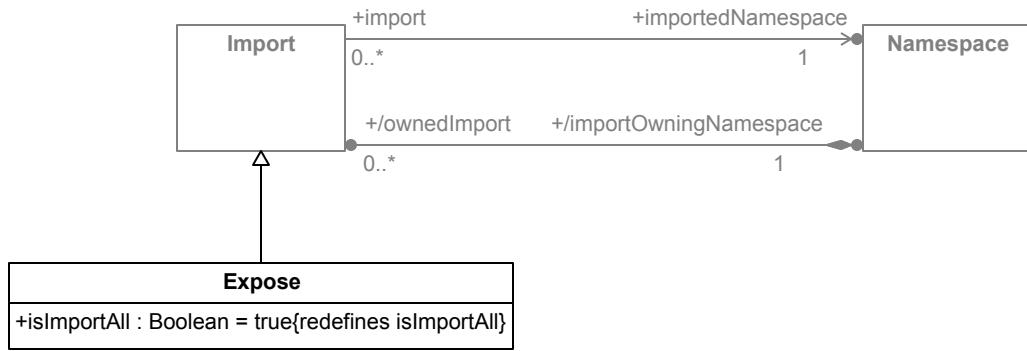


Figure 55. Expose Relationship

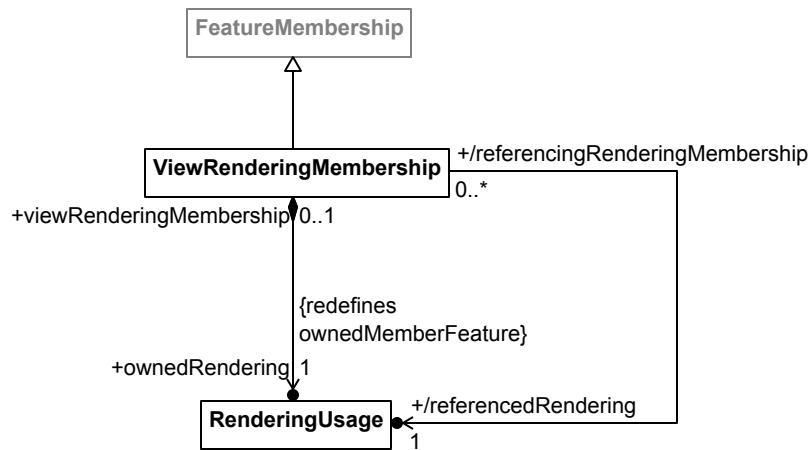
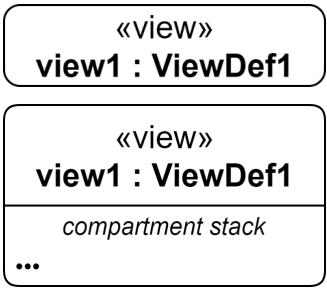
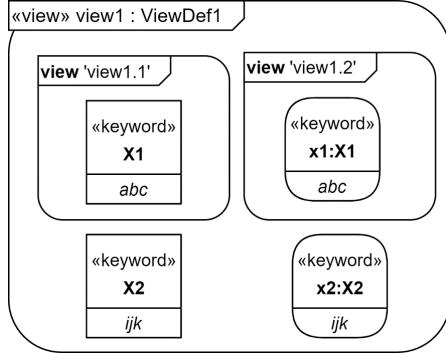
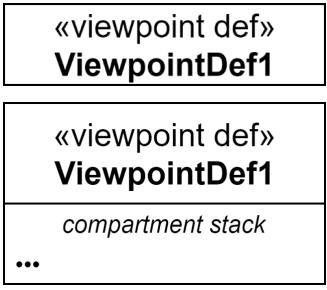


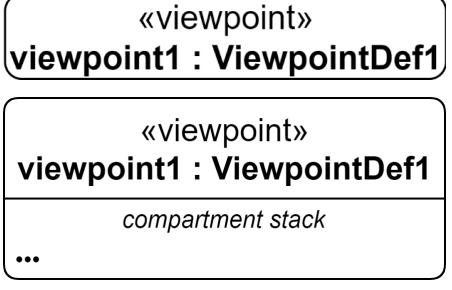
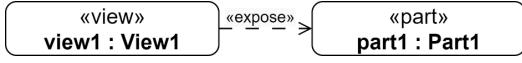
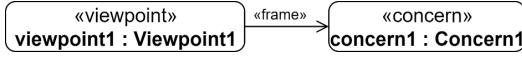
Figure 56. View Rendering Membership

7.25.3 Notation

Table 28. Views and Viewpoints Notation - Representative Usage

Element	Graphical Notation	Textual Notation
<p>View Definition (can include any of the following compartments in the compartment stack: <i>allocations, attributes, constraints, documentation, individuals, metadata, namespaces, relationships, snapshots, timeslices, variants, variant elementusages, views, viewpoint</i>)</p>	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> «view def» ViewDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> «view def» ViewDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> <i>compartment stack</i> ... </div>	<pre> view def ViewDef1; or view def ViewDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
<p>View (can include any of the following compartments in the compartment stack: <i>allocations, attributes, constraints, documentation, individuals, metadata, namespaces, relationships, snapshots, timeslices, variants, variant elementusages, views, viewpoint</i>)</p>		<pre>view view1 : ViewDef1; or view view1 : ViewDef1 { /* members */ }</pre>
<p>View as Diagram</p>		<pre>view view1 : ViewDef1{ view 'view1.1' view 'view1.1.1' «keyword» X1 abc view 'view1.1.2' «keyword» x1:X1 abc view 'view1.2' view 'view1.2.1' «keyword» X2 ijk view 'view1.2.2' «keyword» x2:X2 ijk }</pre>
<p>Viewpoint Definition (can include any of the following compartments in the compartment stack: <i>actors, allocations, attributes, constraints, documentation, frame concerns, individuals, metadata, namespaces, relationships, snapshots, stakeholders, timeslices, variants, variant elementusages, satisfied by view, viewpoint</i>)</p>		<pre>viewpoint def ViewPointDef1; or viewpoint def ViewPointDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
Viewpoint (can include any of the following compartments in the compartment stack: <i>actors, allocations, attributes, constraints, documentation, frame concerns, individuals, metadata, namespaces, relationships, snapshots, stakeholders, timeslices, variants, variant element usages, satisfied by view, viewpoint</i>)		<pre>viewpoint viewpoint1 : ViewPointDef1; or viewpoint viewpoint1 : ViewPointDef1 { /* members } }</pre>
Expose		
Frame		
Frame Concerns Compartment		

7.26 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 Metamodel

8.1 Metamodel Overview

The SysML metamodel extends the KerML metamodel as specified in the KerML specification [KerML].

- The SysML concrete syntax includes a textual notation (see [8.2.2](#)), which is generally distinct from that of KerML, though consistent on common elements (such as packages and expressions), and a complete graphical notation.
- The SysML abstract syntax (see [8.3](#)) imports the KerML abstract syntax, reusing some KerML metaclasses directly, and further specializing most other KerML metaclasses.
- The SysML semantics (see [8.4](#)) are defined by relating the SysML abstract syntax to the semantic models in the Systems Model Library (see [Clause 9](#)), which is based on the Kernel Model Library from KerML, and providing syntactic transformations from SysML models to syntactically equivalent KerML models (including elements that are otherwise implicit in the SysML abstract syntax).

8.2 Concrete Syntax

8.2.1 Concrete Syntax Overview

Concrete syntax specifies the how the language appears to modelers. They construct and review models shown according to the concrete syntax. The SysML concrete syntax includes both a textual notation, described in [8.2.2](#), and a graphical notation, described in [8.2.3](#). Various views of a SysML model may be rendered entirely using the textual notation, entirely using the graphical notation, or using a combination of the two.

8.2.2 Textual Notation

8.2.2.1 Textual Notation Overview

8.2.2.1.1 Lexical Structure

The lexical structure of the SysML textual notation is identical to that of the KerML textual notation [KerML], except for the following two points.

1. The reserved keywords of SysML are the following.

```
about abstract accept action actor alias all allocate allocation analysis
and as assert assoc assume attribute bind block by calc case comment
concern connect connection constraint decide def default defined dependency
do doc else end entry enum event exhibit exit expose feature filter
first flow for fork frame from hastype id if implies import in
include individual inout interface istype item join language merge message
metadata nonunique not objective occurrence of or ordered out package
parallel part perform port private protected public redefines ref render
rendering rep require requirement return satisfy send snapshot specializes
stakeholder state stream subject subsets succession then timeslice to
transition use variant variation verification verify via view viewpoint xor
```

2. The set of special lexical terminals matching either certain keywords or their symbolic equivalents are the following in SysML.

```

DEFINED_BY = ':' | 'defined' 'by'
SPECIALIZES = ':>' | 'specializes'
SUBSETS = ':>' | 'subsets'
REDEFINES = ':>>' | 'redefines'

```

8.2.2.1.2 EBNF Conventions

The *grammar* definition for the SysML textual concrete syntax defines how lexical tokens for an input text are grouped in order to construct an abstract syntax representation of a model (see [8.3](#)). The concrete syntax grammar definition uses an Extended Backus Naur Form (EBNF) notation (see [Table 29](#)) that includes further notations to describe how the concrete syntax maps to the abstract syntax (see [Table 30](#)).

Productions in the grammar formally result in the synthesis of classes in the abstract syntax and the population of their properties (see [Table 31](#)). Productions may also be parameterized, with the parameters typed by abstract syntax classes. Information passed in parameters during parsing allows a production to update the properties of the provided abstract syntax elements as a side-effect of the parsing it specifies. Some productions only update the properties of parameters, without synthesizing any new abstract syntax element.

Table 29. EBNF Notation Conventions

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

Table 30. Abstract Syntax Synthesis Notation

Variable assignment	v = Element	Assign the result of parsing the concrete syntax Element to the local variable v.
Property assignment	x.p = Element	Assign the result of parsing the concrete syntax Element to property p of the abstract syntax element denoted by x.
List property construction	x.p += Element	Add the result of parsing the concrete syntax Element to the list property p of the abstract syntax element denoted by x.
Boolean property assignment	x.p ?= Element	If the concrete syntax Element is parsed, then set the Boolean property p of the abstract syntax element denoted by x to true.
Non-parsing assignment	{ v = value } { x.p = value } { x.p += value } }	Assign (or add) the given value to the variable v or property x.p, without parsing any input.

Name resolution	[QualifiedName]	Parse a QualifiedName, then resolve that name to an Element reference for use as a value in an assignment as above.
------------------------	-----------------	---

Table 31. Grammar Production Definitions

Synthetic production definition	NonterminalElement : AbstractSyntaxElement = ...	Define a production for the NonterminalElement that synthesizes the AbstractSyntaxElement.
Parameterized synthetic production definition	NonterminalElement (p1 : Type1, p2 : Type2, ...) : AbstractSyntaxElement = ...	Define a production for the NonterminalElement that synthesizes the AbstractSyntaxElement, with the given parameters named p1, p2, The types of the parameters must be abstract syntax classes.
Parameterized updating production definition	NonterminalElement (p1 : Type1, p2 : Type2, ...) = ...	Define a production for the NonterminalElement that does not synthesize any new abstract syntax element, but updates properties of its parameters. (Such a production must have at least one parameter.)

8.2.2.2 Elements Textual Notation

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

8.2.2.3 Annotations Textual Notation

8.2.2.3.1 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 }
```

8.2.2.3.2 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

```

8.2.2.3.3 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

```

8.2.2.3.4 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

```

8.2.2.4 Packages Textual Notation

8.2.2.4.1 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 += ElementFilterMember
    | p.ownedRelationship += AliasMember
    | p.ownedRelationship += Import

MemberPrefix (m : Membership) =
    ( ownedRelationship += PrefixDocumentation )*
    ( visibility = VisibilityIndicator )?

PackageMember (p : Package) : Membership
    MemberPrefix(this)
    ( ownedMemberElement = DefinitionElement(this, p)
    | ownedMemberElement = UsageElement(this) )

ElementFilterMember : ElementFilterMembership =
    MemberPrefix(this)
    'filter' condition = OwnedExpression ';'

AliasMember : Membership =
    MemberPrefix(this)
    'alias' memberName = Name 'for'
    memberElement = [QualifiedName] ';''

Import : Import =
    ( ownedRelationship += PrefixDocumentation )*
    ( visibility = VisibilityIndicator )?
    'import' ( isImportAll ?= 'all' )?
    ( ImportedNamespace(this)
    | ImportedFilterPackage(this) ) ';''

ImportedNamespace (i : Import) =
    ( i.importedNamespace = [QualifiedName] )?
    ( importedMemberName = Name | '*' )
    ( '::' isRecursive ?= '**' )?

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

```

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

FilterPackageImport : Import =
    ImportedNamespace (this)

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

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

```

8.2.2.4.2 Package Elements

```

DefinitionElement (m : Membership, n : Namespace) : Element =
    Package(m)
    | Comment(m, n)
    | TextualRepresentation(m, n)
    | AnnotatingFeature(m, n)
    | Dependency(m)
    | AttributeDefinition(m)
    | EnumerationDefinition(m)
    | OccurrenceDefinition(m)
    | IndividualDefinition(m)
    | ItemDefinition(m)
    | PartDefinition(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)
    | UseCaseDefinition(m)
    | ViewDefinition(m)
    | ViewpointDefinition(m)
    | RenderingDefinition(m)

UsageElement (m : Membership) : Usage =
    DefaultReferenceUsage(m)
    | ReferenceUsage(m)
    | AttributeUsage(m)
    | EnumerationUsage(m)
    | OccurrenceUsage(m)
    | IndividualUsage(m)
    | PortionUsage(m)

```

```

| ItemUsage (m)
| PartUsage (m)
| PortUsage (m)
| BindingConnector (m)
| Succession (m)
| ConnectionUsage (m)
| InterfaceUsage (m)
| ActionUsage (m)
| CalculationUsage (m)
| StateUsage (m)
| ConstraintUsage (m)
| AssertConstraintUsage (m)
| RequirementUsage (m)
| ConcernUsage (m)
| CaseUsage (m)
| AnalysisCaseUsage (m)
| VerificationCaseUsage (m)
| UseCaseUsage (m)
| ViewUsage (m)
| ViewpointUsage (m)
| RenderingUsage (m)

```

8.2.2.5 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] )*

```

8.2.2.6 Definition and Usage Textual Notation

8.2.2.6.1 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) SubclassificationPart(d) ?

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

DefinitionBodyItem (t : Type) =
    t.ownedRelationship += OwnedDocumentation
    | t.ownedRelationship += DefinitionMember(t)
    | t.ownedRelationship += VariantUsageMember
    | t.ownedRelationship += NonOccurrenceUsageMember
    | t.ownedRelationship += OccurrenceUsageMember(t)

```

```

| t.ownedRelationship += OccurrenceSuccessionMember(t)
| t.ownedRelationship += AliasMember
| t.ownedRelationship += Import

DefinitionMember (t : Type) : Membership =
    MemberPrefix(this)
    ownedMemberElement = DefinitionElement(this, t)

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

NonOccurrenceUsageMember : FeatureMembership =
    MemberPrefix(this)
    ownedMemberFeature = NonOccurrenceUsageElement(this)

OccurrenceUsageMember : FeatureMembership =
    MemberPrefix(this)
    ownedMemberFeature = OccurrenceUsageElement(this)

OccurrenceSuccessionMember (t : Type) : FeatureMembership =
    s = SourceSuccessionMember(t)
    OccurrenceUsageMember
    TargetEndFor(s.ownedMemberFeature, ownedMemberFeature)

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

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

```

8.2.2.6.2 Usages

```

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

RefPrefix (u : Usage) =
    ( u.direction = FeatureDirection )?
    ( u.isAbstract ?= 'abstract' | u.isVariation ?= 'variation')?
    ( u.isEnd ?= 'end' )?

UsagePrefix (u : Usage) =
    RefPrefix
    ( isReference ?= 'ref' )?

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

UsageDeclaration (u : Usage, m : Membership) =
    Identification(u, m) FeatureSpecializationPart(u) ?

UsageCompletion (u : Usage) =

```

```

ValueOrFlowPart(this) ? UsageBody(this)

UsageBody (u : Usage) =
    DefinitionBody(u)

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

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

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

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

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

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

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

SuccessionItemFlowTo (f : Feature) : SuccessionFlowConnectionUsage =
    '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 }

```

8.2.2.6.3 Reference Usages

```

DefaultReferenceUsage (m : Membership) : ReferenceUsage =
    RefPrefix(this) Usage(this, m)

```

```

ReferenceUsage (m : Membership) : ReferenceUsage =
    RefPrefix(this) 'ref' Usage(this, m)

VariantReference : ReferenceUsage =
    ownedRelationship += OwnedSubsetting
    FeatureSpecialization(this) * UsageBody(this)

```

8.2.2.6.4 Body Elements

```

NonOccurrenceElement (m : Membership) : Usage =
    DefaultReferenceUsage(m)
    | ReferenceUsage(m)
    | AttributeUsage(m)
    | EnumerationUsage(m)
    | BindingConnector(m)
    | Succession(m)

OccurrenceUsageElement (m : Membership) : Usage =
    StructureUsageElement(m) | BehaviorUsageElement(m)

StructureUsageElement (m : Membership) : Feature =
    OccurrenceUsage(m)
    | IndividualUsage(m)
    | PortionUsage(m)
    | EventOccurrenceUsage(m)
    | ItemUsage(m)
    | PartUsage(m)
    | ViewUsage(m)
    | RenderingUsage(m)
    | PortUsage(m)
    | ConnectionUsage(m)
    | InterfaceUsage(m)
    | AllocationUsage(m)
    | Message(m)
    | FlowConnectionUsage(m)
    | SuccessionFlowConnectionUsage(m)

BehaviorUsageElement (m : Membership) : Usage =
    ActionUsage(m)
    | CalculationUsage(m)
    | StateUsage(m)
    | ConstraintUsage(m)
    | RequirementUsage(m)
    | ConcernUsage(m)
    | CaseUsage(m)
    | AnalysisCaseUsage(m)
    | VerificationCaseUsage(m)
    | UseCaseUsage(m)
    | ViewpointUsage(m)
    | PerformActionUsage(m)
    | ExhibitStateUsage(m)
    | IncludeUseCaseUsage(m)
    | AssertConstraintUsage(m)
    | SatisfyRequirementUsage(m)

```

```

VariantUsageElement (m : Membership) : Usage =
  VariantReference
  | ReferenceUsage (m)
  | AttributeUsage (m)
  | BindingConnector (m)
  | Succession (m)
  | OccurrenceUsage (m)
  | IndividualUsage (m)
  | PortionUsage (m)
  | EventOccurrenceUsage (m)
  | ItemUsage (m)
  | PartUsage (m)
  | ViewUsage (m)
  | RenderingUsage (m)
  | PortUsage (m)
  | ConnectionUsage (m)
  | InterfaceUsage (m)
  | AllocationUsage (m)
  | Message (m)
  | FlowConnectionUsage (m)
  | SuccessionFlowConnectionUsage (m)
  | BehaviorUsageElement (m)

```

8.2.2.6.5 Specialization

```

SubclassificationPort (c : Classifier) =
  SPECIALIZES c.ownedRelationship += OwnedSubclassification
  ( ',' c.ownedRelationship += OwnedSubclassification )*

OwnedSubclassification : Subclassification =
  superClassifier = [QualifiedName]

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

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

Typings (f : Feature) =
  TypedBy(f) ( ',' f.ownedRelationship += FeatureTyping )*

TypedBy (f : Feature) =
  DEFINED_BY f.ownedRelationship += FeatureTyping

FeatureTyping : FeatureTyping =
  OwnedFeatureTyping | ConjugatePortTyping

OwnedFeatureTyping : FeatureTyping =
  type = [QualifiedName]

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

```

```

Subsets (f : Feature) =
    SUBSETS f.ownedRelationship += OwnedSubsetting

OwnedSubsetting : Subsetting =
    subsettetedFeature = [QualifiedName]
    | subsettetedFeature = FeatureChain
        { ownedRelatedFeature += subsettetedFeature }

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

Redefines (f : Feature) =
    REDEFINES ownedRelationship += OwnedRedefinition

OwnedRedefinition : Redefinition =
    redefinedFeature = [QualifiedName]
    | redefinedFeature = FeatureChain
        { ownedRelatedFeature += redefinedFeature }

FeatureChain : Feature =
    ownedRelationship += OwnedFeatureChaining
    ( '.' ownedRelationship += OwnedFeatureChaining )+

OwnedFeatureChaining : FeatureChaining =
    chainingFeature = [QualifiedName]

```

8.2.2.6.6 Multiplicity

```

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

```

```

OwnedMultiplicity : Membership =
    ownedMemberElement = MultiplicityRange

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

MultiplicityExpressionMember : Membership =
    ownedMemberElement = ( LiteralExpression | FeatureReferenceExpression )

```

8.2.2.7 Attributes Textual Notation

```

AttributeDefinition (m : Membership) : AttributeDefinition =
    DefinitionPrefix(this)? 'attribute' 'def' Definition(this, m)

AttributeUsage (m : Membership) : AttributeUsage =
    UsagePrefix(this) 'attribute' Usage(this, m)

```

8.2.2.8 Enumerations Textual Notation

```
EnumerationDefinition (m : Membership) : EnumerationDefinition =
    'enum' 'def' DefinitionDeclaration(this, m) EnumerationBody(this)

EnumerationBody (e : EnumerationDefinition) =
    ';' |
    | '{' ( e.ownedRelationship += EnumerationUsageMember )* '}' |

EnumerationUsageMember : VariantMembership =
    MemberPrefix(this) ownedVariantUsage = EnumeratedValue(this)

EnumeratedValue (m : Membership) : EnumerationUsage =
    'enum'? Usage(m)

EnumerationUsage (m : Membership) : EnumerationUsage =
    'enum' Usage(m)
```

8.2.2.9 Occurrences Textual Notation

8.2.2.9.1 Occurrence Definitions

```
OccurrenceDefinitionPrefix (o : OccurrenceDefinition) =
    DefinitionPrefix(o)?
    ( o.isIndividual ?= 'individual'
        o.ownedRelationship += LifeClassMembership )?

OccurrenceDefinition (m : Membership) : OccurrenceDefinition =
    OccurrenceDefinitionPrefix(this) 'occurrence' 'def' Definition(this, m)

IndividualDefinition (m : Membership) : OccurrenceDefinition -
    DefinitionPrefix(this)?
    isIndividual ?= 'individual' 'def' Definition(this, m)
    ownedRelationship += LifeClassMembership

LifeClassMembership : Membership =
    ownedMemberElement = LifeClass

LifeClass : LifeClass =
    {}
```

8.2.2.9.2 Occurrence Usages

```
OccurrenceUsagePrefix (o : OccurrenceUsage) =
    UsagePrefix(o)
    ( o.isIndividual ?= 'individual' )?
    ( o.portionKind = PortionKind
        o.ownedRelationship += PortioningFeatureMember )?

OccurrenceUsage (m : Membership) : OccurrenceUsage =
    OccurrenceUsagePrefix(this) 'occurrence' Usage(this, m)

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

```

PortionUsage (m : Membership) : OccurrenceUsage =
    UsagePrefix(this) ( isIndividual ?= 'individual' )?
    portionKind = PortionKind
    ownedRelationship += PortioningFeatureMember
    Usage(this, m)

PortionKind : PortionKind =
    'snapshot' | 'timeslice'

PortioningFeatureMember : FeatureMembership =
    ownedMemberFeature = PortioningFeature

PortioningFeature : PortioningFeature =
    {}

EventOccurrenceUsage (m : Membership) : EventOccurrenceUsage =
    OccurrenceUsagePrefix(this) 'event'
    ( ownedRelationship += OwnedSubsetting
        FeatureSpecializationPart(this)?
    | 'occurrence' UsageDeclaration(this, m)? )
    UsageCompletion(this)

```

8.2.2.9.3 Occurrence Successions

```

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

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

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

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

ConnectorEndFor (f : Feature) : Feature =
    ownedRelationship += SubsettingTo(f)
    ( ownedRelationship += OwnedMultiplicity )?

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

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

SubsettingTo (f : Feature) : Subsetting =
    { subsettedFeature = f }

```

8.2.2.10 Items Textual Notation

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

ItemUsage (m : Membership) : ItemUsage =
    OccurrenceUsagePrefix(this) 'item' Usage(this, m)
```

8.2.2.11 Parts Textual Notation

```
PartDefinition (m : Membership) : PartDefinition =
    OccurrenceDefinitionPrefix(this) 'part' 'def' Definition(this, m)

PartUsage (m : Membership) : PartUsage =
    OccurrenceUsagePrefix(this) 'part' Usage(this, m)
```

8.2.2.12 Ports Textual Notation

```
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

ConjugatedPortTyping : ConjugatedPortTyping =
    '~' originalPortDefinition = [QualifiedName]

PortUsage (m : Membership) : PortUsage =
    OccurrenceUsagePrefix(this) 'port' Usage(this, m)
```

8.2.2.13 Connections Textual Notation

8.2.2.13.1 Connection Definition and Usage

```
ConnectionDefinition (m : Membership) : ConnectionDefinition =
    OccurrenceDefinitionPrefix(this) 'connection' 'def' Definition(this, m)

ConnectionUsage (m : Membership) : ConnectionUsage =
    OccurrenceUsagePrefix(this)
    ( 'connection' UsageDeclaration(this, m)
    ( 'connect' ConnectorPart(this) )?
    | 'connect' ConnectorPart(this) )
    UsageBody(this)

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

```

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

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

ConnectorEndMember : EndFeatureMembership :
  ( memberName = NAME ':>' )? ownedMemberFeature = ConnectorEnd

ConnectorEnd : Feature =
  ownedRelationship += OwnedSubsetting
  ( ownedRelationship += OwnedMultiplicity )?

```

8.2.2.13.2 Binding Connectors

```

BindingConnector (m : Membership) : BindingConnectorAsUsage =
  UsagePrefix(this) 'bind'
  BindingConnectorDeclaration(this, m) DefinitionBody(m)

BindingConnectorDeclaration (c : BindingConnectorAsUsage, m : Membership) =
  ( UsageDeclaration(c, m) 'as' )?
  c.ownedRelationship += ConnectorEndMember '='
  c.ownedRelationship += ConnectorEndMember

```

8.2.2.13.3 Successions

```

Succession (m : Membership) : SuccessionAsUsage =
  ( isAbstract ?= 'abstract' )? 'succession'
  SuccessionDeclaration(this, m) DefinitionBody(this)

SuccessionDeclaration (s : SuccessionAsUsage, m : Membership) =
  ( UsageDeclaration(s, m)? 'first' )?
  s.ownedRelationship += ConnectorEndMember 'then'
  s.ownedRelationship += ConnectorEndMember

```

8.2.2.13.4 Messages and Flow Connection Usages

```

Message (m : Membership) : FlowConnectionUsage =
  OccurrenceUsagePrefix 'message'
  MessageDeclaration(this, m) DefinitionBody(this)
  { isAbstract = true }

MessageDeclaration (c : FlowConnectionUsage, m : Membership) =
  UsageDeclaration(c, m)
  ( 'of' c.ownedRelationship += ItemFeatureMember )?
  ( 'from' BinaryConnectorPart | ValuePart(c) )?

FlowConnectionUsage (m : Membership) : FlowConnectionUsage =
  OccurrenceUsagePrefix 'stream'
  FlowConnectionDeclaration(this, m) DefinitionBody(this)

```

```

SuccessionFlowConnectionUsage (m : Membership) : SuccessionFlowConnectionUsage =
    OccurrenceUsagePrefix 'flow'
    FlowConnectionDeclaration(this, m) DefinitionBody(this)

FlowConnectionDeclaration (c : FlowConnectionUsage, m : Membership) =
    ( UsageDeclaration(i, m)
        ( 'of' c.ownedRelationship += ItemFeatureMember
        | c.ownedRelationship += EmptyItemFeatureMember )
        'from'
        | c.ownedRelationship += EmptyItemFeatureMember
    )
    c.ownedRelationship += ItemFlowEndMember 'to'
    c.ownedRelationship += ItemFlowEndMember

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

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

EmptyItemFeatureMember : FeatureMembership =
    ownedMemberFeature = EmptyItemFeature

EmptyItemFeature : Feature =
    {}

ItemFlowEndMember : FeatureMembership =
    ownedMemberFeature = ItemFlowEnd

ItemFlowEnd : Feature =
    ( ownedRelationship += ItemFlowEndSubsetting )?
    ownedRelationship += ItemFlowFeatureMember

ItemFlowEndSubsetting : Subsetting =
    subsettetedFeature = [QualifiedName]
    | subsettetedFeature = FeatureChainPrefix
        { ownedRelatedElement += subsettetedFeature }

FeatureChainPrefix : Feature =
    ( ownedRelationship += OwnedFeatureChaining '.' )+
    ownedRelationship += OwnedFeatureChaining '.'

ItemFlowFeatureMember : FeatureMembership =
    ownedMemberFeature = ItemFlowFeature

ItemFlowFeature : Feature =
    ownedRelationship += ItemFlowRedefinition

ItemFlowRefefinition : Redefinition =
    redefinedFeature = [QualifiedName]

```

8.2.2.14 Interfaces Textual Notation

8.2.2.14.1 Interface Definitions

```
InterfaceDefinition (m : Membership) : InterfaceDefinition =
    OccurrenceDefinitionPrefix(this) 'interface' 'def'
    DefinitionDeclaration(this, m) InterfaceBody(this)

InterfaceBody (t : Type) =
    ';' | '{' InterfaceBodyItem(t)* '}''

InterfaceBodyItem (t : Type) =
    t.ownedRelationship += OwnedDocumentation
    | t.ownedRelationship += DefinitionMember(t)
    | t.ownedRelationship += VariantUsageMember
    | t.ownedRelationship += InterfaceNonOccurrenceUsageMember
    | t.ownedRelationship += InterfaceOccurrenceUsageMember(t)
    | t.ownedRelationship += InterfaceOccurrenceSuccessionMember(t)
    | t.ownedRelationship += AliasMember
    | t.ownedRelationship += Import

InterfaceNonOccurrenceMember : FeatureMembership =
    MemberPrefix(this) ownedMemberFeature = InterfaceNonOccurrenceUsageElement(this)

InterfaceNonOccurrenceUsageElement (m : Membership) : Usage =
    ReferenceUsage(m)
    | AttributeUsage(m)
    | EnumerationUsage(m)
    | BindingConnector(m)
    | Succession(m)

InterfaceOccurrenceUsageMember : FeatureMembership =
    ownedMemberFeature = InterfaceOccurrenceUsageElement(this)

InterfaceOccurrenceSuccessionMember (t : Type) : FeatureMembership =
    s = SourceSuccessionMember(t)
    InterfaceOccurrenceUsageMember
    TargetEndFor(s.ownedMemberFeature, ownedMemberFeature)

InterfaceOccurrenceUsageElement (m : Membership) : Feature =
    DefaultInterfaceEnd(m) | StructureUsageElement(m) | BehaviorUsageElement(m)

DefaultInterfaceEnd (m : Membership) : PortUsage =
    ( direction = FeatureDirection )?
    ( isAbstract ?= 'abstract' | isVariation ?= 'variation')?
    isEnd ?= 'end' Usage(this, m)
```

8.2.2.14.2 Interface Usages

```
InterfaceUsage (m : Membership) : InterfaceUsage =
    OccurrenceUsagePrefix(this) 'interface'
```

```

InterfaceUsageDeclaration(this, m) InterfaceBody(this)

InterfaceUsageDeclaration (i : InterfaceUsage, m : Membership) =
    UsageDeclaration(i, m) ( 'connect' InterfacePart(i) )?
    | InterfacePart(i)

InterfacePart (i: InterfaceUsage) =
    BinaryInterfacePart(i) | NaryInterfacePart(i)

BinaryInterfacePart (i : InterfaceUsage) =
    i.ownedRelationship += InterfaceEndMember 'to'
    i.ownedRelationship += InterfaceEndMember

NaryInterfacePart (i : InterfaceUsage) =
    '(' i.ownedRelationship += InterfaceEndMember ','
        i.ownedRelationship += InterfaceEndMember
        ( ',' i.ownedRelationship += InterfaceEndMember )* ')'

InterfaceEndMember : EndFeatureMembership =
    ( memberName = Name ':>' )? ownedMemberFeature = InterfaceEnd

InterfaceEnd : PortUsage :
    ownedRelationship += OwnedSubsetting
    ( ownedRelationship += OwnedMultiplicity )?

```

8.2.2.15 Allocations Textual Notation

```

AllocationDefinition (m : Membership) : AllocationDefinition =
    OccurrenceDefinitionPrefix(this) 'allocation' 'def' Definition(this, m)

AllocationUsage (m : Membership) : AllocationUsage =
    OccurrenceUsagePrefix(this)
    AllocationUsageDeclaration(this, m) UsageBody(this)

AllocationUsageDeclaration (a : AllocationUsage, m : Membership) =
    'allocation' UsageDeclaration(a, m)
    ( 'allocate' ConnectorPart(a) )?
    | 'allocate' ConnectorPart(a)

```

8.2.2.16 Actions Textual Notation

8.2.2.16.1 Action Definitions

```

ActionDefinition (m : Membership) : ActionDefinition =
    OccurrenceDefinitionPrefix(m) 'action' 'def'
    ActionDefinitionDeclaration(this, m) ActionBody(this)

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

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

```

```

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

ActionBodyItem (t : Type) =
  NonBehaviorBodyItem(t)
  | t.ownedRelationship += ActionBehaviorMember(t)
    ( t.ownedRelationship += ActionTargetSuccessionMember(t) )*
  | t.ownedRelationship += GuardedSuccessionMember

NonBehaviorBodyItem (t : Type) =
  t.ownedRelationship += OwnedDocumentation
  | t.ownedRelationship += Import
  | t.ownedRelationship += AliasMember
  | t.ownedRelationship += DefinitionMember(t)
  | t.ownedRelationship += VariantUsageMember
  | t.ownedRelationship += NonOccurrenceUsageMember
  | t.ownedRelationship += StructureUsageMember
  | t.ownedRelationship += StructureSuccessionMember(t)

StructureSuccessionMember (t : Type) : FeatureMembership =
  s = SourceSuccessionMember(t)
  StructureUsageMember
  TargetEndFor(s.ownedMemberFeature, ownedMemberFeature)

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

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

InitialNodeMember : FeatureMembership =
  MemberPrefix(this) 'first' memberFeature = [QualifiedName] ';'

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

ActionTargetSuccessionMember (t : Type) : FeatureMembership =
  MemberPrefix(this) ownedMemberFeature = ActionTargetSuccession(t) ','

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

```

8.2.2.16.2 Action Usages

```

ActionUsage (m : Membership) : ActionUsage =
  OccurrenceUsagePrefix(this) 'action'
  ActionUsageDeclaration(this, m) ActionBody(this)

ActionUsageDeclaration (a : ActionUsage, m : Membership) =

```

```

UsageDeclaration(a, m) ( ValuePart(a) | ActionUsageParameterList(a) )?

PerformActionUsage (m : Membership) : PerformActionUsage =
  OccurrenceUsagePrefix(this) 'perform'
  PerformActionUsageDeclaration(this, m) ActionBody(this)

PerformActionUsageDeclaration (a : PerformActionUsage, m : Membership) =
  ( a.ownedRelationship += OwnedSubsetting FeatureSpecializationPart(a) ?
    | 'action' UsageDeclaration(a, m)
    ( ValuePart(a) | ActionUsageParameterList(a) )?

ActionUsageParameterList (f : Feature) =
  '(' ( f.ownedRelationship += ActionUsageParameterMember
    ( ',' f.ownedRelationship += ActionUsageParameterMember )* )? ')'

ActionUsageParameterMember : ParameterMembership =
  ownedMemberParameter = ActionUsageParameter(this)

ActionUsageParameter (m : Membership) : Usage =
  ( Parameter(m) | EmptyParameter )
  ValueOrFlowPart(this)?

```

8.2.2.16.3 Action Parameters

```

EmptyParameterMember : ParameterMembership =
  ownedMemberFeature = EmptyParameter

EmptyParameter : ReferenceUsage :
  {}

ParameterMember : ParameterMembership =
  ownedMemberParameter = Parameter(this)

Parameter(m : Membership) : Feature =
  ReferenceParameterDeclaration(m)
  | AttributeParameterDeclaration(m)
  | OccurrenceUsageDeclaration(m)
  | ItemParameterDeclaration(m)
  | PartParameterDeclaration(m)
  | RenderingParameterDeclaration(m)
  | ActionParameterDeclaration(m)
  | CalculationParameterDeclaration(m)
  | StateParameterDeclaration(m)
  | ConstraintParameterDeclaration(m)
  | RequirementParameterDeclaration(m)
  | ConcerParameterDeclaration(m)
  | AnalysisCaseParameterDeclaration(m)
  | VerificationCaseDeclaration(m)
  | UseCaseParameterDeclaration(m)
  | ViewpointParamterDeclaration(m)

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

```

```

AttributeParameterDeclaration (m : Membership) : AttributeUsage =
    'attribute' ParameterDeclaration(this, m)

OccurrenceParameterDeclaration (m : Membership) : OccurrenceUsage =
    'occurrence' ParameterDeclaration(this, m)

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

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

ViewParameterDeclaration (m : Membership) : ViewUsage =
    'view' ParameterDeclaration(this, m)

RenderingParameterDeclaration (m : Membership) : RenderingUsage =
    'rendering' ParameterDeclaration(this, m)

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

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

StateParameterDeclaration (m : Membership) : StateUsage =
    'state' ParameterDeclaration(this, m)

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

RequirementParameterDeclaration (m : Membership) : RequirementUsage =
    'requirement' ParameterDeclaration(this, m)

ConcernParameterDeclaration (m : Membership) : ConcernUsage =
    'concern' ParameterDeclaration(this, m)

AnalysisCaseParameterDeclaration (m : Membership) : AnaysisCaseUsage =
    'analysis' ParameterDeclaration(this, m)

VerificationCaseDeclaration (m : Membership) : VerificationCaseUsage =
    'verification' ParameterDeclaration(this, m)

UseCaseParameterDeclaration (m : Membership) : UseCaseUsage =
    'use' 'case' ParameterDeclaration(this, m)

ViewpointParameterDeclaration (m : Membership) : ViewpointUsage =
    'viewpoint' ParameterDeclaration(this, m)

ParameterDeclaration (u : Usage, m : Membership) =
    Identification(u, m) ParameterSpecializationPart(u)

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

```

```

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

8.2.2.16.4 Action Nodes

ActionNode (m : Membership) : ActionUsage =
    SendNode(m) | AcceptNode(m) | ControlNode(m)

AcceptNode (m : Membership) : AcceptActionUsage =
    OccurrenceUsagePrefix(this)
    AcceptNodeDeclaration(this, m) ActionBody(this)

AcceptNodeDeclaration (a : AcceptActionUsage, m : Membership) =
    ( 'action' UsageDeclaration(a, m) )?
    'accept' AcceptParameterPart(a)

AcceptParameterPart (a : AcceptActionUsage) =
    a.ownedRelationship += ItemParameterMember
    ( 'via' a.ownedRelationship += NodeParameterMember
    | a.ownedRelationship += EmptyParameterMember )

ItemParameterMember : ParameterMembership =
    ownedMemberParameter = ItemParameter(this)

ItemParameter (m : Membership) : ReferenceUsage =
    Identification(this, m) ItemParameterSpecializationPart
    | ownedRelationship += OwnedFeatureTyping
    ( ownedRelationship += OwnedMultiplicity )?

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

SendNode (m : Membership) : SendActionUsage =
    OccurrenceUsagePrefix(this)
    SendNodeDeclaration(this, m) ActionBody(this)

SendNodeDeclaration (a : SendActionUsage, m : Membership) =
    ( 'action' UsageDeclaration(a, m) )?
    'send' ownedRelationship += NodeParameterMember
    'to' ownedRelationship += NodeParameterMember

NodeParameterMember : ParameterMembership =
    ownedMemberParameter = NodeParameter

NodeParameter : ReferenceUsage =
    ownedRelationship += FeatureBinding

FeatureBinding : FeatureValue =
    value = ownedExpression

ControlNode (m : Membership) : ControlNode =
    MergeNode(m) | DecisionNode(m) | JoinNode(m) | ForkNode(m)

```

```

MergeNode (m : Membership) : MergeNode =
    OccurrenceUsagePrefix(this)
    isComposite ?= 'merge' UsageDeclaration(this, m) ';'

DecisionNode (m : Membership) : DecisionNode =
    OccurrenceUsagePrefix(this)
    isComposite ?= 'decide' UsageDeclaration(this, m) ';'

JoinNode (m : Membership) : JoinNode =
    OccurrenceUsagePrefix(this)
    isComposite ?= 'join' UsageDeclaration(this, m) ';'

ForkNode (m : Membership) : ForkNode =
    OccurrenceUsagePrefix(this)
    isComposite ?= 'fork' UsageDeclaration(this, m) ';'

```

8.2.2.16.5 Action Successions

```

ActionTargetSuccession (t : Type) : Feature =
    TargetSuccession(t) | GuardedTargetSuccession(t) | DefaultTargetSuccession(t)

TargetSuccessionMember (t : Type) : FeatureMembership =
    ownedMemberFeature = TargetSuccession(t)

TargetSuccession (t : Type) : SuccessionAsUsage =
    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' )?
    s = TransitionSourceMember
    { ownedRelationship += s }
    ownedRelationship += GuardExpressionMember
    'then' ownedRelationship += TransitionSuccessionMember(s.memberElement)

```

8.2.2.17 States Textual Notation

8.2.2.17.1 State Definitions

```

StateDefinition (m : Membership) : StateDefinition =
    OccurrenceDefinitionPrefix(this) 'state' 'def'
    ActionDeclaration (this, m) StateDefBody(this)

StateDefBody (s : StateDefinition) =
    ';;'
    | ( s.isParallel ?= 'parallel' )?
    '{' StateBodyItem(s)* '}'

```

```

StateBodyItem (t : Type) =
    NonBehaviorBodyItem(t)
    | t.ownedRelationship =
        ( BehaviorUsageMember(t)
        | BehaviorUsageSuccessionMember(t) )
        ( t.ownedRelationship = TargetTransitionUsageMember(t) )*
    | t.ownedRelationship += TransitionUsageMember
    | EntryActionMember(t)
    | t.ownedRelationship += DoActionMember
    | t.ownedRelationship += ExitActionMember

BehaviorUsageSuccessionMember (t : Type) : FeatureMembership =
    s = SourceSuccessionMember(t)
    BehaviorUsageMember
    TargetEndFor(s.ownedMemberFeature, ownedMemberFeature)

EntryActionMember (t : Type) : StateSubactionMembership =
    MemberPrefix(this) kind = 'entry'
    ownedMemberFeature = StateActionUsage
    { t.ownedRelationship += this }
    ( t.ownedRelationship += EntryTransitionMember(this) )*

DoActionMember : StateSubactionMembership =
    MemberPrefix(this) kind = 'do'
    ownedMemberFeature = StateActionUsage

ExitActionMember : StateSubactionMembership =
    MemberPrefix(this) kind = 'exit'
    ownedMemberFeature = StateActionUsage

EntryTransitionMember (f : Feature) : FeatureMembership :
    MemberPrefix(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) : AcceptActionUsage =
    AcceptNodeDeclaration(this, m) ActionBody(this)

StateSendActionUsage (m : Membership) : SendActionUsage
    SendNodeDeclaration(this, m) ActionBody(this)

```

```

TransitionUsageMember : FeatureMembership =
    MemberPrefix(this) ownedMemberFeature = TransitionUsage(this) ','

TargetTransitionUsageMember (t : Type) : FeatureMembership =
    MemberPrefix(this) ownedMemberFeature = TargetTransitionUsage(t) ','

```

8.2.2.17.2 State Usages

```

StateUsage (m : Membership) : StateUsage =
    OccurrenceUsagePrefix(this) 'state'
    ActionUsageDeclaration(this, m) StateUsageBody(this)

StateUsageBody (s : StateUsage) =
    ';' |
    | ( s.isParallel ?= 'parallel' )?
    | '{' StateBodyItem(s)* '}' |

ExhibitStateUsage (m : Membership) : ExhibitStateUsage =
    OccurrenceUsagePrefix(this) 'exhibit'
    ( ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this) ?
    | 'state' UsageDeclaration(this, m) )
    ( ValuePart(this) | ActionUsageParameterList(this) )?
    StateUsageBody(this)

```

8.2.2.17.3 Transition Usages

```

TransitionUsage (m : Membership) : TransitionUsage =
    'transition' ( UsageDeclaration(this, m) 'first' )?
    s = TransitionSourceMember
    { ownedRelationship += s }
    ownedRelationship += EmptyParameterMember
    ( ownedRelationship += EmptyParameterMember
        ownedRelationship += TriggerActionMember )?
    ( ownedRelationship += GuardExpressionMember )?
    ( ownedRelationship += EffectBehaviorMember )?
    'then' ownedRelationship += TransitionSuccessionMember(s.memberElement)

TargetTransitionUsage (t : Type) : TransitionUsage =
    ownedRelationship += EmptyParameterMember
    ( 'transition'
        ( ownedRelationship += EmptyParameterMember
            ownedRelationship += TriggerActionMember )?
        ( ownedRelationship += GuardExpressionMember )?
        ( ownedRelationship += EffectBehaviorMember )?
    | ownedRelationship += EmptyParameterMember
        ownedRelationship += TriggerActionMember
        ( ownedRelationship += GuardExpressionMember )?
        ( ownedRelationship += EffectBehaviorMember )?
    | ownedRelationship += GuardExpressionMember
        ( ownedRelationship += EffectBehaviorMember )?
    )?
    'then' ownedRelationship += TargetSuccessionMember(t)

```

```

TransitionSourceMember : Membership =
    memberElement = [QualifiedName]
    | memberElement = FeatureChain
        { ownedRelatedElement += FeatureChain }

TriggerActionMember : TransitionFeatureMembership =
    'accept' { kind = 'trigger' } ownedMemberFeature = TriggerAction

TriggerAction : AcceptActionUsage =
    AcceptParameterPart(this)

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(this)* '}' )?

TransitionAcceptActionUsage (m : Membership) : AcceptActionUsage =
    AcceptNodeDeclaration(this, m) ( '{' ActionBodyItem(this)* '}' )?

TransitionSendActionUsage (m : Membership) : SendActionUsage =
    SendNodeDeclaration(this, m) ( '{' ActionBodyItem(this)* '}' )?

TransitionSuccessionMember (f : Feature) : FeatureMembership =
    ownedMemberFeature = TransitionSuccession(f)

TransitionSuccession (f : Feature) : Succession =
    ownedRelationship += ConnectorEndMemberFor(f)
    ownedRelationship += ConnectorEndMember

```

8.2.2.18 Calculations Textual Notation

8.2.2.18.1 Calculation Definitions

```

CalculationDefinition (m : Membership) : CalculationDefinition =
    OccurrenceDefinitionPrefix(this) 'calc' 'def'
    CalculationDeclaration(this, m)
    ( CalculationBody(this)
    | '=' ownedRelationship += ResultExpressionMember ';' )
)

CalculationDeclaration (c : CalculationDefinition, m : Membership) =
    DefinitionDeclaration(c, m)
    ( ParameterList(c) ReturnParameterPart(c) ? )?

```

```

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 =
    MemberPrefix(this)? 'return' ownedMemberParameter = UsageElement(this)

ResultExpressionMember : ResultExpressionMembership =
    MemberPrefix(this)? ownedResultExpression = OwnedExpression

```

8.2.2.18.2 Calculation Usages

```

CalculationUsage (m : Membership) : CalculationUsage =
    OccurrenceUsagePrefix(this) 'calc'
    CalculationUsageDeclaration(this, m) CalculationBody(this)

CalculationUsageDeclaration (u : Usage, m : Membership) =
    UsageDeclaration(this, m) ( ValuePart(u) | CalculationUsageParameterPart(u) )?

CalculationUsageParameterPart (u : Usage) =
    ActionUsageParameterList(u)
    ( ownedRelationship += CalculationReturnParameterMember )?

CalculationReturnParameterMember : ReturnParameterMembership =
    'return'? ownedMemberParameter = ActionUsageParameter(this)

```

8.2.2.19 Constraints Textual Notation

```

ConstraintDefinition (m : Membership) : ConstraintDefinition =
    OccurrenceDefinitionPrefix(this) 'constraint' 'def'
    ConstraintDeclaration(this, m) CalculationBody(this)

ConstraintDeclaration (c : ConstrainDefinition, m : Membership) =
    DefinitionDeclaration(c, m) ParameterList(c)?

ConstraintUsage (m : Membership) : ConstraintUsage =
    OccurrenceUsagePrefix(this) 'constraint'
    CalculationUsageDeclaration(this, m) CalculationBody(this)

AssertConstraintUsage (m : Membership) : AssertConstraintUsage =
    OccurrenceUsagePrefix(this) 'assert' ( isNegated ?= 'not' )?
    ( ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this) )

```

```

| 'constraint' UsageDeclaration(this, m) )
CalculationUsageParameterPart(this) CalculationBody(this)

```

8.2.2.20 Requirements Textual Notation

8.2.2.20.1 Requirement Definitions

```

RequirementDefinition (m : Membership) : RequirementDefinition =
    OccurrenceDefinitionPrefix(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 += FramedConcernMember
    | t.ownedRelationship += RequirementVerificationMember
    | t.ownedRelationship += ActorMember
    | t.ownedRelationship += StakeholderMember

SubjectMember : SubjectMembership =
    MemberPrefix(this) ownedSubjectParameter = SubjectUsage(this)

SubjectUsage (m : Membership) : ReferenceUsage =
    'subject' Usage(m)

RequirementConstraintMember : RequirementConstraintMembership =
    MemberPrefix(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)

FramedConcernMember : FramedConcernMembership =
    MemberPrefix(this)? 'frame'
    ownedMemberFeature = FramedConcernUsage(this)

FramedConcernUsage (m : Membership) : ConcernUsage =
    ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this)?
    CalculationParameterPart(this) RequirementBody(this)
    | 'concern' CalculationUsageDeclaration(this, m) CalculationBody(this)

ActorMember : ActorMembership =
    MemberPrefix(this) ownedSubjectParameter = ActorUsage(this)

```

```

ActorUsage (m : Membership) : PartUsage =
    'actor' Usage(m)

StakeholderMember : StakeholderMembership =
    MemberPrefix(this) ownedSubjectParameter = SubjectUsage(this)

StakeholderUsage (m : Membership) : StakeholderUsage =
    'stakeholder' Usage(m)

```

8.2.2.20.2 Requirement Usages

```

RequirementUsage (m : Membership) : RequirementUsage =
    OccurrenceUsagePrefix(this) 'requirement'
    CalculationUsageDeclaration(this, m) RequirementBody(this)

SatisfyRequirementUsage : SatisfyRequirementUsage =
    UsagePrefix(this)? OccurrenceUsagePrefix(this) 'satisfy'
    ( ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this) ?
    | 'requirement' UsageDeclaration(this, m) )
    ( ValuePart(this) | ActionParameterList(this) )
    ( 'by' ownedRelationship += EmptySubjectParameterMember
    ownedRelationship += SatisfactionConnectorMember(this) )?
    RequirementBody(this)

EmptySubjectParameterMember : SubjectMembership =
    ownedSubjectParameter = EmptySubjectParameter

EmptySubjectParameter : ReferenceUsage =
    {}

SatisfactionConnectorMember (s : SatisfyRequirementUsage) : Membership =
    ownedMemberFeature = SatisfactionConnector(s)

SatisfactionConnector (s : SatisfyRequirementUsage) : BindingConnector =
    ownedRelationship += ConnectorEndMemberFor(s.subjectParameter)
    ownedRelationship += ConnectorEndMember

```

8.2.2.20.3 Concerns

```

ConcernDefinition (m : Membership) : ConcernDefinition =
    OccurrenceDefinitionPrefix(this) 'concern' 'def'
    ConstraintDeclaration(this, m) RequirementBody(this)?

ConcernUsage (m : Membership) : ConcernUsage =
    OccurrenceUsagePrefix(this) 'concern'
    CalculationUsageDeclaration(this, m) RequirementBody(this)

```

8.2.2.21 Cases Textual Notation

```

CaseDefinition (m : Membership) : CaseDefinition =
    OccurrenceDefinitionPrefix(this) 'case' 'def'
    CalculationDeclaration(this, m) CaseBody(this)

CaseUsage (m : Membership) : CaseUsage =

```

```

OccurrenceUsagePrefix(this) 'case'
CalculationUsageDeclaration(this, m) CaseBody(this)

CaseBody (t : Type) =
  ';'*
| '{' CaseBodyItem(t)*
  ( t.ownedRelationship += ResultExpressionMember )?
' }'

CaseBodyItem (t : Type) =
  ActionBodyItem(t)
| t.ownedRelationship += SubjectMember
| t.ownedRelationship += ActorMember
| t.ownedRelationship += ObjectiveMember

ObjectiveMember : ObjectiveMembership =
  MemberPrefix(this) 'objective'
  ownedObjectiveRequirement = ObjectiveRequirementUsage(this)

ObjectiveRequirementUsage (m : Membership) : RequirementUsage =
  CalculationUsageDeclaration(this, m) RequirementBody(this)

```

8.2.22 Analysis Cases Textual Notation

```

AnalysisCaseDefinition (m : Membership) : AnalysisCaseDefinition =
  OccurrenceDefinitionPrefix(this) 'analysis' 'def'
  CalculationDeclaration(this, m) CaseBody(this)

AnalysisCaseUsage (m : Membership) : CaseUsage =
  OccurrenceUsagePrefix(this) 'analysis'
  CalculationUsageDeclaration(this, m) CaseBody(this)

```

8.2.23 Verification Cases Textual Notation

```

VerificationCaseDefinition (m : Membership) : VerificationCaseDefinition =
  OccurrenceDefinitionPrefix(this) 'verification' 'def'
  CalculationDeclaration(this, m) CaseBody(this)

VerificationCaseUsage (m : Membership) : VerificationCaseUsage =
  OccurrenceUsagePrefix(this) 'verification'
  CalculationUsageDeclaration(this, m) CaseBody(this)

RequirementVerificationMember : RequirementVerificationMembership =
  MemberPrefix kind = RequirementVerificationKind(this)
  ownedRequirement = RequirementVerificationUsage(this)

RequirementVerificationKind (m : RequirementVerificationMembership) =
  'verify' { m.kind = 'requirement' }

RequirementVerificationUsage (m : Membership) : RequirementUsage =
  ownedRelationship += OwnedSubsetting FeatureSpecialization(this)*
  CalculationUsageParameterPart(this)? RequirementBody(this)
  | 'requirement' CalculationUsageDeclaration(this, m) RequirementBody(this)

```

8.2.2.24 Use Cases Textual Notation

```
UseCaseDefinition (m : Membership) : UseCaseDefinition =
    OccurrenceDefinitionPrefix(this) 'use' 'case' 'def'
    CalculationDeclaration(this, m) CaseBody(this)

UseCaseUsage (m : Membership) : UseCaseUsage =
    OccurrenceUsagePrefix(this) 'use' 'case'
    CalculationUsageDeclaration(this, m) CaseBody(this)

IncludeUseCaseUsage (m : Membership) : IncludeUseCaseUsage :
    OccurrenceUsagePrefix(this) 'include'
    ( ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this) ?
    | 'use' 'case' UsageDeclaration(this, m) )
    ( ValuePart(this) | ActionUsageParameterList(this) )?
    CaseBody(this)
```

8.2.2.25 Views and Viewpoints Textual Notation

8.2.2.25.1 View Definitions

```
ViewDefinition (m : Membership) : ViewDefinition =
    OccurrenceDefinitionPrefix(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 =
    MemberPrefix(this) 'render'
    ownedRendering = ViewRenderingUsage

ViewRenderingUsage : RenderingUsage =
    ownedRelationship += OwnedSubsetting FeatureSpecializationPart(this)?
    UsageBody(this)
```

8.2.2.25.2 View Usages

```
ViewUsage (m : Membership) : ViewUsage =
    OccurrenceUsagePrefix(this) '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 = VisibilityIndicator )?
'Expose' ( ImportedNamespace | ImportedFilterPackage ) ';'

```

8.2.2.25.3 Viewpoints

```

ViewpointDefinition (m : Membership) : ViewpointDefinition =
    OccurrenceDefinitionPrefix(this) 'viewpoint' 'def'
    ConstraintDeclaration(this, m) RequirementBody(this)

ViewpointUsage (m : Membership) : ViewpointUsage =
    OccurrenceUsagePrefix(this) 'viewpoint'
    CalculationUsageDeclaration(this, m) RequirementBody(this)

```

8.2.2.25.4 Renderings

```

RenderingDefinition (m : Membership) : RenderingDefinition =
    OccurrenceDefinitionPrefix(this) 'rendering' 'def'
    Definition(this, m)

RenderingUsage (m : Membership) : RenderingUsage =
    OccurrenceUsagePrefix(this) 'rendering'
    Usage(this, m)

```

8.2.3 Graphical Notation

8.2.3.1 Graphical Notation Overview

The SysML Graphical Notation is expressed using a simplified form of the EBNF notation used to define the SysML Textual Notation. This Graphical BNF has been extended to include productions with a mixture of graphical and textual elements. The following table summarizes the conventions used:

Table 32. Graphical BNF Conventions

Non-terminal element	non-terminal-element
Non-terminal element production (graphical)	non-terminal-element = graphical-elements
Non-terminal element production (textual)	non-terminal-element = textual-elements
Graphical line that connects other elements	&element graphical-line &element
Literal text element	'terminal'
Terminal text element without line breaks	string
Terminal text element with line breaks	text-block
Sequential elements	element1 element2
Alternative elements	element1 element2
Optional elements (zero or one)	element ?
Repeated elements (zero or more)	element *

Repeated elements (one or more)	element +
Grouping	(elements ...)

A graphical production contains a two-dimensional layout of graphical and textual elements including graphical shapes and lines. Shapes may contain other elements nested within these shapes. Generally speaking, graphical elements specify only containment and connectivity of graphical and textual elements out of which they are built. Shapes within the graphical notation may generally be relocated anywhere within a given graphical layout. They may also have any of their graphical elements stretched as necessary to hold their contents.

Lines that connect other graphical elements may be composed of one or more straight or curved line segments. Any of these line segments may contain a semicircular jump symbol where the segment overlaps a line segment of another connecting line.

A textual production contains only other textual productions. Terminal textual symbols include literal text elements, simple strings without line breaks, and multi-line blocks of text. Non-terminal textual productions can contain sample formatting, such as bold or italic fonts and permissible line breaks.

Submission Note. Only a limited part of the graphical notation is formally specified in this submission. The rest will be covered in the final submission.

8.2.3.2 Elements and Relationships Graphical Notation

```

graphical-view =
    (element-node) *

element-node =
    relationship-path-with-name
    | annotation-node
    | namespace-node

relationship-path-with-name =
    (relationship-name)?
    relationship-path-without-name

relationship-path-without-name =
    annotation
    | namespace-relationship
    | dependency-relationship
    | type-relationship

```

8.2.3.3 Annotations Graphical Notation

```

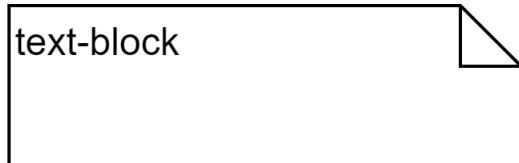
annotation =
    &annotated-element annotation-link &annotation-node

annotation-link =
    ●-----<br/>
    -----<br/>
    ●-----<br/>
    -----<br/>

annotation-node =
    comment-annotation-node

```

```
| documentation-annotation-node  
| textual-representation-annotation-node  
| metadata-feature-annotation-node  
  
comment-annotation-node =  
    comment-without-keyword  
    | comment-with-keyword  
    | comment-with-name  
  
comment-without-keyword =
```



```
comment-with-keyword =
```

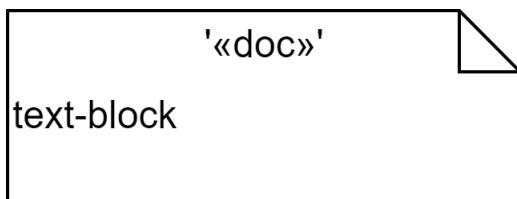


```
comment-with-name =
```

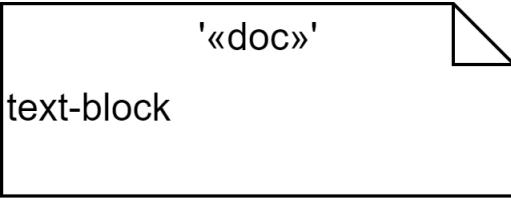


```
documentation-annotation-node =  
    documentation-annotation-without-name  
    | documentation-annotation-with-name
```

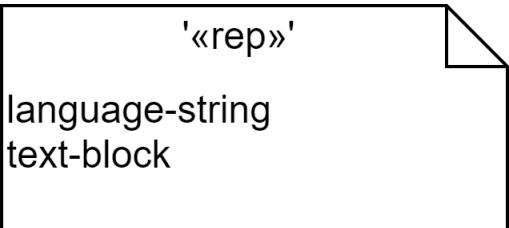
```
documentation-annotation-without-name =
```



```
documentation-annotation-with-name =
```

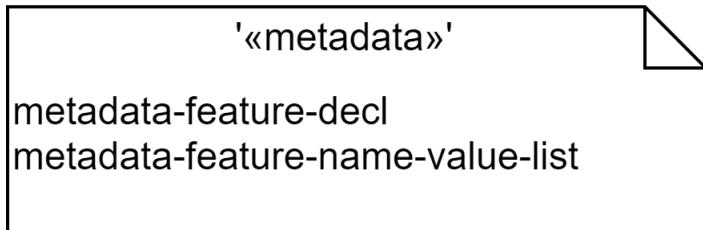


```
textual-representation-annotation-node =
```



```
language-string = string
```

```
metadata-feature-annotation-node =
```



```
metadata-feature-name-value-list =  
( metadata-feature-name '=' expression-text )+
```

```
metadata-feature-decl = string  
metadata-feature-name = string  
expression-text = string
```

A comment node may be attached to zero, one, or more than one annotated elements. All other annotation nodes must be attached to one and only one annotated element.

8.2.3.4 Namespaces and Packages Graphical Notation

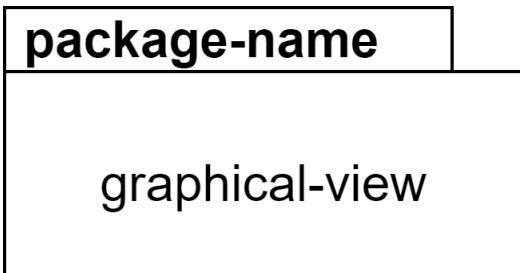
```
namespace-node =  
    package-node  
    | type-node  
  
namespace-relationship =  
    import  
    | recursive-import  
    | owned-membership  
    | unowned-membership  
  
package-node =
```

```
package-with-name-inside  
| package-with-name-in-tab  
| imported-package-with-name-inside  
| imported-package-with-name-in-tab
```

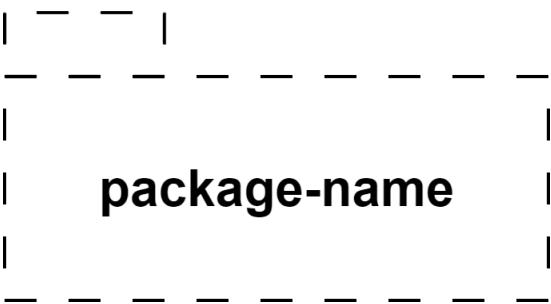
package-with-name-inside =



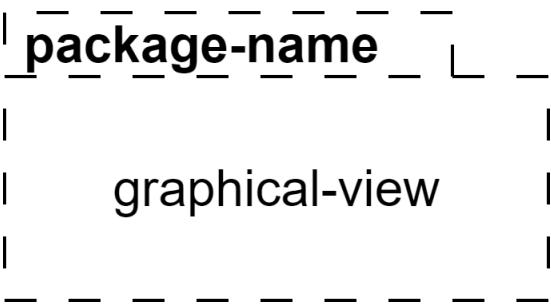
package-with-name-in-tab =



imported-package-with-name-inside =



imported-package-with-name-in-tab =



```

package-name = string

import =
    &namespace-node --> &namespace-node
        '«import»'

recursive-import =
    &namespace-node --> &namespace-node
        '«import [**]»'

owned-membership =
    &namespace-node ⊕ ----- &element-node

unowned-membership =
    &namespace-node ○ ----- &element-node

```

8.2.3.5 Dependencies Graphical Notation

```

dependency-path =
    binary-dependency
    | n-ary-dependency

binary-dependency =
    &element-node -----> &element-node

n-ary-dependency =
    &n-ary-association-dot (n-ary-dependency-client-or-supplier-link &element-node) +
    n-ary-dependency-client-or-supplier-link =
        n-ary-dependency-client-link
        | n-ary-dependency-supplier-link

n-ary-association-dot =
    ●

n-ary-dependency-client-link =
    -----
    ----->

n-ary-dependency-supplier-link =
    ----->

```

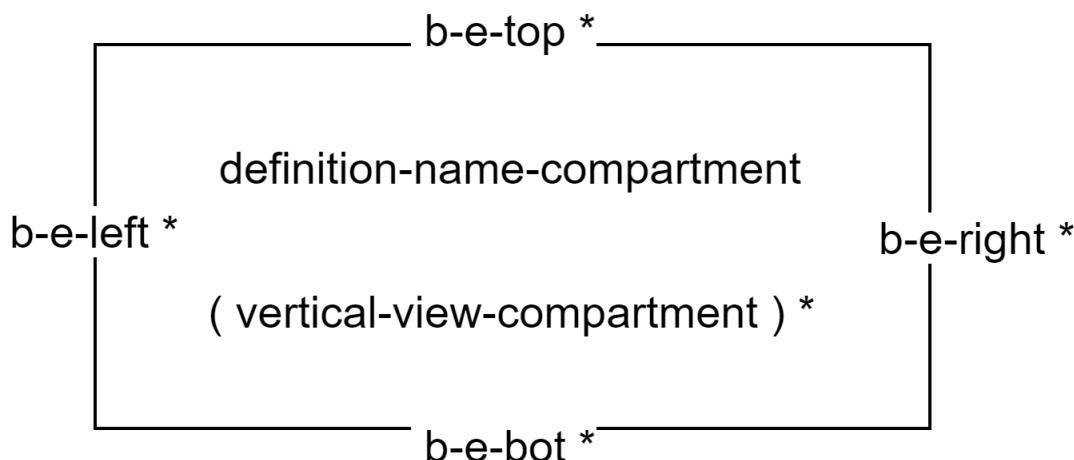
An n-ary dependency must have two or more client elements or two or more supplier elements.

8.2.3.6 Definition and Usage Graphical Notation

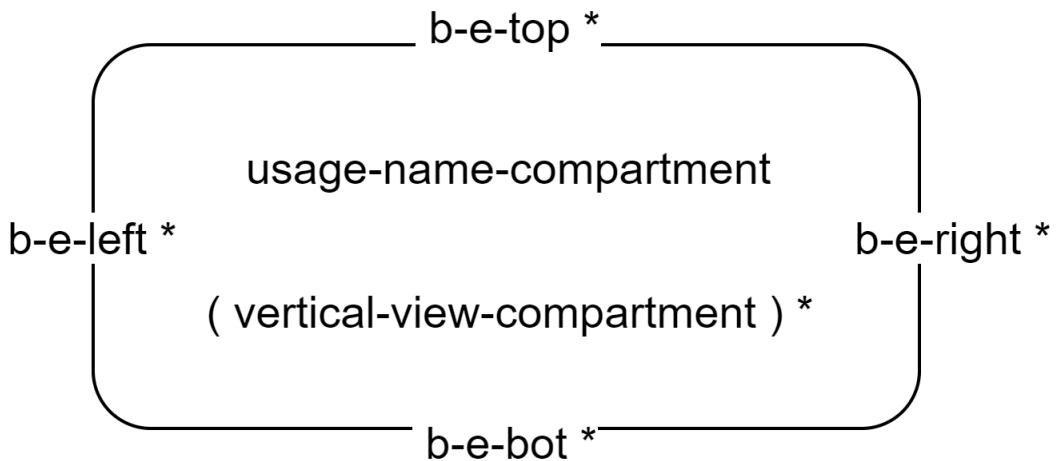
```
type-node =
    definition-node
| usage-node

type-relationship =
    specialization
| subsetting
| definition
| redefinition
| feature-membership
| variant-membership
| portion-membership
| connection-definition
| connection
| binding-connection
| item-flow
| item-flow-as-node
| message
| interface
| interface-as-node
| allocation
| succession
| perform
| perform-action
| transition
| exhibit
| satisfy
| expose
| frame

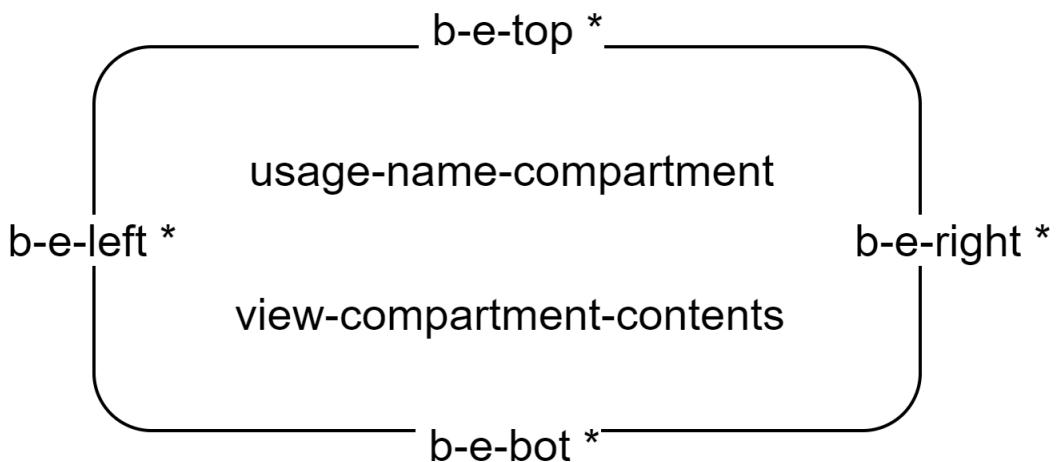
definition-node =
```



```
usage-node =
```



|



```
vertical-view-compartment =
```

view-compartment-contents

```
view-compartment-contents =
```

```
    actions-compartment
    | actions-compartment-graphical
    | actors-compartment
    | allocations-compartment
    | analyses-compartment
    | assert-constraints-compartment
    | attributes-compartment
```

```

| body-compartment
| calcs-compartment
| connections-compartment
| constraints-compartment
| directed-features-compartment
| documentation-compartment
| enums-compartment
| exhibited-by-compartment
| exhibit-states-compartment
| exposes-compartment
| frame-concerns-compartment
| include-actions-compartment
| individuals-compartment
| interfaces-compartment
| item-flows-compartment
| items-compartment
| metadata-compartment
| namespaces-compartment
| occurrences-compartment
| parameters-compartment
| parts-compartment
| parts-compartment-graphical
| perform-actions-compartment
| perform-actions-compartment-graphical
| performed-by-compartment
| ports-compartment
| queries-compartment
| related-elements-compartment
| relationships-compartment
| rendered-content-compartment
| renderings-compartment
| requirements-compartment
| satisfied-by-compartment
| satisfy-requirements-compartment
| shapes-compartment
| snapshots-compartment
| stakeholders-compartment
| states-compartment
| states-compartment-graphical
| timeslices-compartment
| variant-parts-compartment
| variants-compartment
| verification-methods-compartment
| verifications-compartment
| verified-by-compartment
| verify-requirements-compartment
| viewpoint-compartment
| viewpoint-compartment
| views-compartment

b-e-top =
    part-on-boundary-top
    | plain-port-top
    | in-port-top

```

```

| out-port-top
| inout-port-top
| parameter-top

b-e-bot =
    part-on-boundary-bot
| plain-port-bot
| in-port-bot
| out-port-bot
| inout-port-bot
| parameter-bot

b-e-left =
    part-on-boundary-left
| plain-port-left
| in-port-left
| out-port-left
| inout-port-left
| parameter-left

b-e-right =
    part-on-boundary-right
| plain-port-right
| in-port-right
| out-port-right
| inout-port-right
| parameter-right

```

8.2.3.7 Attributes Graphical Notation

8.2.3.8 Enumerations Graphical Notation

8.2.3.9 Occurrences Graphical Notation

8.2.3.10 Items Graphical Notation

8.2.3.11 Parts Graphical Notation

8.2.3.12 Ports Graphical Notation

8.2.3.13 Connections Graphical Notation

8.2.3.14 Interfaces Graphical Notation

8.2.3.15 Allocations Graphical Notation

8.2.3.16 Actions Graphical Notation

8.2.3.17 States Graphical Notation

8.2.3.18 Calculations Graphical Notation

8.2.3.19 Constraints Graphical Notation

8.2.3.20 Requirements Graphical Notation

8.2.3.21 Cases Graphical Notation

8.2.3.22 Analysis Cases Graphical Notation

8.2.3.23 Verification Cases Graphical Notation

8.2.3.24 Use Cases Graphical Notation

8.2.3.25 Views and Viewpoints Graphical Notation

8.3 Abstract Syntax

8.3.1 Abstract Syntax Overview

The *abstract syntax* is the common underlying syntactic representation for SysML models. The SysML textual or graphical notations (see [8.2](#)) provide for concrete presentation of models in the abstract syntax presentation. This concrete syntax notation may also be parsed to create or update the abstract syntax representation of models. The semantics for SysML models are then formally defined on the abstract syntax representation (see [8.4](#)).

The SysML abstract syntax is specified as a MOF model [MOF] that is an extension of the KerML abstract syntax model [KerML]. Each of the subsequent abstract subclauses describes one package in the abstract syntax model, including one or more overview diagrams and descriptions of each of the elements in the package. In the diagrams, metaclasses and relationships from the KerML abstract syntax are shown in gray. See [KerML] for the description of these elements.

8.3.2 Dependencies Abstract Syntax

8.3.2.1 Overview

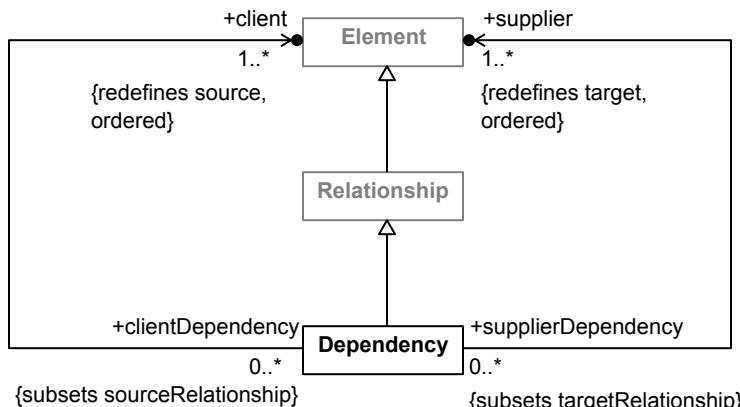


Figure 57. Dependencies

8.3.2.2 Dependency

Description

A Dependency is a Relationship that indicates that one or more `client` Elements require one or 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, ordered}

The Element or Elements dependent on the supplier elements.

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

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

Operations

No operations.

Constraints

No constraints.

8.3.3 Definition and Usage Abstract Syntax

8.3.3.1 Overview

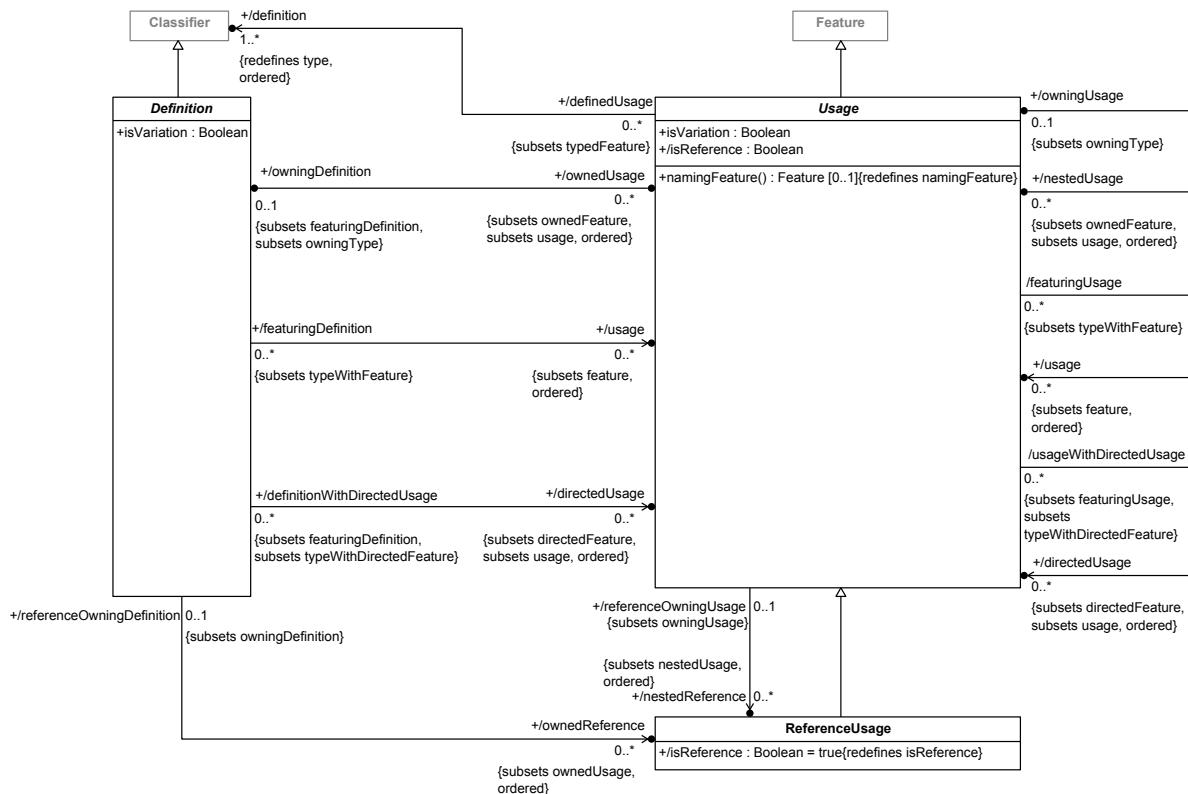


Figure 58. Definition and Usage

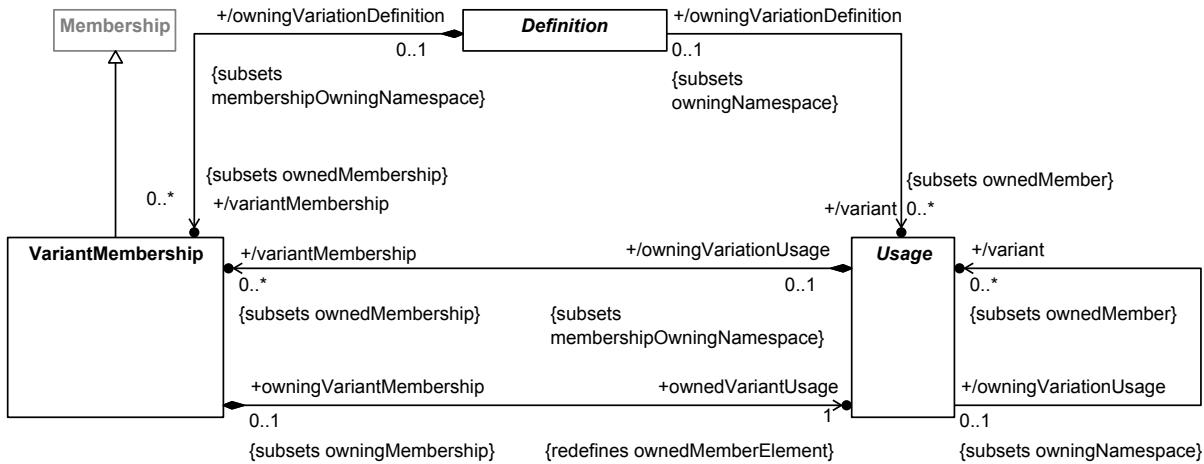


Figure 59. Variant Membership

8.3.3.2 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

`/directedUsage : Usage [0..*] {subsets usage, directedFeature, ordered}`

The usages of this Definition that are directedFeatures.

`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 ownedOccurrence, ordered}`

The ActionUsages that are ownedUsages of this Definition.

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

The AllocationUsages that are `ownedUsages` of this Definition.

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

The AnalysisCaseUsages that are `ownedUsages` of this Definition.

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

The AttributeUsages that are `ownedUsages` of this Definition.

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

The CalculationUsages that are `ownedUsages` of this Definition.

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

The CaseUsages that are `ownedUsages` of this Definition.

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

The ConcernUsages that are `ownedUsages` of this Definition.

`/ownedConnection : ConnectorAsUsage [0..*] {subsets ownedPart, ordered}`

The ConnectorAsUsages that are `ownedUsages` of this Definition. Note that this list includes BindingConnectorAsUsages and SuccessionAsUsages, even though these are ConnectorAsUsages but not ConnectionUsages.

`/ownedConstraint : ConstraintUsage [0..*] {subsets ownedOccurrence, ordered}`

The ConstraintUsages that are `ownedUsages` of this Definition.

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

The EnumerationUsages that are `ownedUsages` of this Definition.

`/ownedFlow : FlowConnectionUsage [0..*] {subsets ownedConnection}`

The FlowConnectionUsages that are `ownedUsages` of this Definition.

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

The InterfaceUsages that are `ownedUsages` of this Definition.

`/ownedItem : ItemUsage [0..*] {subsets ownedOccurrence, ordered}`

The ItemUsages that are `ownedUsages` of this Definition.

`/ownedOccurrence : OccurrenceUsage [0..*] {subsets ownedUsage, ordered}`

The OccurrenceUsages that are `ownedUsages` of this Definition.

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

The PartUsages that are ownedUsages of this Definition.

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

The PortUsages that are ownedUsages of this Definition.

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

The ReferenceUsages that are ownedUsages of this Definition.

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

The usages of this Definition that are RenderingUsages.

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

The RequirementUsages that are ownedUsages of this Definition.

/ownedState : StateUsage [0..*] {subsets ownedAction, ordered}

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, ordered}

The Usages that are ownedFeatures of this Definition.

/ownedUseCase : UseCaseUsage [0..*] {subsets ownedCase, ordered}

The UseCaseUsages that are ownedUsages of this Definition.

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

The ownedUsages of this Definition that are VerificationCaseUsages.

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

The ownedUsages of this Definition that are ViewUsages.

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

The ownedUsages of this Definition that are ViewpointUsages.

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

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, 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)

8.3.3.3 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

/isReference : Boolean {redefines isReference}

Always true for a ReferenceUsage.

Operations

No operations.

Constraints

No constraints.

8.3.3.4 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

/definition : Classifier [1..*] {redefines type, ordered}

The Classifiers that are the types of this Usage. Nominally, these are Definitions, but other kinds of Kernel Classifiers are also allowed, to permit use of Classifiers from the Kernel Library.

/directedUsage : Usage [0..*] {subsets usage, directedFeature, ordered}

The usages of this Usage that are directedFeatures.

/isReference : Boolean

Whether this Usage is a reference Usage, derived as the negation of isComposite.

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 nestedOccurrence, ordered}

The ActionUsages that are nestedUsages of this Usage.

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

The AllocationUsages that are nestedUsages of this Usage.

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

The AnalysisCaseUsages that are nestedUsages of this Usage.

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

The AttributeUsages that are nestedUsages of this Usage.

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

The CalculationUsages that are nestedUsages of this Usage.

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

The CaseUsages that are nestedUsages of this Usage.

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

The ConcernUsages that are nestedUsages of this Usage.

/nestedConnection : ConnectorAsUsage [0..*] {subsets nestedPart, ordered}

The ConnectorAsUsages that are nestedUsages of this Usage. Note that this list includes BindingConnectorAsUsages and SuccessionAsUsages, even though these are ConnectorAsUsages but not ConnectionUsages.

/nestedConstraint : ConstraintUsage [0..*] {subsets nestedOccurrence, ordered}

The ConstraintUsages that are nestedUsages of this Usage.

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

The EnumerationUsages that are nestedUsages of this Usage.

/nestedFlow : FlowConnectionUsage [0..*] {subsets nestedConnection}

The FlowConnectionUsages that are nestedUsages of this Usage.

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

The InterfaceUsages that are nestedUsages of this Usage.

/nestedItem : ItemUsage [0..*] {subsets nestedOccurrence, ordered}

The ItemUsages that are nestedUsages of this Usage.

/nestedOccurrence : OccurrenceUsage [0..*] {subsets nestedUsage, ordered}

The OccurrenceUsages that are nestedUsages of this Usage.

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

The PartUsages that are nestedUsages of this Usage.

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

The PortUsages that are nestedUsages of this Usage.

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

The ReferenceUsages that are nestedUsages of this Usage.

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

The RenderingUsages that are nestedUsages of this Usage.

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

The RequirementUsages that are nestedUsages of this Usage.

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

The StateUsages that are nestedUsages of this Usage.

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

The TransitionUsages that are nestedUsages of this Usage.

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

The Usages that are ownedFeatures of this Usage.

/nestedUseCase : UseCaseUsage [0..*] {subsets nestedCase, ordered}

The UseCaseUsages that are nestedUsages of this Usage.

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

The VerificationCaseUsages that are nestedUsages of this Usage.

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

The ViewUsages that are nestedUsages of this Usage.

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

The ViewpointUsages of this Usage that are nestedUsages.

/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, ordered}

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, 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 subsetteted Feature (unless that Feature is the owner of the usage).

```
body: if not owningMembership.oclIsKindOf(VariantMembership) then
      self.oclAsType(Feature).namingFeature()
else
  let namingFeature : Feature = firstSubsettetedFeature() in
  if namingFeature = owner then null
  else namingFeature
endif
```

Constraints

usageVariantMembership

[no documentation]

variantMembership = ownedMembership->selectByKind(VariantMembership)

usageIsReference

[no documentation]

isReference = not isComposite

usageIsVariationMembership

[no documentation]

isVariation implies variantMembership = ownedMembership

usageNonVariationMembership

[no documentation]

not isVariation implies variantMembership->isEmpty()

usageVariant

[no documentation]

```
variant = variantMembership.ownedVariantUsage
```

8.3.3.5 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.

8.3.4 Attributes Abstract Syntax

8.3.4.1 Overview

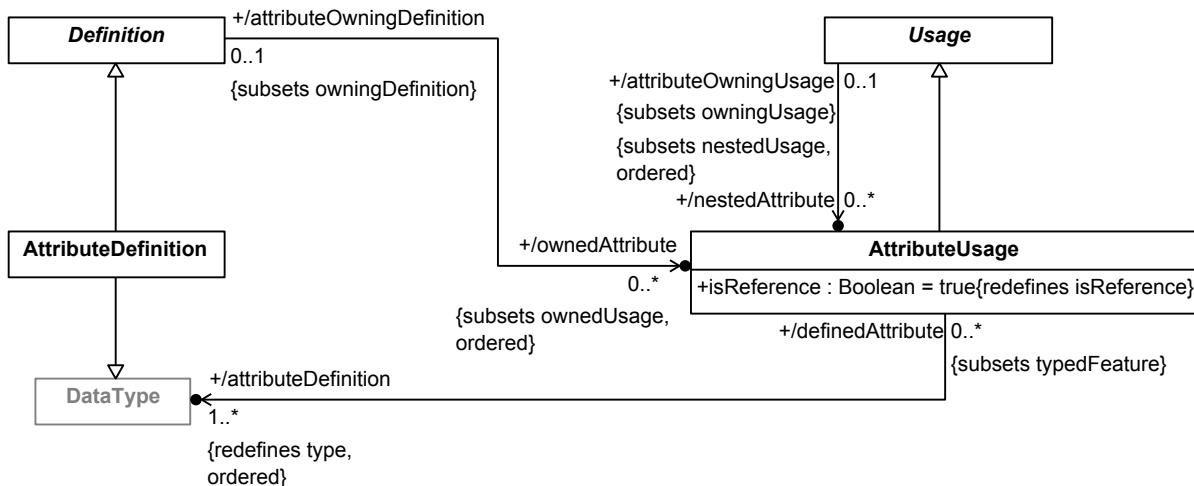


Figure 60. Attribute Definition and Usage

8.3.4.2 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, ordered}`

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.

`isReference : Boolean {redefines isReference}`

Always true for an AttributeUsage.

Operations

No operations.

Constraints

No constraints.

8.3.4.3 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.

8.3.5 Enumerations Abstract Syntax

8.3.5.1 Overview

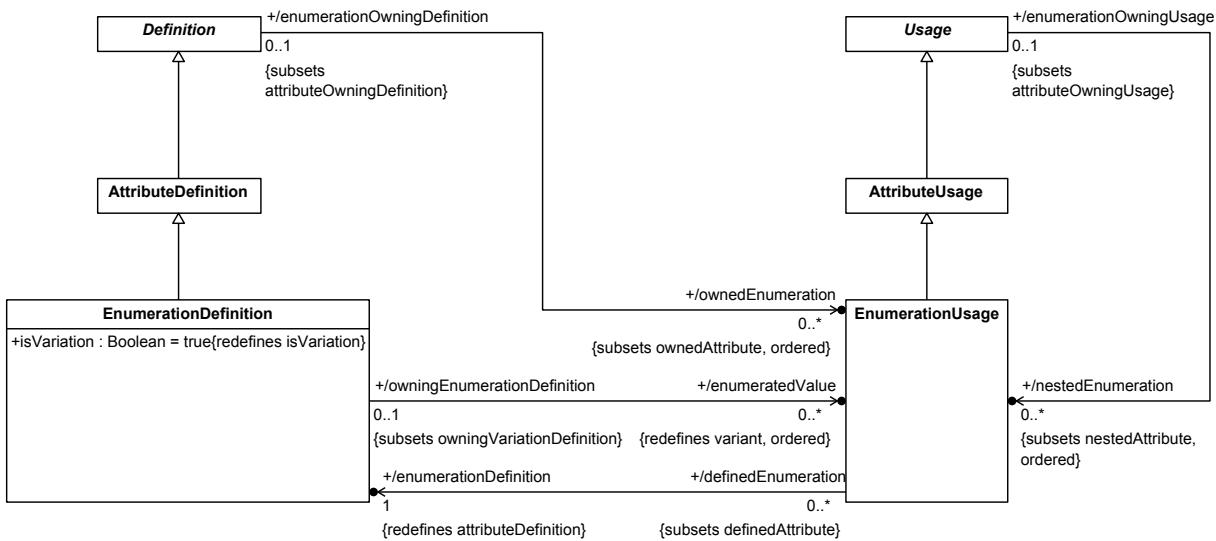


Figure 61. Enumeration Definition and Usage

8.3.5.2 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, ordered}

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.

8.3.5.3 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.

8.3.6 Occurrences Abstract Syntax

8.3.6.1 Overview

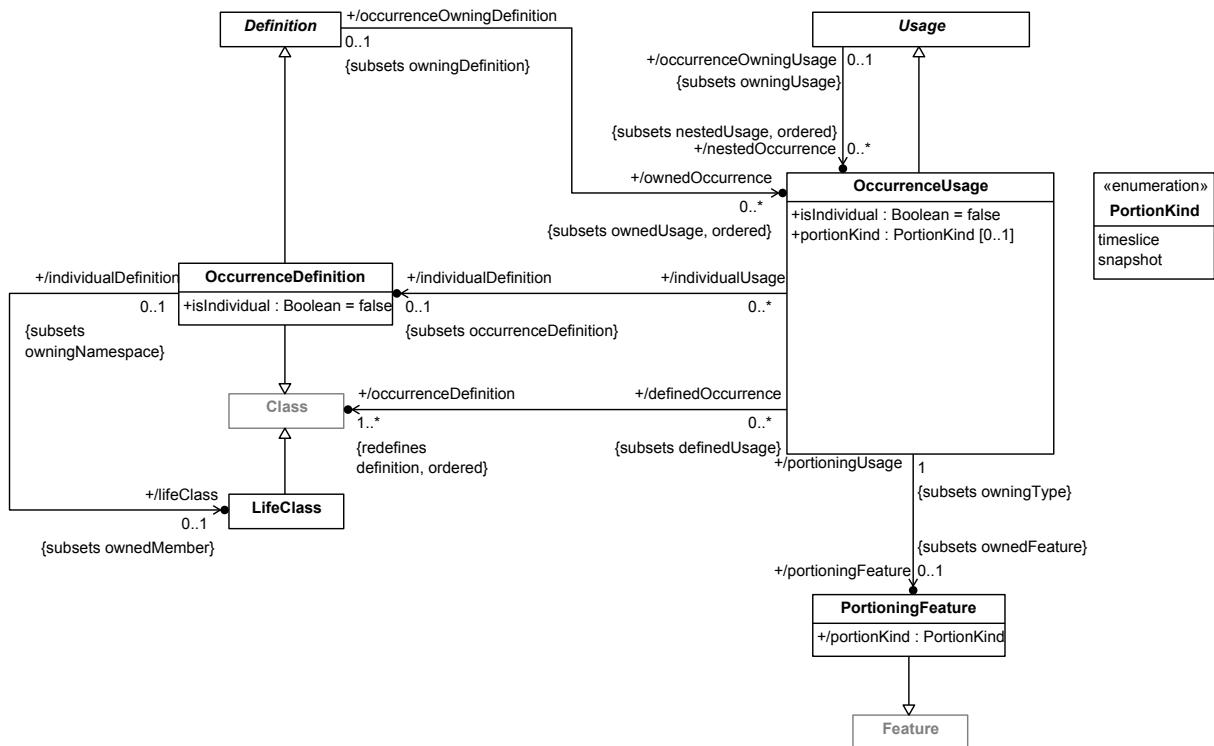


Figure 62. Occurrence Definition and Usage

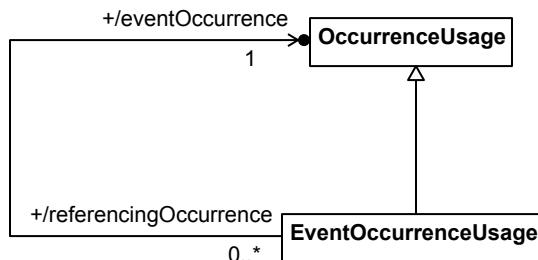


Figure 63. Event Occurrences

8.3.6.2 EventOccurrenceUsage

Description

A **EventOccurrenceUsage** is an **OccurrenceUsage** that represents another **OccurrenceUsage** occurring as a suboccurrence of the containing occurrence of the **EventOccurrenceUsage**. The referenced **OccurrenceUsage** performed (which may be the **EventOccurrenceUsage** itself) is related to the **EventOccurrenceUsage** by a **Subsetting** relationship.

If the **EventOccurrenceUsage** is owned by an **Occurrence**, then it also subsets the **suboccurrence** property of that **Occurrence** (as defined in the library model for Part).

General Classes

OccurrenceUsage

Attributes

/eventOccurrence : OccurrenceUsage

The OccurrenceUsage referenced as an event by this EventOccurrenceUsage. It is the `subsettetFeature` of the first owned Subsetting Relationship of the EventOccurrenceUsage.

Operations

No operations.

Constraints

No constraints.

8.3.6.3 LifeClass

Description

A LifeClass is a Class that specializes both the *Base::Life Class* from the Kernel Library and a single OccurrenceDefinition, and has a multiplicity of 0..1. This constrains the OccurrenceDefinition to have at most one instance that is a complete Life.

General Classes

Class

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.6.4 OccurrenceDefinition

Description

An OccurrenceDefinition is a Definition of a Class of individuals that have an independent life over time and potentially an extent over space. This includes both structural things and behaviors that act on such structures.

If `isIndividual` is true, then the OccurrenceDefinition is constrained to represent an individual thing. The instances of such an OccurrenceDefinition 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".

An OccurrenceDefinition must subclass, directly or indirectly, the base Class *Occurrence* from the Kernel model library.

General Classes

Class
Definition

Attributes

isIndividual : Boolean

Whether this OccurrenceDefinition is constrained to represent single individual.

/lifeClass : LifeClass [0..1] {subsets ownedMember}

If `isIndividual` is true, a LifeClass that specializes this OccurrenceDefinition, restricting it to represent an individual.

Operations

No operations.

Constraints

occurrenceDefinitionLifeClass

An OccurrenceDefinition has a `lifeClass` if and only if `isIndividual = true`, in which case the `lifeClass` must specialize the OccurrenceDefinition.

```
if not isIndividual then lifeClass = null
else
    lifeClass <> null and
    lifeClass.allSupertypes () ->includes (self)
endif
```

8.3.6.5 OccurrenceUsage

Description

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

An OccurrenceUsage must subset, directly or indirectly, the base Feature *occurrences* from the Kernel model library.

General Classes

Usage

Attributes

/individualDefinition : OccurrenceDefinition [0..1] {subsets occurrenceDefinition}

The one occurrenceDefinition that has isIndividual = true (if any).

isIndividual : Boolean

Whether this OccurrenceUsage represents the usage of the specific individual (or portion of it) represented by its individualDefinition.

/occurrenceDefinition : Class [1..*] {redefines definition, ordered}

The Classes that are the types of this OccurrenceUsage. Nominally, these are OccurrenceDefinitions, but other kinds of Kernel Classes are also allowed, to permit use of Classes from the Kernel Library.

/portioningFeature : PortioningFeature [0..1] {subsets ownedFeature}

A PortioningFeature typed by the occurrenceDefinitions of this OccurrenceUsage, thereby restricting the values of the OccurrenceUsage to be general portions, time slices or snapshots of instances of those definitions, consistence with the portionKind.

portionKind : PortionKind [0..1]

The kind of portion of the instances of the occurrenceDefinition represented by this OccurrenceUsage, if it is so restricted.

Operations

No operations.

Constraints

occurrenceUsageIndividualUsage

If an OccurrenceUsage has isIndividual = true, then it must have a single individualDefinition.

isIndividual implies individualDefinition <> null

occurrenceUsageIndividualDefinition

The individualDefinition of an OccurrenceUsage is the occurrenceDefinition that is an OccurrenceDefinition with isIndividual = true, if any.

```
let individualDefinitions : Sequence(OccurrenceDefinition) =
    occurrenceDefinition->
        selectByKind(OccurrenceDefinition)->
            select(isIndividual) in
    if individualDefinitions->isEmpty() then null
    else individualDefinitions->at(1) endif
```

occurrenceUsagePortioning

An OccurrenceUsage has a portioningFeature if and only if it has a portionKind and, if so, the portionKind of the portioningFeature must be the same as that of the OccurrenceUsage and the types of the portioningFeature must be the same as the occurrenceDefinitions of the OccurrenceUsage.

```

if portionKind = null then portioningFeature = null
else
    portioningFeature <> null and
    portionKind = portioningFeature.portionKind and
    occurrenceDefinition.asSet() = portioningFeature.type.asSet()
endif

```

8.3.6.6 PortioningFeature

Description

A PortioningFeature is a Feature that is a redefinition of one of the Features *timeSliceOf* or *snapshotOf* of the *portionOfLife* of each of the types of its *portioningUsage*.

General Classes

Feature

Attributes

/portionKind : PortionKind

Whether this PortionFeature is a redefinition of *timeSliceOf* or *snapshotOf*.

Operations

No operations.

Constraints

portioningFeaturePortionKind

The *portionKind* of a PortioningFeature is *timeslice* or *snapshot*, if the PortioningFeature is a direct Redefinition of *timeSliceOf* or *snapshotOf*, respectively.

8.3.6.7 PortionKind

Description

PortionKind is an enumeration of the possible special kinds of Occurrence portions that can be represented by an OccurrenceUsage.

General Classes

No general classes.

Literal Values

snapshot

A snapshot of an Occurrence (a time slice with zero duration).

timeslice

A time slice of an Occurrence (a portion over time).

8.3.7 Items Abstract Syntax

8.3.7.1 Overview

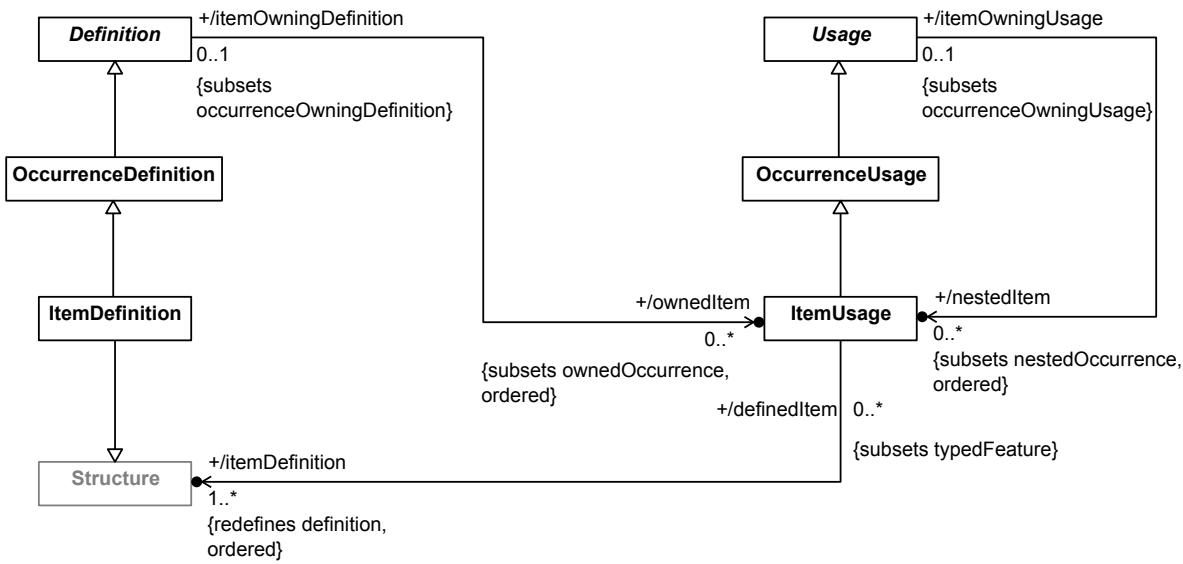


Figure 64. Item Definition and Usage

8.3.7.2 ItemDefinition

Description

An ItemDefinition is an OccurrenceDefinition 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

OccurrenceDefinition
Structure

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.7.3 ItemUsage

Description

An ItemUsage is a Usage whose type is a Structure. 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

OccurrenceUsage

Attributes

/itemDefinition : Structure [1..*] {redefines definition, ordered}

The Structures that are the *definitions* 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.

8.3.8 Parts Abstract Syntax

8.3.8.1 Overview

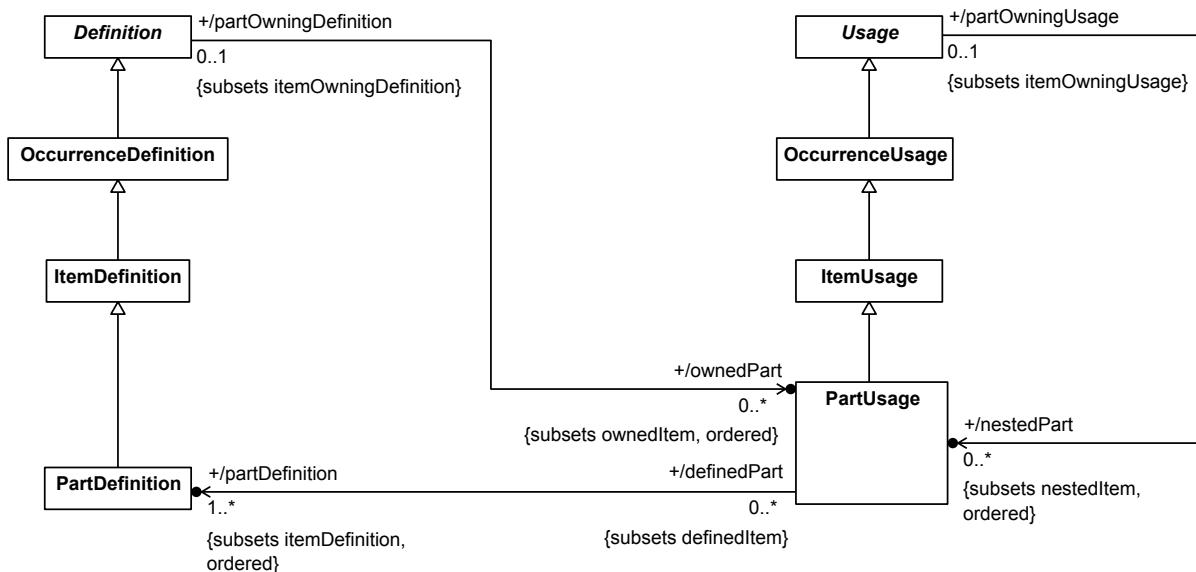


Figure 65. Part Definition and Usage

8.3.8.2 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.

8.3.8.3 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, ordered}

The itemDefinitions of this PartUsage that are PartDefinitions.

Operations

No operations.

Constraints

No constraints.

8.3.9 Ports Abstract Syntax

8.3.9.1 Overview

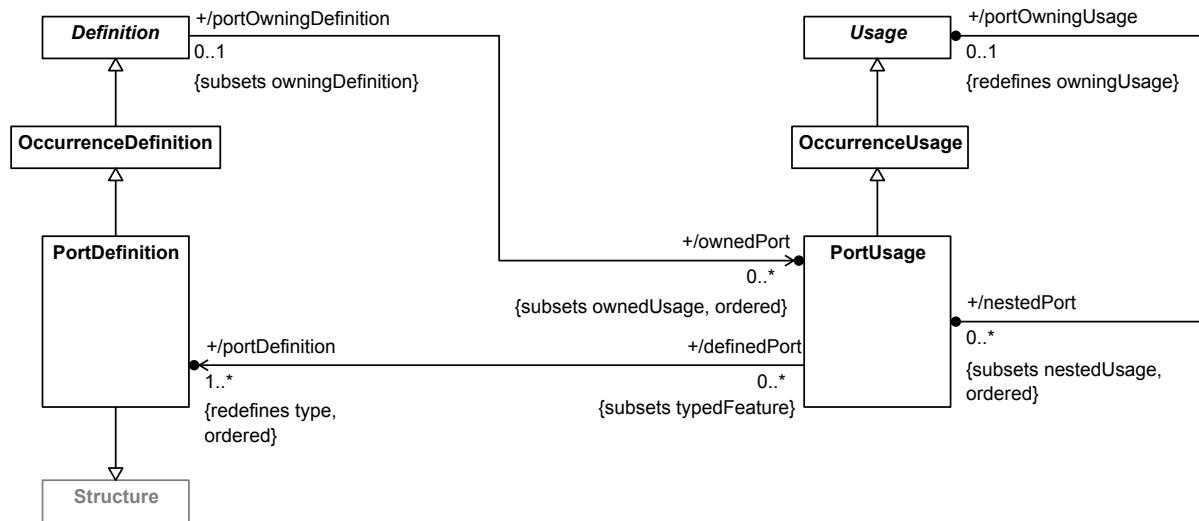


Figure 66. Port Definition and Usage

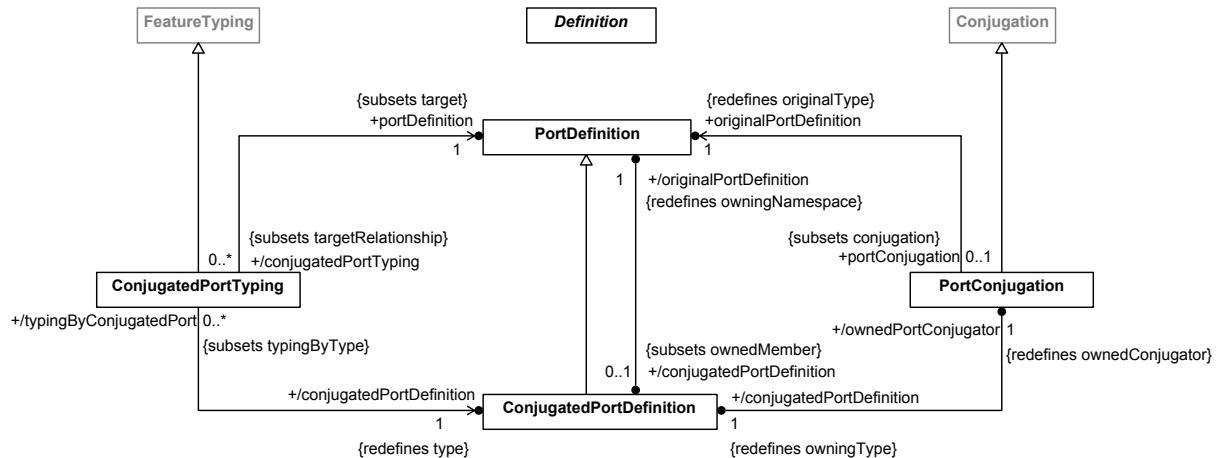


Figure 67. Port Conjugation

8.3.9.2 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 **originalPortDefinition**, 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 to its originalPortDefinition.

Operations

No operations.

Constraints

conjugatedPortDefinitionConjugatedPortDefinitionIsEmpty

[no documentation]

conjugatedPortDefinition = null

conjugatedPortDefinitionOriginalPortDefinition

[no documentation]

originalPortDefinition = ownedPortConjugator.originalPortDefinition

8.3.9.3 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
```

8.3.9.4 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.

8.3.9.5 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

OccurrenceDefinition

Structure

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))
```

8.3.9.6 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

OccurrenceUsage

Attributes

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

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.

8.3.10 Connections Abstract Syntax

8.3.10.1 Overview

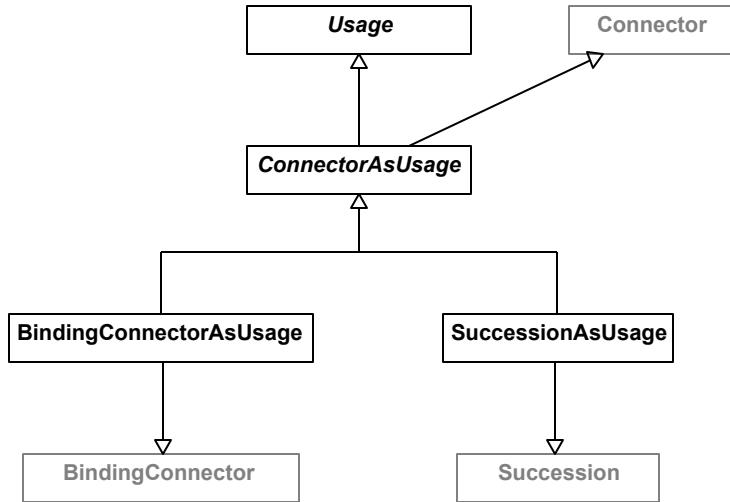


Figure 68. Connectors as Usages

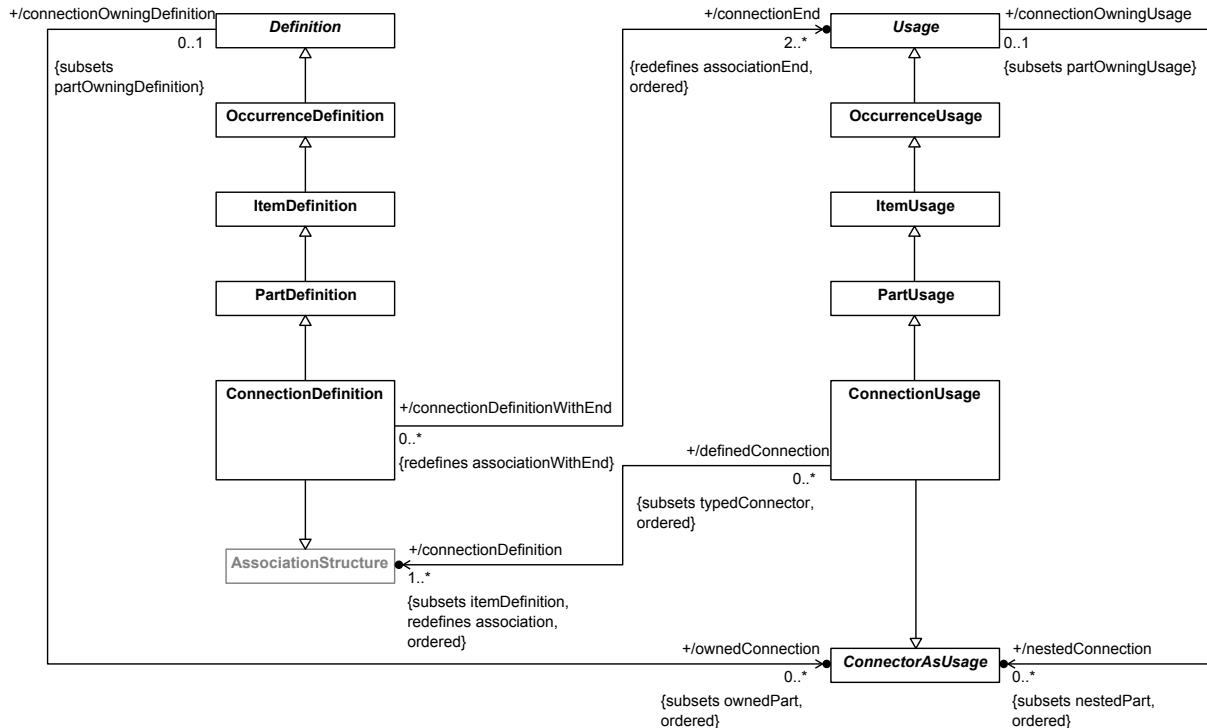


Figure 69. Connection Definition and Usage

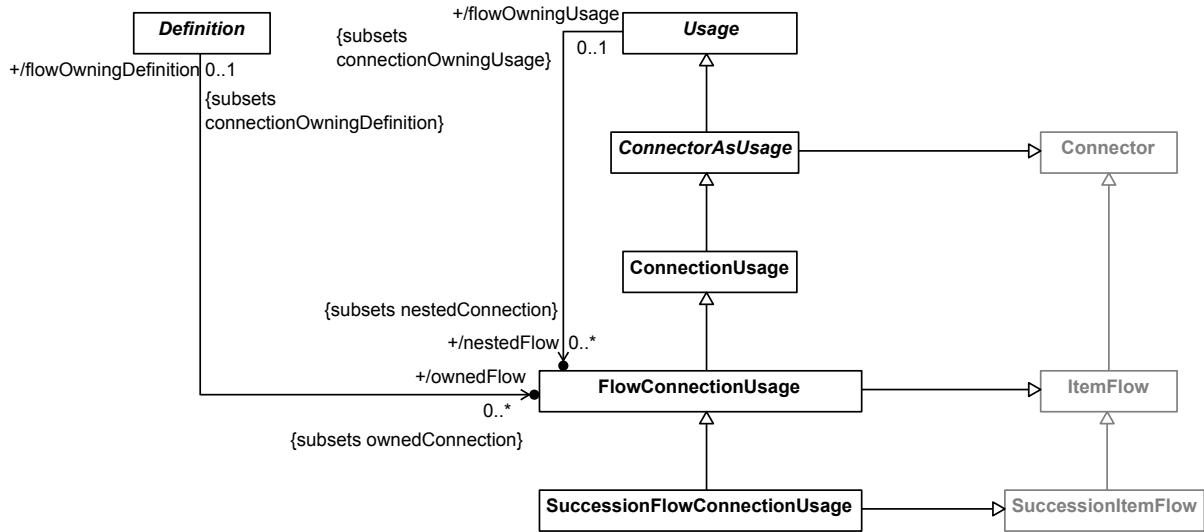


Figure 70. Flow Connections

8.3.10.2 BindingConnectorAsUsage

Description

A BindingConnectorAsUsage is both a BindingConnector and a ConnectorAsUsage.

General Classes

BindingConnector
ConnectorAsUsage

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.10.3 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, ordered}

The Usages that define the things related by the ConnectionDefinition.

Operations

No operations.

Constraints

No constraints.

8.3.10.4 ConnectionUsage

Description

A ConnectionUsage is a ConnectorAsUsage that is also a PartUsage. 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 AssociationStructures from the Kernel Library (such as the default BinaryLinkObject).

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

General Classes

ConnectorAsUsage
PartUsage

Attributes

/connectionDefinition : AssociationStructure [1..*] {subsets itemDefinition, redefines association, ordered}

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

Operations

No operations.

Constraints

No constraints.

8.3.10.5 ConnectorAsUsage

Description

A ConnectorAsUsage is both a Connector and a Usage. ConnectorAsUsage cannot itself be instantiated in a SysML model, but it is the base class for the concrete classes BindingConnectorAsUsage, SuccessionAsUsage and ConnectionUsage.

General Classes

Usage
Connector

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.10.6 FlowConnectionUsage

Description

A FlowConnectionUsage is a ConnectionUsage that is also an ItemFlow.

General Classes

ConnectionUsage
ItemFlow

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.10.7 SuccessionAsUsage

Description

A SuccessionAsUsage is both a ConnectorAsUsage and a Succession.

General Classes

Succession
ConnectorAsUsage

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.10.8 SuccessionFlowConnectionUsage

Description

A SuccessionFlowConnectionUsage is a FlowConnectionUsage that is also a SuccessionItemFlow.

General Classes

SuccessionItemFlow
FlowConnectionUsage

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.11 Interfaces Abstract Syntax

8.3.11.1 Overview

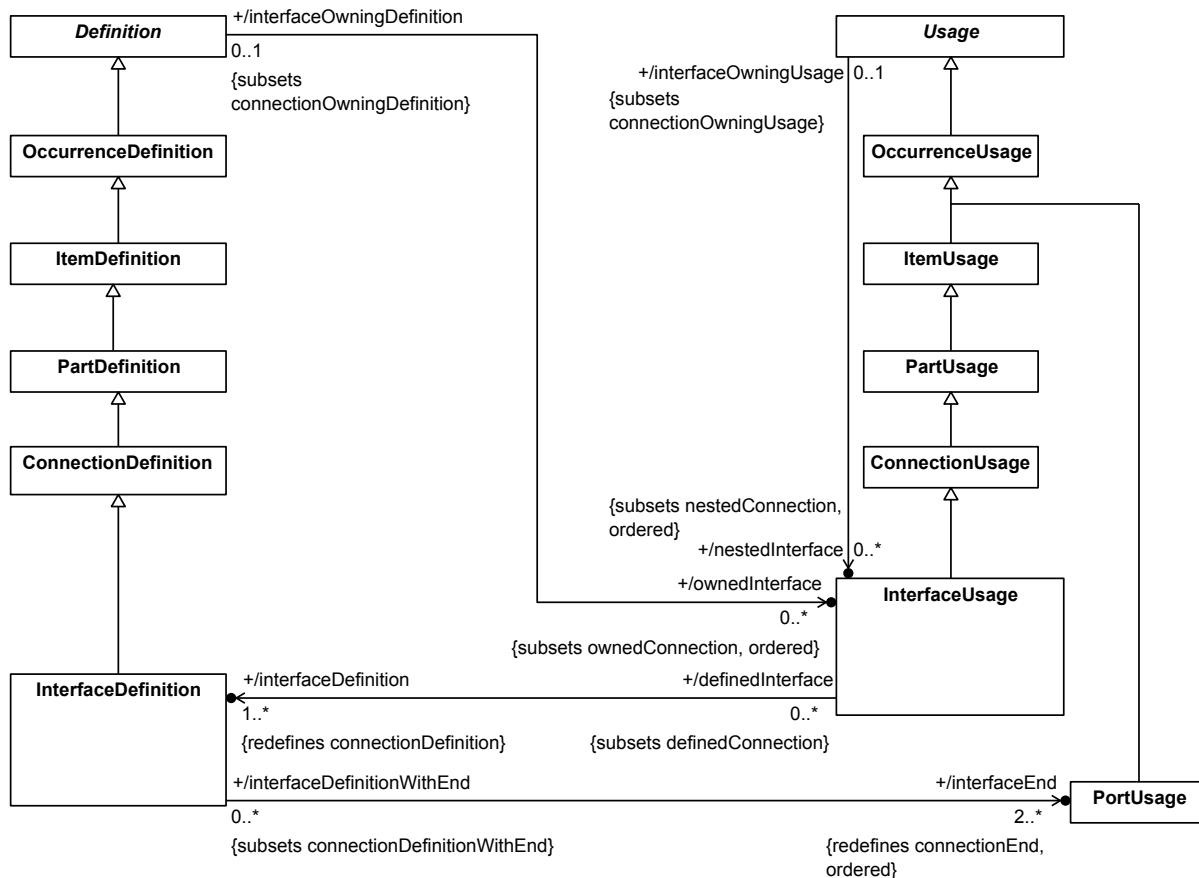


Figure 71. Interface Definition and Usage

8.3.11.2 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**, **ordered**}

The **PortUsages** that are the **associationEnds** of this **InterfaceDefinition**.

Operations

No operations.

Constraints

No constraints.

8.3.11.3 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 [1..*] {redefines connectionDefinition}
```

The InterfaceDefinitions that type this InterfaceUsage.

Operations

No operations.

Constraints

No constraints.

8.3.12 Allocations Abstract Syntax

8.3.12.1 Overview

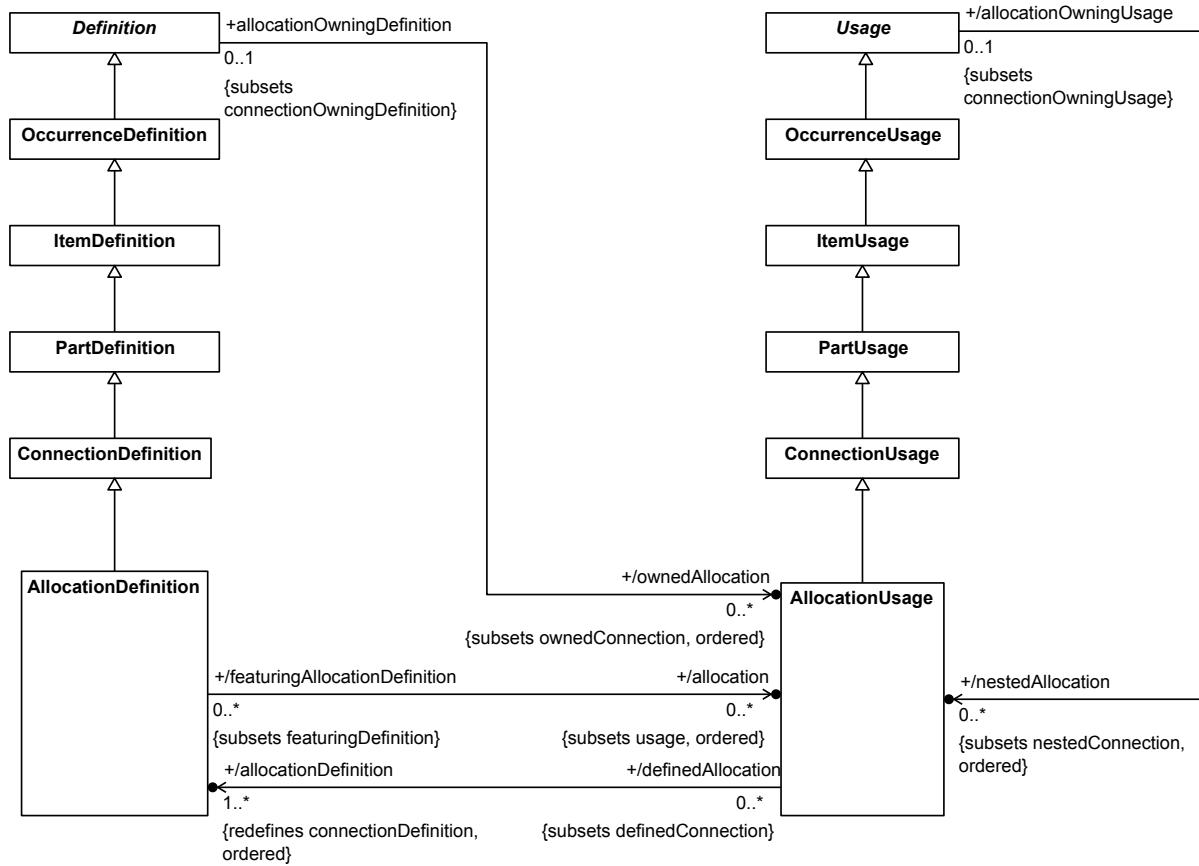


Figure 72. Allocation Definition and Usage

8.3.12.2 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, ordered}

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

Operations

No operations.

Constraints

No constraints.

8.3.12.3 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, ordered}

The AllocationDefinitions that are the types of this AllocationUsage.

Operations

No operations.

Constraints

No constraints.

8.3.13 Actions Abstract Syntax

8.3.13.1 Overview

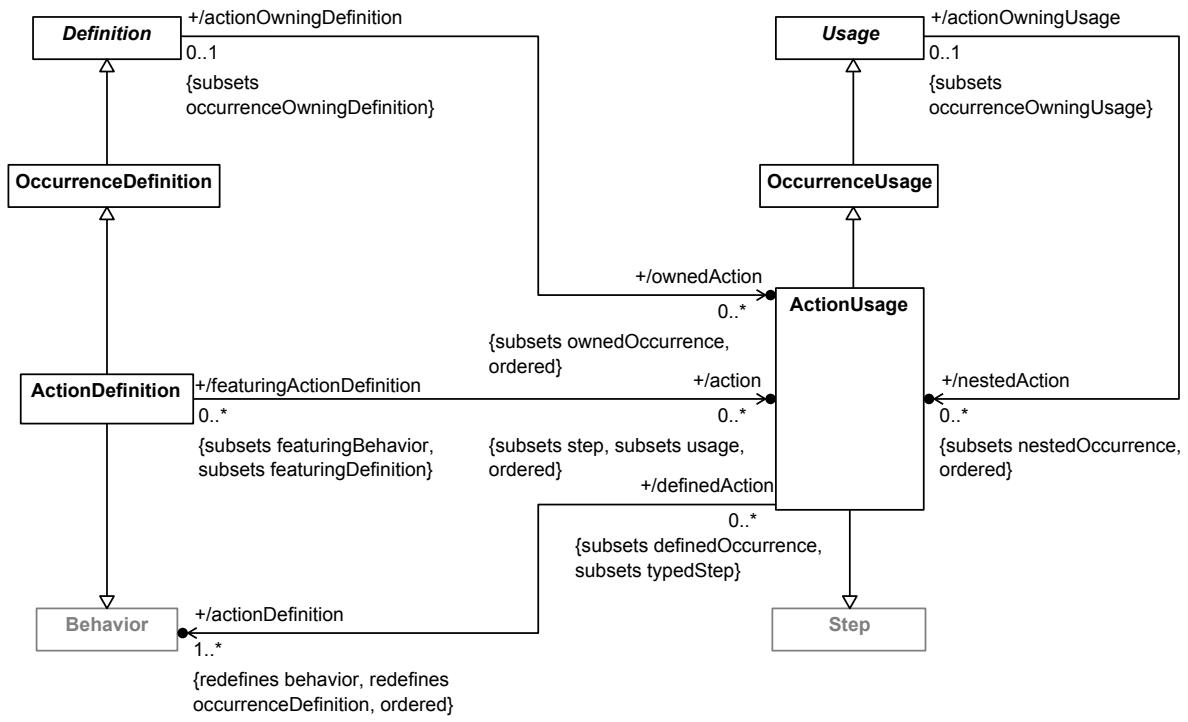


Figure 73. Action Definition and Usage

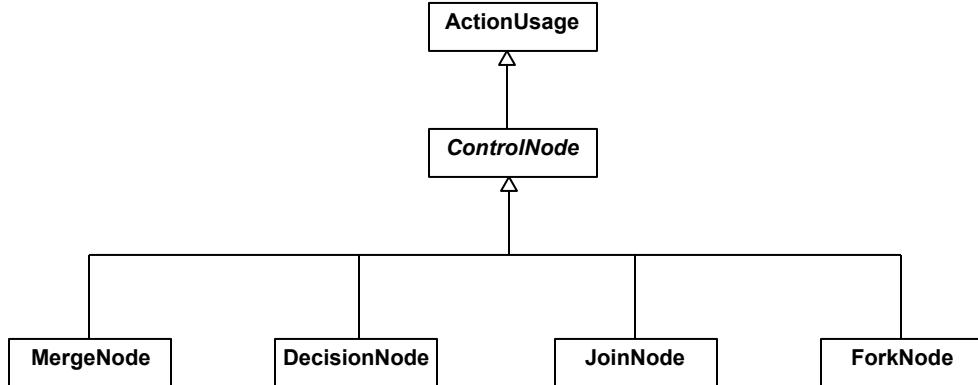


Figure 74. Control Nodes

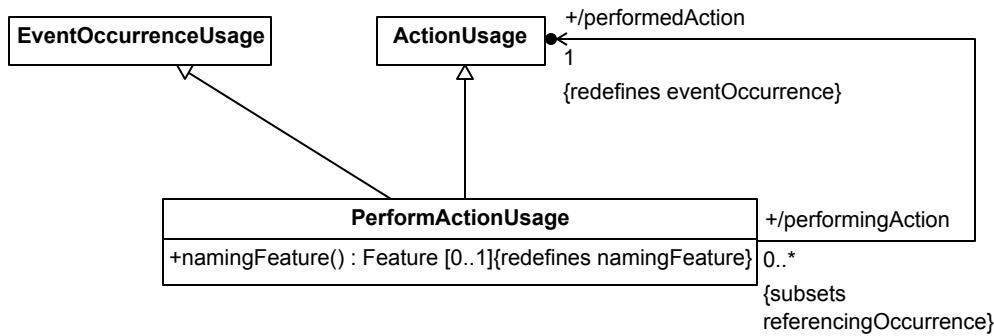


Figure 75. Action Performance

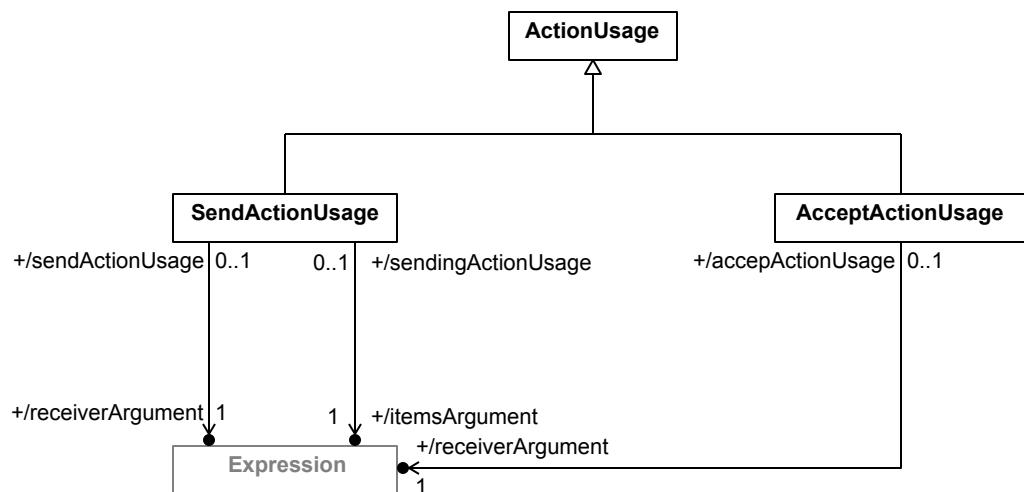


Figure 76. Send and Accept Actions

8.3.13.2 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

An **Expression** whose result is bound to the **receiver** input parameter of this **AcceptActionUsage**.

Constraints

No constraints.

8.3.13.3 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

OccurrenceDefinition
Behavior

Attributes

/action : ActionUsage [0..*] {subsets step, usage, ordered}

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.

8.3.13.4 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

OccurrenceUsage
Step

Attributes

/actionDefinition : Behavior [1..*] {redefines behavior, occurrenceDefinition, ordered}

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.

8.3.13.5 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.

8.3.13.6 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.

8.3.13.7 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.

8.3.13.8 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.

8.3.13.9 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.

8.3.13.10 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 PartDefinition or PartUsage, then it also subsets the ActionUsage `Part::performedAction` from the Systems model library.

General Classes

EventOccurrenceUsage
ActionUsage

Attributes

/performedAction : ActionUsage {redefines eventOccurrence}

The ActionUsage to be performed by this PerformedActionUsage. It is the `subsettetedFeature` of the first owned Subsetting Relationship of the PerformedActionUsage.

Constraints

No constraints.

8.3.13.11 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

An Expression whose result is bound to the `items` input parameter of this SendActionUsage.

/receiverArgument : Expression

An Expression whose result is bound to the `receiver` input parameter of this SendActionUsage.

Constraints

No constraints.

8.3.13.12 TransferActionUsage

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

8.3.14 States Abstract Syntax

8.3.14.1 Overview

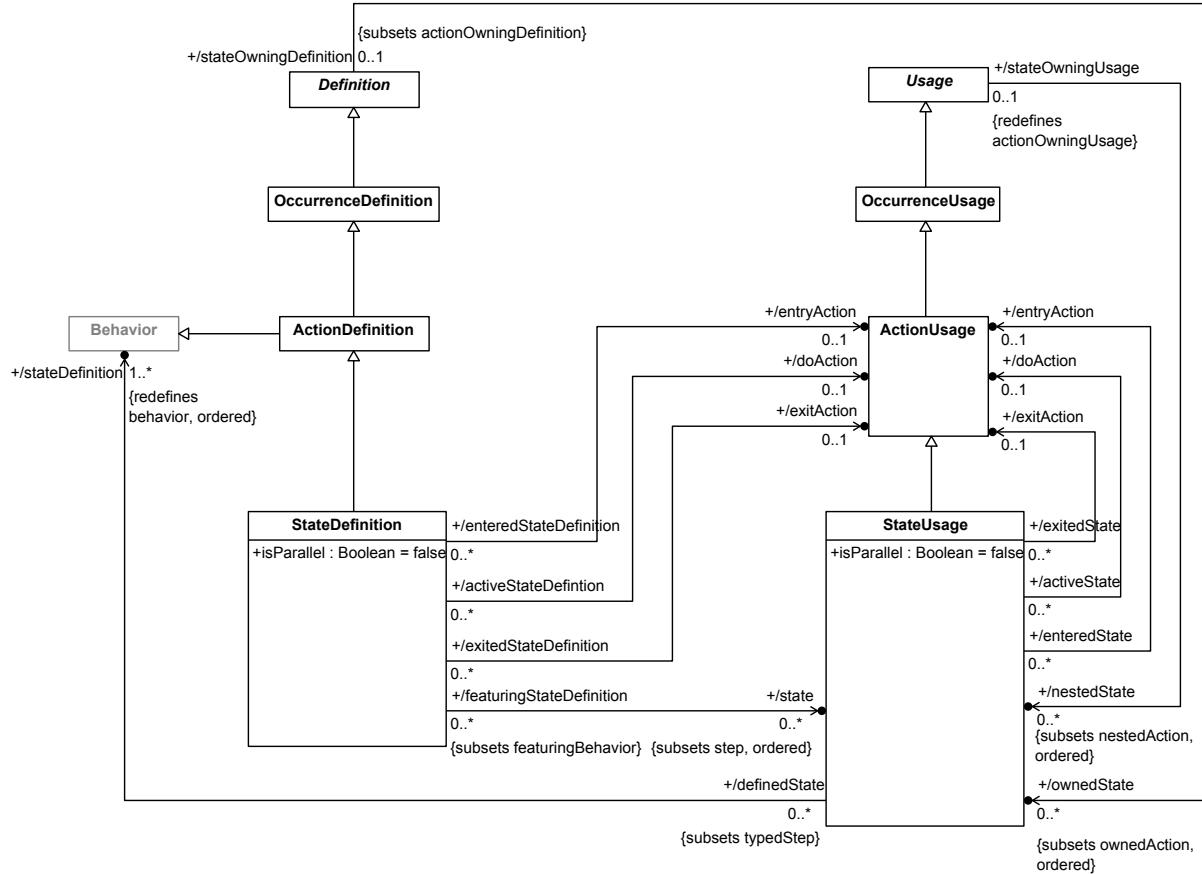


Figure 77. State Definition and Usage

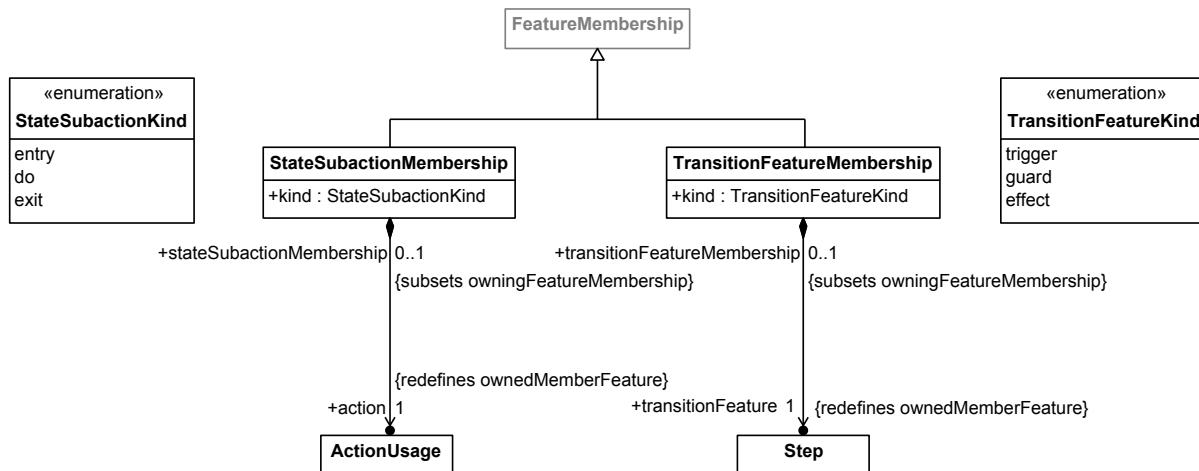


Figure 78. State Membership

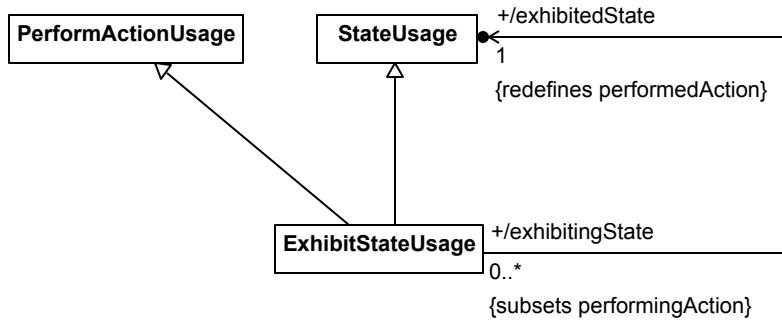


Figure 79. State Exhibition

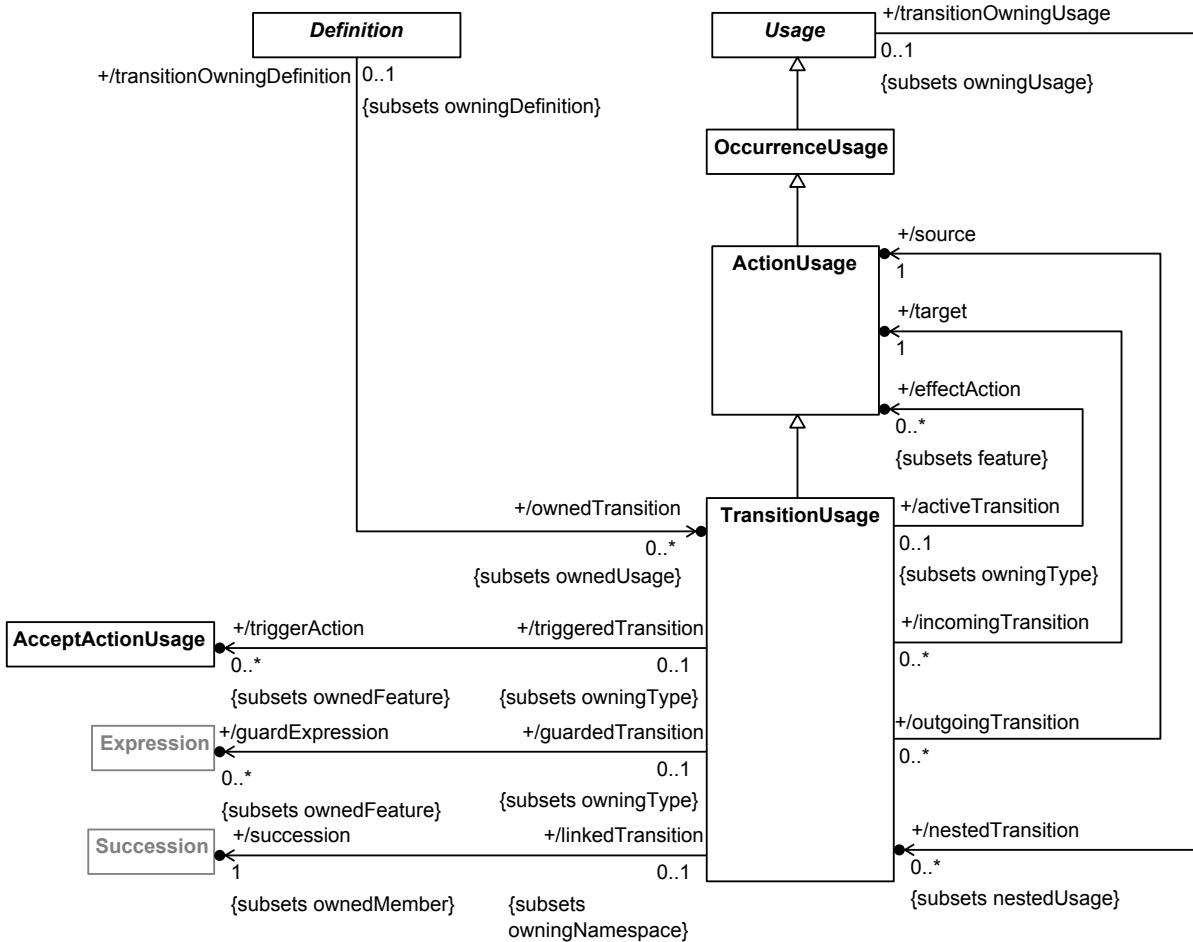


Figure 80. Transition Usage

8.3.14.2 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. An **ExhibitStateUsage** is also a **PerformActionUsage**, with its **exhibitedState** as the **performedAction**.

If the ExhibitStateUsage is owned by a PartDefinition or PartUsage, then it also subsets the StateUsage *Part::exhibitedStates* from the Systems model library.

General Classes

StateUsage
PerformActionUsage

Attributes

/exhibitedState : StateUsage {redefines performedAction}

The StateUsage to be exhibited by the ExhibitStateUsage. It is the *subsettetedFeature* of the first owned Subsetting Relationship of the ExhibitStateUsage.

Operations

No operations.

Constraints

No constraints.

8.3.14.3 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.

8.3.14.4 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.

8.3.14.5 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`.

isParallel : Boolean

Whether the ownedStates of this StateDefinition are to all be performed in parallel. If true, none of the ownedStates may have any incoming or outgoing transitions. If false, only one ownedState may be performed at a time.

/state : StateUsage [0..*] {subsets step, ordered}

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

stateDefinitionIsParallelGeneralization

Every generalization of a StateDefinition that is also a StateDefinition must have the same value for isParallel as this StateDefinition.

```
ownedGeneralization.general->
    selectByKind(StateDefinition).isParallel->
        forAll(p | p = isParallel)
```

8.3.14.6 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 PartDefinition or PartUsage) 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.

isParallel : Boolean

Whether the nestedStates of this StateDefinition are to all be performed in parallel. If true, none of the nestedStates may have any incoming or outgoing transitions. If false, only one nestedState may be performed at a time.

/stateDefinition : Behavior [1..*] {redefines behavior, ordered}

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

stateUsagesParallelGeneralization

Every generalization of a StateUsage that is also a StateDefinition or a StateUsage must have the same value for isParallel as this StateUsage.

```
let general : Sequence(Type) = ownedGeneralization.general in
general ->
  selectByKind(StateDefinition).isParallel->
    forAll(p | p = isParallel) and
general ->
  selectByKind(StateUsage).isParallel->
    forAll(p | p = isParallel)
```

8.3.14.7 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.

8.3.14.8 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.

8.3.14.9 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.

8.3.15 Calculations Abstract Syntax

8.3.15.1 Overview

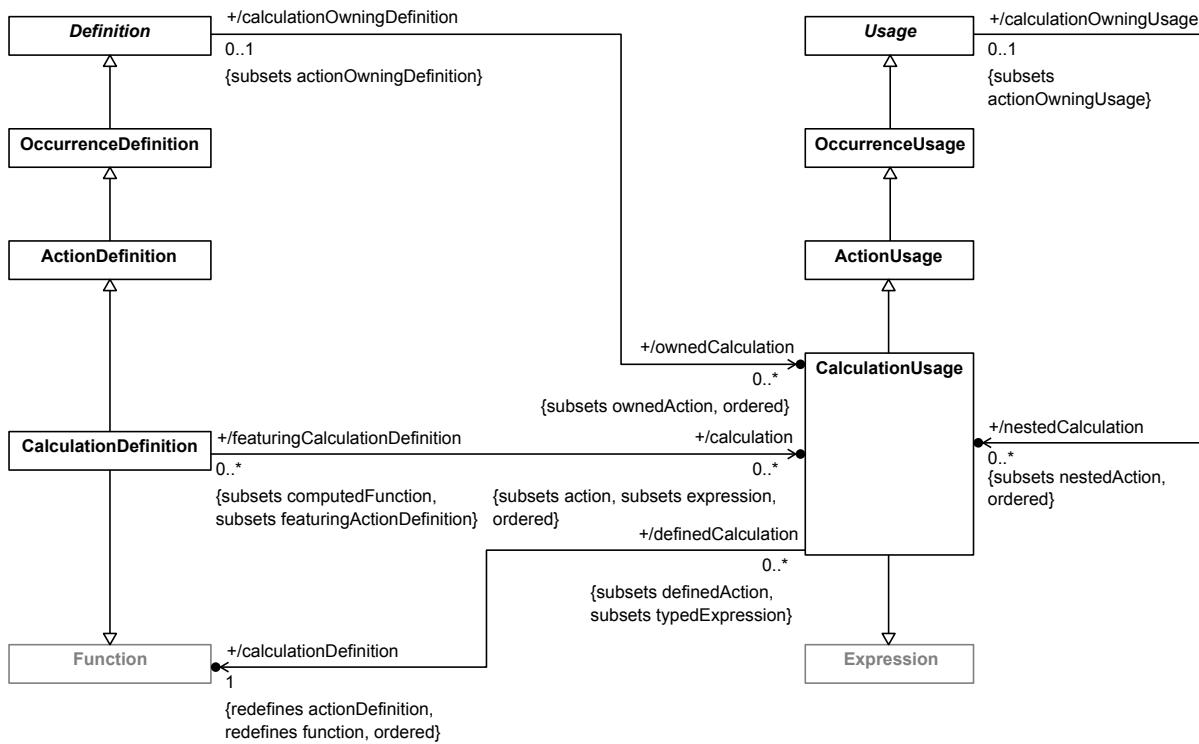


Figure 81. Calculation Definition and Usage

8.3.15.2 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, ordered}

The CalculationUsages that are actions in this CalculationDefinition.

Operations

No operations.

Constraints

No constraints.

8.3.15.3 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, ordered}

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.

8.3.16 Constraints Abstract Syntax

8.3.16.1 Overview

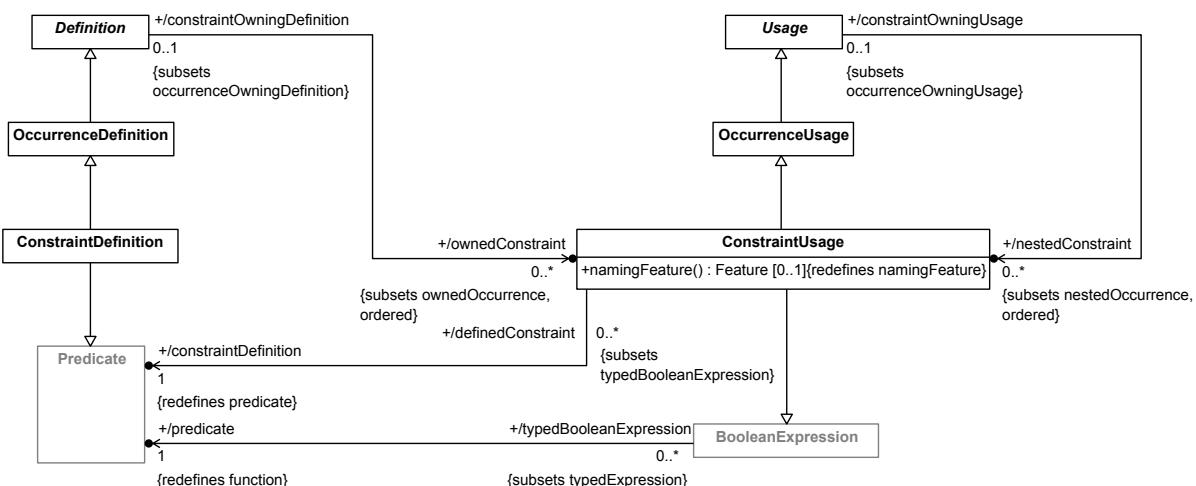


Figure 82. Constraint Definition and Usage

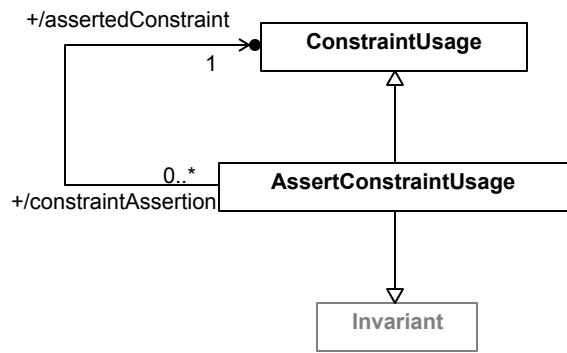


Figure 83. Constraint Assertion

8.3.16.2 AssertConstraintUsage

Description

An **AssertConstraintUsage** is a **ConstraintUsage** that is also an **Invariant** and, so, is asserted to be true (by default). 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 `subsettectedFeature` of the first owned Subsetting Relationship of the **AssertConstraintUsage**.

Operations

No operations.

Constraints

No constraints.

8.3.16.3 ConstraintDefinition

Description

A **ConstraintDefinition** is an **OccurrenceDefinition** 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

OccurrenceDefinition
Predicate

Attributes

No attributes.

Operations

No operations.

Constraints

No constraints.

8.3.16.4 ConstraintUsage

Description

A ConstraintUsage is a OccurrenceUsage 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

OccurrenceUsage
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.

8.3.17 Requirements Abstract Syntax

8.3.17.1 Overview

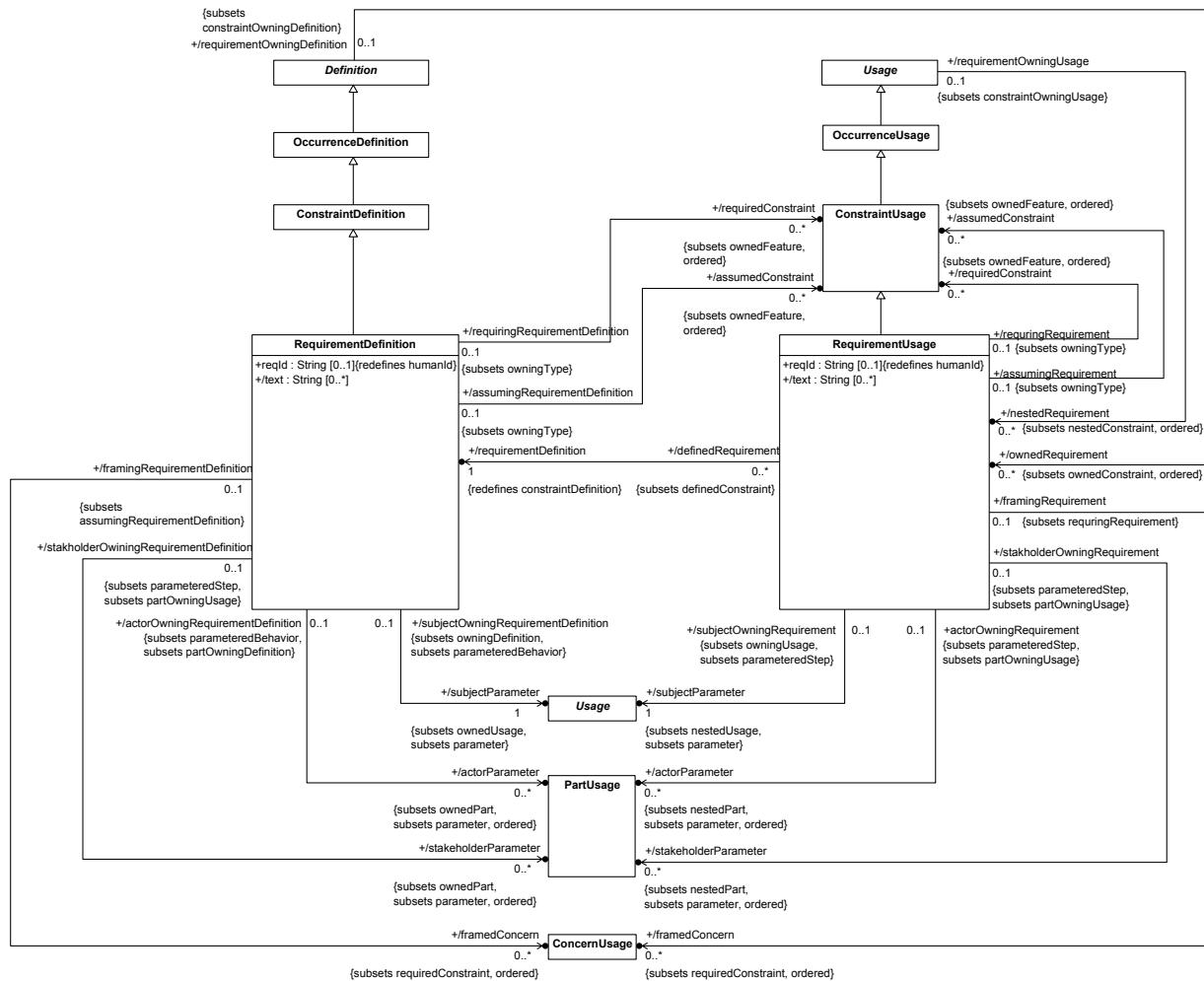


Figure 84. Requirement Definition and Usage

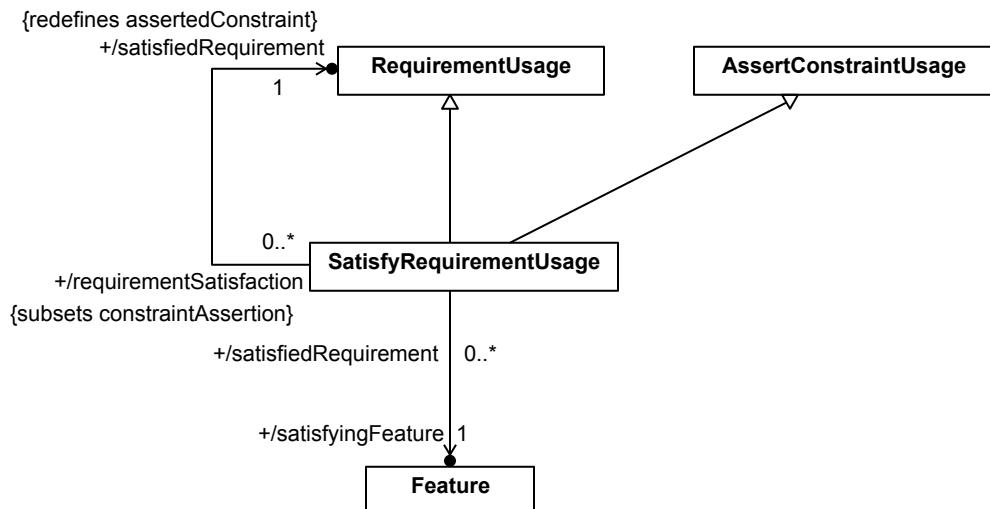


Figure 85. Requirement Satisfaction

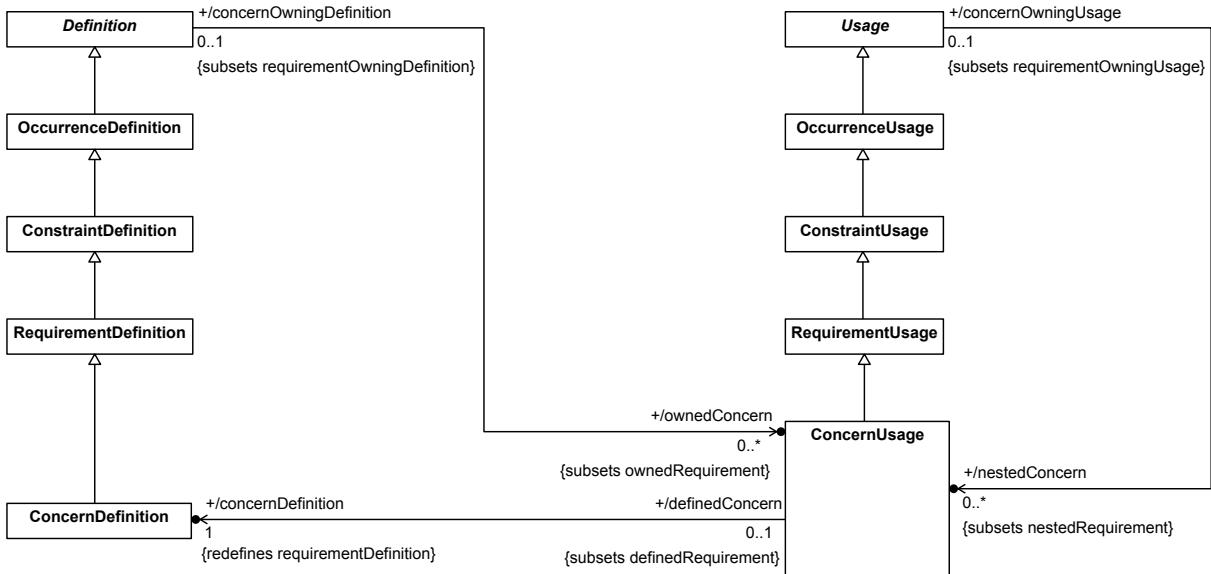


Figure 86. Concern Definition and Usage

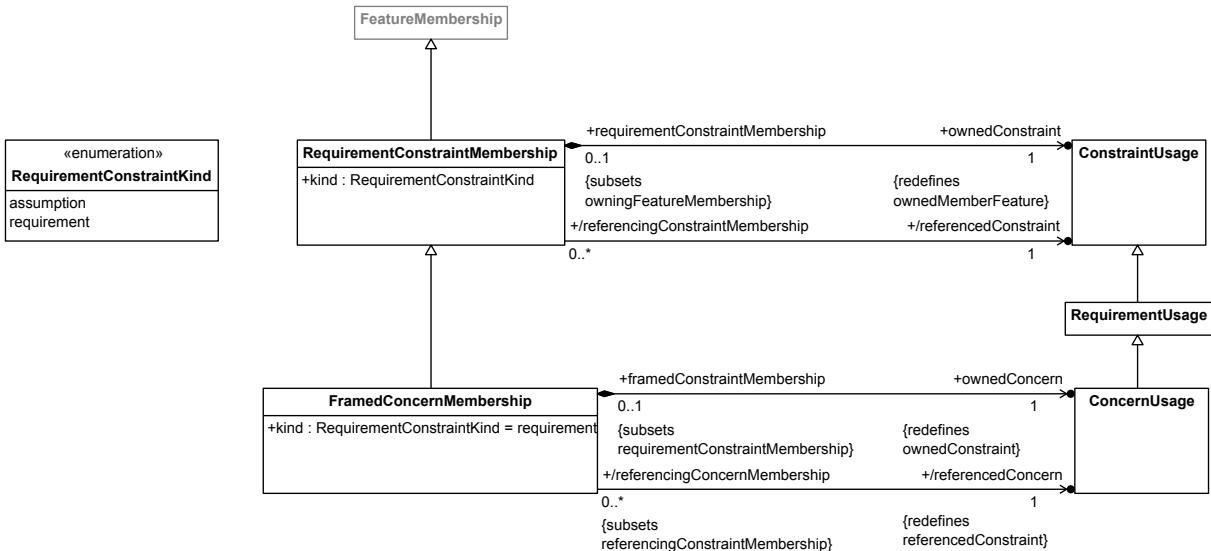


Figure 87. Requirement Constraint Membership

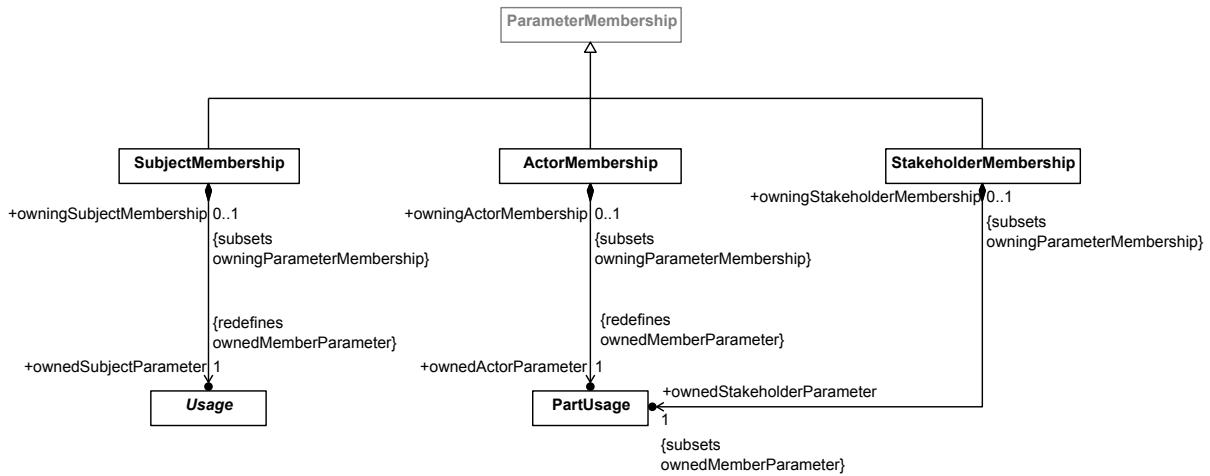


Figure 88. Requirement Parameter Memberships

8.3.17.2 ActorMembership

Description

An ActorMembership is a ParameterMembership that identifies a PartUsage as an actor parameter, which specifies a role played by an entity external in interaction with the parametered element.

General Classes

ParameterMembership

Attributes

ownedActorParameter : PartUsage {redefines ownedMemberParameter}

The PartUsage specifying the actor.

Operations

No operations.

Constraints

No constraints.

8.3.17.3 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.

8.3.17.4 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

Attributes

/concernDefinition : ConcernDefinition {redefines requirementDefinition}

The ConcernDefinition that is the single type of this ConcernUsage.

Operations

No operations.

Constraints

No constraints.

8.3.17.5 FramedConcernMembership

Description

A FramedConcernMembership is a RequirementConstraintMembership for a framed ConcernUsage of a RequirementDefinition or RequirementUsage. The *ownedConstraint* of a FramedConcernMembership 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.

8.3.17.6 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.

8.3.17.7 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.

8.3.17.8 RequirementDefinition

Description

A RequirementDefinition is a ConstraintDefinition that defines a requirement as a constraint that is used in the context of a specification of a that a valid solution must satisfy. The specification is relative to a specified subject, possibly in collaboration with one or more external actors.

A RequirementDefinition must subclass, directly or indirectly, the base RequirementDefinition *RequirementCheck* from the Systems model library.

General Classes

ConstraintDefinition

Attributes

/actorParameter : PartUsage [0..*] {subsets ownedPart, parameter, ordered}

The parameters of this RequirementDefinition that are owned via ActorMemberships, which must subset, directly or indirectly, the PartUsage *actors* of the base RequirementDefinition *RequirementCheck* from the Systems model library.

/assumedConstraint : ConstraintUsage [0..*] {subsets ownedFeature, ordered}

The owned ConstraintUsages that represent assumptions of this RequirementDefinition, derived as the `ownedConstraints` of the RequirementConstraintMemberships of the RequirementDefinition with `kind = assumption`.

/framedConcern : ConcernUsage [0..*] {subsets requiredConstraint, ordered}

The Concerns framed by this RequirementDefinition, derived as the ownedConcerns of all FramedConcernMemberships of the RequirementDefinition.

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, ordered}

The owned ConstraintUsages that represent requirements of this RequirementDefinition, derived as the ownedConstraints of the RequirementConstraintMemberships of the RequirementDefinition with kind = requirement.

/stakeholderParameter : PartUsage [0..*] {subsets ownedPart, parameter, ordered}

The parameters of this RequirementDefinition that are owned via StakeholderMemberships, which must subset, directly or indirectly, the PartUsage stakeholders of the base RequirementDefinition RequirementCheck from the Systems model library.

/subjectParameter : Usage {subsets parameter, ownedUsage}

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.

8.3.17.9 RequirementUsage

Description

A RequirementUsage is a Usage of a RequirementDefinition.

A RequirementUsage must subset, directly or indirectly, the base RequirementUsage requirementChecks from the Systems model library.

General Classes

ConstraintUsage

Attributes

/actorParameter : PartUsage [0..*] {subsets nestedPart, parameter, ordered}

The parameters of this RequirementUsage that are owned via ActorMemberships, which must subset, directly or indirectly, the PartUsage *actors* of the base RequirementDefinition *RequirementCheck* from the Systems model library.

/assumedConstraint : ConstraintUsage [0..*] {subsets ownedFeature, ordered}

The owned ConstraintUsages that represent assumptions of this RequirementUsage, derived as the ownedConstraints of the RequirementConstraintMemberships of the RequirementUsage with kind = assumption.

/framedConcern : ConcernUsage [0..*] {subsets requiredConstraint, ordered}

The Concerns framed by this RequirementUsage, derived as the ownedConcerns of all FramedConcernMemberships of the RequirementUsage.

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, ordered}

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.

/stakeholderParameter : PartUsage [0..*] {subsets nestedPart, parameter, ordered}

The parameters of this RequirementUsage that are owned via StakeholderMemberships, which must subset, directly or indirectly, the PartUsage *stakeholders* of the base RequirementDefinition *RequirementCheck* from the Systems model library.

/subjectParameter : Usage {subsets parameter, nestedUsage}

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.

8.3.17.10 SatisfyRequirementUsage

Description

A SatisfyRequirementUsage is an AssertConstraintUsage that asserts, by default, that a satisfied RequirementUsage is true for a specific satisfyingSubject, or, if isNegated = true, that the RequirementUsage is false. 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 subsettectedFeature 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.

8.3.17.11 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.

8.3.17.12 StakeholderMembership

Description

A `StakeholderMembership` is a `ParameterMembership` that identifies a `PartUsage` as a stakeholder parameter, which specifies a role played by an entity with Concerns framed by the parametered requirement.

General Classes

`ParameterMembership`

Attributes

`ownedStakeholderParameter` : `PartUsage` {subsets `ownedMemberParameter`}

The `PartUsage` specifying the stakeholder.

Operations

No operations.

Constraints

No constraints.

8.3.18 Cases Abstract Syntax

8.3.18.1 Overview

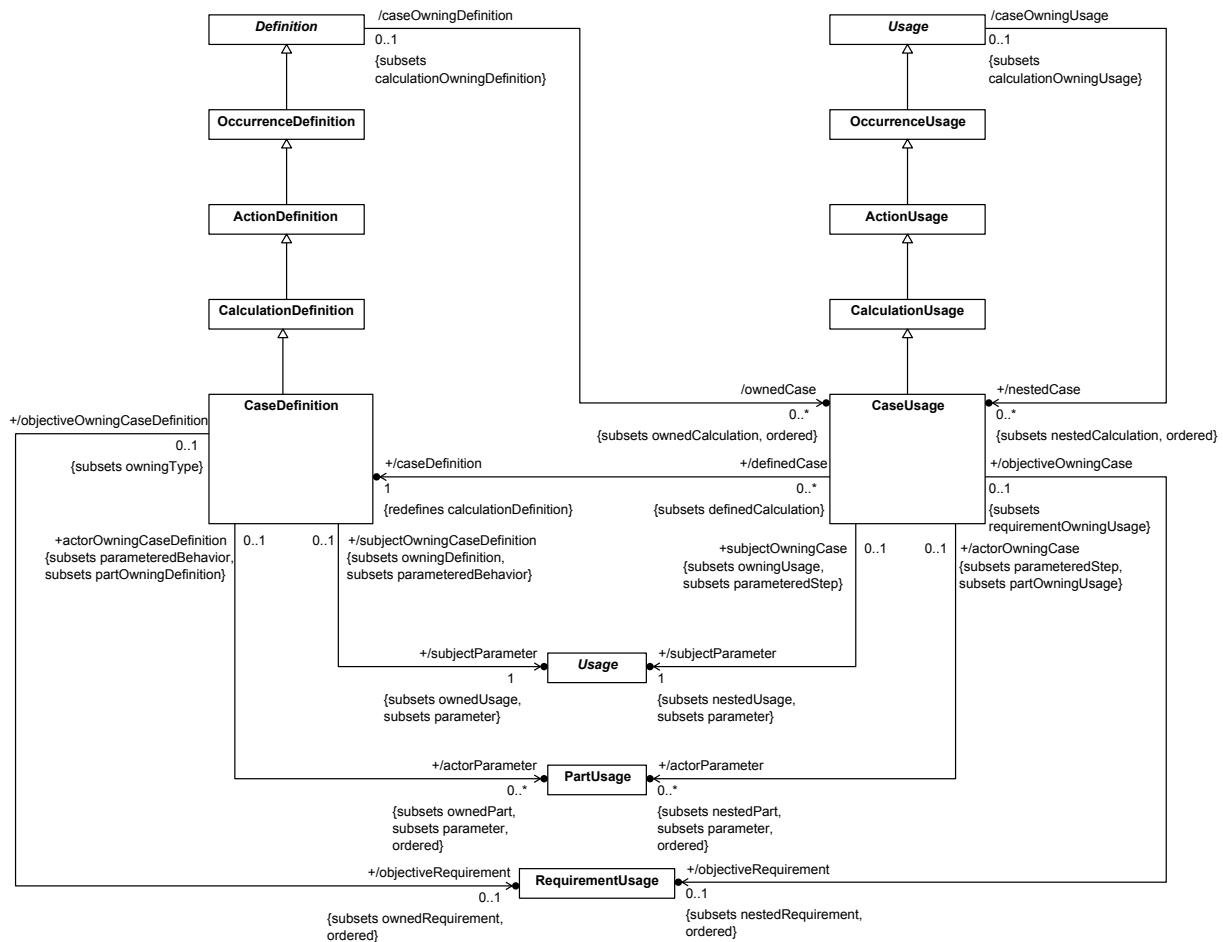


Figure 89. Case Definition and Usage

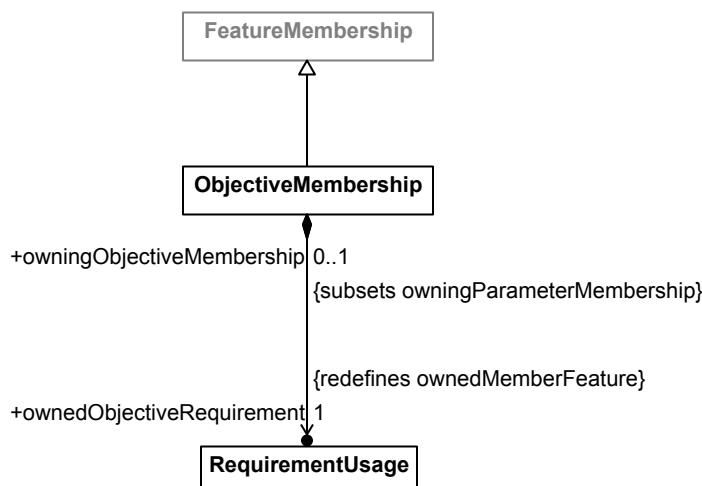


Figure 90. Case Membership

8.3.18.2 CaseDefinition

Description

A CaseDefinition is a CalculationDefinition for a process, often involving collecting evidence or data, relative to a subject, possibly involving the collaboration of one or more other actors, 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

/actorParameter : PartUsage [0..*] {subsets parameter, ownedPart, ordered}

The parameters of this CaseDefinition that are owned via ActorMemberships, which must subset, directly or indirectly, the PartUsage *actors* of the base CaseDefinition *Case* from the Systems model library.

/objectiveRequirement : RequirementUsage [0..1] {subsets ownedRequirement, ordered}

The ownedFeature of this CaseDefinition 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, ownedUsage}

The parameter of this CaseDefinition 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.

8.3.18.3 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 owned by a CaseDefinition or CaseUsage, it must subset the CaseUsage *Cases::subcases*.

General Classes

CalculationUsage

Attributes

/actorParameter : PartUsage [0..*] {subsets nestedPart, parameter, ordered}

The parameters of this CaseUsage that are owned via ActorMemberships, which must subset, directly or indirectly, the PartUsage *actors* of the base CaseDefinition *Case* from the Systems model library.

/caseDefinition : CaseDefinition {redefines calculationDefinition}

The CaseDefinition that is the type of this CaseUsage.

/objectiveRequirement : RequirementUsage [0..1] {subsets nestedRequirement, ordered}

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, nestedUsage}

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.

8.3.18.4 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.

8.3.19 Analysis Cases Abstract Syntax

8.3.19.1 Overview

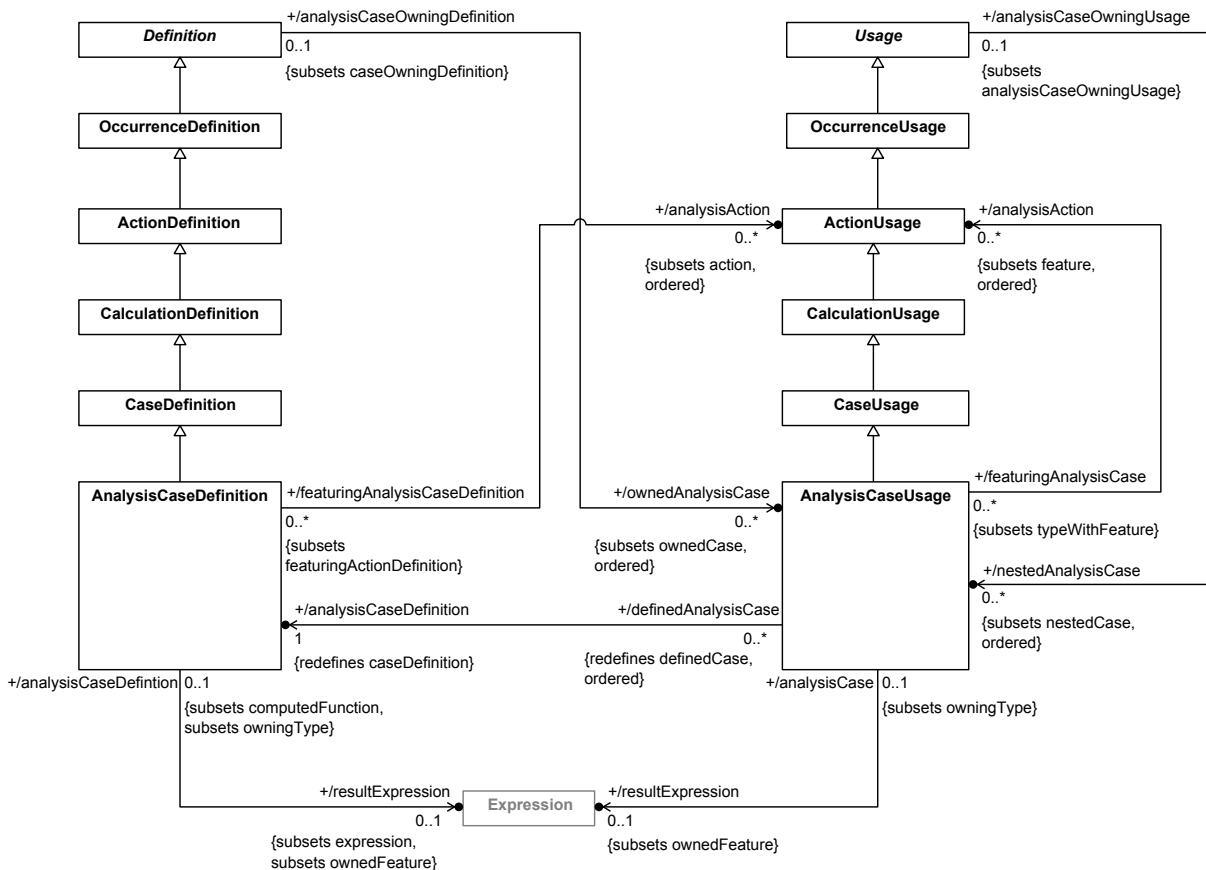


Figure 91. Analysis Case Definition and Usage

8.3.19.2 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, ordered}

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.

8.3.19.3 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, ordered}

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 owned 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.

8.3.20 Verification Cases Abstract Syntax

8.3.20.1 Overview

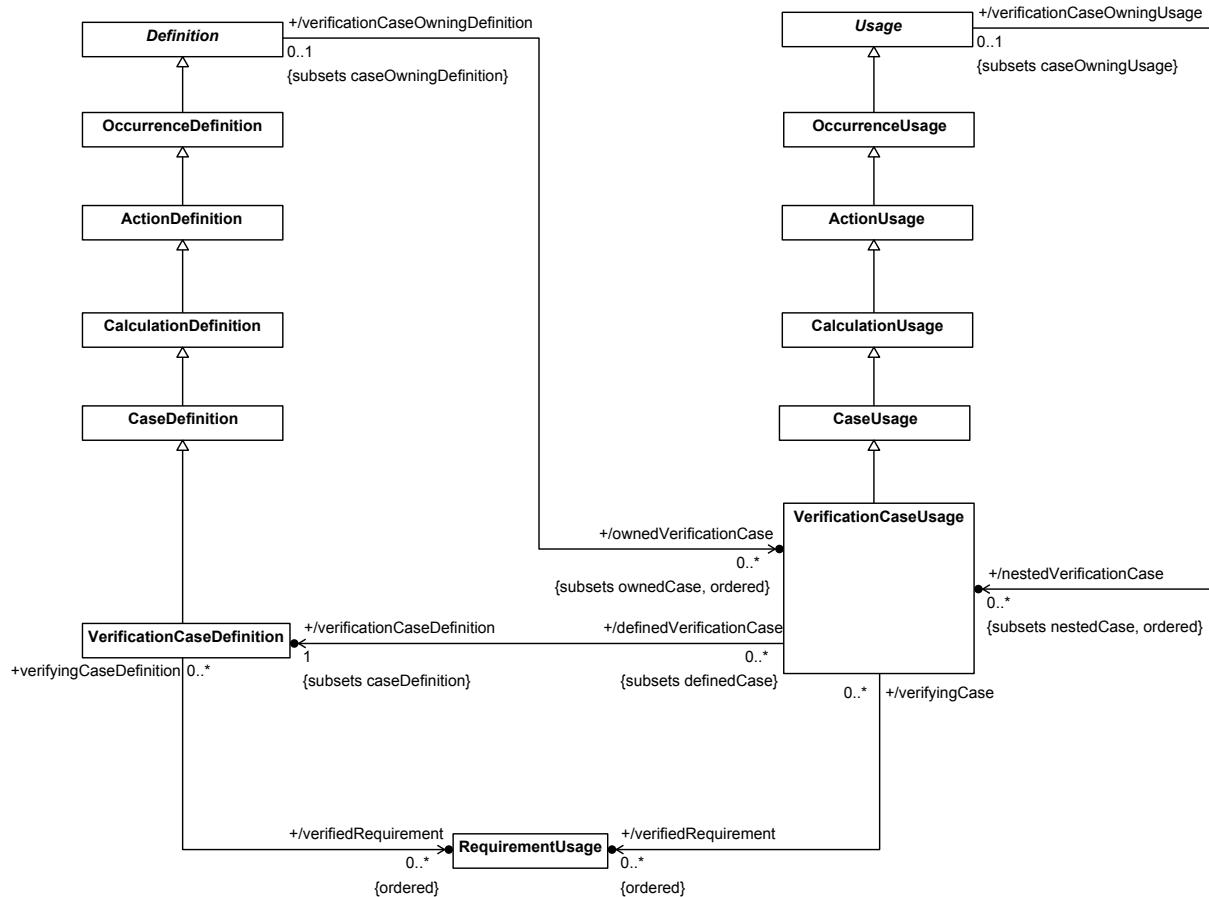


Figure 92. Verification Case Definition and Usage

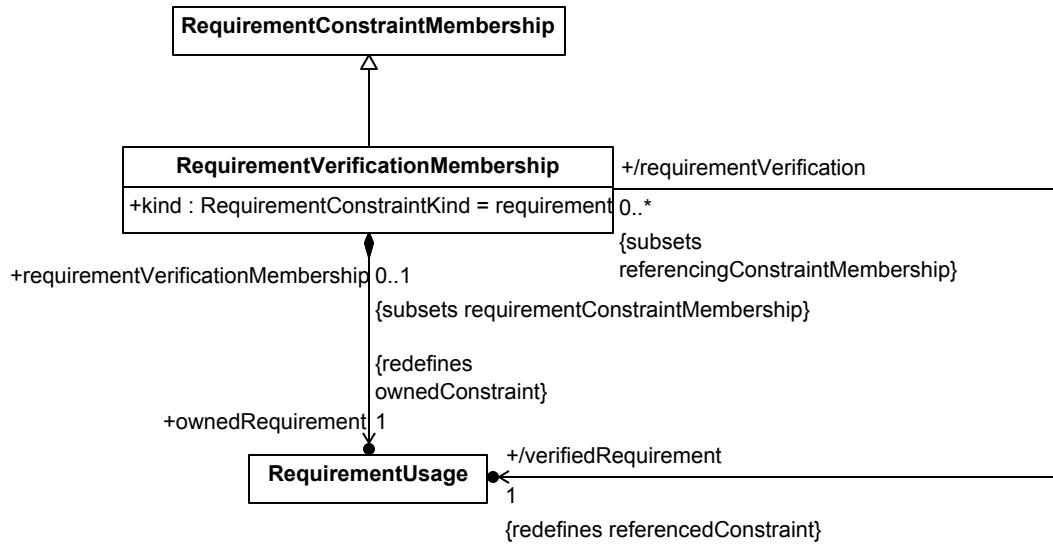


Figure 93. Verification Membership

8.3.20.2 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.

8.3.20.3 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..*] {ordered}

The RequirementUsages verified by this VerificationCaseDefinition, derived as the `verifiedRequirements` of all RequirementVerificationMemberships of the `objectiveRequirement`.

Operations

No operations.

Constraints

No constraints.

8.3.20.4 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..*] {ordered}

The RequirementUsages verified by this VerificationCaseUsage, derived as the `verifiedRequirements` of all RequirementVerificationMemberships of the `objectiveRequirement`.

Operations

No operations.

Constraints

No constraints.

8.3.21 Use Cases Abstract Syntax

8.3.21.1 Overview

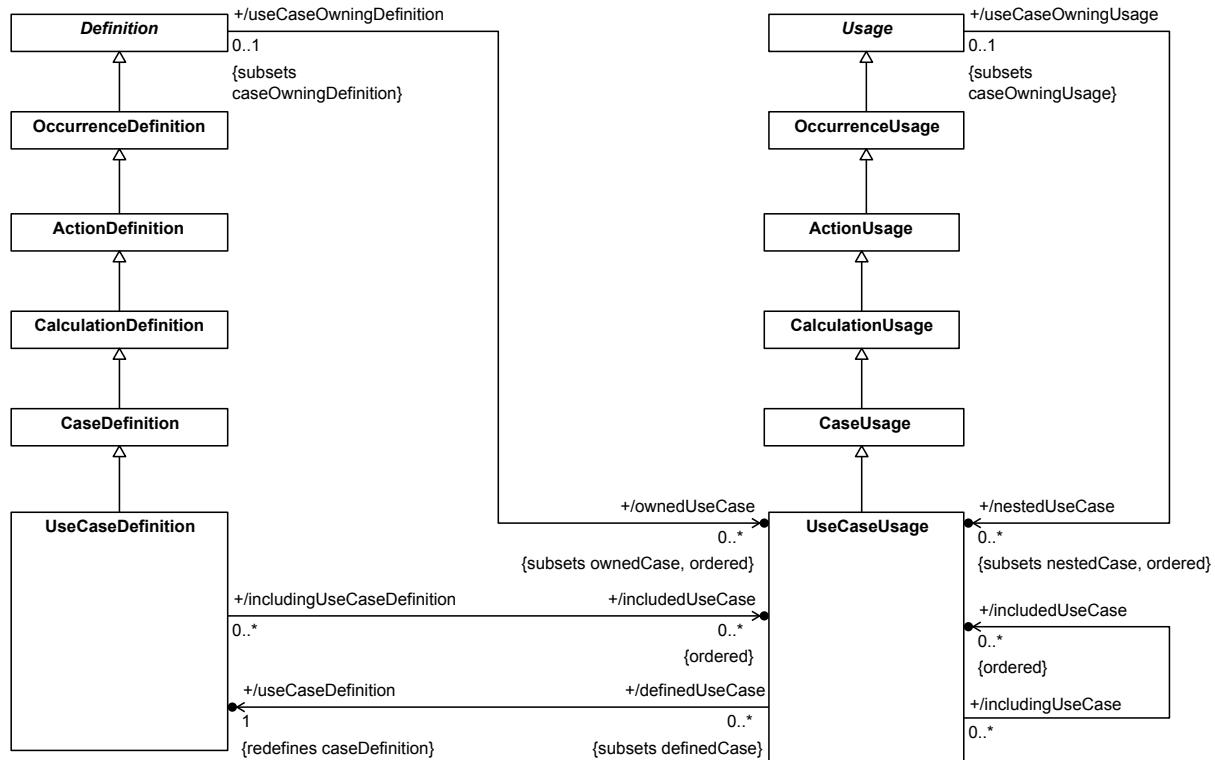


Figure 94. Use Case Definition and Usage

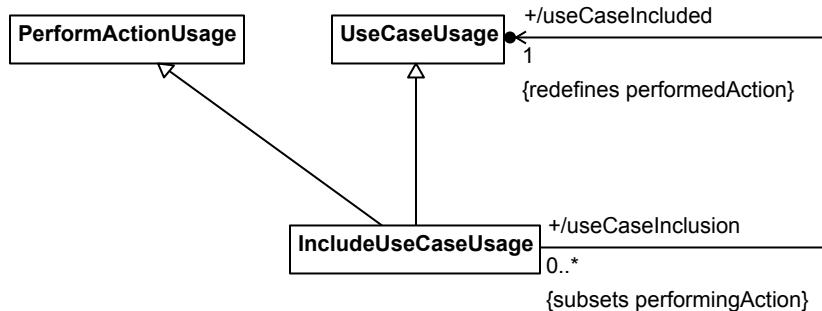


Figure 95. Use Case Inclusion

8.3.21.2 IncludeUseCaseUsage

Description

An **IncludeUseCaseUsage** is a **UseCaseUsage** that represents the inclusion of a **UseCaseUsage** by a **UseCaseDefinition** or **UseCaseUsage**. The **UseCaseUsage** to be included (which may be the **IncludeUseCaseUsage** itself) is related to the **includedUseCase** by a **Subsetting Relationship**. An **IncludeUseCaseUsage** is also a **PerformActionUsage**, with its **includedUseCase** as the **performedAction**.

If the **IncludeUseCaseUsage** is owned by a **UseCaseDefinition** or **UseCaseUsage**, then it also subsets the **UseCaseUsage** *UseCase::includedUseCases* from the Systems model library.

General Classes

PerformActionUsage
UseCaseUsage

Attributes

/useCaseIncluded : **UseCaseUsage** {redefines performedAction}

The **UseCaseUsage** to be included by this **IncludeUseCaseUsage**. It is the **subsettectedFeature** of the first owned **Subsetting Relationship** of the **IncludeUseCaseUsage**.

Operations

No operations.

Constraints

No constraints.

8.3.21.3 UseCaseDefinition

Description

A **UseCaseDefinition** is a **CaseDefinition** that specifies a set of actions performed by its subject, in interaction with one or more actors external to the subject. The objective is to yield an observable result that is of value for one or more of the actors.

A **UseCaseDefinition** must subclass, directly or indirectly, the base **UseCaseDefinition** *UseCase* from the Systems model library.

General Classes

CaseDefinition

Attributes

/includedUseCase : **UseCaseUsage** [0..*] {ordered}

The **UseCaseUsages** that are included by this **UseCaseDefinition**. Derived as the **includedUseCase** of the **IncludeUseCaseUsages** owned by this **UseCaseDefinition**.

Operations

No operations.

Constraints

No constraints.

8.3.21.4 UseCaseUsage

Description

A UseCaseUsage is a Usage of a UseCaseDefinition.

A UseCaseUsage must subset, directly or indirectly, either the base UseCaseUsage *useCases* from the Systems model library. If it is owned by a UseCaseDefinition or UseCaseUsage then it must subset the UseCaseUsage *UseCase::subUseCases*.

General Classes

CaseUsage

Attributes

/includedUseCase : UseCaseUsage [0..*] {ordered}

The UseCaseUsages that are included by this UseCaseUsage. Derived as the `includedUseCase` of the `IncludeUseCaseUsages` owned by this UseCaseUsage.

/useCaseDefinition : UseCaseDefinition {redefines caseDefinition}

The UseCaseDefinition that is the type of this UseCaseUsage.

Operations

No operations.

Constraints

No constraints.

8.3.22 Views Abstract Syntax

8.3.22.1 Overview

Submission Note. This is a preliminary abstract syntax model for views and viewpoints. A complete model will be provided in the revised submission.

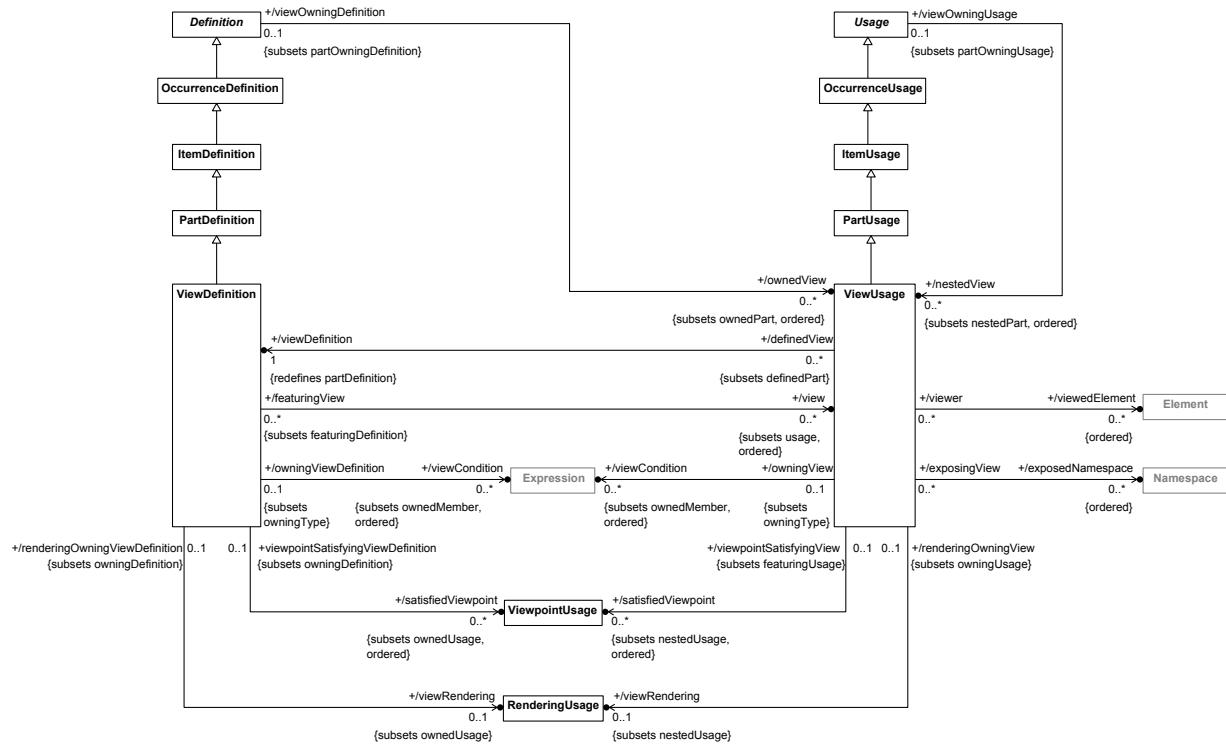


Figure 96. View Definition and Usage

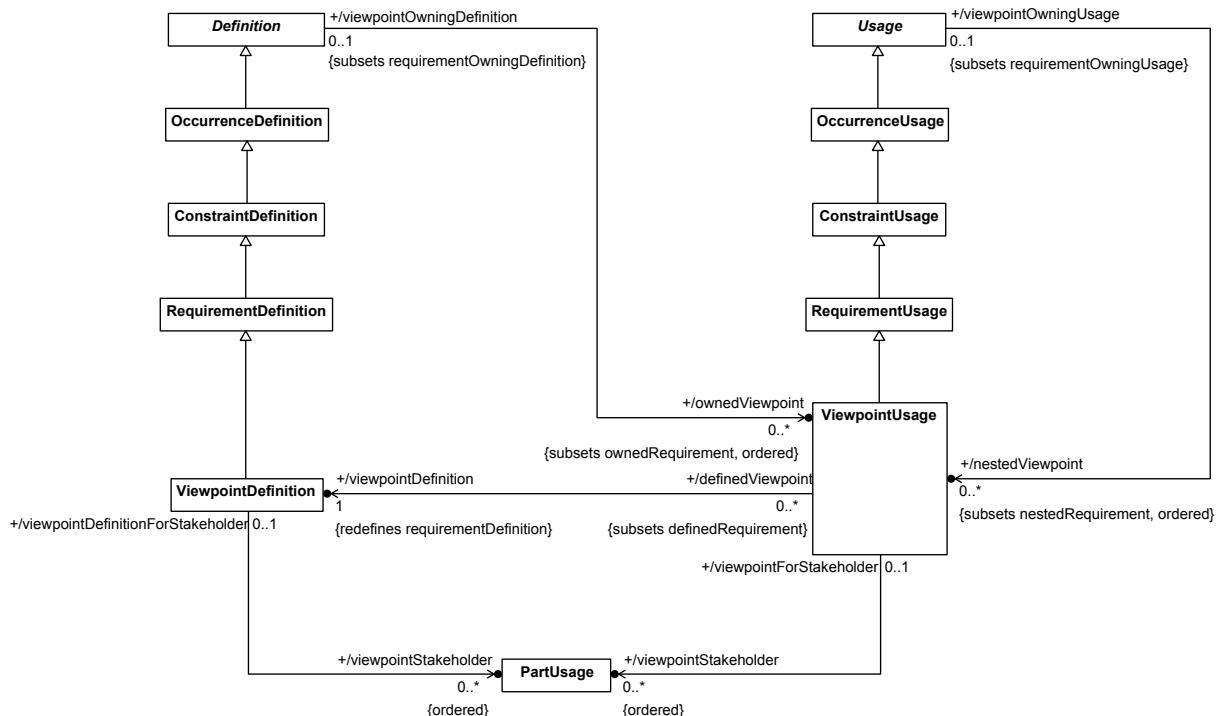


Figure 97. Viewpoint Definition and Usage

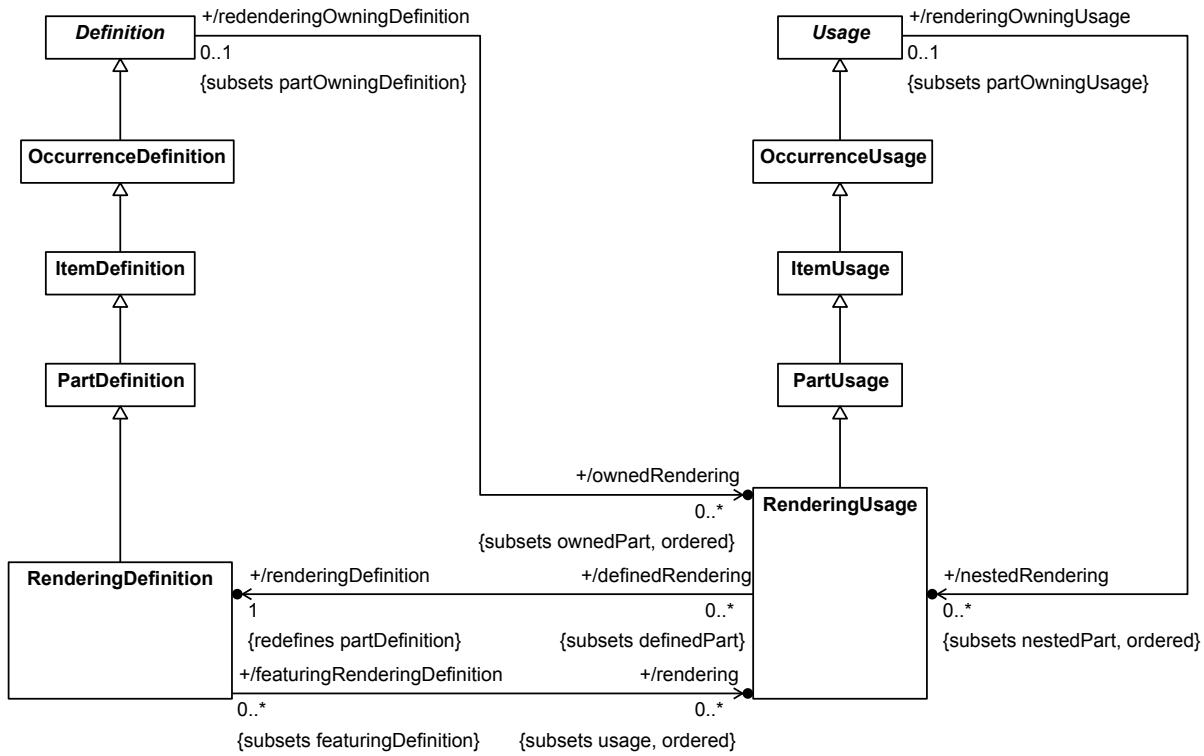


Figure 98. Rendering Definition and Usage

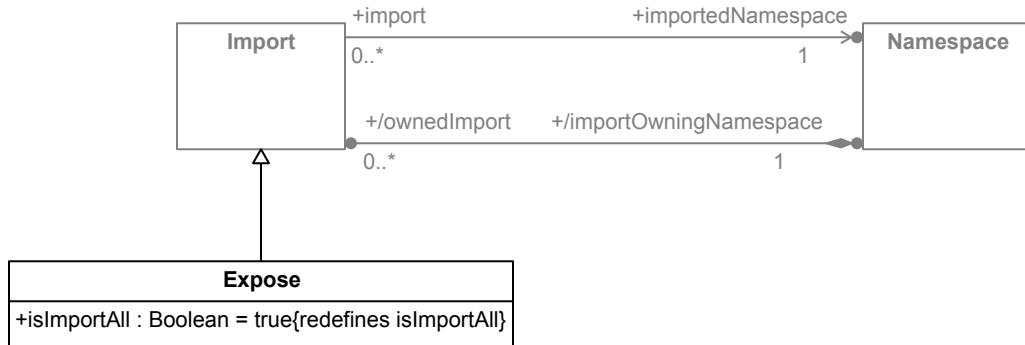


Figure 99. Expose Relationship

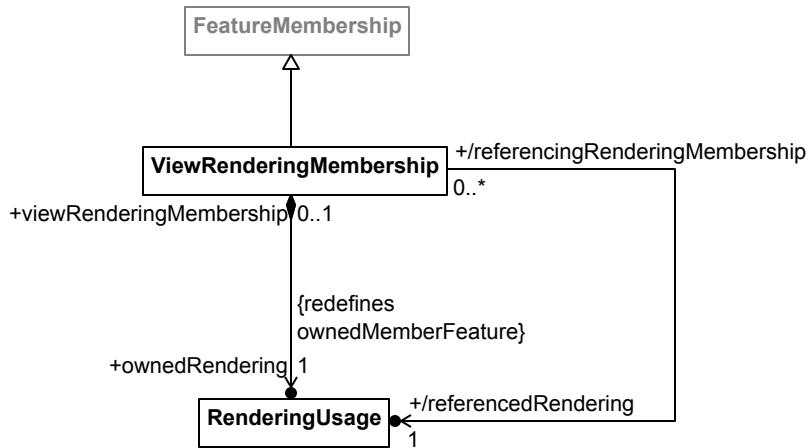


Figure 100. View Rendering Membership

8.3.22.2 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. Visibility is always ignored for an Expose (i.e., `isImportAll = true`).

General Classes

Import

Attributes

`isImportAll : Boolean {redefines isImportAll}`

An Expose always imports all Elements, regardless of visibility.

Operations

No operations.

Constraints

`exposeIsImportAll`

An Expose always imports all Elements, regardless of visibility.

`isImportAll`

8.3.22.3 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, ordered}`

The usages of a `RenderingDefinition` that are `RenderingUsages`.

Operations

No operations.

Constraints

No constraints.

8.3.22.4 `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.

8.3.22.5 `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, ordered}`

The `ownedUsages` of this ViewDefinition that are ViewpointUsages for viewpoints satisfied by the ViewDefinition.

`/view : ViewUsage [0..*] {subsets usage, ordered}`

The `usages` of this ViewDefinition that are ViewUsages.

`/viewCondition : Expression [0..*] {subsets ownedMember, ordered}`

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.

8.3.22.6 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 : PartUsage [0..*] {ordered}

The features that identify the stakeholders with concerns framed by this ViewpointDefinition, derived as the owned and inherited stakeholderParameters of the framedConcerns of this ViewpointDefinition.

Operations

No operations.

Constraints

No constraints.

8.3.22.7 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 : PartUsage [0..*] {ordered}

The features that identify the stakeholders with concerns addressed by this ViewpointUsage, derived as the owned and inherited stakeholderParameters of the framedConcerns of this ViewpointUsage.

Operations

No operations.

Constraints

No constraints.

8.3.22.8 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 {redefines ownedMemberFeature}

/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.

8.3.22.9 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..*] {ordered}

The Namespaces that are exposed by this ViewUsage, derived as the Namespaces related to the ViewUsage by Expose Relationships.

/satisfiedViewpoint : ViewpointUsage [0..*] {subsets nestedUsage, ordered}

The nestedUsages of this ViewUsage that are ViewpointUsages for (additional) viewpoints satisfied by the ViewUsage.

/viewCondition : Expression [0..*] {subsets ownedMember, ordered}

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..*] {ordered}

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.

8.4 Semantics

8.4.1 Semantics Overview

Submission Note. The Semantics subclauses are intended to detail the formal semantics of SysML in terms of the semantics of KerML. This will be provided in the final submission.

8.4.2 Definition and Usage Semantics

8.4.3 Attributes Semantics

8.4.4 Enumerations Semantics

8.4.5 Occurrences Semantics

8.4.6 Items Semantics

8.4.7 Parts Semantics

8.4.8 Ports Semantics

8.4.9 Connections Semantics

8.4.10 Interfaces Semantics

8.4.11 Allocations Semantics

8.4.12 Actions Semantics

8.4.13 States Semantics

8.4.14 Calculations Semantics

8.4.15 Constraints Semantics

8.4.16 Requirements Semantics

8.4.17 Cases Semantics

8.4.18 Analysis Cases Semantics

8.4.19 Verification Cases Semantics

8.4.20 Use Cases Semantics

8.4.21 View Semantics

9 Model Libraries

9.1 Model Libraries Overview

The SysML model libraries are an integral part of the language. The Systems Model Library (see [9.2](#)) is used any time a Definition or Usage element is instantiated in a user model, providing a bridge to the semantic models in the Kernel Model Library [KerML, Clause 8]. For example, any ItemDefinition or ItemUsage must directly or indirectly specialize the based ItemDefinition *Item* from the *Items* library model, where *Item* specializes the Kernel Class *Object*, giving Items the semantics of structural Objects.

SysML also includes a set of domain libraries, which provide models of fundamental concepts from domains of particular importance in systems engineering. These models are normative and available for use in all SysML user models. The following domain libraries are included.

- The *Metadata Domain Library* contains models of attribute definitions for a useful set of standard metadata annotations (see [9.3](#); see also [7.3](#) on Annotations).
- The *Analysis Domain Library* contains models of concepts useful in carrying out analyses of systems. In particular, it includes frameworks for state space representation of systems and for performing trade-off studies (see [9.4](#)).
- The *Quantities and Units Domain Library* contains a comprehensive set of models for scalar, vector and tensor quantities, including quantity value and unit definitions covering the ISO 80000 and ISO 8601-1 standards (see [9.5](#)).

Submission Note. It is expected that additional domain libraries will be included in the final submission, including at least one for basic geometry.

The normative definition of all library models is given in the SysML textual notation files for them associated with this specification. The documentation on these models provided in this clause is either derived from the model files themselves or gives additional overview information on the use of the models, and is therefore also considered normative.

Submission Note. This clause currently does not include any graphical representation for the library models. Consideration will be given to including such diagrams in the final submission, as the tooling for providing accurate visualization of SysML v2 models improves.

9.2 Systems Model Library

9.2.1 Overview

The Systems Model Library includes models for the base types of all kinds of Definition and Usage elements in SysML. Each of the following subclauses describes a library model package corresponding to the elements in the similarly named abstract syntax package (see [8.3](#)). For example, the *Attributes* library model package (see [9.2.2](#)) includes the *Attribute* and *attributes* types that are the base types for all AttributeDefinitions and AttributeUsages (respectively) as specified in the *Attributes* abstract syntax package (see [8.3.4](#)).

9.2.2 Attributes

9.2.2.1 Attributes Overview

This package defines the base types for attributes and related structural elements in the SysML language.

9.2.2.2 Elements

9.2.2.2.1 attributeValues <AttributeUsage>

Description

attributeValues is the base feature for all AttributeUsages.

General Classes

AttributeValue
dataValues

Attributes

No attributes.

Constraints

No constraints.

9.2.2.2 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

WithValue

Attributes

No attributes.

Constraints

No constraints.

9.2.3 Items

9.2.3.1 Items Overview

This package defines the base types for items and related structural elements in the SysML language.

9.2.3.2 Elements

9.2.3.2.1 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

```
checkedConstraints : ConstraintCheck [0..*] {subsets enactedPerformance}  
subitems : Item [0..*] {subsets happensDuring?!"}
```

Constraints

No constraints.

9.2.3.2.2 items <ItemUsage>

Description

items is the base feature of all ItemUsages.

General Classes

objects
Item

Attributes

No attributes.

Constraints

No constraints.

9.2.4 Parts

9.2.4.1 Parts Overview

This package defines the base types for parts and related structural elements in the SysML language.

9.2.4.2 Elements

9.2.4.2.1 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}
```

StateActions that are exhibited by this Part.

```
performedActions : Action [0..*] {subsets enactedPerformance}
```

Actions that are performed by this Part.

portsOnPart : Port [0..*] {subsets happensDuring?¹}

Ports that are owned by this Part.

subparts : Part [0..*] {subsets subitems}

Constraints

No constraints.

9.2.4.2.2 parts <PartUsage>

Description

parts is the base feature of all PartUsages.

General Classes

Part
items

Attributes

No attributes.

Constraints

No constraints.

9.2.5 Ports

9.2.5.1 Ports Overview

This package defines the base types for ports and related structural elements in the SysML language.

9.2.5.2 Elements

9.2.5.2.1 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

Object

Attributes

subports : Port [0..*] {subsets happensDuring?¹}

Constraints

No constraints.

9.2.5.2.2 ports <PortUsage>

Description

`ports` is the base feature of all PortUsages.

General Classes

Port
objects

Attributes

No attributes.

Constraints

No constraints.

9.2.6 Connections

9.2.6.1 Connections Overview

This package defines the base types for connections and related structural elements in the SysML language.

9.2.6.2 Elements

9.2.6.2.1 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.

9.2.6.2.2 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.

9.2.7 Interfaces

9.2.7.1 Interfaces Overview

This package defines the base types for interfaces and related structural elements in the SysML language.

9.2.7.2 Elements

9.2.7.2.1 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.

9.2.7.2.2 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.

9.2.8 Allocations

9.2.8.1 Allocations Overview

9.2.8.2 Elements

9.2.8.2.1 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.

9.2.8.2.2 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.

9.2.9 Actions

9.2.9.1 Actions Overview

This package defines the base types for actions and related behavioral elements in the SysML language.

9.2.9.2 Elements

9.2.9.2.1 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.

9.2.9.2.2 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..*] {subsets subperformances}

The subperformances of this action that are actions.

subtransitions : TransitionAction [0..*]

Constraints

No constraints.

9.2.9.2.3 actions <ActionUsage>

Description

actions is the base feature for all ActionUsages.

General Classes

Action
performances

Attributes

No attributes.

Constraints

No constraints.

9.2.9.2.4 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.

9.2.9.2.5 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.

9.2.9.2.6 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.

9.2.9.2.7 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.

9.2.9.2.8 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.

9.2.9.2.9 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.

9.2.10 States

9.2.10.1 States Overview

This package defines the base types for states and related behavioral elements in the SysML language.

9.2.10.2 Elements

9.2.10.2.1 StateAction <StateDefinition>

Description

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.

9.2.10.2.2 stateActions <StateUsage>

Description

stateActions is the base feature for all StateUsages.

General Classes

actions
StateAction

Attributes

No attributes.

Constraints

No constraints.

9.2.10.2.3 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}
triggerPayload : Anything [0..*]

Constraints

No constraints.

9.2.10.2.4 transitionActions <TransitionUsage>

Description

transitionActions is the base feature for all TransitionUsages.

General Classes

actions
Action

TransitionAction

Attributes

No attributes.

Constraints

No constraints.

9.2.11 Calculations

9.2.11.1 Calculations Overview

This package defines the base types for calculations and related behavioral elements in the SysML language.

9.2.11.2 Elements

9.2.11.2.1 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.

9.2.11.2.2 calculations <CalculationUsage>

Description

calculations is the base Feature for all CalculationUsages.

General Classes

Calculation
actions
evaluations

Attributes

No attributes.

Constraints

No constraints.

9.2.12 Constraints

9.2.12.1 Constraints Overview

This package defines the base types for constraints and related behavioral elements in the SysML language.

9.2.12.2 Elements

9.2.12.2.1 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.

9.2.12.2.2 constraintChecks <ConstraintUsage>

Description

constraintChecks is the base feature of all ConstraintUsages.

General Classes

booleanEvaluations

ConstraintCheck

Attributes

No attributes.

Constraints

No constraints.

9.2.13 Requirements

9.2.13.1 Requirements Overview

This package defines the base types for requirements and related behavioral elements in the SysML language.

9.2.13.2 Elements

9.2.13.2.1 ConcernCheck <ConcernDefinition>

Description

ConcernCheck is the most general class for concern checking. ConcernCheck is the base type of all ConcernDefinitions.

General Classes

RequirementCheck

Attributes

No attributes.

Constraints

No constraints.

9.2.13.2.2 concernChecks <ConcernUsage>

Description

concernChecks is the base feature of all ConcernUsages.

General Classes

requirementChecks
ConcernCheck

Attributes

No attributes.

Constraints

No constraints.

9.2.13.2.3 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.

9.2.13.2.4 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.

9.2.13.2.5 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.

9.2.13.2.6 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.

9.2.13.2.7 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.

9.2.13.2.8 RequirementCheck <RequirementDefinition>

Description

RequirementCheck is the most general class for requirements checking. RequirementCheck is the base type of all RequirementDefinitions.

General Classes

ConstraintCheck

Attributes

actors : Part [0..*]

The Parts that fill the role of actors for this RequirementCheck.

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.

stakeholders : Part [0..*]

The Parts that represent stakeholders interested in the requirement being checked.

subject : Anything

The entity that is being checked for satisfaction of the required constraints.

Constraints

[no name]

[no documentation]

allTrue(assumptions) implies allTrue(constraints)

9.2.13.2.9 requirementChecks <RequirementUsage>

Description

requirementChecks is the base feature of all RequirementUsages.

General Classes

RequirementCheck
constraintChecks

Attributes

No attributes.

Constraints

No constraints.

9.2.13.2.10 Stakeholder <StakeholderDefinition>

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

9.2.13.2.11 stakeholders <StakeholderUsage>

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

9.2.14 Cases

9.2.14.1 Cases Overview

This package defines the base types for cases and related behavioral elements in the SysML language.

9.2.14.2 Elements

9.2.14.2.1 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

actors : Part [0..*]

The Parts that fill the role of actors for this Case.

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.

9.2.14.2.2 cases <CaseUsage>

Description

cases is the base feature of all CaseUsages.

General Classes

calculations
Case

Attributes

No attributes.

Constraints

No constraints.

9.2.15 Analysis Cases

9.2.15.1 Analysis Cases Overview

This package defines the base types for analysis cases and related behavioral elements in the SysML language.

9.2.15.2 Elements

9.2.15.2.1 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.

9.2.15.2.2 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.

9.2.15.2.3 analysisCases <AnalysisCaseUsage>

Description

analysisCases is the base feature of all AnalysisCaseUsages.

General Classes

AnalysisCase
cases

Attributes

No attributes.

Constraints

No constraints.

9.2.16 Verification Cases

9.2.16.1 Verification Cases Overview

This package defines the base types for verification cases and related behavioral elements in the SysML language.

9.2.16.2 Elements

9.2.16.2.1 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.

9.2.16.2.2 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.

9.2.16.2.3 verificationCases <VerificationCaseUsage>

Description

verificationCases is the base feature of all VerificationCaseUsages.

General Classes

VerificationCase
cases

Attributes

No attributes.

Constraints

No constraints.

9.2.16.2.4 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.

9.2.17 Use Cases

9.2.17.1 Use Cases Overview

9.2.17.2 Elements

9.2.17.2.1 UseCase

Description

UseCase is the most general class of performances of UseCaseDefinitions. UseCase is the base class of all UseCaseDefinitions.

General Classes

Case

Attributes

includedUseCases : UseCase [0..*] {subsets subUseCases}

Other UseCases included by this UseCase (i.e., as modeled by an IncludeUseCaseUsage).

subUseCases : UseCase [0..*] {subsets subcases}

Other UseCases carried out as part of the performance of this UseCase.

Constraints

No constraints.

9.2.17.2.2 useCases

Description

useCases is the base feature of all UseCaseUsages.

General Classes

UseCase
cases

Attributes

No attributes.

Constraints

No constraints.

9.2.18 Views

9.2.18.1 Views Overview

This package defines the base types for views, viewpoints, renderings and related elements in the SysML language.

9.2.18.2 Elements

9.2.18.2.1 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.

9.2.18.2.2 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.

9.2.18.2.3 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.

9.2.18.2.4 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.

9.2.18.2.5 GraphicalRendering <RenderingDefinition>

Description

A GraphicalRendering is a Rendering of a View into a Graphical format.

General Classes

Rendering

Attributes

No attributes.

Constraints

No constraints.

9.2.18.2.6 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.

9.2.18.2.7 renderings <RenderingUsage>

Description

renderings is the base feature of all RenderingUsages.

General Classes

Rendering
parts

Attributes

No attributes.

Constraints

No constraints.

9.2.18.2.8 TabularRendering <RenderingDefinition>

Description

A TabularRendering is a Rendering of a View into a tabular format.

General Classes

Rendering

Attributes

No attributes.

Constraints

No constraints.

9.2.18.2.9 TextualRendering <RenderingDefinition>

Description

A TextualRendering is a Rendering of a View into a textual format.

General Classes

Rendering

Attributes

No attributes.

Constraints

No constraints.

9.2.18.2.10 View <ViewDefinition>

Description

View is the base type of all ViewDefinitions.

General Classes

Part

Attributes

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.

viewRendering : Rendering [0..1]

The Rendering of this View.

Constraints

No constraints.

9.2.18.2.11 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

subject : View {redefines subject}

The subject of this ViewpointCheck, which must be a View.

Constraints

No constraints.

9.2.18.2.12 viewpointChecks <ViewpointUsage>

Description

viewpointChecks is the base feature of all ViewpointUsages.

General Classes

ViewpointCheck
requirementChecks

Attributes

No attributes.

Constraints

No constraints.

9.2.18.2.13 viewpointConformance <SatisfyRequirementUsage>

Description

General Classes

RequirementCheck

Attributes

viewpointSatisfactions : ViewpointCheck [0..*] {subsets constraints}

The required ViewpointChecks.

Constraints

No constraints.

9.2.18.2.14 views <ViewUsage>

Description

views is the base feature of all ViewUsages.

General Classes

parts

View

Attributes

No attributes.

Constraints

No constraints.

9.3 Metadata Domain Library

9.3.1 Metadata Domain Library Overview

9.3.2 Modeling Metadata

9.3.3 Risk Metadata

9.4 Analysis Domain Library

9.4.1 Analysis Domain Library Overview

The Analysis Domain Library provides library models supporting the modeling of analysis cases (see [7.22](#)).

9.4.2 Analysis Tooling

9.4.3 Sampled Functions

9.4.4 State Space Representation

9.4.4.1 State Space Representation Overview

State Space Representation (SSR) is a foundational dynamical systems representation, commonly used in control systems. In this representation, a system is described by a set of *state variables* whose evolution by a *state equation* (not that this is a different conception of "state" than used in the behavioral state modeling constructs described in [7.17](#)). The system outputs are then given by an *output equation*. This representation provides a description of the quantitative stateful behavior of the target system in an explicit manner so that external solvers can compute the behavior by properly integrating input and state variables.

Mathematically, let \mathbf{x} be a vector of state variables and $\dot{\mathbf{x}}$ be its time derivative, \mathbf{u} be a vector of inputs, and \mathbf{y} be a vector of outputs. Then the state equation has the form

$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}),$$

for some system-specific function f , and the output equation has the form

$$\mathbf{y} = g(\mathbf{x}, \mathbf{u}),$$

for some system-specific function g .

The *StateSpaceRepresentation* library model is a model of this representation in terms of SysML actions and calculations. These can be used in combination with end-user action models to describe system functional behaviors.

9.4.4.2 Elements

The model library defines the action definitions, *StateSpaceDynamics* defines input, corresponding to \mathbf{u} , output, corresponding to \mathbf{y} , and *stateSpace*, corresponding to \mathbf{x} , of the state space representation.

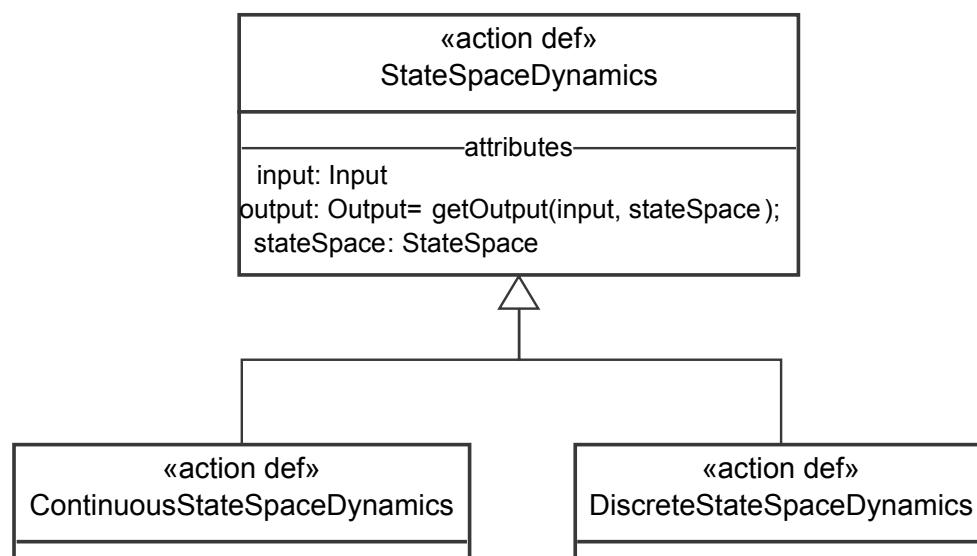


Figure 101. State Space Representation action definitions

9.4.5 Trade Studies

9.4.5.1 Trade Studies Overview

This package provides a simple framework for defining trade-off study analysis cases.

9.4.5.2 Elements

9.4.5.2.1 MaximizeObjective

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

9.4.5.2.2 MinimizeObjective

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

9.4.5.2.3 ObjectiveFunction

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

9.4.5.2.4 TradeStudy

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

9.4.5.2.5 TradeStudyObjective

Description

General Classes

Model Library Element Description

Attributes

No attributes.

Constraints

No constraints.

9.5 Quantities and Units Domain Library

9.5.1 Quantities and Units Domain Library Overview

For any system model, a solid foundation for the representation of physical quantities, their units, scales, and quantity dimensions, as well as coordinate systems is essential. Quantity attributes are needed to specify many characteristics of a system of interest and its elements. The foundation should be a shareable resource that can be reused in models within and across project as well as organizations in order to facilitate collaboration and model interoperability.

The Quantities and Units Domain Library defines reusable model elements for physical quantities, including vector and tensor quantities, quantity dimensions, measurement units, measurement scales, coordinate systems and their transformations. It also provides for the definition of coherent systems of quantities and systems of units, as well as collections of operators and functions to support unambiguous quantity expressions, automated unit or scale conversion, and coordinate transformations.

The most widely accepted, scrutinized, and globally used foundational specification 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 domain 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 the representation of the ISQ and the SI, but also of any other systems of quantities and units.

The data model in this library includes precise representation of the relationships between quantities, units, scales and quantity dimensions with unambiguous semantics. 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.

This library further contains lower level packages that represent actual quantities, units and scales as specified in parts 3 to 14 of the ISO/IEC 80000 series. These packages are intended to provide a broad common basis, that can be extended and tailored for use by particular communities of practice and industry sectors.

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, including their relationships with ISQ quantities, and their conversion factors with respect to SI units as specified in [NIST SP-811].

9.5.2 Quantities

9.5.2.1 Quantities Overview

Taxonomy

The Quantities package defines the root elements to represent quantities and their values.

TensorQuantityValue (an AttributeDefinition) and tensorQuantities (an AttributeUsage) are defined to represent quantities at the most general level, and can represent any n^{th} order tensor quantity. Then, VectorQuantityValue and vectorQuantities are defined as order 1 specializations of the tensor quantity concepts, and finally, ScalarQuantityValue and scalarQuantities as order 0 specializations of the vector quantity concepts.

Quantity Values

A quantity value is defined as a tuple of:

- a sequence of one or more mathematical numbers (as AttributeUsage num),
- a measurement reference (as AttributeUsage mRef).

For a ScalarQuantityValue the sequence of numbers collapses to one single number, and the measurement reference is typically a measurement unit or scale. For a VectorQuantityValue there must be as many numbers as needed to define the magnitude and direction of the vector quantity, and a measurement reference that typically specifies a coordinate system, e.g., a sequence of 3 numbers for the vector components in a standard ortho-normal Cartesian 3D vector space with the same measurement unit on each of the axes. For a TensorQuantityValue the measurement reference must establish a reference frame compliant with the full dimensionality of the tensor quantity involved.

Note: The specification of a quantity value as a tuple of its numerical value and a measurement reference has the big advantage that the type of a quantity value becomes independent from the choice of measurement reference. For example: a power expressed as 1.5 watt has the same type (AttributeDefinition *ForceValue*) as a power expressed as 1500 milliwatt. In SysML v1 the choice of measurement reference (i.e. measurement unit) was embedded in the value property type.

Free versus Bound Quantities and Vector Spaces

A TensorQuantityValue can be defined with respect to a free vector space product or a bound vector space product. Similarly, a VectorQuantityValue can be defined with respect to a free or a bound vector space, and a ScalarQuantityValue with respect to a free or bound number line, which can be regarded as the one-dimensional version of a vector space. In a free vector space, vectors can be added and vectors can be multiplied by a scalar number, where both operations yield a new free vector. Free vectors have only magnitude and direction. A bound vector space includes a particular choice of origin, and vectors in such a space can not be added nor multiplied by scalars. AttributeUsage `isBound` is used to capture this: `isBound == false` specifies a free, while `isBound == true` specifies a bound vector space (product).

Examples that (informally) illustrate the distinction between free and bound vector quantities are given by pairs of quantities of the same quantity dimension:

1. Displacement vector (free) and the position vector (bound), both of quantity dimension length.
Two displacement vectors can be added and yield a resulting displacement vector. A displacement vector can also be multiplied with a scalar factor, which changes only its magnitude. Two position vectors can not be added, nor can a position vector be multiplied by a scalar number. A position vector is always bound to the origin of its bound vector space. One can however subtract one position vector from another, and the result is a displacement vector! Namely, the displacement to get from the position denoted by the first vector to that of the second vector.
2. Duration (free scalar) and time instant (bound scalar), both of quantity dimension time.
Durations can be added and multiplied, time instants cannot. Time instants values can only be specified with respect to a measurement reference that is time scale with some particular choice of zero. Such a time scale is the same as a (time) coordinate axis in a the one-dimensional bound vector space.

9.5.2.2 Elements

9.5.2.2.1 scalarQuantities <AttributeUsage>

Description

AttributeUsage `scalarQuantities : ScalarQuantityValue[*] nonunique` is the subset of `vectorQuantities` that defines a top-level general self-standing attribute that can be used to consistently specify scalar quantities of Occurrences.

Any particular scalar quantity attribute is specified by subsetting `scalarQuantities`. In other words, the co-domain of a scalar quantity attribute is a suitable specialization of `ScalarQuantityValue`.

General Classes

`vectorQuantities`
`ScalarQuantityValue`

Attributes

No attributes.

Constraints

No constraints.

9.5.2.2.2 ScalarQuantityValue <AttributeDefinition>

Description

A ScalarQuantityValue is an abstract AttributeDefinition that specializes VectorQuantityValue. It represents a scalar quantity value as a tuple of a Number num and a ScalarMeasurementReference mRef. By definition it has order zero. The ScalarMeasurementReference is typically a MeasurementUnit or a MeasurementScale.

General Classes

TensorQuantityValue
VectorQuantityValue

Attributes

mRef : ScalarMeasurementReference {redefines mRef}

Specification of the ScalarMeasurementReference for the value of the scalar quantity.

Constraints

oneElement

[no documentation]

dimensions[1] == 1

order == 0

9.5.2.2.3 tensorQuantities <AttributeUsage>

Description

AttributeUsage tensorQuantities : TensorQuantityValue[*] nonunique defines a top-level general self-standing attribute that can be used to consistently specify quantities of Occurrences.

Any particular tensor quantity attribute is specified by subsetting tensorQuantities. In other words, the co-domain of a tensor quantity attribute is a suitable specialization of TensorQuantityValue.

General Classes

TensorQuantityValue
dataValues

Attributes

No attributes.

Constraints

No constraints.

9.5.2.2.4 TensorQuantityValue <AttributeDefinition>

Description

A TensorQuantityValue is an abstract AttributeDefinition and a specialization of *Collections::Array* that represents a tensor quantity value as a sequence of Numbers and a TensorMeasurementReference.

The dimensionality of the tensor quantity is specified in `dimensions`, from which the `order` of the tensor is derived. Typically in engineering the name 'tensor' is used if its order is 2 or greater, but mathematically a tensor can have order 0 or 1.

A `TensorQuantityValue` must have the same `dimensions` and `order` as the `TensorMeasurementReference` that it references via Attribute `mRef`.

It is possible to specify the contravariant and covariant order of a tensor quantity through the Attributes `contravariantOrder` and `covariantOrder`, of which the sum must be equal to the overall `order`. In applications where it is not important to distinguish between contravariant and covariant tensors (or vectors), the convention is use contravariant by default and make the `contravariantOrder` equal to `order`.

General Classes

`Collections::Array`

Attributes

`contravariantOrder` : Positive

The number of contravariant indices of the tensor quantity.

`covariantOrder` : Positive

The number of covariant indices of the tensor quantity.

`dimensions` : Positive [1..*] {redefines `dimensions`, ordered, nonunique}

A sequence of positive integer numbers that define the dimensionality of the tensor quantity.

Examples: for a second order 3D tensor `dimensions` = (3, 3) ; for fourth order 2D tensor `dimensions` = (2, 2, 2, 2) ;

The `dimensions` must be the same as the `dimensions` of the associated `mRef`.

`isBound` : Boolean

Assertion whether this tensor quantity is defined in a free (`isBound == false`) or bound (`isBound == true`) vector space product.

`mRef` : `TensorMeasurementReference`

Specification of the `TensorMeasurementReference` for the value of the tensor quantity.

`num` : Number [0..*] {redefines `elements`, ordered, nonunique}

Sequence of numbers that specify the numerical value of the tensor quantity.

`order` : Natural {redefines `rank`}

Order of the tensor quantity. The order is derived to be equal to the size of `dimensions`.

Constraints

```
orderSum  
[no documentation]  
  
contravariantOrder + covariantOrder == order  
  
matchingDimensions  
[no documentation]  
  
dimensions == mRef.dimensions
```

9.5.2.2.5 vectorQuantities <AttributeUsage>

Description

AttributeUsage vectorQuantities : VectorQuantityValue[*] nonunique is the subset of tensorQuantities that defines a top-level general self-standing attribute that can be used to consistently specify vector quantities of Occurrences.

Any particular vector quantity attribute is specified by subsetting vectorQuantities. In other words, the co-domain of a vector quantity attribute is a suitable specialization of VectorQuantityValue.

General Classes

VectorQuantityValue
tensorQuantities

Attributes

No attributes.

Constraints

No constraints.

9.5.2.2.6 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}

Specification of the VectorMeasurementReference for the value of the vector quantity.

Constraints

[no name]

[no documentation]

order == 1

9.5.3 Units and Scales

9.5.3.1 Units and Scales Overview

This package defines the general AttributeDefinitions and AttributeUsages to construct measurement references, including measurement units and scales (including ordinal, logarithmic and cyclic scales), unit conversions, coordinate systems, coordinate transformations, as well as measurement unit prefixes to denote multiples (such as 'mega') and sub-multiples (such as 'nano'). It also defines concepts to represent quantity dimensions, which form the basis for quantity dimensional analysis.

9.5.3.2 Elements

9.5.3.2.1 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.

9.5.3.2.2 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/IEC-80000-1] as a decimal or binary multiple or sub-multiple.

Constraints

No constraints.

9.5.3.2.3 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]
```

9.5.3.2.4 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.

9.5.3.2.5 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.

9.5.3.2.6 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.

Implementation note: In order to enable quantity value conversion between an IntervalScale and another measurement scale, the offset (sometimes also called zero shift) between the source and target scales must be known. This offset can be indirectly defined through a ScaleValueMapping, see scaleValueMapping of MeasurementScale. This will be aligned with, and possibly replaced by, a 1D coordinate transformation, so that scalar and vector transformations are handled in the same way.

General Classes

MeasurementScale

Attributes

isBound : Boolean {redefines isBound}

For an IntervalScale isBound is always true, since the scale must include a definition of what zero means.

Constraints

No constraints.

9.5.3.2.7 LogarithmBaseKind

Description

General Classes

No general classes.

Literal Values

10
2
e

9.5.3.2.8 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:

$$v = f * \log_{\text{base}}((x / x_{\text{ref}})^a)$$

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.

9.5.3.2.9 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.

9.5.3.2.10 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.

General Classes

ScalarMeasurementReference

Attributes

unitConversion : UnitConversion [0..1]

AttributeUsage `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}

AttributeUsage `unitPowerFactors` specifies a product of powers of base units, that define the quantity dimension of this measurement unit.

Constraints

No constraints.

9.5.3.2.11 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.

9.5.3.2.12 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.

9.5.3.2.13 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.

9.5.3.2.14 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 where 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 where 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.

9.5.3.2.15 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
```

9.5.3.2.16 TensorMeasurementReference <AttributeDefinition>

Description

A TensorMeasurementReference is an abstract AttributeDefinition and a specialization of `Collections::Array`, 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.

As an example a Cartesian 3-dimensional 3x3 moment of inertia tensor would have the following attributes:

```
attribute def Cartesian3dMomentOfInertiaMeasurementReference :> TensorMeasurementReference
  attribute :>> dimensions = (3, 3);
  attribute :>> isBound = false;
  attribute :>> mRefs: MomentOfInertiaUnit[9];
}
```

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

Collections::Array

Attributes

`isBound` : Boolean

`longName` : String

`mRefs` : `ScalarMeasurementReference` [0..*] {redefines elements, ordered, nonunique}

`order` : Natural {redefines rank}

Constraints

No constraints.

9.5.3.2.17 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.

9.5.3.2.18 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.

9.5.3.2.19 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.

9.5.3.2.20 VectorMeasurementReference <AttributeDefinition>

Description

A VectorMeasurementReference is a specialization of MeasurementReference for vector quantities that are typed by a VectorQuantityValue. Its order is one. It implicitly 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
```

9.5.4 ISQ

9.5.4.1 ISQ Overview

The ISQ package specifies a complete set of predefined quantity types for the International System of Quantities (ISQ) as specializations of TensorQuantityValue, VectorQuantityValue, ScalarQuantityValue, and SystemOfQuantities, that capture the all quantities defined in ISO/IEC 80000 parts 3 to 13. It also defines all TensorMeasurementReference, VectorMeasurementReference and ScalarMeasurementReference specialization needed to define concrete measurement reference AttributeDefinitions needed to specify the actual measurement units, scales and coordinate systems in other library packages.

The ISQ package comprises the following sub-packages via import:

- ISQBase for ISO/IEC 80000 base quantities and general concepts
- ISQSpaceTime for [ISO 80000-3] "Space and Time"
- ISQMechanics for [ISO 80000-4] "Mechanics"
- ISQThermodynamics for [ISO 80000-5] "Thermodynamics"
- ISQELECTROMAGNETISM for [IEC 80000-6] "Electromagnetism"
- ISQLight for [ISO 80000-7] "Light"
- ISQAcoustics for [ISO 80000-8] "Acoustics"
- ISOChemistryMolecular for [ISO 80000-9] "Physical chemistry and molecular physics"
- ISQAtomicNuclear for [ISO 80000-10] "Atomic and nuclear physics"
- ISQCharacteristicNumbers for [ISO 80000-11] "Characteristic numbers"
- ISQCondensedMatter for [ISO 80000-12] "Condensed matter physics"
- ISQInformation for [IEC 80000-13] "Information science and technology"

Since package `ISQ` imports all other sub-packages, the statement `import ISQ::*;` suffices to make the whole `ISQ` available in a user model.

9.5.4.2 Elements

9.5.5 SI Prefixes

9.5.5.1 SI Prefixes Overview

This package specifies the SI unit prefixes as specified in [ISO 80000-1], so that they can be used in automated unit conversion.

ISO/IEC 80000-1 unit prefixes for decimal multiples and sub-multiples. See also https://en.wikipedia.org/wiki/Unit_prefix.

Name	Symbol	Value
yocto	y	10^{-24}
zepto	z	10^{-21}
atto	a	10^{-18}
femto	f	10^{-15}
pico	p	10^{-12}
nano	n	10^{-9}
micro	μ	10^{-6}
milli	m	10^{-3}
centi	c	10^{-2}
deci	d	10^{-1}
deca	da	10^1
hecto	h	10^2
kilo	k	10^3

Name	Symbol	Value
mega	M	10^6
giga	G	10^9
tera	T	10^{12}
peta	P	10^{15}
exa	E	10^{18}
zetta	Z	10^{21}
yotta	Y	10^{24}

ISO/IEC 80000-1 prefixes for binary multiples, i.e. multiples of 1024 (= 2^{10}). See also https://en.wikipedia.org/wiki/Binary_prefix.

Name	Symbol	Value
kibi	Ki	1024
mebi	Mi	1024^2
giibi	Gi	1024^3
tebi	Ti	1024^4
pebi	Pi	1024^5
exbi	Ei	1024^6
zebi	Zi	1024^7
yobi	Yi	1024^8

9.5.5.2 Elements

9.5.6 SI

9.5.6.1 SI Overview

This package specifies the (majority of) the measurement unit as defined in ISO/IEC 80000 parts 3 to 13, the International System of (Measurement) Units -- Système International d'Unités (SI).

The statement `import SI:::*`; suffices to make the all SI units available in a user model.

9.5.6.2 Elements

9.5.7 US Customary Units

9.5.7.1 US Customary Units Overview

This package specifies the US Customary Units as defined in [NIST-SP-811] Annex B, including the official NIST conversion factors to corresponding SI units.

The statement `import USCustomaryUnits::*`; suffices to make the all US Customary Units available in a user model.

9.5.7.2 Elements

9.5.8 Time

9.5.8.1 Time Overview

The Time library package specifies concepts to support time-related quantities and metrology, beyond the quantities duration and time as defined in [ISO 80000-3]. The package includes the following specifications:

- `TimeInstantValue :> ScalarQuantityValue (AttributeDefinition)`.
- `timeInstant: : TimeInstantValue :> scalarQuantities (AttributeUsage)`.
- `TimeScale :> IntervalScale (AttributeDefinition)`.
- `DateTime : TimeInstantValue (AttributeDefinition)` for a date and time-of-day on a calendar `TimeScale`.
- `Date : TimeInstantValue (AttributeDefinition)` for a date on a calendar `TimeScale`.
- `TimeOfDay : TimeInstantValue (AttributeDefinition)` for a time instant on a 24 hour day `TimeScale`.
- `UTC : TimeScale` for the Coordinated Universal Time (UTC) time scale as defined in ITU-R TF.460-6. See also https://en.wikipedia.org/wiki/Coordinated_Universal_Time.
- `UtcTimeInstantValue :> DateTime (AttributeDefinition)` and `utcTimeInstant (AttributeUsage)`.
- `Iso8601DateTime :> UtcTimeInstantValue` as well as additional AttributeDefinitions and Calculations to handle and convert ISO 8601-1 timestamps in string encoding, such as "2021-08-30^23:30:00.123456Z".

Representations of the Gregorian calendar date and time of day as specified by the ISO 8601-1 standard are used.

Reference: ISO 8601-1:2019 (First edition) "Date and time — Representations for information interchange — Part 1: Basic rules", see also <https://www.iso.org/standard/70907.html>.

9.5.8.2 Elements

9.5.8.2.1 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 a an `IntervalScale` that defines the (proleptic) Gregorian Calendar.

Attributes

No attributes.

9.5.8.2.2 TimeOfDay

Description

ISO 8601 time instant within an ISO 8601 24 hour day.

Attributes

No attributes.

A Annex: Conformance Test Suite

Submission Note. A conformance test suite will be provided in the revised submission.

B Annex: Example Model

B.1 Introduction

The example presented in this Annex is intended to illustrate how SysML can be used to model a system. The example used is a simple vehicle model. The model is represented using a combination of textual and graphical notation.

Submission Note. Since prototyping of the visualization of SysML v2 models is still ongoing, there are some inconsistencies between the diagrams in this annex and the graphical notation as currently specified in the main body of this document. These inconsistencies will be resolved in the final submission.

B.2 Model Organization

The *SimpleVehicleModel* is organized into a hierarchy of packages, where some packages contain nested packages. The *Definitions* package contains nested packages for part definitions, attribute definitions, port definitions, item definitions, action definitions, requirements definitions, and other kinds of definition elements. The *VehicleConfigurations* package contains two design configurations that are modeled as usages of the definition elements from the *Definitions* package. Each vehicle configuration contains packages to contain its parts, actions, states, and requirements. This model also includes separate packages for *VehicleAnalysis* and *VehicleVerification*. Separate packages can be created for many other aspects of the model. Examples include a package to represent how the vehicle is used in a particular context, a package that represents the superset model with variation points from which each vehicle configuration can be derived, and a package to specify user-defined views.

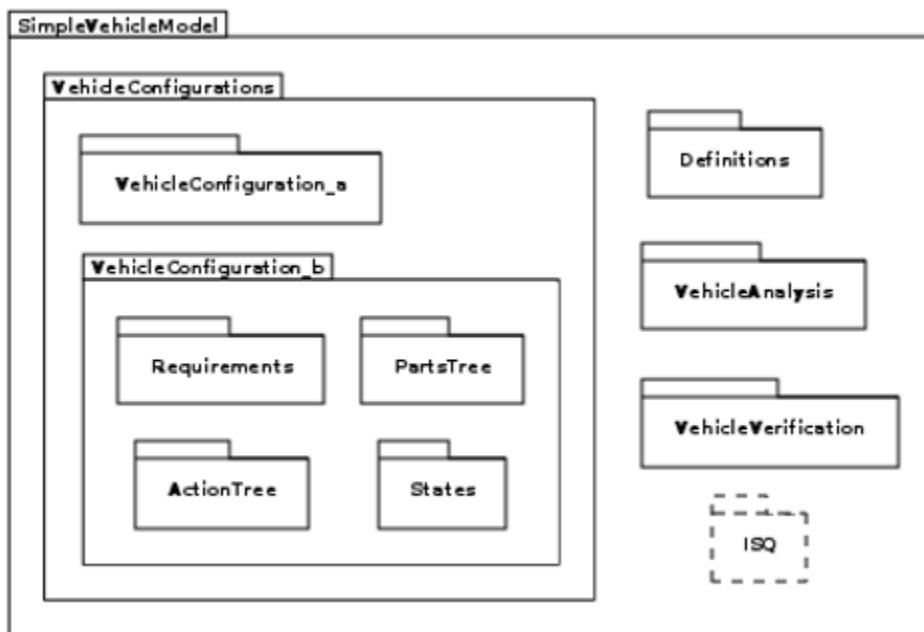


Figure 102. Model Organization for SimpleVehicleModel

```
package SimpleVehicleModel{
    import ISQ::.*;
    package Definitions { ... }
    package VehicleConfigurations{
        package VehicleConfiguration_a{ ... }
```

```

package VehicleConfiguration_b{
    package PartsTree{ ... }
    package ActionTree{ ... }
    package States{ ... }
    package Requirements{ ... }
}
}

package VehicleAnalysis{ ... }
package VehicleVerification{ ... }
}

```

The packages of a system model are often organized and managed to enable team members to work collaboratively on different aspects of the model, where each package contains cohesive content that can be worked on independently. Although not done for this example, the *VehicleConfigurations* package would typically import packages for each major system element (e.g., subsystem) to aid in collaborative development.

B.3 Definitions

The *Definitions* package contains a nested *PartDefinition* package that contains definitions for the parts that are used to represent the vehicle configurations. This includes the part definition for a *Vehicle*, whose features include attributes, ports, actions, and states.

«part def»
Vehicle
<i>attributes</i>
Tmax:> temperature acceleration:> acceleration brakePedalDepressed: Boolean cargoMass:> mass dryMass:> mass electricalPower:> power maintenanceTime: DateTime mass:> mass position:> length velocity:> speed
<i>ports</i>
ignitionCmdPort: IgnitionCmdPort pwrCmdPort: PwrCmdPort vehicleToRoadPort: VehicleToRoadPort
<i>actions</i>
perform providePower perform provideBraking perform controlDirection perform performSelfTest perform applyParkingBrake perform senseTemperature
<i>states</i>
exhibit providePower

Figure 103. Part Definition for Vehicle

```

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 ignitionCmdPort: IgnitionCmdPort;
port pwrCmdPort: PwrCmdPort;
port vehicleToRoadPort: VehicleToRoadPort;
perform action providePower;
perform action provideBraking;
perform action controlDirection;
perform action performSelfTest;
perform action applyParkingBrake;
perform action senseTemperature;
exhibit state vehicleStates;
}

```

The attributes called *mass*, *dryMass*, and *cargoMass* are each a kind of the base *mass* attribute imported from the standard SysML *ISQ* library model (see [9.5.4](#)). Values of the attribute quantities contained in this library can then be assigned standard units from the *SI* (see [9.5.6](#)) or *USCustomaryUnits* (see [9.5.7](#)) library models. For example, the value of the *mass* of the *Vehicle* can be assigned the unit of kilogram (*SI*::kg). The *Vehicle* also contains other quantity attributes such as its *position* and *velocity*.

The *Vehicle* contains three ports called *ignitionCmdPort*, *pwrCmdPort* and *vehicleToRoadPort*, which are interaction points that provide ignition and fuel commands to the vehicle, and transfer vehicle torque to the road. The *Vehicle* performs the action *providePower* to accelerate the vehicle, and other actions that include *performSelfTest* and *applyParkingBrake*. In addition, the *Vehicle* exhibits its *vehicleStates*.

The *Vehicle* represents a class of individual vehicles which is defined by its attributes, ports, actions, and states. Other part definitions can be specified in a similar way to build a reusable library of part definitions.

The part definition for *FuelTank* contains an attribute called *mass* and an item called *fuel*. Items often are used to represent something that flows through a system or is stored by a system. The fuel is not considered to be part of the *FuelTank*, so *fuel* is modeled as a referential feature (shown graphically using the white diamond symbol instead of the black diamond). The *fuel* contains an attribute called *fuelMass*. The *FuelTank* contains an attribute called *fuelMassMax*, which represents the maximum amount of fuel that a *FuelTank* can store.

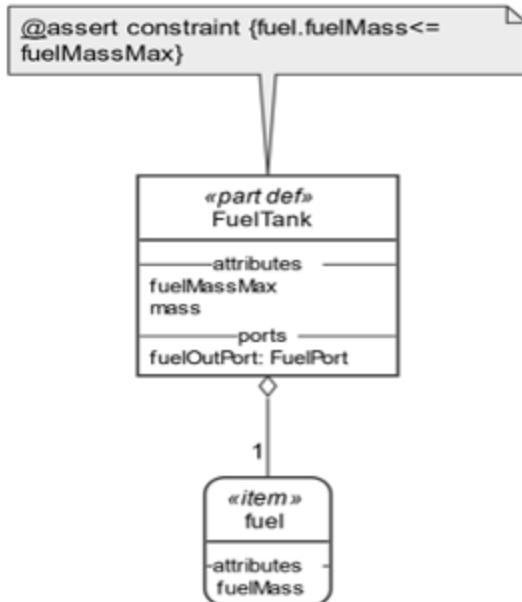


Figure 104. Part Definition for FuelTank Referencing Fuel it Stores

```

part def FuelTank{
    attribute mass :> ISQ::mass;
    ref item fuel:Fuel{
        attribute redefines fuelMass;
    }
    attribute fuelMassMax:>ISQ::mass;
    assert constraint {fuel.fuelMass<=fuelMassMax}
    port fuelOutPort:FuelPort;
}
  
```

A constraint is imposed that the `fuelMass` must be less than or equal to the `fuelMassMax`. The constraint is asserted to be true because, if the `fuelMass` exceeds the `fuelMassMax`, the model would be invalid and the model validation should generate an error. If `assert` is not used with the constraint, the model could evaluate the constraint to be false, and the model validation should not generate an error.

Although not included here, the fuel could also contain an attribute to represent the kind of fuel as gas or diesel. The attribute could be defined by an enumeration `FuelKind` with the literal values `gas` and `diesel`.

The part definition for `Axle` contains the attribute `mass`. `FrontAxle` is a specialization of `Axle` that inherits its `mass` attribute and contains an additional attribute called `steeringAngle`.

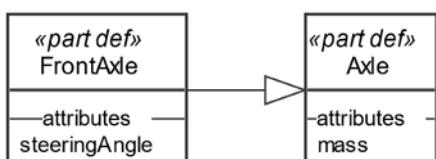


Figure 105. Axle and its Subclass FrontAxle

```

part def Axle{
    attribute mass:>ISQ::mass;
}
part def FrontAxle:>Axe{
  
```

```

        attribute steeringAngle:ScalarValues::Real;
}

```

The *Definitions* package also contains several other kinds of definition elements. The attribute definition *TorqueValue* is imported from the standard SysML *ISO* library model (see 9.5.4). The port definition *FuelCmdPort* contains an item called *fuelCmd* that can flow into the port. The *fuelCmd* is defined by the item definition *FuelCmd*. The item definition *Fuel* contains a *mass* attribute. The action definition *ProvidePower* contains an input item *fuelCmd*. It also contains an output attribute called *wheelToRoadTorque* that has multiplicity of 2, and is defined by the attribute definition *TorqueValue*. Although not shown in the diagram, the alias *Torque* is used in place of *TorqueValue*.

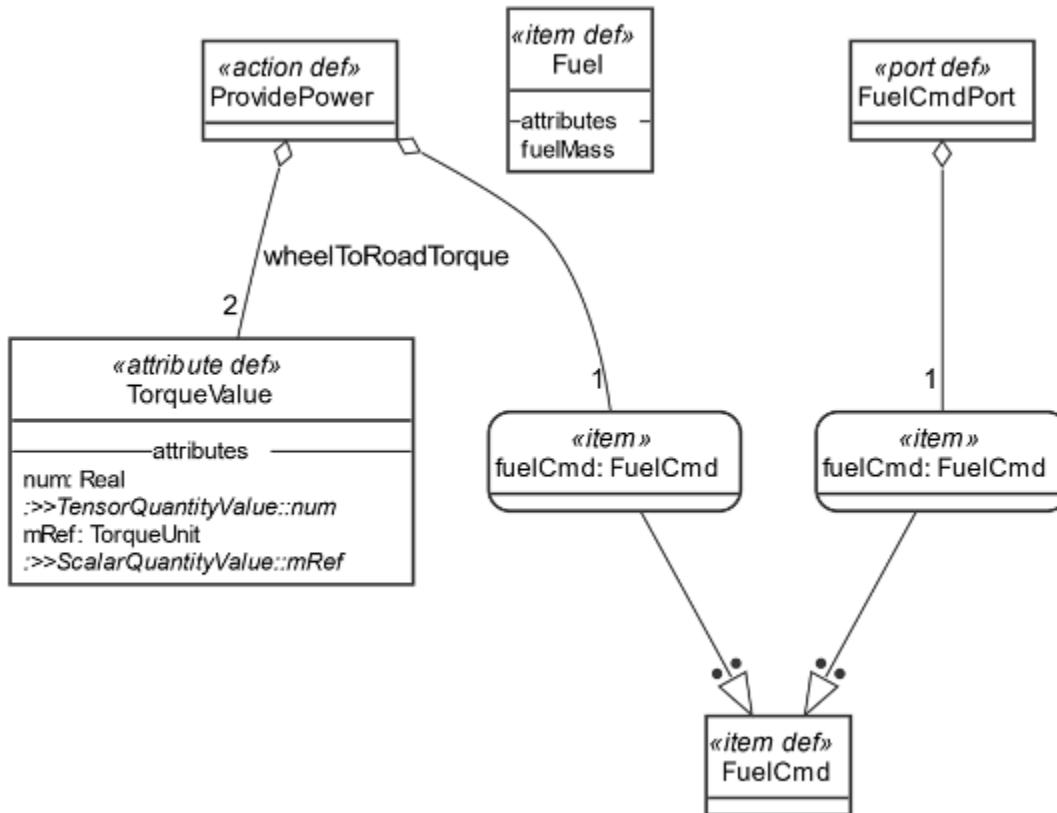


Figure 106. Example Definition Elements

```

alias Torque for ISO::TorqueValue;
port def FuelCmdPort{
    in item fuelCmd:FuelCmd;
}

item def FuelCmd;

item def Fuel{
    attribute fuelMass:>ISO::mass;
}

action def ProvidePower {
    in item fuelCmd:FuelCmd;
}

```

```

    out wheelToRoadTorque:Torque[2];
}

```

B.4 Parts

The *VehicleConfigurations* package contains two usages of the *Vehicle* part definition called *vehicle_a* and *vehicle_b*. The *vehicle_b* configuration is shown below. The part *vehicle_b* redefines the *mass* property inherited from its definition *Vehicle* by further constraining it to be the sum of its *dryMass*, *cargoMass*, and *fuelMass*.

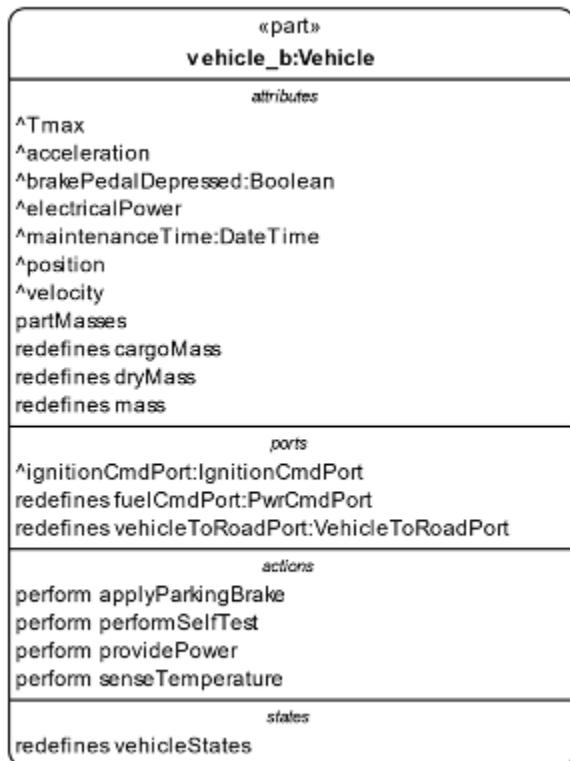


Figure 107. Part Usage for vehicle_b

```

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 default 0 [kg];
    attribute partMases=
        (fuelTank.mass, frontAxleAssembly.mass, rearAxleAssembly.mass,
         engine.mass, transmission.mass, driveshaft.mass);
    port fuelCmdPort redefines pwrCmdPort;
    port vehicleToRoadPort redefines vehicleToRoadPort{
        port wheelToRoadPort1;
        port wheelToRoadPort2;
    }
    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;
}
...
}

```

A *parts tree* is a representation of the decomposition of a part into its constituent parts. Different part usages with the same definition, such as *vehicle_a* and *vehicle_b*, can have different decompositions. As shown below, *vehicle_b* is composed of several parts, including an *engine*, *starterMotor*, *transmission*, *driveshaft*, *rearAxleAssembly*, *frontAxleAssembly*, *fuelTank*, and *vehicleSoftware*. The *rearAxleAssembly* contains a *rearAxle* and two *frontWheels* as designated by the multiplicity [2]. The *frontAxleAssembly* contains a *frontAxle*, *differential*, *rearWheel1*, and *rearWheel2*.

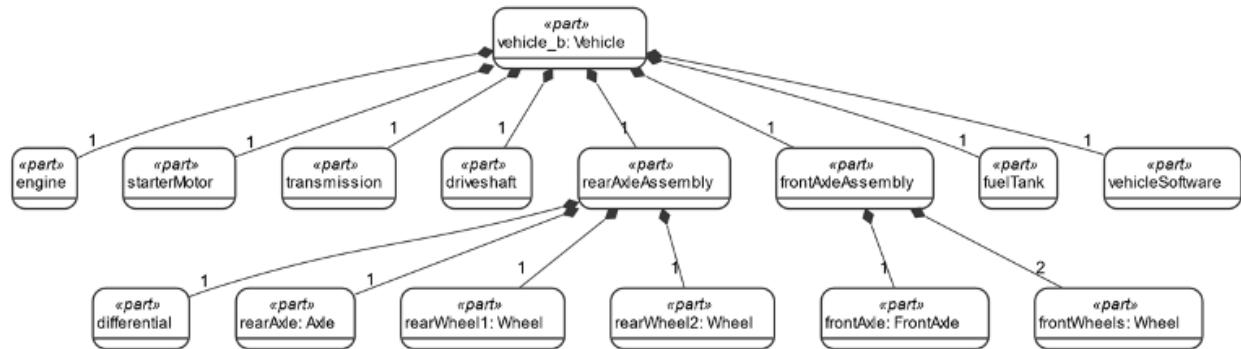


Figure 108. Parts Tree for vehicle_b

```

part vehicle_b:Vehicle {
    ...
    part engine;
    part starterMotor;
    part transmission;
    part driveshaft;
    part rearAxleAssembly{
        part differential;
        part rearAxle:Axle;
        part rearWheel1:Wheel;
        part rearWheel2:Wheel;
    }
    part frontAxleAssembly{
        part frontAxle:FrontAxe;
        part frontWheels:Wheel[2];
    }
    part fuelTank;
    part vehicleSoftware;
}

```

The *VehicleConfigurations* package also contains the *engine4Cyl* variant that subsets engine from the *vehicle_b* configuration. In general, an *engine* can contain 4 to 8 *cylinders*. The *engine4Cyl* variant constrains the number of *cylinders* to 4, and then provides separate features to represent each of its 4 cylinders, subsetting the full set. (See also the example of variability modeling in [B.12](#).)

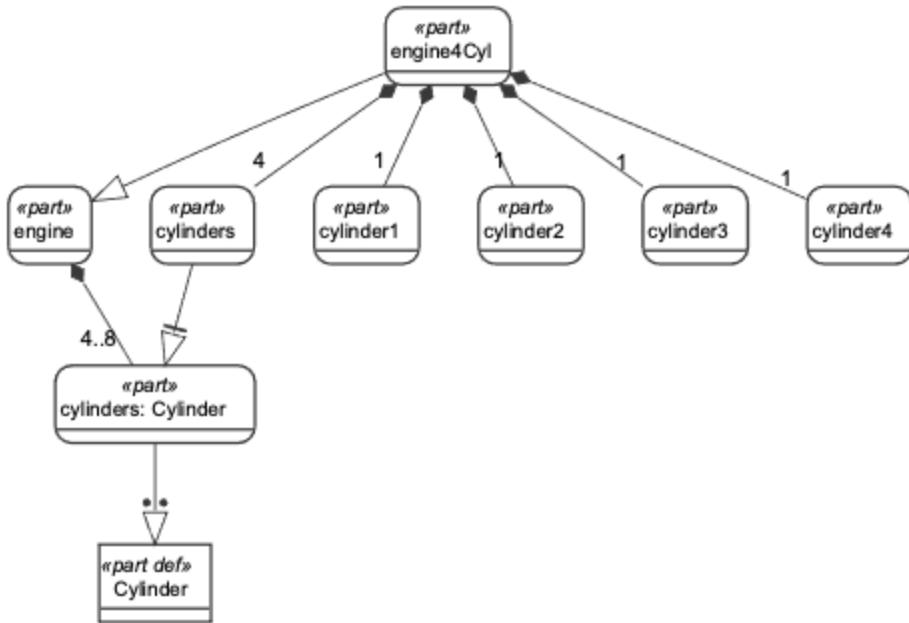


Figure 109. Variant engine4Cyl

```

part engine{
    part cylinders:Cylinder [4..8] ordered;
}

part engine4Cyl:>engine{
    part redefines cylinders[4];
    part cylinder1[1] subsets cylinders;
    part cylinder2[1] subsets cylinders;
    part cylinder3[1] subsets cylinders;
    part cylinder4[1] subsets cylinders;
}

```

B.5 Parts Interconnection

The various constituent parts of *vehicle_b* are interconnected via their ports. The *fuelCmdPort* on *vehicle_b* is delegated to the *fuelCmdPort* on the engine using a binding connection. The *controlPort* on the *vehicleController* is connected to the *engineControlPort* on the engine. The *controlPort* is defined by *ControlPort* and the *engineControlPort* is defined by a port definition that is the conjugate of the *ControlPort* (that is, the directions of all its directed features are reversed relative to those of the original port definition).

The *drivePwrPort* on the engine is connected to the *clutchPort* on the transmission by an interface. The interface is defined by an interface definition whose port at one end of the interface is defined as *DrivePwrPort* and whose port at the other end of the interface is defined as *ClutchPort*. The *DrivePwrPort* contains the directed feature *out engineTorque:Torque*. The *ClutchPort* is the conjugate of the *DrivePwrPort* implying it contains the directed feature *in engineTorque:Torque*.

Connections can be made directly between nested parts without having to establish a connection between the corresponding composite parts. For example, the port on the *driveShaft* can connect directly to a port on the *differential* without having to connect first to the *rearAxleAssembly* that composes the differential. Ports

can also be nested within a composite port as shown by the `vehicleToRoadPort`, which contains a nested port for each rear wheel.

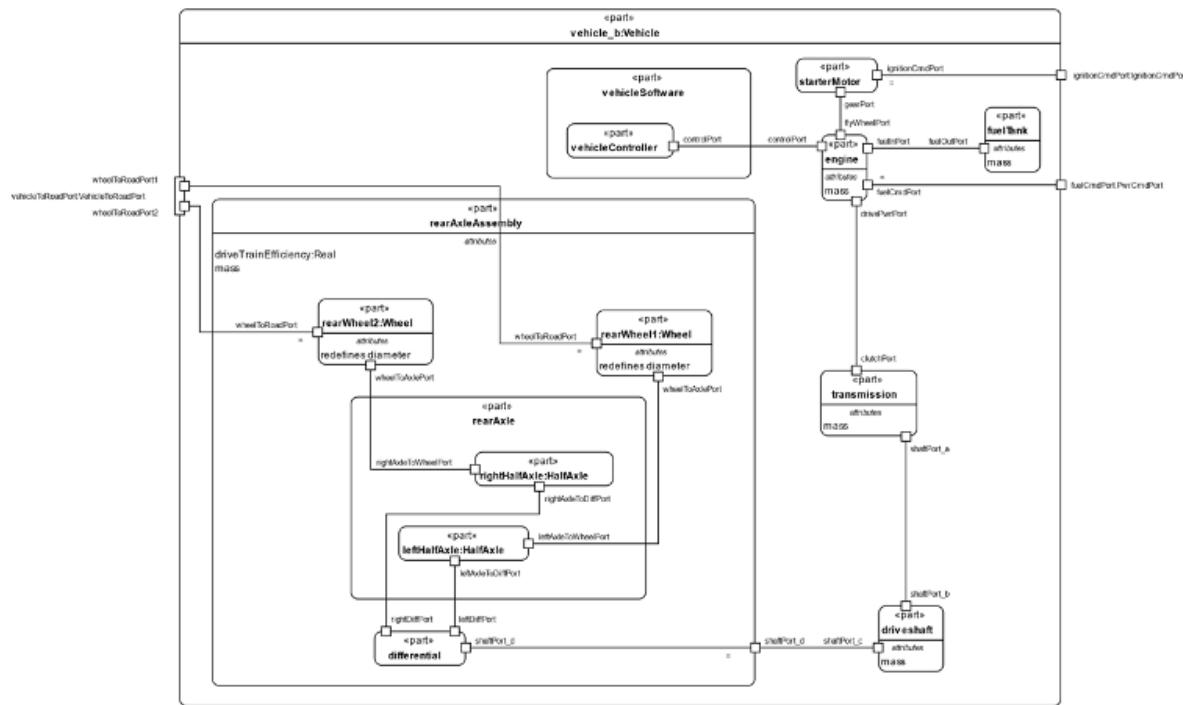


Figure 110. Parts Interconnection for vehicle_b

```

part vehicle_b : Vehicle{
    port fuelCmdPort redefines pwrCmdPort;
    port vehicleToRoadPort redefines vehicleToRoadPort{
        port wheelToRoadPort1;
        port wheelToRoadPort2;
    }
    part fuelTank{
        attribute mass :> ISQ::mass;
        item fuel{
            attribute fuelMass;
        }
        port fuelOutPort;
    }
    part rearAxleAssembly{
        attribute mass :> ISQ::mass;
        attribute driveTrainEfficiency:Real = 0.6;
        port shaftPort_d;
        part rearWheel1:Wheel{
            attribute redefines diameter;
            port :>>wheelToRoadPort;
            port :>>wheelToAxePort;
        }
        part rearWheel2:Wheel{
            attribute redefines diameter;
            port :>>wheelToRoadPort;
        }
    }
}
```

```

        port :>>wheelToAxlePort;
    }
part differential{
    port shaftPort_d;
    port leftDiffPort;
    port rightDiffPort;
}
part rearAxe{
    part leftHalfAxe:HalfAxe{
        port leftAxeToDiffPort;
        port leftAxeToWheelPort;
    }
    part rightHalfAxe:HalfAxe{
        port rightAxeToDiffPort;
        port rightAxeToWheelPort;
    }
}
bind shaftPort_d = differential.shaftPort_d;
connect differential.leftDiffPort
    to rearAxe.leftHalfAxe.leftAxeToDiffPort;
connect differential.rightDiffPort
    to rearAxe.rightHalfAxe.rightAxeToDiffPort;
connect rearAxe.leftHalfAxe.leftAxeToWheelPort
    to rearWheel1.wheelToAxlePort;
connect rearAxe.rightHalfAxe.rightAxeToWheelPort
    to rear-Wheel2.wheelToAxlePort;
}

part starterMotor{
    port ignitionCmdPort;
    port gearPort;
}
part engine{
    attribute mass;
    port fuelCmdPort;
    port drivePwrPort;
    port fuelInPort;
    port flyWheelPort;
    port controlPort;
}
part transmission{
    attribute mass;
    port clutchPort;
    port shaftPort_a;
}
part driveshaft{
    attribute mass;
    port shaftPort_b;
    port shaftPort_c;
}
part vehicleSoftware{
    part vehicleController {
        port controlPort;
}

```

```

        }

    }

//connections
bind engine.fuelCmdPort = fuelCmdPort;
bind starterMotor.ignitionCmdPort = ignitionCmdPort;

interface engineToTransmissionInterface:EngineToTransmissionInterface
    connect engine.drivePwrPort to transmission.clutchPort;

interface fuelInterface:FuelInterface
    connect fuelTank.fuelOutPort to engine.fuelInPort;

connect vehicleSoftware.vehicleController.controlPort
    to engine.controlPort;
connect starterMotor.gearPort
    to engine.flyWheelPort;
connect transmission.shaftPort_a
    to driveshaft.shaftPort_b;
connect driveshaft.shaftPort_c
    to rearAxleAssembly.shaftPort_d;
bind rearAxleAssembly.rearWheel1.wheelToRoadPort
    = vehicleToRoadPort.wheelToRoadPort1;
bind rearAxleAssembly.rearWheel2.wheelToRoadPort
    = vehicleToRoadPort.wheelToRoadPort2;
}

```

B.6 Actions

The definition and usage pattern applies not only to parts and part definitions, but to most constructs in SysML. As shown below, the action *providePower* is defined by the action definition *ProvidePower*. The action *providePower* contains actions to *generateTorque*, *amplifyTorque*, *transferTorque*, and *distributeTorque*, each of which have their own definitions.

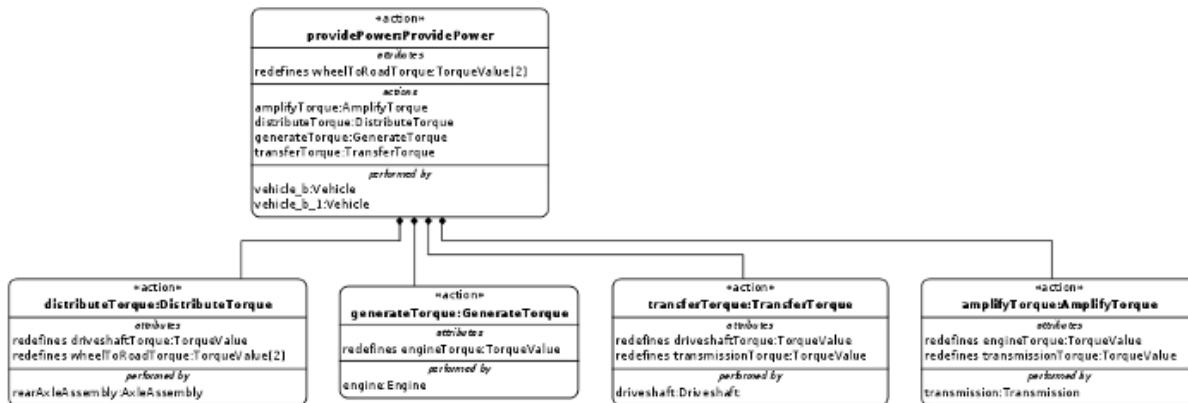


Figure 111. Action *providePower*

```

action providePower:ProvidePower{
    in fuelCmd:FuelCmd;
    out wheelToRoadTorque:Torque[2];
}

```

```

action generateTorque:GenerateTorque{
    in fuelCmd:FuelCmd;
    out engineTorque:Torque;
}
action amplifyTorque:AmplifyTorque{
    in engineTorque:Torque;
    out transmissionTorque:Torque;
}
action transferTorque:TransferTorque{
    in transmissionTorque:Torque;
    out driveshaftTorque:Torque;
}
action distributeTorque:DistributeTorque{
    in driveshaftTorque:Torque;
    out wheelToRoadTorque:Torque[2];
}
...
}

```

As shown in [Fig. 107](#), the part vehicle_b performs the action *providePower*. The subparts of vehicle_b then perform the appropriate subactions of *providePower*. For example, the part *engine* performs the action *generateTorque*.

The output of each of the subactions of *providePower* is connected by a streaming flow connection to the input of the next subaction, except for *distributeTorque*, whose outputs are bound to the outputs of *providePower*.

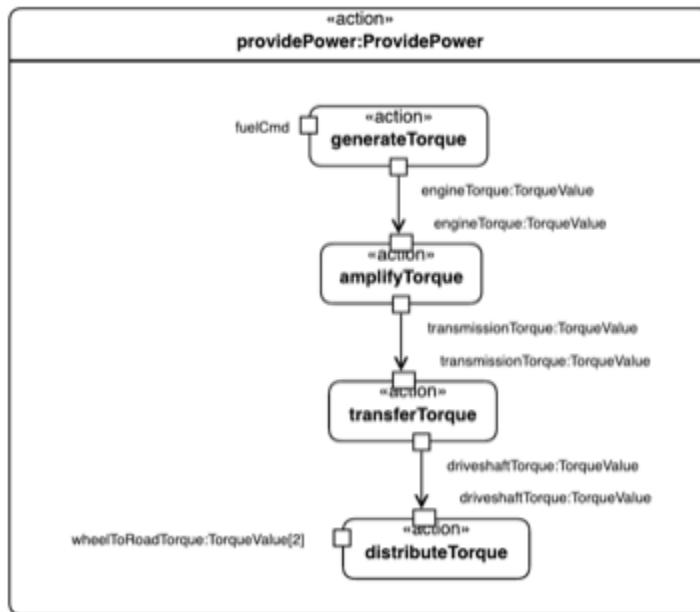


Figure 112. Action flow for providePower

```

action providePower:PowerProvider{
    ...
    bind fuelCmd = generateTorque.fuelCmd;
    stream generateTorque.engineTorque to amplifyTorque.engineTorque;
    stream amplifyTorque.transmissionTorque to transferTorque.transmissionTorque;
}

```

```

        stream transferTorque.driveshaftTorque to distributeTorque.driveshaftTorque;
        bind distributeTorque.wheelToRoadTorque = wheelToRoadTorque;
    }
}

```

The *transportPassenger* action models the use of a *Vehicle* to transport passengers. This action has subactions *passenger1GetInVehicle* and *driverGetInVehicle* that are performed concurrently after the start of the action. After both these actions complete, an accept action is triggered upon receipt of an *IgnitionCmd*. After this, the actions *driveVehicleToDestination* and *providePower* can proceed concurrently. Once these are both completed, then the actions *passenger1GetOutOfVehicle* and *driverGetOutOfVehicle* are preformed concurrently, after which the *transportPassenger* action is done.

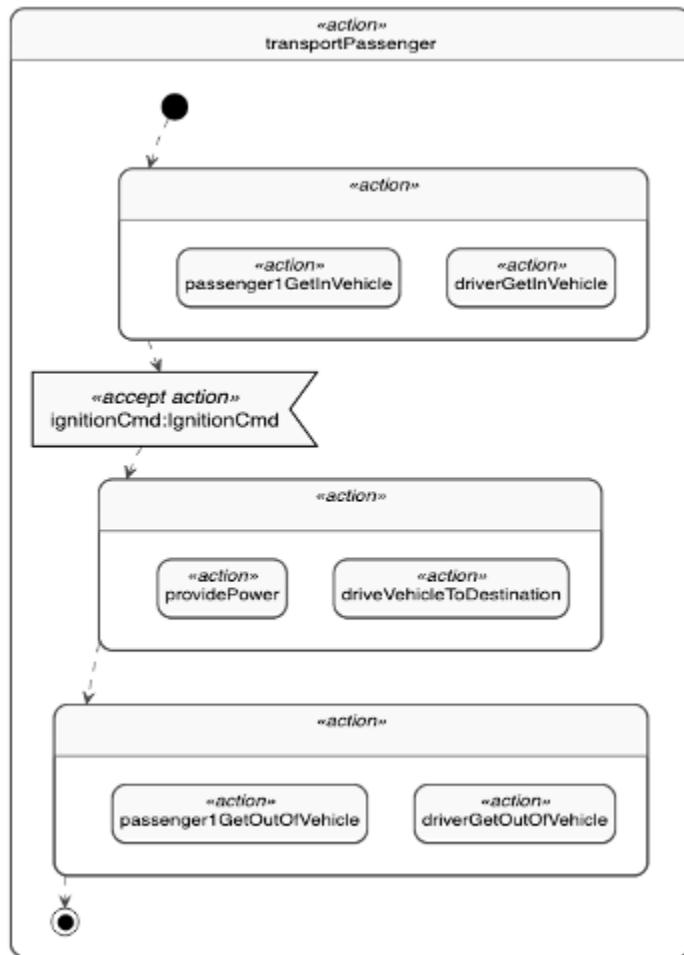


Figure 113. Action flow for transportPassenger

```

action transportPassenger{
    first start;
    then action {
        action driverGetInVehicle;
        action passenger1GetInVehicle;
    }
    then action ignitionTrigger accept ignitionCmd:IgnitionCmd;
    then action {
        action driveVehicleToDestination;
    }
}

```

```

        action providePower;
    }
then action {
    action driverGetOutOfVehicle;
    action passenger1GetOutOfVehicle;
}
then done;
}

```

B.7 States

The states of a *Vehicle* enable selected actions to be performed. The state *vehicleStates* is the top-level state in a state tree. The state *vehicleStates* is a parallel state, so its substates *operatingStates* and *healthStates* are performed concurrently. The states *operatingStates* and *healthStates* are not parallel, so only one of each of their substates can be active at any given time.

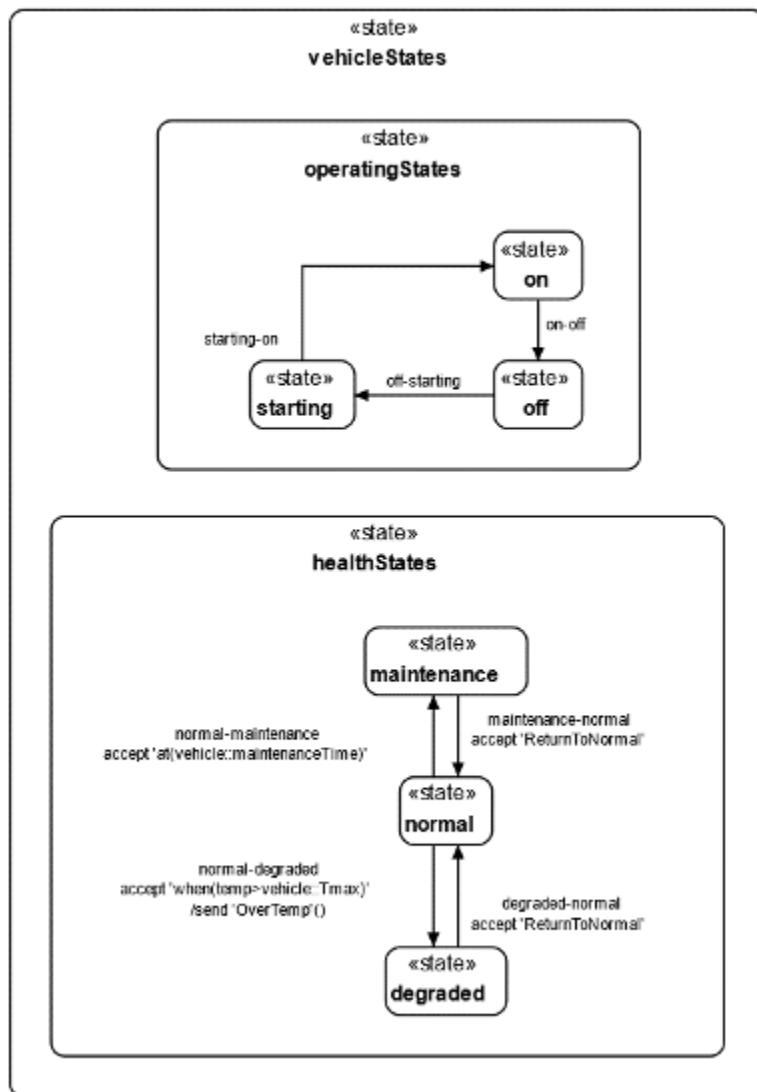


Figure 114. Vehicle States

```

state vehicleStates parallel {
    ref vehicle:Vehicle;
    ref controller;

    state operatingStates {
        ...
    }

    state healthStates {
        ...
    }
}

```

Note that the state `vehicleStates` has a referential feature `vehicle:Vehicle`. This allows the substates of `vehicleStates` to access the features of `Vehicle`, such as the attribute `brakePedalDepressed`. This pattern enables any usage of `Vehicle` to reuse `vehicleStates`. For example, the part `vehicle_b`, which is defined by `Vehicle`, exhibits `vehicleStates`, binding itself to the `vehicle` referenced by `vehicleStates`.

The `operatingStates` are further decomposed into `off`, `starting`, and `on` states, with an entry transition to the `off` substate. Upon receipt of an `ignitionCmd`, the `off-starting` transition fires if the `ignitionCmd` is in the `on` position and the `brakePedalDepressed` is true. A `StartSignal` is sent to the `controller` as part of this transition, after which `operatingStates` enters its `starting` substate. The state `operatingStates` also includes transitions from the substates `starting` to `on` and `on` to `off`.

The `ignitionCmd` is defined by the item definition `IgnitionCmd`, which contains an attribute defined by an enumeration with values `on` and `off`. This pattern is used to represent a variety of signals that may be sent by send actions and accepted by accept actions.

The `on` state has an entry action to `performSelfTest`, which is performed upon entry to the state. When the entry action completes, the do action to `providePower` starts, and it continues to be performed until the state is exited. Prior to exiting the state, the exit action to `applyParkingBrake` is performed. The state also has a constraint that the `electricalPower` must not exceed 500 watts.

```

state operatingStates {
    entry action initial;

    state off;
    state starting;
    state on {
        entry vehicle.performSelfTest;
        do vehicle.providePower;
        exit vehicle.applyParkingBrake;
        constraint {vehicle.electricalPower<=500 [W] }
    }

    transition initial then off;

    transition 'off-starting'
        first off
        accept ignitionCmd:IgnitionCmd via vehicle.ignitionCmdPort
            if ignitionCmd.ignitionOnOff==IgnitionOnOff::on and vehicle.brakePedalDepresse
        do send StartSignal() to controller

```

```

        then starting;

transition 'starting-on'
    first starting
    accept VehicleOnSignal
    then on;

transition 'on-off'
    first on
    accept VehicleOffSignal
    do send OffSignal() to controller
    then off;
}

}

```

The `healthStates` are decomposed into `normal`, `maintenance` and `degraded` states. Starting in the `normal` state, `healthStates` continually monitors the vehicle temperature and, when the temperature exceeds the allowed maximum, it transitions to the `degraded` state and notifies the `controller`. It also transitions from `normal` to `maintenance` whenever it is time for vehicle maintenance. In either case, it transitions back to the `normal` state on receipt of a `ReturnToNormal` signal.

```

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;
}

```

B.8 Requirements

The requirement definition *MassRequirement* has a shall statement that "The actual mass shall be less than the required mass". This statement is formalized using attributes for *massRequired* and *massActual* and the constraint expression $\{massActual \leq massRequired\}$.

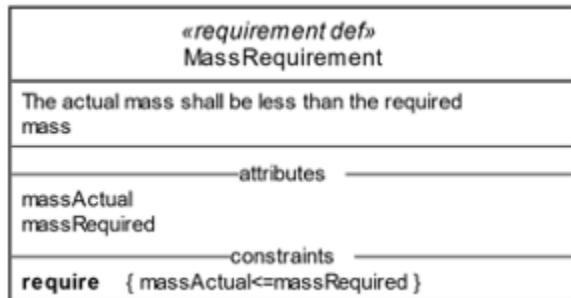


Figure 115. Requirement Definition MassRequirement

```
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}
}
```

The *vehicleSpecification* is a requirement group that contains other requirements. It has a dependency to *marketSurvey* that indicates its requirements are dependent on the market survey. The subject of the *vehicleSpecification* is *vehicle:Vehicle*, which enables the requirements contained in the specification to reference the features of *vehicle*. One of the requirements contained in the specification is the *vehicleMassRequirement*, which is defined by *MassRequirement*. The attribute *massRequired* for this usage is redefined to have a specific value of 2000 kg in the context of this *vehicleSpecification*. The attribute *massActual* is redefined to be the sum of the *dryMass* and *fuelMassActual* of the *vehicle*, where the *fuelMassActual* is assumed to be a full tank of gas that weighs 60 kg.

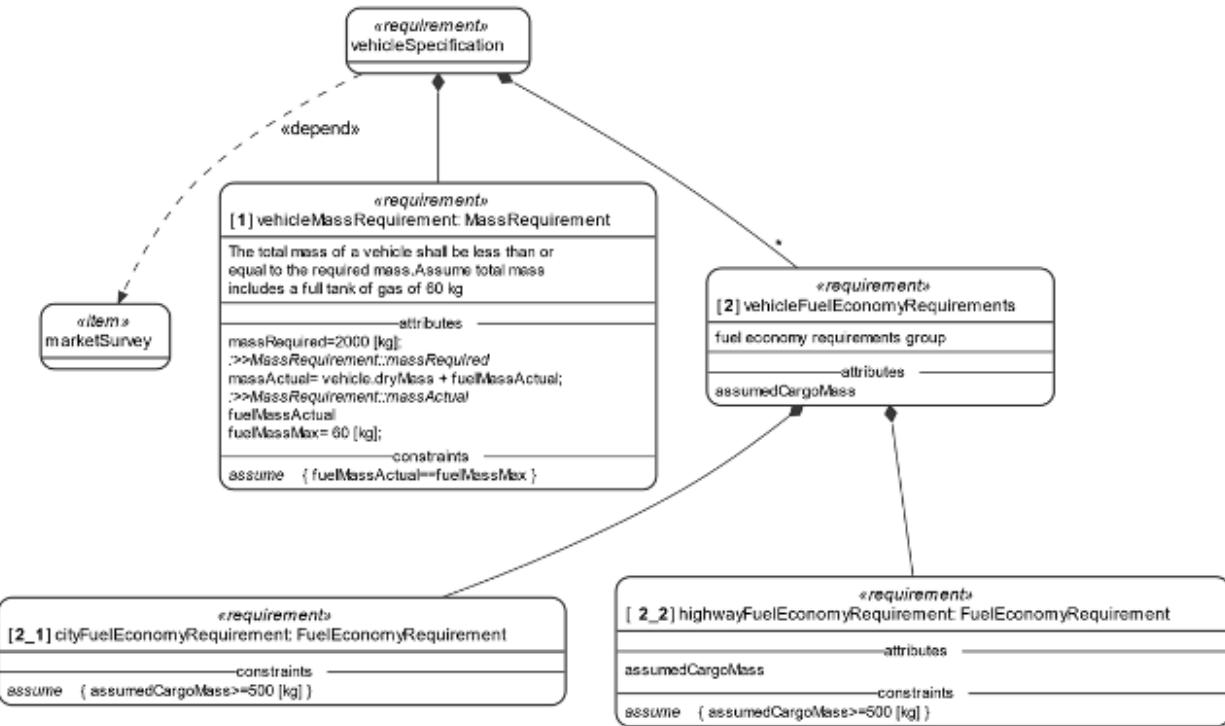


Figure 116. Requirements Group vehicleSpecification

In order to evaluate whether *vehicle_b* satisfies the *vehicleMassRequirement*, the *massActual* must be bound to the *mass* of *vehicle_b*. This is accomplished by asserting that *vehicle_b* satisfies the *vehicleSpecification* while binding the *actualMass* of the requirement to the *mass* of *vehicle_b*. Asserting *vehicle_b* satisfies the requirement is equivalent to imposing the mass constraint contained in the requirement on *vehicle_b*.

B.9 Analysis

The *FuelEconomyAnalysisModel* package contains an analysis case called *fuelEconomyAnalysis*. The objective for this analysis case is to determine whether the vehicle design configuration can satisfy the fuel economy requirements. Its subject is the part *vehicle_b*. The analysis case accepts a nominal driving scenario as an input, and returns a *calculatedFuelEconomy* in *KilometersPerLitres* as an output.

The analysis includes the following calculations to determine the result:

- *TraveledDistance* (*scenario*)
- *AverageTravelTimePerDistance* (*scenario*)
- *ComputeBSFC* (*vehicle_b.engine*)
- *BestFuelConsumptionPerDistance* (*vehicle_b.mass*, *bsfc*, *tpd_avg*, *distance*)
- *IdlingFuelConsumptionPerTime* (*vehicle_b.engine*)
- *FuelConsumption* (*f_a*, *f_b*, *tpd_avg*)

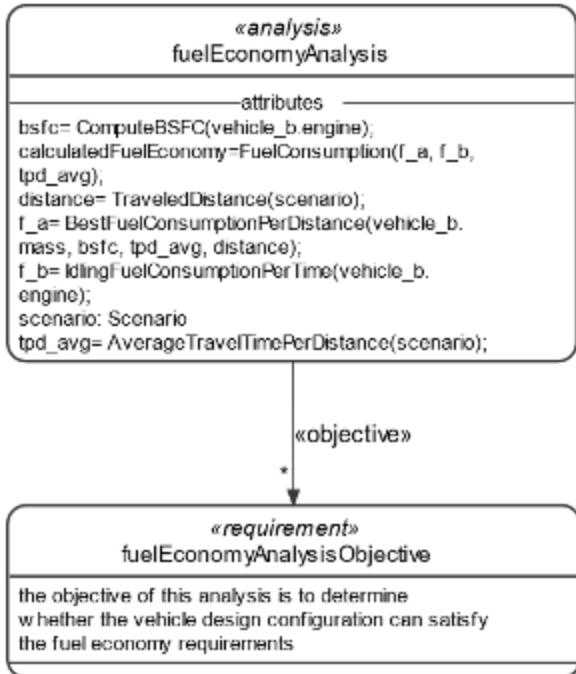


Figure 117. Analysis Case fuelEconomyAnalysis

```

analysis fuelEconomyAnalysis {
    in attribute scenario: Scenario;

    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 */
    }

    attribute distance = TraveledDistance(scenario);
    attribute tpd_avg = AverageTravelTimePerDistance(scenario);
    attribute bsfc = ComputeBSFC(vehicle_b.engine);
    attribute f_a =
        BestFuelConsumptionPerDistance(vehicle_b.mass, bsfc, tpd_avg, distance);
    attribute f_b = IdlingFuelConsumptionPerTime(vehicle_b.engine);
    return attribute calculatedFuelEconomy:>distancePerVolume =
        FuelConsumption(f_a, f_b, tpd_avg);
}

```

B.10 Verification

The simple verification case *massTests* is a usage of the verification case definition *MassTest*. The verification objective is to verify the *vehicleMassRequirement*. The subject of the verification case is *vehicle_b*. The verification case includes actions to *weighVehicle* and *evaluatePassFail*.

The *massVerificationSystem* performs the *massTests*. It is composed of an *operator* and a *scale*. The *scale* performs the action to *weighVehicle*, and the *operator* performs the action to

`evaluatePassFail`. The verification case returns a verdict of `pass` or `fail` based on whether the measured mass satisfies the mass requirement.

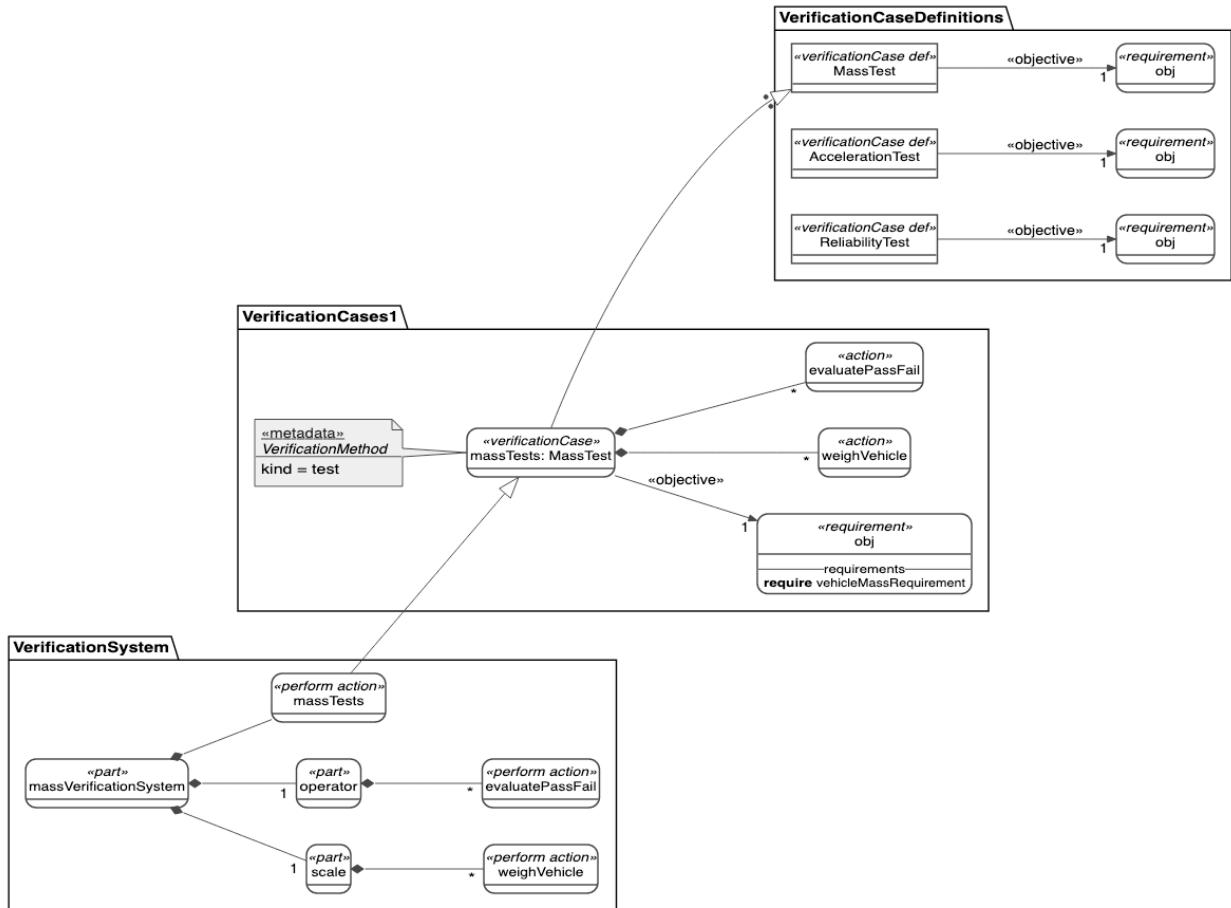


Figure 118. Vehicle Mass Verification Test

```

package VerificationCaseDefinitions{
    verification def MassTest;
    verification def AccelerationTest;
    verification def ReliabilityTest;
}
package VerificationCases1{
    verification massTests:MassTest {
        subject = vehicle_b;
        objective {
            verify vehicleSpecification.vehicleMassRequirement{
                redefines massActual=weighVehicle.massMeasured;
            }
        }
        metadata VerificationMethod{
            kind = test;
        }
        action weighVehicle {
            out massMeasured:>ISQ::mass;
        }
    }
}

```

```

        then action evaluatePassFail {
            in massMeasured:>ISQ::mass;
            out verdict = PassIf(
                vehicleSpecification::vehicleMassRequirement(vehicle_b)
            );
        }
        return :>> verdict = evaluatePassFail.verdict;
    }
}
package VerificationSystem{
    part massVerificationSystem{
        perform massTests;
        part scale{
            perform massTests.weighVehicle;
        }
        part operator{
            perform massTests.evaluatePassFail;
        }
    }
}

```

B.11 View and Viewpoint

The *SafetyEngineer* is a stakeholder with a concern for *VehicleSafety*. The *safetyViewpoint* frames this concern. The view *vehiclePartsTree_Safety* is a *PartsTreeView* that satisfies the *SafetyViewpoint*, and, therefore, addresses the *VehicleSafety* concern.

The view definition *TreeView* defines views that are rendered as tree diagrams. The view definition *PartsTreeView* specializes *TreeView* with a filter condition that only *PartUsages* should be included in the view. The view usage *vehiclePartsTree_Safety* adds the further condition to only include parts that have the metadata annotation for *Safety*. This view then exposes all the nested parts of *vehicle_b*, such that those parts meeting all the filter criteria are rendered in a tree diagram.

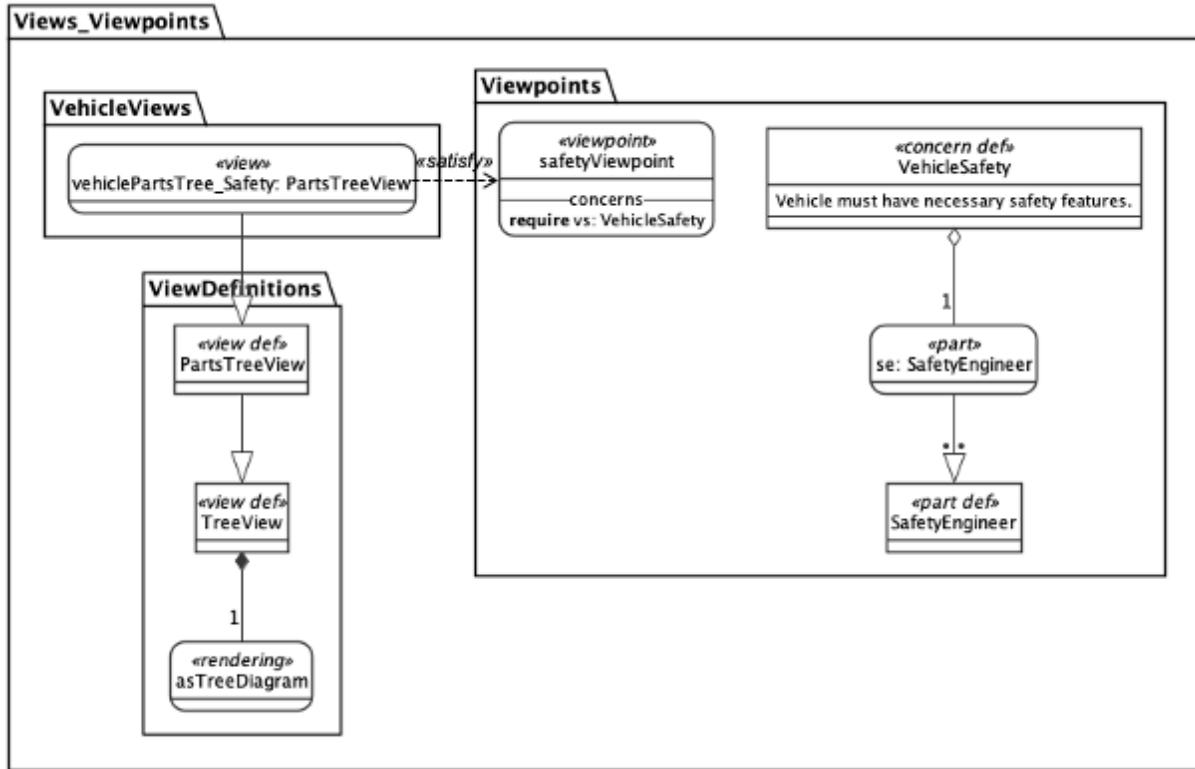


Figure 119. Vehicle Safety View

```

package Viewpoints{
    part def SafetyEngineer;
    concern def VehicleSafety {
        doc /* Vehicle must have necessary safety features. */
        stakeholder se:SafetyEngineer;
    }
    viewpoint safetyViewpoint{
        frame concern vs:VehicleSafety;
    }
}
package ViewDefinitions{
    view def TreeView {
        render asTreeDiagram;
    }
    view def PartsTreeView:>TreeView {
        filter @SysML::PartUsage;
    }
}
package VehicleViews{
    view vehiclePartsTree_Safety:PartsTreeView{
        satisfy safetyViewpoint;
        filter @Safety;
        expose vehicle_b:**;
    }
}

```

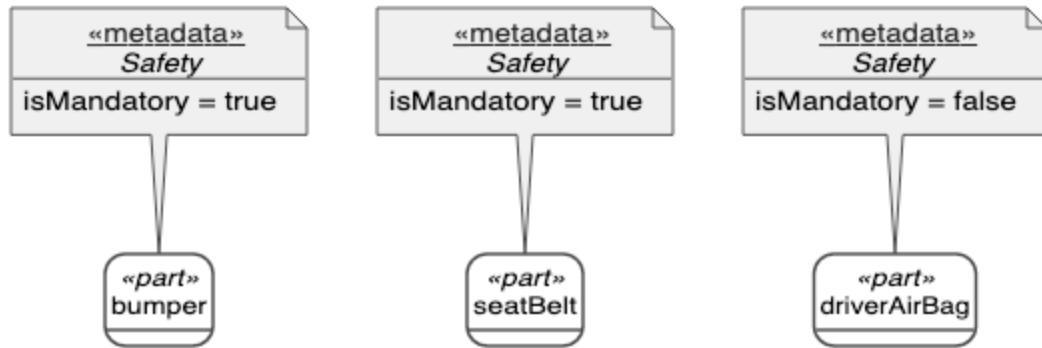


Figure 120. Rendering of view vehiclePartsTree_Safety

B.12 Variability

The part *vehicleFamily* models a family of *Vehicles* that allows variations in the subparts *engine*, *transmission* and *sunroof*. In particular, the part *engine* has two variants, *engine4Cyl* and *engine6Cyl*, which constrain *engine.cylinders* to have multiplicity 4 and 6, respectively. The part *cylinders* of *engine6Cyl* has an attribute *diameter* that is also a variation point, with two variants for *smallDiameter* and *largeDiameter*. There are also two choices for the *transmission* and a *sunroof* is optional. Note that the choice of a selected variant at one variation point can constrain the available choices at another variation point.

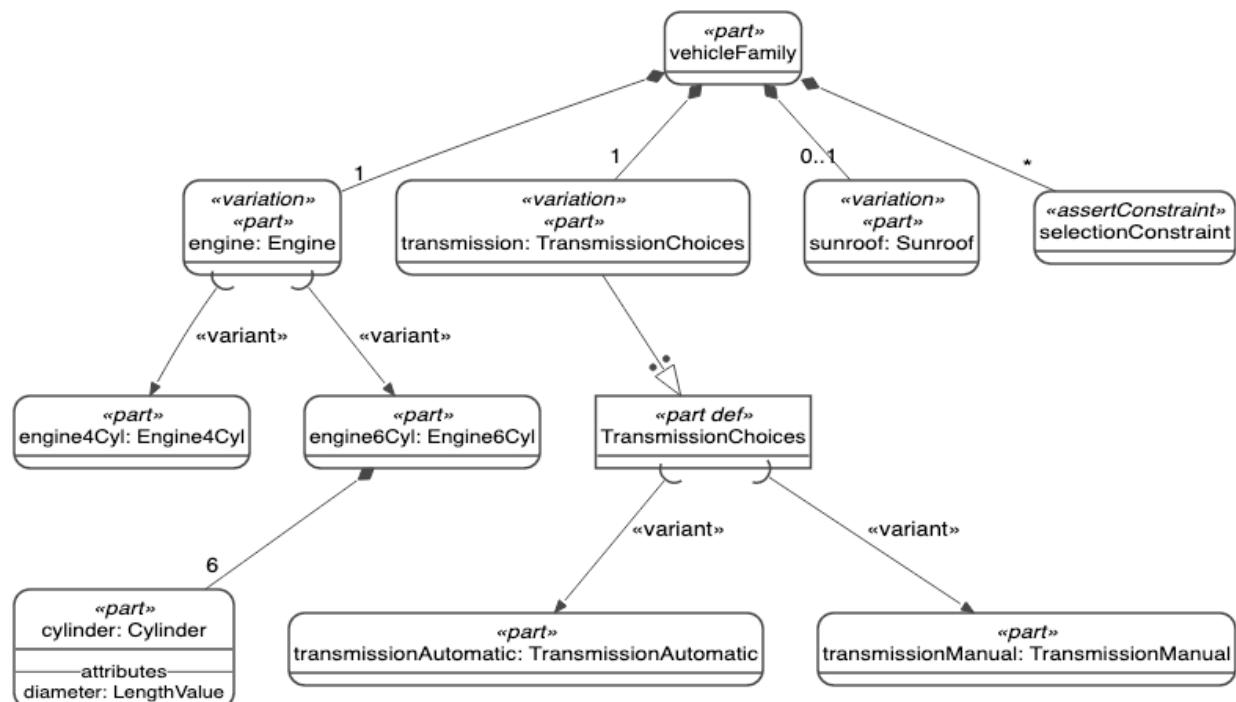


Figure 121. Variability Model for vehicleFamily

```

variation part def TransmissionChoices:>Transmission {
    variant part transmissionAutomatic:TransmissionAutomatic;
    variant part transmissionManual:TransmissionManual;
}
abstract part vehicleFamily:>vehicle_a{

```

```

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 part transmission:TransmissionChoices;
variation part sunroof:Sunroof;
assert constraint selectionConstraint {
    (engine==engine::engine4Cyl and
     transmission==TransmissionChoices::transmissionManual) xor
    (engine==engine::engine6Cyl and
     transmission==TransmissionChoices::transmissionAutomatic)
}
}

```

B.13 Individuals

The part definition *Vehicle* represents a class of individual vehicles with common characteristics. The parts *vehicle_a* and *vehicle_b* are usages of *Vehicle* with different part decompositions. There can be many individual vehicles that conform to *vehicle_a* or *vehicle_b*.

The individual part definition *Vehicle_1* is a specialization of *Vehicle* that restricts the part definition to a single individual. A usage *vehicle_1* of this definition represents that individual within a specific context. This usage can also subset *vehicle_b*, indicating that the individual identified by *Vehicle_1* conforms to *vehicle_b* in the context of the usage *vehicle_1*. The individual usage *vehicle_1* then inherits the parts hierarchy and other features of *vehicle_b*.

Additional individual definitions *FrontAxleAssembly_1*, *FrontAxle_1*, *Wheel_1*, *Wheel_2*, etc., are similarly specializations of their respective part definitions. The *vehicle1.frontAxleAssembly* can then be constrained to be a usage of *FrontAxleAssembly_1*, whose *frontAxle* is a usage of *FrontAxle_1*, whose *wheels* are *Wheel_1* and *Wheel_2*, and so forth. In this way, an entire hierarchy of individual parts can be provided for *vehicle_1*.

(Note that individual definition and usages can be created for any definition and usage element. An individual action for example, represents a particular performance of an action, with individual inputs and outputs.)

The part definition *VehicleRoadContext* defines a context containing *vehicle:Vehicle* and *road:Road* subparts. The individual definition *VehicleRoadContext_1* is a specialization of *VehicleRoadContext* whose subparts are constrained to be usages of the individual definitions *Vehicle_1* and *Road_1*.

As shown below, there is an individual usage of *VehicleRoadContext_1* that has a time slice *t0_t2_a* with three snapshots *t0_a*, *t1_a* and *t2_a*, at the times *t0*, *t1* and *t2*, respectively. Each of these context snapshots contains snapshots of *Vehicle_1* and *Road_1* at the respective times. Each of the vehicle and road snapshots are characterized by specific values for their attributes. In addition, the vehicle snapshot contains snapshots of its individual parts (consistent with the decomposition of *vehicle_1*).

An analysis may be used to compute the values of the attributes for each snapshot. The analysis results reflect the time history of the entities, which may be visualized using typical time-based plots and data representations.

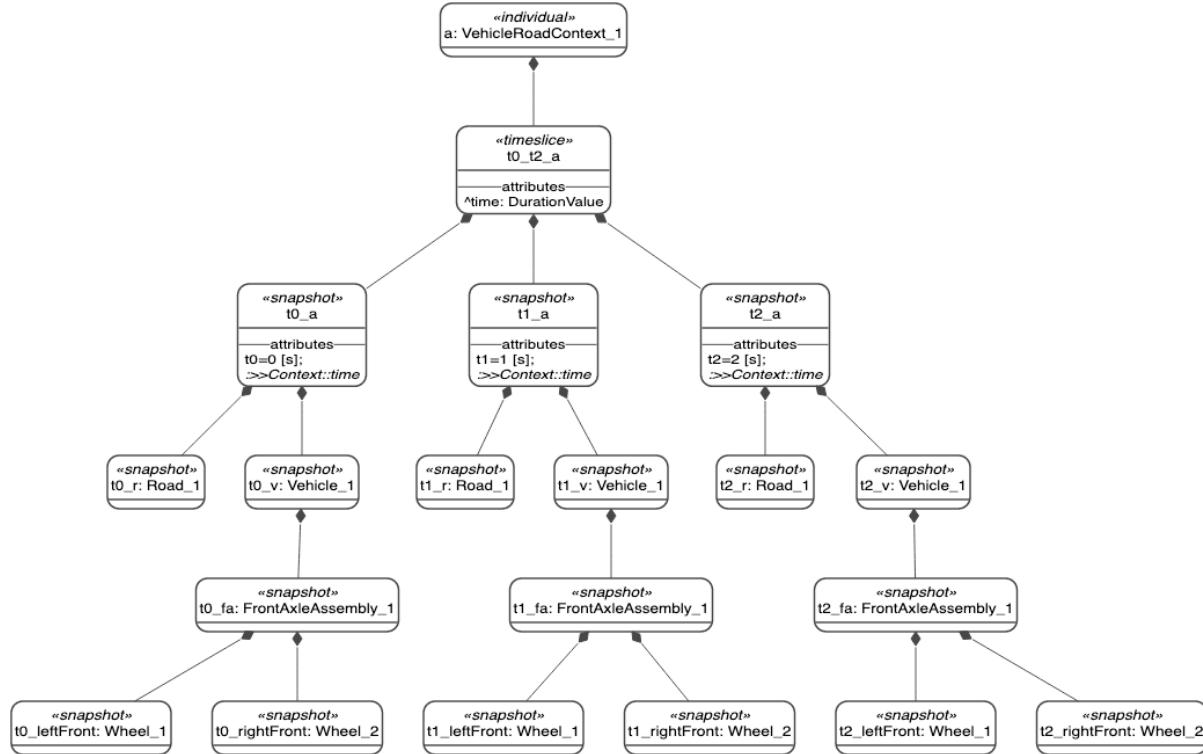


Figure 122. Vehicle Individuals and Snapshots

```

individual a:VehicleRoadContext_1{
    timeslice t0_t2_a{
        snapshot t0_a {
            attribute t0 redefines time=0 [s];
            snapshot t0_r:Road_1{
                :>>incline=0;
                :>>friction=.1;
            }
            snapshot t0_v:Vehicle_1{
                :>>position=0 [m];
                :>>velocity=0 [m];
                :>>acceleration=1.96 [m/s**2];
            }
            snapshot t0_fa:FrontAxleAssembly_1{
                snapshot t0_leftFront:Wheel_1;
                snapshot t0_rightFront:Wheel_2;
            }
        }
    }
    snapshot t1_a{
        attribute t1 redefines time=1 [s];
        snapshot t1_r:Road_1{
            :>>incline=0;
            :>>friction=.1;
        }
        snapshot t1_v:Vehicle_1{
    }
}
  
```

```

        :>>position=.98 [m];
        :>>velocity=1.96 [m/s];
        :>>acceleration=1.96 [m/s**2];
        snapshot t1_fa:FrontAisleAssembly_1{
            snapshot t1_leftFront:Wheel_1;
            snapshot t1_rightFront:Wheel_2;
        }
    }
}

snapshot t2_a{
    attribute t2 redefines time=2 [s];
    snapshot t2_r:Road_1{
        :>>incline =0;
        :>>friction=.1;
    }
    snapshot t2_v:Vehicle_1{
        :>>position=3.92 [m];
        :>>velocity=3.92 [m/s];
        :>>acceleration=1.96 [m/s**2];
        snapshot t2_fa:FrontAisleAssembly_1{
            snapshot t2_leftFront:Wheel_1;
            snapshot t2_rightFront:Wheel_2;
        }
    }
}
}
}

```

C Annex: SysML v1 to SysML v2 Transformation

C.1 General

C.1.1 Overview

This annex describes a transformation that specifies a semantic translation from SysML v1 [SysMLv1] to SysML v2 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 and 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 provide "SysMLv2-like" read-only access to its SysMLv1 models for external applications.

Submission Note: For this revised submission the transformation specification will cover a restricted scope only, which will be extended in the final submission. In this 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 and stereotypes that are part of the SysML v1 specification and the 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, the 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 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` static operation is provided for this purpose. It returns a (possibly null) value, based on the type of the target metaclass.

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 definition. In other words, assume a mapping is specified as the class "A" (i.e., that has A typing its "from" property), then it applies 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 its "from" property).

It is possible to restrict the applicability of a mapping specification to a specific subset of objects. This is achieved by the "filter" static operation that is evaluated against each candidate object. Only objects for which this "filter" operation returns "true" shall be translated according to the specifications of that mapping class. By default, the filter operation always returns "true".

Some mapping classes have one or more qualifiers for their "to" 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 above.

C.2 Mappings

C.2.1 Overview

C.2.2 Generic Mappings

C.2.2.1 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 can be 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.

C.2.2.2 Generic Mappings to KerML

C.2.2.2.1 Overview

Table 33. List of all Overview Mapping Specifications

Mapping Class	SysML v2 Concept
GenericToAnnotatingElement_Mapping	AnnotatingElement
GenericToAnnotation_Mapping	Annotation
GenericToAssociation_Mapping	Association
GenericToBehavior_Mapping	Behavior
GenericToClassifier_Mapping	Classifier
GenericToConjugation_Mapping	Conjugation
GenericToConnector_Mapping	Connector
GenericToElement_Mapping	Element
GenericToExpression_Mapping	Expression
GenericToFeature_Mapping	Feature
GenericToFeatureMembership_Mapping	FeatureMembership
GenericToFeatureTyping_Mapping	FeatureTyping
GenericToFeatureValue_Mapping	FeatureValue

Mapping Class	SysML v2 Concept
GenericToFunction_Mapping	Function
GenericToGeneralization_Mapping	Specialization
GenericToImport_Mapping	Import
GenericToMembership_Mapping	Membership
GenericToNamespace_Mapping	Namespace
GenericToPackage_Mapping	Package
GenericToParameterMembership_Mapping	ParameterMembership
GenericToPredicate_Mapping	Predicate
GenericToRelationship_Mapping	Relationship
GenericToReturnParameterMembership_Mapping	ReturnParameterMembership
GenericToStep_Mapping	Step
GenericToType_Mapping	Type

C.2.2.2.2 Mapping Specifications

C.2.2.2.2.1 GenericToAnnotatingElement_Mapping

Description

*** not specified yet ***

General Mappings

GenericToElement_Mapping

Mapping Target

AnnotatingElement

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 34. Table GenericToAnnotatingElement_Mapping Rules

Target Property	Target Value
AnnotatingElement::annotation	result = Set{}

C.2.2.2.2.2 GenericToAnnotation_Mapping

Description

*** not specified yet ***

General Mappings

GenericToRelationship_Mapping

Mapping Target

Annotation

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 35. Table GenericToAnnotation_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Annotation::annotatedElement	(abstract rule)
Annotation::annotatingElement	(abstract rule)
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}
Annotation::owningAnnotatedElement	result = null

C.2.2.2.2.3 GenericToAssociation_Mapping

Description

*** not specified yet ***

General Mappings

GenericToRelationship_Mapping
GenericToClassifier_Mapping

Mapping Target

Association

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 36. Table GenericToAssociation_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)

Target Property	Target Value
Type::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.2.2.4 GenericToBehavior_Mapping

Description

*** not specified yet ***

General Mappings

GenericToClassifier_Mapping

Mapping Target

Behavior

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 37. Table GenericToBehavior_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.2.5 GenericToClassifier_Mapping

Description

*** not specified yet ***

General Mappings

GenericToType_Mapping

Mapping Target

Classifier

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 38. Table GenericToClassifier_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.2.6 GenericToConjugation_Mapping

Description

*** not specified yet ***

General Mappings

GenericToRelationship_Mapping

Mapping Target

Conjugation

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 39. Table GenericToConjugation_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Conjugation::conjugatedType	(abstract rule)
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Conjugation::originalType	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.2.2.7 GenericToConnector_Mapping**Description**

*** not specified yet ***

General Mappings

GenericToFeature_Mapping
GenericToRelationship_Mapping

Mapping Target

Connector

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 40. Table GenericToConnector_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
Connector::isDirected	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}

Target Property	Target Value
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.2.2.8 GenericToElement_Mapping

Description

This is the general abstract class to be used as an ancestor for any class mapping specification.

General Mappings

No general mappings.

Mapping Target

Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 41. Table GenericToElement_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.2.2.9 GenericToExpression_Mapping

Description

*** not specified yet ***

General Mappings

GenericToStep_Mapping

Mapping Target

Expression

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 42. Table GenericToExpression_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.2.2.10 GenericToFeature_Mapping

Description

*** not specified yet ***

General Mappings

GenericToType_Mapping

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 43. Table GenericToFeature_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.2.2.2.11 GenericToFeatureMembership_Mapping**Description**

*** not specified yet ***

General Mappings

GenericToMembership_Mapping

Mapping Target

FeatureMembership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 44. Table GenericToFeatureMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}

Target Property	Target Value
FeatureMembership::direction	
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureMembership::isComposite	
FeatureMembership::isDerived	
FeatureMembership::isPort	
FeatureMembership::isPortion	
FeatureMembership::isReadOnly	
FeatureMembership::memberFeature	(abstract rule)
Element::ownedAnnotation	result = Set{}
FeatureMembership::ownedMemberFeature	result = null
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
FeatureMembership::owningType	(abstract rule)
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.2.2.12 GenericToFeatureTyping_Mapping

Description

*** not specified yet ***

General Mappings

GenericToGeneralization_Mapping

Mapping Target

FeatureTyping

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 45. Table GenericToFeatureTyping_Mapping 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)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}
FeatureTyping::type	(abstract rule)
FeatureTyping::typedFeature	(abstract rule)

C.2.2.2.2.13 GenericToFeatureValue_Mapping

Description

*** not specified yet ***

General Mappings

GenericToMembership_Mapping

Mapping Target

FeatureValue

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 46. Table GenericToFeatureValue_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
FeatureValue::featureWithValue	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureValue::isDefault	result = false
FeatureValue::isInitial	result = false
Element::ownedAnnotation	result = Set{}

Target Property	Target Value
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}
FeatureValue::value	(abstract rule)

C.2.2.2.14 GenericToFunction_Mapping

Description

*** not specified yet ***

General Mappings

GenericToBehavior_Mapping

Mapping Target

Function

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 47. Table GenericToFunction_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.2.15 GenericToGeneralization_Mapping

Description

*** not specified yet ***

General Mappings

GenericToRelationship_Mapping

Mapping Target

Specialization

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 48. Table GenericToGeneralization_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Specialization::general	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}
Specialization::specific	(abstract rule)

C.2.2.2.16 GenericToImport_Mapping

Description

*** not specified yet ***

General Mappings

GenericToRelationship_Mapping

Mapping Target

Import

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 49. Table GenericToImport_Mapping 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)
Import::importedMemberName	result = null
Import::importedNamespace	(abstract rule)
Import::importOwningPackage	(abstract rule)
Import::isImportAll	result = false
Import::isRecursive	result = false
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}
Import::visibility	result = KerML::VisibilityKind::public

C.2.2.2.2.17 GenericToMembership_Mapping

Description

*** not specified yet ***

General Mappings

GenericToRelationship_Mapping

Mapping Target

Membership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 50. Table GenericToMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Membership::memberElement	(abstract rule)
Membership::memberName	result = null
Membership::membershipOwningNamespace	(abstract rule)
Element::ownedAnnotation	result = Set{}

Target Property	Target Value
Membership::ownedMemberElement	result = null
Element::ownedRelationship	result = Set{}
Membership::visibility	result = KerML::VisibilityKind::public

C.2.2.2.2.18 GenericToNamespace_Mapping

Description

*** not specified yet ***

General Mappings

GenericToElement_Mapping

Mapping Target

Namespace

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 51. Table GenericToNamespace_Mapping Rules

Target Property	Target Value
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	

C.2.2.2.2.19 GenericToPackage_Mapping

Description

*** not specified yet ***

General Mappings

GenericToNamespace_Mapping

Mapping Target

Package

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 52. Table GenericToPackage_Mapping 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)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.2.2.2.20 GenericToParameterMembership_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeatureMembership_Mapping

Mapping Target

ParameterMembership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 53. Table GenericToParameterMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
ParameterMembership::direction	
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Membership::memberElement	(abstract rule)
Membership::memberName	result = null
ParameterMembership::memberParameter	(abstract rule)
Membership::membershipOwningNamespace	(abstract rule)
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = null
ParameterMembership::ownedMemberParameter	result = null
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}

Target Property	Target Value
Membership::visibility	result = KerML::VisibilityKind::public

C.2.2.2.2.21 GenericToPredicate_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFunction_Mapping

Mapping Target

Predicate

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 54. Table GenericToPredicate_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.2.2.22 GenericToRelationship_Mapping

Description

*** not specified yet ***

General Mappings

GenericToElement_Mapping

Mapping Target

Relationship

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 55. Table GenericToRelationship_Mapping Rules

Target Property	Target Value
Relationship::ownedRelatedElement	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.2.2.23 GenericToReturnParameterMembership_Mapping

Description

*** not specified yet ***

General Mappings

GenericToParameterMembership_Mapping

Mapping Target

ReturnParameterMembership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 56. Table GenericToReturnParameterMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
ReturnParameterMembership::direction	
FeatureMembership::direction	
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureMembership::isComposite	
FeatureMembership::isDerived	
FeatureMembership::isPort	
FeatureMembership::isPortion	

Target Property	Target Value
FeatureMembership::isReadOnly	
FeatureMembership::memberFeature	(abstract rule)
Membership::memberName	result = null
Element::ownedAnnotation	result = Set{}
FeatureMembership::ownedMemberFeature	result = null
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
FeatureMembership::owningType	(abstract rule)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.2.2.24 GenericToStep_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeature_Mapping

Mapping Target

Step

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 57. Table GenericToStep_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	

Target Property	Target Value
Element::ownedRelationship	result = Set{}

C.2.2.2.2.25 GenericToType_Mapping

Description

*** not specified yet ***

General Mappings

GenericToNamespace_Mapping

Mapping Target

Type

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 58. Table GenericToType_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{}
Element::ownedRelationship	result = Set{}

C.2.2.3 Generic Mappings to Systems

C.2.2.3.1 Overview

Table 59. List of all Overview Mapping Specifications

Mapping Class	SysML v2 Concept
GenericToConjugatedPortDefinition_Mapping	ConjugatedPortDefinition
GenericToConjugatedPortTyping_Mapping	ConjugatedPortTyping
GenericToConstraintDefinition_Mapping	ConstraintDefinition
GenericToDefinition_Mapping	Definition
GenericToItemDefinition_Mapping	ItemDefinition

Mapping Class	SysML v2 Concept
GenericToPortConjugation_Mapping	PortConjugation
GenericToPortDefinition_Mapping	PortDefinition
GenericToUsage_Mapping	Usage

C.2.2.3.2 Mapping Specifications

C.2.2.3.2.1 GenericToConjugatedPortDefinition_Mapping

Description

*** not specified yet ***

General Mappings

GenericToPortDefinition_Mapping

Mapping Target

ConjugatedPortDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 60. Table GenericToConjugatedPortDefinition_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
Definition::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
ConjugatedPortDefinition::ownedPortConjugator	(abstract rule)
Element::ownedRelationship	result = Set{}
Definition::variantMembership	result = Set{}

C.2.2.3.2.2 GenericToConjugatedPortTyping_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeatureTyping_Mapping

Mapping Target

ConjugatedPortTyping

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 61. Table GenericToConjugatedPortTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Specialization::general	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
ConjugatedPortTyping::portDefinition	(abstract rule)
Relationship::source	result = Set{}
Specialization::specific	(abstract rule)
Relationship::target	result = Set{}

C.2.2.3.2.3 GenericToConstraintDefinition_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFunction_Mapping

Mapping Target

ConstraintDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 62. Table GenericToConstraintDefinition_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.3.2.4 GenericToDefinition_Mapping

Description

*** not specified yet ***

General Mappings

GenericToClassifier_Mapping

Mapping Target

Definition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 63. Table GenericToDefinition_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
Definition::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}
Definition::variantMembership	result = Set{}

C.2.2.3.2.5 GenericToItemDefinition_Mapping

Description

*** not specified yet ***

General Mappings

GenericToDefinition_Mapping

Mapping Target

ItemDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 64. Table GenericToItemDefinition_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.3.2.6 GenericToPortConjugation_Mapping

Description

*** not specified yet ***

General Mappings

GenericToConjugation_Mapping

Mapping Target

PortConjugation

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 65. Table GenericToPortConjugation_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
PortConjugation::conjugatedType	(abstract rule)
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
PortConjugation::originalPortDefinition	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.2.3.2.7 GenericToPortDefinition_Mapping

Description

*** not specified yet ***

General Mappings

GenericToDefinition_Mapping

Mapping Target

PortDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 66. Table GenericToPortDefinition_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.2.3.2.8 GenericToUsage_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeature_Mapping

Mapping Target

Usage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 67. Table GenericToUsage_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
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}
Usage::variantMembership	result = Set{}

C.2.3 SysML v1.6

C.2.3.1 Overview

C.2.3.2 Activities

C.2.3.2.1 Overview

Table 68. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
Continuous	Feature	Continuous_Mapping
ControlOperator		*** not specified yet ***
Discrete	Feature	Discrete_Mapping
NoBuffer		*** not specified yet ***
Optional	Feature	Optional_Mapping
Overwrite		*** not specified yet ***
Probability		*** not specified yet ***
Rate	Feature	Rate_Mapping

C.2.3.2.2 Mapping Specifications

C.2.3.2.2.1 Continuous_Mapping

Description

A SysML::Continuous parameter is mapped to a SysMLv2::Feature.

General Mappings

Rate_Mapping

Mapping Source

Parameter

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::Activities::Rate'), result =  
thisModule.hasStereotypeApplied(from, 'SysML::Activities::Continuous')
```

Table 69. Table Continuous_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = from.isOrdered
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.3.2.2 Discrete_Mapping

Description

A SysML::Discrete parameter is mapped to a SysMLv2::Feature.

General Mappings

Rate_Mapping

Mapping Source

Parameter

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::Activities::Rate'), result =
thisModule.hasStereotypeApplied(from, 'SysML::Activities::Discrete')
```

Table 70. Table Discrete_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = from.isOrdered
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.3.2.2.3 Optional_Mapping

Description

A SysML::Optional parameter is mapped to a SysMLv2::Feature.

General Mappings

Parameter_Mapping

Mapping Source

Parameter

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = thisModule.hasStereotypeApplied(from, 'SysML::Activities::Optional')`

Table 71. Table Optional_Mapping Rules

Target Property	Target Value
Element::aliasId	<code>result = Set{}</code>
Feature::direction	<code>result = null</code>
Element::documentation	<code>result = Set{}</code>
Element::humanId	<code>result = null</code>
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	<code>result = false</code>
Feature::isComposite	<code>result = false</code>
Feature::isDerived	<code>result = false</code>
Feature::isEnd	<code>result = false</code>
Feature::isOrdered	<code>result = false</code>
Feature::isPortion	<code>result = false</code>
Feature::isReadOnly	<code>result = false</code>
Type::isSufficient	<code>result = false</code>
Feature::isUnique	<code>result = true</code>
Element::ownedAnnotation	<code>result = Set{}</code>
Type::ownedFeatureMembership	<code>result = Set{}</code>
Namespace::ownedImport	<code>result = Set{}</code>
Namespace::ownedMembership	

Target Property	Target Value
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.3.2.2.4 Rate_Mapping

Description

A SysML::Rate parameter is mapped to a SysMLv2::Feature.

General Mappings

Parameter_Mapping

Mapping Source

Parameter

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = thisModule.hasStereotypeApplied(from, 'SysML::Activities::Rate')`

Table 72. Table Rate_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true

Target Property	Target Value
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.3.3 Allocations

C.2.3.3.1 Overview

Table 73. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
Allocate	AllocationUsage	Allocate_Mapping
AllocateActivityPartition		*** not specified yet ***

C.2.3.3.2 Mapping Specifications

C.2.3.3.2.1 Allocate_Mapping

Description

A SysML::Allocate relationship is mapped to a SysMLv2::AllocationUsage.

General Mappings

GenericToUsage_Mapping
ElementMain_Mapping

Mapping Source

Dependency

Mapping Target

AllocationUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = thisModule.hasStereotypeApplied(from, 'SysML::Allocations::Allocate')`

Table 74. Table Allocate_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null

Target Property	Target Value
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.3.4 Blocks

C.2.3.4.1 Overview

Table 75. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
AdjunctProperty	Feature	AdjunctProperty_Mapping
BindingConnector	BindingConnector	BindingConnector_Mapping
Block	PartDefinition	Block_Mapping
BoundReference	Feature	BoundReference_Mapping
ClassifierBehaviorProperty	Feature	ClassifierBehaviorProperty_Mapping
ConnectorProperty	Feature	_ConnectorProperty_Mapping
DirectedRelationshipPropertyPath		*** not specified yet ***
DistributedProperty		*** not specified yet ***
ElementPropertyPath		*** not specified yet ***
EndPathMultiplicity	Feature	EndPathMultiplicity_Mapping

SysML v1 Concept	SysML v2 Concept	Mapping Class
NestedConnectorEnd		*** not specified yet ***
ParticipantProperty		*** not specified yet ***
PropertySpecificType		*** not specified yet ***
ValueType	AttributeDefinition	ValueType_Mapping

C.2.3.4.2 Mapping Specifications

C.2.3.4.2.1 AdjunctProperty_Mapping

Description

*** not specified yet ***

General Mappings

StructuralFeature_Mapping

Mapping Source

Property

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::AdjunctProperty')
```

Table 76. Table AdjunctProperty_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false

Target Property	Target Value
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.3.4.2.2 BindingConnector_Mapping

Description

*** not specified yet ***

General Mappings

Connector_Mapping

Mapping Source

Connector

Mapping Target

BindingConnector

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::BindingConnector')

Table 77. Table BindingConnector_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false

Target Property	Target Value
Connector::isDirected	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.3.4.2.3 Block_Mapping

Description

A SysML::Block is mapped to a SysMLv2::PartDefinition.

General Mappings

Class_Mapping
GenericToPartDefinition_Mapping

Mapping Source

BehavioredClassifier

Mapping Target

PartDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

$$\text{result} = \text{thisModule.hasStereotypeApplied(from, 'SysML::Blocks::Block')}$$

Table 78. Table Block_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Sequence(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = ((src.ownedElement-toFeatureMS) - Set{from.classifierBehavior}) in let relationships: Sequence(UML::Element) = toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e))) ->union(toFeatureMS->collect(e thisModule.resolve_PropertyMembership_Mapping((e)))) in if from.classifierBehavior.oclIsUndefined() then relationships else relationships->append(thisModule.resolve_ClassifierBehaviorMembership_Mapping(from.classifierBehavior)) endif

C.2.3.4.2.4 BoundReference_Mapping

Description

*** not specified yet ***

General Mappings

StructuralFeature_Mapping
ElementMain_Mapping

Mapping Source

Property

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::BoundReference')

Table 79. Table BoundReference_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.3.4.2.5 ClassifierBehaviorProperty_Mapping

Description

*** not specified yet ***

General Mappings

StructuralFeature_Mapping

Mapping Source

Property

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::ClassifierBehaviorProperty')
```

Table 80. Table ClassifierBehaviorProperty_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.3.4.2.6 ConnectorProperty_Mapping

Description

*** not specified yet ***

General Mappings

AdjunctProperty_Mapping

Mapping Source

Property

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::AdjunctProperty'), result =
thisModule.hasStereotypeApplied(from, 'SysML::Blocks::ConnectorProperty')
```

Table 81. Table ConnectorProperty_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Feature::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = from.isOrdered
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.3.4.2.7 EndPathMultiplicity_Mapping

Description

*** not specified yet ***

General Mappings

StructuralFeature_Mapping
ElementMain_Mapping

Mapping Source

Property

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::EndPathMultiplicity')`

Table 82. Table EndPathMultiplicity_Mapping Rules

Target Property	Target Value
Element::aliasId	<code>result = Set{}</code>
Feature::direction	<code>result = null</code>
Element::documentation	<code>result = Set{}</code>
Element::humanId	<code>result = null</code>
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	<code>result = false</code>
Feature::isComposite	<code>result = false</code>
Feature::isDerived	<code>result = false</code>
Feature::isEnd	<code>result = false</code>
Feature::isOrdered	<code>result = false</code>
Feature::isPortion	<code>result = false</code>
Feature::isReadOnly	<code>result = false</code>
Type::isSufficient	<code>result = false</code>
Feature::isUnique	<code>result = true</code>
Element::ownedAnnotation	<code>result = Set{}</code>
Type::ownedFeatureMembership	<code>result = Set{}</code>
Namespace::ownedImport	<code>result = Set{}</code>
Namespace::ownedMembership	
Feature::ownedMembership	<code>result = Set{}</code>
Element::ownedRelationship	<code>result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)</code>
Feature::owningFeatureMembership	<code>result = null</code>

C.2.3.4.2.8 Part_Mapping

Description

*** not specified yet ***

General Mappings

StructuralFeature_Mapping

Mapping Source

Property

Mapping Target

PartUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = let p : OclAny = src.oclAsType(UML::Property) in if not p.ocllsUndefined() then  
p.type.ocllsKindOf(UML::Class) and Helper.hasStereotypeApplied(p, 'SysML::Blocks::Block') else false  
endif
```

Table 83. Table Part_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}

Target Property	Target Value
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.3.4.2.9 ValueType_Mapping

Description

*** not specified yet ***

General Mappings

`DataType_Mapping`

Mapping Source

`DataType`

Mapping Target

`AttributeDefinition`

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::ValueType')`

Table 84. Table ValueType_Mapping Rules

Target Property	Target Value
<code>Element::aliasId</code>	<code>result = Set{}</code>
<code>Element::documentation</code>	<code>result = Set{}</code>
<code>Element::humanId</code>	<code>result = null</code>
<code>Element::identifier</code>	<i>(abstract rule)</i>
<code>Classifier::isAbstract</code>	<code>result = from.isAbstract</code>
<code>Type::isSufficient</code>	<code>result = false</code>
<code>Element::ownedAnnotation</code>	<code>result = Set{}</code>
<code>Type::ownedFeatureMembership</code>	<code>result = Set{}</code>
<code>Namespace::ownedImport</code>	<code>result = Set{}</code>
<code>Namespace::ownedMembership</code>	

Target Property	Target Value
Classifier::ownedRelationship	result = let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = from.ownedElement.excluding(toFeatureMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toFeatureMS->collect(e PropertyMembership_Mapping.getMapped(e)))

C.2.3.5 Model Elements

C.2.3.5.1 Overview

Table 85. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
Conform		*** not specified yet ***
ElementGroup		ElementGroup_Mapping
Expose		*** not specified yet ***
Problem		Problem_Mapping
Rationale		Rationale_Mapping
Stakeholder	PartDefinition	Stakeholder_Mapping
View		*** not specified yet ***
Viewpoint		*** not specified yet ***

C.2.3.5.2 Mapping Specifications

C.2.3.5.2.1 ElementGroup_Mapping

Description

*** not specified yet ***

General Mappings

GenericToPackage_Mapping

Mapping Source

Comment

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = thisModule.hasStereotypeApplied(from, 'SysML::ModelElements::ElementGroup')`

Table 86. Table ElementGroup_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.3.5.2.2 Problem_Mapping**Description**

*** not specified yet ***

General Mappings

Comment_Mapping

ElementMain_Mapping

Mapping Source

Comment

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = not Helper.hasStereotypeApplied(src, 'SysML::ModelElements::Problem'), result =
thisModule.hasStereotypeApplied(from, 'SysML::ModelElements::Problem')
```

Table 87. Table Problem_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
AnnotatingElement::annotation	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)

C.2.3.5.2.3 Rationale_Mapping

Description

*** not specified yet ***

General Mappings

Comment_Mapping

ElementMain_Mapping

Mapping Source

Comment

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = not Helper.hasStereotypeApplied(src, 'SysML::ModelElements::Problem'), result =  
thisModule.hasStereotypeApplied(from, 'SysML::ModelElements::Rationale')
```

Table 88. Table Rationale_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
AnnotatingElement::annotation	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)

C.2.3.5.2.4 Stakeholder_Mapping

Description

*** not specified yet ***

General Mappings

GenericToPartDefinition_Mapping

ElementMain_Mapping

Mapping Source

Classifier

Mapping Target

PartDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::ModelElements::Stakeholder')
```

Table 89. Table Stakeholder_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.3.5.2.5 StakeholderToConcernMapping

Description

*** not specified yet ***

General Mappings

GenericToUsage_Mapping

Mapping Source

Classifier

Mapping Target

ConcernUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::ModelElements::Stakeholder')
```

Table 90. Table StakeholderToConcernMapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}

Target Property	Target Value
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.3.6 PortsAndFlows

C.2.3.6.1 Overview

Table 91. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
AcceptChangeStructuralFeatureEventAction		*** not specified yet ***
AddFlowPropertyValueOnNestedPortAction		*** not specified yet ***
ChangeStructuralFeatureEvent		*** not specified yet ***
ConjInterfaceBlock		*** not specified yet ***
DirectedFeature		*** not specified yet ***
FlowProperty		*** not specified yet ***
FullPort		FullPort_Mapping
InterfaceBlock	PortDefinition	InterfaceBlock_Mapping
InvocationOnNestedPortAction		*** not specified yet ***

SysML v1 Concept	SysML v2 Concept	Mapping Class
ItemFlow		*** not specified yet ***
ProxyPort		ProxyPort_Mapping
TriggerOnNestedPort		*** not specified yet ***

C.2.3.6.2 Mapping Specifications

C.2.3.6.2.1 FullPort_Mapping

Description

*** not specified yet ***

General Mappings

Port_Mapping

Mapping Source

Port

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

```
result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::FullPort')
```

Table 92. Table FullPort_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Feature::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = from.isOrdered
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false

Target Property	Target Value
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.3.6.2.2 InterfaceBlock_Mapping

Description

*** not specified yet ***

General Mappings

Interface_Mapping
ElementMain_Mapping
Block_Mapping

Mapping Source

Class

Mapping Target

PortDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::Block'), result = thisModule.hasStereotypeApplied(from, 'SysML::Ports&Flows::InterfaceBlock')

Table 93. Table InterfaceBlock_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract

Target Property	Target Value
Type::isSufficient	result = false
Definition::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Sequence(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = ((src.ownedElement-toFeatureMS) - Set{from.classifierBehavior}) in let relationships: Sequence(UML::Element) = toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e))) ->union(toFeatureMS->collect(e thisModule.resolve_PropertyMembership_Mapping((e)))) in if from.classifierBehavior.oclIsUndefined() then relationships else relationships->append(thisModule.resolve_ClassifierBehaviorMembership_Mapping(from.classifierBehavior)) endif
Definition::variantMembership	result = Set{}

C.2.3.6.2.3 ProxyPort_Mapping

Description

*** not specified yet ***

General Mappings

Port_Mapping

Mapping Source

Port

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

result = thisModule.hasStereotypeApplied(from, 'SysML::Blocks::ProxyPort')

Table 94. Table ProxyPort_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null

Target Property	Target Value
Element::identifier	(abstract rule)
Feature::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = from.isOrdered
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.3.7 Requirements

C.2.3.7.1 Overview

Table 95. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class
AbstractRequirement		*** not specified yet ***
Copy		*** not specified yet ***
DeriveReqt		*** not specified yet ***
Refine		*** not specified yet ***
Requirement	RequirementDefinition	Requirement_Mapping
Satisfy		*** not specified yet ***
TestCase		*** not specified yet ***
Trace		*** not specified yet ***
Verify		*** not specified yet ***

C.2.3.7.2 Mapping Specifications

C.2.3.7.2.1 Requirement_Mapping

Description

A SysML::Requirement is mapped to a SysMLv2::RequirementDefinition.

General Mappings

GenericToConstraintDefinition_Mapping
ElementMain_Mapping

Mapping Source

Class

Mapping Target

RequirementDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = thisModule.hasStereotypeApplied(from, 'SysML::Requirements::Requirement')`

Table 96. Table Requirement_Mapping Rules

Target Property	Target Value
Element::aliasId	<code>result = Set{}</code>
Element::documentation	<code>result = Set{}</code>
Element::humanId	<code>result = null</code>
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	<code>result = false</code>
Type::isSufficient	<code>result = false</code>
Element::ownedAnnotation	<code>result = Set{}</code>
Type::ownedFeatureMembership	<code>result = Set{}</code>
Namespace::ownedImport	<code>result = Set{}</code>
Namespace::ownedMembership	
Element::ownedRelationship	<code>result = Set{}</code>

C.2.4 UML4SysML

C.2.4.1 Overview

C.2.4.2 Actions

C.2.4.2.1 Overview

Table 97. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
AcceptCallAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
AcceptEventAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
Action	ActionUsage	Action_Mapping ActionMembership_Mapping	
ActionInputPin		InputPin_Mapping	ActionInputPin.owner.oclIsTypeOf()
AddStructuralFeatureValueAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
AddVariableValueAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
BroadcastSignalAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
CallAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
CallBehaviorAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
CallOperationAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
Clause	Feature Relationship	ElementOwnership_Mapping	
ClearAssociationAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ClearStructuralFeatureAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ClearVariableAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ConditionalNode	PartDefinition ActionUsage	Block_Mapping Action_Mapping ActionMembership_Mapping	hasStereotypeApplied(ConditionalN 'SysML::Blocks::Block')
CreateLinkAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
CreateLinkObjectAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
CreateObjectAction	ActionUsage	Action_Mapping ActionMembership_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
DestroyLinkAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
DestroyObjectAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ExpansionRegion	PartDefinition ActionUsage	Block_Mapping Action_Mapping ActionMembership_Mapping	hasStereotypeApplied(ExpansionRegion, 'SysML::Blocks::Block')
InputPin		InputPin_Mapping	InputPin.owner.oclIsTypeOf(UML::Pin)
InvocationAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
LinkAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
LinkEndCreationData	Feature Relationship	ElementOwnership_Mapping	
LinkEndData	Feature Relationship	ElementOwnership_Mapping	
LinkEndDestructionData	Feature Relationship	ElementOwnership_Mapping	
LoopNode	PartDefinition ActionUsage	Block_Mapping Action_Mapping ActionMembership_Mapping	hasStereotypeApplied(LoopNode, 'SysML::Blocks::Block')
OpaqueAction		OpaqueAction_Mapping	
OutputPin		OutputPin_Mapping	OutputPin.owner.oclIsTypeOf(UML::Pin)
Pin		Pin_Mapping PinMembership_Mapping	Pin.owner.oclIsTypeOf(UML::Activator) null
RaiseExceptionAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReadExtentAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReadIsClassifiedObjectAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReadLinkAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReadLinkObjectEndAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReadSelfAction		ReadSelfAction_Mapping	
ReadStructuralFeatureAction		ReadStructuralFeatureAction_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
ReadVariableAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReclassifyObjectAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReduceAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
RemoveStructuralFeatureValueAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
RemoveVariableValueAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ReplyAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
SendObjectAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
SendSignalAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
SequenceNode	PartDefinition ActionUsage	Block_Mapping Action_Mapping ActionMembership_Mapping	hasStereotypeApplied(SequenceNod 'SysML::Blocks::Block')
StartClassifierBehaviorAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
StartObjectBehaviorAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
StructuralFeatureAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
StructuredActivityNode	PartDefinition ActionUsage	Block_Mapping Action_Mapping ActionMembership_Mapping	hasStereotypeApplied(StructuredAct 'SysML::Blocks::Block')
TestIdentityAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
UnmarshallAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
ValuePin		ValuePin_Mapping	ValuePin.owner.oclIsTypeOf(UML:
ValueSpecificationAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
VariableAction	ActionUsage	Action_Mapping ActionMembership_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
WriteLinkAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
WriteStructuralFeatureAction	ActionUsage	Action_Mapping ActionMembership_Mapping	
WriteVariableAction	ActionUsage	Action_Mapping ActionMembership_Mapping	

C.2.4.2.2 Mapping Specifications

C.2.4.2.2.1 Action_Mapping

Description

*** not specified yet ***

General Mappings

GenericToStep_Mapping
GenericToUsage_Mapping
ElementMain_Mapping

Mapping Source

Action

Mapping Target

ActionUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 98. Table Action_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false

Target Property	Target Value
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
ActionUsage::ownedRelationship	result = let toParameterMS: Set(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Pin)) in let toElementOMS: Set(UML::Element) = (src.ownedElement-toParameterMS) in toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e)))->union(toParameterMS->collect(e thisModule.resolve_PinMembership_Mapping((e))))
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.4.2.2 InputPin_Mapping

Description

*** not specified yet ***

General Mappings

Pin_Mapping
ElementMain_Mapping

Mapping Source

InputPin

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 $\text{result} = \text{from.owner.oclIsTypeOf(UML::Activity)}$

Table 99. Table InputPin_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(<i>abstract rule</i>)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = from.isOrdered
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.4.2.2.3 OpaqueAction_Mapping**Description**

*** not specified yet ***

General Mappings

Action_Mapping

Mapping Source

OpaqueAction

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 100. Table OpaqueAction_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Usage::variantMembership	result = Set{}

C.2.4.2.2.4 OutputPin_Mapping

Description

*** not specified yet ***

General Mappings

Pin_Mapping
ElementMain_Mapping

Mapping Source

OutputPin

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = from.owner.ocllsTypeOf(UML::Activity)`

Table 101. Table OutputPin_Mapping Rules

Target Property	Target Value
Element::aliasId	<code>result = Set{}</code>
Feature::direction	<code>result = null</code>
Element::documentation	<code>result = Set{}</code>
Element::humanId	<code>result = null</code>
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	<code>result = false</code>
Feature::isComposite	<code>result = false</code>
Feature::isDerived	<code>result = false</code>
Feature::isEnd	<code>result = false</code>
Feature::isOrdered	<code>result = from.isOrdered</code>
Feature::isPortion	<code>result = false</code>
Feature::isReadOnly	<code>result = false</code>
Type::isSufficient	<code>result = false</code>
Feature::isUnique	<code>result = from.isUnique</code>
Element::ownedAnnotation	<code>result = Set{}</code>
Type::ownedFeatureMembership	<code>result = Set{}</code>
Namespace::ownedImport	<code>result = Set{}</code>
Namespace::ownedMembership	
Feature::ownedMembership	<code>result = Set{}</code>
Feature::ownedRelationship	<code>result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.ocllsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif</code>
Feature::owningFeatureMembership	<code>result = null</code>

C.2.4.2.2.5 Pin_Mapping

Description

*** not specified yet ***

General Mappings

ActivityParameter_Mapping

Mapping Source

Pin

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = from.owner.ocllsTypeOf(UML::Activity)`

Table 102. Table Pin_Mapping Rules

Target Property	Target Value
Element::aliasId	<code>result = Set{}</code>
Feature::direction	<code>result = null</code>
Element::documentation	<code>result = Set{}</code>
Element::humanId	<code>result = null</code>
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	<code>result = false</code>
Feature::isComposite	<code>result = false</code>
Feature::isDerived	<code>result = false</code>
Feature::isEnd	<code>result = false</code>
Feature::isOrdered	<code>result = from.isOrdered</code>
Feature::isPortion	<code>result = false</code>
Feature::isReadOnly	<code>result = false</code>
Type::isSufficient	<code>result = false</code>
Feature::isUnique	<code>result = from.isUnique</code>
Element::ownedAnnotation	<code>result = Set{}</code>
Type::ownedFeatureMembership	<code>result = Set{}</code>
Namespace::ownedImport	<code>result = Set{}</code>
Namespace::ownedMembership	

Target Property	Target Value
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.4.2.2.6 ReadSelfAction_Mapping

Description

*** not specified yet ***

General Mappings

Action_Mapping

Mapping Source

ReadSelfAction

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 103. Table ReadSelfAction_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false

Target Property	Target Value
Type::isSufficient	result = false
Feature::isUnique	result = true
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Usage::variantMembership	result = Set{}

C.2.4.2.2.7 ReadStructuralFeatureAction_Mapping

Description

*** not specified yet ***

General Mappings

Action_Mapping

Mapping Source

ReadStructuralFeatureAction

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 104. Table ReadStructuralFeatureAction_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false

Target Property	Target Value
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Usage::variantMembership	result = Set{}

C.2.4.2.2.8 ValuePin_Mapping

Description

*** not specified yet ***

General Mappings

Pin_Mapping
ElementMain_Mapping

Mapping Source

ValuePin

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 $\text{result} = \text{from.owner.occlIsTypeOf(UML::Activity)}$

Table 105. Table ValuePin_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = from.isOrdered
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Feature::owningFeatureMembership	result = null

C.2.4.3 Activities

C.2.4.3.1 Overview

Table 106. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
Activity	ActionDefinition	Activity_Mapping	
ActivityEdge	Feature Relationship	ElementOwnership_Mapping	
ActivityFinalNode	ActionUsage	ActivityFinalNode_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
ActivityGroup	Feature Relationship	ElementOwnership_Mapping	
ActivityNode	Feature Relationship	ElementOwnership_Mapping	
ActivityParameterNode	Element	ActivityParameterNode_Mapping	
ActivityPartition	Feature Relationship	ElementOwnership_Mapping	
CentralBufferNode		TypedElementToFeatureTyping_Mapping	not CentralBufferNode.type.oclIsUndefined() or CentralBufferNode.oclIsKindOf(UML::ValueObject)
ControlFlow	Succession Subclassification	ControlFlow_Mapping	
ControlNode	ActionUsage	ControlNode_Mapping	
DataStoreNode		TypedElementToFeatureTyping_Mapping	not DataStoreNode.type.oclIsUndefined() or DataStoreNode.oclIsKindOf(UML::ValueObject)
DecisionNode	DecisionNode	DecisionNode_Mapping	
ExceptionHandler	Feature Relationship	ElementOwnership_Mapping	
ExecutableNode	Feature Relationship	ElementOwnership_Mapping	
FinalNode	ActionUsage	ControlNode_Mapping	
FlowFinalNode	Element	FlowFinalNode_Mapping	
ForkNode	ForkNode	ForkNode_Mapping	
InitialNode	FeatureMembership	ControlNode_Mapping	
InterruptibleActivityRegion	Feature Relationship	ElementOwnership_Mapping	
JoinNode	JoinNode	JoinNode_Mapping	
MergeNode	MergeNode	MergeNode_Mapping	
ObjectFlow	Feature Relationship	ElementOwnership_Mapping	
ObjectNode		TypedElementToFeatureTyping_Mapping	not ObjectNode.type.oclIsUndefined() or ObjectNode.oclIsKindOf(UML::ValueObject)
Variable	Feature MultiplicityRange	MultiplicityBound_Mapping MultiplicityBoundOwnership_Mapping MultiplicityMembership_Mapping UpperBoundValueOwnership_Mapping MultiplicityBoundTyping_Mapping MultiplicityElement_Mapping LowerBoundValueOwnership_Mapping	

C.2.4.3.2 Mapping Specifications

C.2.4.3.2.1 Activity_Mapping

Description

A UML4SysML::Activity is mapped to a SysMLv2::ActionDefinition.

General Mappings

Behavior_Mapping

Mapping Source

Activity

Mapping Target

ActionDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 107. Table Activity_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Sequence(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = ((src.ownedElement-toFeatureMS) - Set{from.classifierBehavior}) in let relationships: Sequence(UML::Element) = toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e))) ->union(toFeatureMS->collect(e thisModule.resolve_PropertyMembership_Mapping((e)))) in if from.classifierBehavior.oclIsUndefined() then relationships else relationships->append(thisModule.resolve_ClassifierBehaviorMembership_Mapping(from.classifierBehavior)) endif

Target Property	Target Value
ActionDefinition::ownedRelationship	<pre> result = let toParameterMS: Set(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Parameter)) in let ignoreParameterNodes: Set(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::ActivityParameterNode)) in let toFeatureMSproperties: Set(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toFeatureMSactions: Set(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Action)) in let toElementOMS: Set(UML::Element) = (((src.ownedElement-toFeatureMSproperties)-toFeatureMSactions)-toParameterMS)-ignoreParameterNodes) in toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e)))->union(toFeatureMSproperties->collect(e thisModule.resolve_PropertyMembership_Mapping((e))))->union(toFeatureMSactions->collect(e thisModule.resolve_ActionMembership_Mapping((e))))->union(toParameterMS->collect(e thisModule.resolve_ParameterMembership_Mapping((e)))) </pre>

C.2.4.3.2.2 ActivityFinalNode_Mapping

Description

*** not specified yet ***

General Mappings

ElementMain_Mapping
GenericToStep_Mapping

Mapping Source

ActivityFinalNode

Mapping Target

ActionUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 108. Table ActivityFinalNode_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false

Target Property	Target Value
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.4.3.2.3 ActivityParameterNode_Mapping

Description

A UML4SysML::ActivityParameterNode is mapped to nothing. Only the UML4SysML::Parameter element is mapped to SysML v2.

General Mappings

ElementMain_Mapping

Mapping Source

ActivityParameterNode

Mapping Target

Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 109. Table ActivityParameterNode_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null

Target Property	Target Value
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.4.3.2.4 ControlFlow_Mapping

Description

*** not specified yet ***

General Mappings

ElementMain_Mapping

GenericToConnector_Mapping

Mapping Source

ControlFlow

Mapping Target

Succession

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 110. Table ControlFlow_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false

Target Property	Target Value
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.4.3.2.5 ControlNode_Mapping

Description

*** not specified yet ***

General Mappings

GenericToUsage_Mapping
ElementMain_Mapping

Mapping Source

ControlNode

Mapping Target

ActionUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 111. Table ControlNode_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null

Target Property	Target Value
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.4.3.2.6 DecisionNode_Mapping

Description

*** not specified yet ***

General Mappings

ControlNode_Mapping

Mapping Source

DecisionNode

Mapping Target

DecisionNode

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 112. Table DecisionNode_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Usage::variantMembership	result = Set{}

C.2.4.3.2.7 FlowFinalNode_Mapping

Description

*** not specified yet ***

General Mappings

ElementMain_Mapping

Mapping Source

FlowFinalNode

Mapping Target

Element

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 113. Table FlowFinalNode_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.4.3.2.8 ForkNode_Mapping

Description

*** not specified yet ***

General Mappings

ControlNode_Mapping

Mapping Source

ForkNode

Mapping Target

ForkNode

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 114. Table ForkNode_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false

Target Property	Target Value
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Usage::variantMembership	result = Set{}

C.2.4.3.2.9 JoinNode_Mapping

Description

*** not specified yet ***

General Mappings

ControlNode_Mapping

Mapping Source

JoinNode

Mapping Target

JoinNode

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 115. Table JoinNode_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Usage::variantMembership	result = Set{}

C.2.4.3.2.10 MergeNode_Mapping

Description

*** not specified yet ***

General Mappings

ControlNode_Mapping

Mapping Source

MergeNode

Mapping Target

MergeNode

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 116. Table MergeNode Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Usage::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
Usage::variantMembership	result = Set{}

C.2.4.4 Classification

C.2.4.4.1 Overview

Table 117. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
BehavioralFeature	PartDefinition	Block_Mapping	hasStereotypeApplied(BehavioralFeature)
Classifier	PartDefinition ConcernUsage PartDefinition	Stakeholder_Mapping StakeholderToConcern_Mapping Block_Mapping	hasStereotypeApplied(Classifier, 'SysML::ModelElements::Stakeholder') hasStereotypeApplied(Classifier, 'SysML::ModelElements::Stakeholder') hasStereotypeApplied(Classifier, 'SysML::ModelElements::Concern')
Feature	Feature Relationship	ElementOwnership_Mapping	
Generalization	Subclassification	Generalization_Mapping	
GeneralizationSet	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
InstanceSpecification	Feature Feature FeatureTyping Feature	InstanceSpecification_Mapping InstanceSpecificationToFeatureTyping_Mapping	
InstanceValue		SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
Operation	Feature Relationship Feature Relationship PartDefinition	ElementOwnership_Mapping ElementOwnership_Mapping Block_Mapping	hasStereotypeApplied(Operation, 'SysML::Operations::Function')
Parameter	Feature	ParameterMembership_Mapping Parameter_Mapping	
ParameterSet	Feature Relationship	ElementOwnership_Mapping	
Property	PartUsage Feature Feature AttributeUsage Feature Feature	AttributeMembership_Mapping Part_Mapping PropertyMembership_Mapping AdjunctProperty_Mapping BoundReference_Mapping Attribute_Mapping ClassifierBehaviorProperty_Mapping EndPathMultiplicity_Mapping	let p : OclAny = src.oclAsType(UML::Property) p.oclIsUndefined() then p.type.oclIsKindOf(UML::Helper) Helper.hasStereotypeApplied(p, 'SysML::Blocks::BoundReference') endif hasStereotypeApplied(Property, 'SysML::Properties::BoundReference') hasStereotypeApplied(Property, 'SysML::Properties::EndPathMultiplicity') src.oclAsType(UML::Property).owner.oclIsKindOf(UML::Classifier) hasStereotypeApplied(Property, 'SysML::Blocks::ClassifierBehaviorProperty') hasStereotypeApplied(Property, 'SysML::Blocks::EndPathMultiplicity')

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
RedefinableElement	Feature Relationship	ElementOwnership_Mapping	
Slot	Feature	SlotMembership_Mapping Slot_Mapping SlotToFeatureTyping_Mapping	
StructuralFeature	Feature	StructuralFeatureMembership_Mapping	
Substitution		Realization_Mapping	

C.2.4.4.2 Mapping Specifications

C.2.4.4.2.1 Classifier_Mapping

Description

*** not specified yet ***

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 118. Table Classifier_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}

Target Property	Target Value
Namespace::ownedMembership	
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Classifier::ownedRelationship	result = let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = from.ownedElement.excluding(toFeatureMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toFeatureMS->collect(e PropertyMembership_Mapping.getMapped(e)))

C.2.4.4.2.2 LowerBoundTyping_Mapping

Description

*** not specified yet ***

General Mappings

MultiplicityBoundTyping_Mapping

Mapping Source

MultiplicityElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 119. Table LowerBoundTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Specialization::general	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Specialization::specific	(abstract rule)

Target Property	Target Value
Relationship::target	result = Set{}
FeatureTyping::type	(abstract rule)
typedFeature	result = self.lowerBound.to
FeatureTyping::typedFeature	(abstract rule)

C.2.4.4.2.3 MultiplicityBound_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeature_Mapping

Mapping Source

MultiplicityElement

Mapping Target

Feature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 120. Table MultiplicityBound_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedRelationship	result = Set{}

C.2.4.4.2.4 MultiplicityBoundOwnership_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeatureMembership_Mapping

Mapping Source

MultiplicityElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 121. Table MultiplicityBoundOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
isComposite	result = true
Membership::memberElement	(abstract rule)
memberFeature	(abstract rule)
Membership::memberName	result = null
Membership::membershipOwningNamespace	(abstract rule)
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = null
ownedMemberFeature	result = self.memberFeature()
ownedRelatedElement	result = Set{self.ownedMemberFeature()}
Element::ownedRelationship	result = Set{}
owningType	result = MultiplicityElement_Mapping.getMapped(from)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.4.4.2.5 MultiplicityBoundTyping_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeatureTyping_Mapping

Mapping Source

MultiplicityElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 122. Table MultiplicityBoundTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Specialization::general	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Specialization::specific	(abstract rule)
Relationship::target	result = Set{}
type	result = Helper.getScalarValueTypeByName('Integer')
typedFeature	(abstract rule)

C.2.4.4.2.6 MultiplicityElement_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeature_Mapping

Mapping Source

MultiplicityElement

Mapping Target

MultiplicityRange

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 123. Table MultiplicityElement_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
MultiplicityRange::isUnique	result = from.isUnique
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
MultiplicityRange::ownedRelationship	result = Set{lBoundOwnership.to, uBoundOwnership.to}

C.2.4.4.2.7 MultiplicityLowerBound_Mapping

Description

*** not specified yet ***

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 124. Table MultiplicityLowerBound_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(<i>abstract rule</i>)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let rels: Set(KerML::Relationship) = Set{self.lowerBoundTyping.to} in if from.lowerValue.oclIsUndefined() then rels else rels.including(LowerBoundValueOwnership_Mapping.getMapped(from)) endif
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.4.4.2.8 MultiplicityMembership_Mapping**Description**

*** not specified yet ***

General Mappings

GenericToFeatureMembership_Mapping

Mapping Source

MultiplicityElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 125. Table MultiplicityMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
isComposite	result = true
Membership::memberElement	(abstract rule)
memberFeature	result = self.multiplicityElement.to
Membership::memberName	result = null
Membership::membershipOwningNamespace	(abstract rule)
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = null
ownedMemberFeature	result = self.memberFeature()
ownedRelatedElement	result = Set{self.ownedMemberFeature()}
Element::ownedRelationship	result = Set{}
owningType	result = StructuralFeature_Mapping.getMapped(from)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.4.4.2.9 MultiplicityLowerBoundOwnership_Mapping

Description

*** not specified yet ***

General Mappings

MultiplicityBoundOwnership_Mapping

Mapping Source

MultiplicityElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 126. Table MultiplicityLowerBoundOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
FeatureMembership::direction	
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureMembership::isComposite	
FeatureMembership::isDerived	
FeatureMembership::isPort	
FeatureMembership::isPortion	
FeatureMembership::isReadOnly	
FeatureMembership::memberFeature	(abstract rule)
memberFeature	self.lowerBound.to
memberName	result = 'lowerBound'
Element::ownedAnnotation	result = Set{}
FeatureMembership::ownedMemberFeature	result = null
ownedMemberFeature	result = self.memberFeature()
Relationship::ownedRelatedElement	result = Set{}
ownedRelatedElement	result = Set{self.ownedMemberFeature()}
Element::ownedRelationship	result = Set{}
FeatureMembership::owningType	(abstract rule)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.4.4.2.10 MultiplicityUpperBound_Mapping

Description

*** not specified yet ***

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 127. Table MultiplicityUpperBound_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Feature::ownedRelationship	result = let rels: Set(KerML::Relationship) = Set{self.upperBoundTyping.to} in if from.upperValue.oclIsUndefined() then rels else rels.including(UpperBoundValueOwnership_Mapping.getMapped(from)) endif
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.4.4.2.11 MultiplicityUpperBoundOwnership_Mapping

Description

*** not specified yet ***

General Mappings

MultiplicityBoundOwnership_Mapping

Mapping Source

MultiplicityElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 128. Table MultiplicityUpperBoundOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
FeatureMembership::direction	
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
FeatureMembership::isComposite	
FeatureMembership::isDerived	
FeatureMembership::isPort	
FeatureMembership::isPortion	
FeatureMembership::isReadOnly	
FeatureMembership::memberFeature	(abstract rule)
memberFeature	result = self.upperBound.to
memberName	result = 'upperBound'
Element::ownedAnnotation	result = Set{}
FeatureMembership::ownedMemberFeature	result = null
ownedMemberFeature	result = self.memberFeature()
Relationship::ownedRelatedElement	result = Set{}
ownedRelatedElement	result = Set{self.ownedMemberFeature()}
Element::ownedRelationship	result = Set{}
FeatureMembership::owningType	(abstract rule)
Membership::visibility	result = KerML::VisibilityKind::public

C.2.4.4.2.12 StructuralFeature_Mapping

Description

*** not specified yet ***

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 129. Table StructuralFeature_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
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	
Feature::ownedRelationship	result = let typing: KerML::FeatureTyping = TypedElementToFeatureTyping_Mapping.getMapped(from) in if typing.oclIsUndefined() then Set{self.multiplicityMembership.to} else Set{self.multiplicityMembership.to, typing} endif
Element::ownedRelationship	result = Set{}

C.2.4.4.2.13 TypedElementToFeatureTyping_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeatureTyping_Mapping

Mapping Source

TypedElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

$\text{result} = \text{not from.type.oclIsUndefined()} \text{ and not from.oclIsKindOf(UML::ValueSpecification)}$

Table 130. Table TypedElementToFeatureTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	$\text{result} = \text{Set}\{\}$
Element::documentation	$\text{result} = \text{Set}\{\}$
Specialization::general	(abstract rule)
Element::humanId	$\text{result} = \text{null}$
Element::identifier	(abstract rule)
Element::ownedAnnotation	$\text{result} = \text{Set}\{\}$
Relationship::ownedRelatedElement	$\text{result} = \text{Set}\{\}$
Element::ownedRelationship	$\text{result} = \text{Set}\{\}$
Relationship::owningRelatedElement	$\text{result} = \text{null}$
Relationship::source	$\text{result} = \text{Set}\{\}$
Specialization::specific	(abstract rule)
Relationship::target	$\text{result} = \text{Set}\{\}$
type	$\text{result} = \text{if from.type.oclIsKindOf(UML::PrimitiveType) then Helper.getScalarValueType(from.type) else Classifier_Mapping.getMapped(from.type) endif}$
typedFeature	$\text{result} = \text{StructuralFeature_Mapping.getMapped(from)}$

C.2.4.4.2.14 UpperBoundTyping_Mapping

Description

*** not specified yet ***

General Mappings

MultiplicityBoundTyping_Mapping

Mapping Source

MultiplicityElement

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 131. Table UpperBoundTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Specialization::general	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Specialization::specific	(abstract rule)
Relationship::target	result = Set{}
type	result = Helper.getScalarValueTypeByName('UnlimitedNatural')
FeatureTyping::type	(abstract rule)
FeatureTyping::typedFeature	(abstract rule)
typedFeature	result = self.upperBound.to

C.2.4.5 CommonBehavior

C.2.4.5.1 Overview

Table 132. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
AnyReceiveEvent	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
Behavior	Behavior	Behavior_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
CallEvent	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
ChangeEvent	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
Event	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
FunctionBehavior	TextualRepresentation ActionDefinition	OpaqueBehavior_Mapping	
MessageEvent	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
OpaqueBehavior	TextualRepresentation ActionDefinition	OpaqueBehavior_Mapping	
SignalEvent	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
TimeEvent	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
Trigger	Feature Relationship	ElementOwnership_Mapping	

C.2.4.5.2 Mapping Specifications

C.2.4.5.2.1 Behavior

Description

*** not specified yet ***

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 133. Table Behavior Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Sequence(UML::Element) = src.ownedElement->select(e e.ocllsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = ((src.ownedElement-toFeatureMS) - Set{from.classifierBehavior}) in let relationships: Sequence(UML::Element) = toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e))) ->union(toFeatureMS->collect(e thisModule.resolve_PropertyMembership_Mapping((e)))) in if from.classifierBehavior.ocllsUndefined() then relationships else relationships->append(thisModule.resolve_ClassifierBehaviorMembership_Mapping(from.classifierBehavior)) endif
Behavior::ownedRelationship	result = let toParameterMS: Set(UML::Element) = from.ownedElement->select(e e.ocllsKindOf(UML::Parameter)) in let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.ocllsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = from.ownedElement.excluding(toFeatureMS).excluding(toParameterMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toFeatureMS->collect(e PropertyMembership_Mapping.getMapped(e))) ->union(toParameterMS->collect(e ParameterMembership_Mapping.getMapped(e)))

C.2.4.6 CommonStructure

C.2.4.6.1 Overview

Table 134. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
Abstraction		Abstraction_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
Comment	Comment Annotation	Comment_Mapping CommentToAnnotation_Mapping ElementGroup_Mapping	not Helper.hasStereotypeApplied(src, 'SysML::ModelElements::Problem') hasStereotypeApplied(Comment, 'SysML::ModelElements::ElementGroup')
Constraint		Constraint_Mapping	
Dependency	Dependency AllocationUsage	Dependency_Mapping Allocate_Mapping	hasStereotypeApplied(Dependency, 'SysML::Allocations::Allocate')
DirectedRelationship	Relationship	DirectedRelationship_Mapping	
Element	Feature Relationship	ElementOwnership_Mapping	
ElementImport		ElementImport_Mapping	
MultiplicityElement	Feature MultiplicityRange	MultiplicityBound_Mapping MultiplicityBoundOwnership_Mapping MultiplicityMembership_Mapping UpperBoundValueOwnership_Mapping MultiplicityBoundTyping_Mapping MultiplicityElement_Mapping LowerBoundValueOwnership_Mapping	
NamedElement	Feature Relationship	ElementOwnership_Mapping	
Namespace	PartDefinition	Block_Mapping	hasStereotypeApplied(Namespace, 'SysML::Blocks::Block')
PackageableElement	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
PackageImport	Import	PackageImport_Mapping	
ParameterableElement	Feature Relationship	ElementOwnership_Mapping	
Realization		Realization_Mapping	
Relationship	Relationship	Relationship_Mapping	
Type	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
TypedElement		TypedElementToFeatureTyping_Mapping	not TypedElement.type.oclIsUndefined() and TypedElement.oclIsKindOf(UML::ValueSpec)
Usage		Usage_Mapping	

C.2.4.6.2 Mapping Specifications

C.2.4.6.2.1 Abstraction Mapping

Description

There is no way to represent the "mapping" property on the target metaclass

General Mappings

Dependency_Mapping

Mapping Source

Abstraction

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 135. Table Abstraction Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
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)
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.2.4.6.2.2 Comment_Mapping

Description

test

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:

```
result = not Helper.hasStereotypeApplied(src, 'SysML::ModelElements::Problem')
```

Table 136. 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.2.4.6.2.3 CommentToAnnotation_Mapping

Description

*** not specified yet ***

General Mappings

GenericToAnnotation_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 137. 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	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Annotation::owningAnnotatedElement	result = null
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.4.6.2.4 Dependency_Mapping**Description**

*** not specified yet ***

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 138. 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{}

Target Property	Target Value
Element::humanId	result = null
Element::identifier	(abstract rule)
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)
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.2.4.6.2.5 DirectRelationship_Mapping

Description

*** not specified yet ***

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 139. Table DirectRelationship_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}

Target Property	Target Value
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
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.2.4.6.2.6 ElementMain_Mapping

Description

This is the general abstract class to be used as an ancestor for any class mapping specification.

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 140. Table ElementMain_Mapping Rules

Target Property	Target Value
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)

C.2.4.6.2.7 ElementOwnership_Mapping

Description

General Mappings

GenericToRelationship_Mapping

Mapping Source

Element

Mapping Target

Relationship

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 141. Table ElementOwnership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
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.2.4.6.2.8 ElementOwningMembership_Mapping

Description

*** not specified yet ***

General Mappings

ElementOwnership_Mapping
 GenericToMembership_Mapping

Mapping Source

Element

Mapping Target

Membership

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 142. Table ElementOwningMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)

Target Property	Target Value
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::membershipOwningNamespace	result = Namespace_Mapping.getMapped(from.owner)
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = self.target()->at(1)
Relationship::ownedRelatedElement	result = Set{}
Membership::ownedRelatedElement	result = self.target()
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.2.4.6.2.9 Namespace_Mapping

Description

*** not specified yet ***

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 143. Table Namespace_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::ownedRelationship	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e))
Element::ownedRelationship	result = Set{}

C.2.4.6.2.10 Relationship_Mapping

Description

*** not specified yet ***

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 144. 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.2.4.7 InformationFlows

C.2.4.7.1 Overview

Table 145. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
InformationFlow	Relationship	DirectedRelationship_Mapping	
InformationItem	ItemDefinition Relationship	InformationItem_Mapping InformationFlow_Mapping	

C.2.4.7.2 Mapping Specifications

C.2.4.8 Interactions

C.2.4.8.1 Overview

Table 146. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
ActionExecutionSpecification	Feature Relationship	ElementOwnership_Mapping	
BehaviorExecutionSpecification	Feature Relationship	ElementOwnership_Mapping	
CombinedFragment	Feature Relationship	ElementOwnership_Mapping	
ConsiderIgnoreFragment	Feature Relationship	ElementOwnership_Mapping	
Continuation	Feature Relationship	ElementOwnership_Mapping	
DestructionOccurrenceSpecification	Feature Relationship	ElementOwnership_Mapping	
ExecutionOccurrenceSpecification	Feature Relationship	ElementOwnership_Mapping	
ExecutionSpecification	Feature Relationship	ElementOwnership_Mapping	
Gate	Feature Relationship	ElementOwnership_Mapping	
GeneralOrdering	Feature Relationship	ElementOwnership_Mapping	
Interaction	Interaction	Interaction_Mapping	
InteractionConstraint		Constraint_Mapping	
InteractionFragment	Feature Relationship	ElementOwnership_Mapping	
InteractionOperand	PartDefinition	Block_Mapping	hasStereotypeApplied(InteractionOperand, 'SysML::Blocks::Block')
InteractionUse	Feature Relationship	ElementOwnership_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
Lifeline	Feature Relationship	ElementOwnership_Mapping	
Message	Feature Relationship	ElementOwnership_Mapping	
MessageEnd	Feature Relationship	ElementOwnership_Mapping	
MessageOccurrenceSpecification	Feature Relationship	ElementOwnership_Mapping	
OccurrenceSpecification	Feature Relationship	ElementOwnership_Mapping	
PartDecomposition	Feature Relationship	ElementOwnership_Mapping	
StateInvariant	Feature Relationship	ElementOwnership_Mapping	

C.2.4.8.2 Mapping Specifications

C.2.4.8.2.1 Interaction_Mapping

Description

A UML4SysML::Interaction is mapped to a SysMLv2::Interaction.

General Mappings

Behavior_Mapping

Mapping Source

Interaction

Mapping Target

Interaction

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 147. Table Interaction_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)

Target Property	Target Value
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Sequence(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = ((src.ownedElement-toFeatureMS) - Set{from.classifierBehavior}) in let relationships: Sequence(UML::Element) = toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e))) ->union(toFeatureMS->collect(e thisModule.resolve_PropertyMembership_Mapping((e)))) in if from.classifierBehavior.oclIsUndefined() then relationships else relationships->append(thisModule.resolve_ClassifierBehaviorMembership_Mapping(from.classifierBehavior)) endif

C.2.4.9 Packages

C.2.4.9.1 Overview

Table 148. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	File
Extension	FeatureMembership MetadataFeatureValue Annotation AnnotatingFeature MetadataFeature ConnectionDefinition	AssociationToMetadataFeatureMembership_Mapping AssociationToMetadataFeatureValue_Mapping AssociationIsDerived_Mapping AssociationToAnnotatingFeature_Mapping AssociationToMetadataFeature_Mapping Association_Mapping	
ExtensionEnd	PartUsage Feature Feature AttributeUsage Feature Feature	AttributeMembership_Mapping Part_Mapping PropertyMembership_Mapping AdjunctProperty_Mapping BoundReference_Mapping Attribute_Mapping ClassifierBehaviorProperty_Mapping EndPathMultiplicity_Mapping	let p : OclAny = src.oclAsType(UML::Object) p.oclIsUndefined() then p.type.oclHelper.hasStereotypeApplied(p, 'StereotypeName') endif hasStereotypeApplied(ExtensionEnd) 'SysML::Blocks::AdjunctProperty' hasStereotypeApplied(ExtensionEnd) 'SysML::Blocks::BoundReference' src.oclAsType(UML::Property).owningElement hasStereotypeApplied(ExtensionEnd) 'SysML::Blocks::ClassifierBehavior' hasStereotypeApplied(ExtensionEnd) 'SysML::Blocks::EndPathMultiplicity'
Image	Feature Relationship	ElementOwnership_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	File
Model	Package	Package_Mapping	
Package	Package	Package_Mapping	
PackageMerge	Relationship	DirectedRelationship_Mapping	
Profile	Package	Package_Mapping	
ProfileApplication	Relationship	DirectedRelationship_Mapping	
Stereotype	RequirementDefinition OccurrenceDefinition PortDefinition	Requirement_Mapping Class_Mapping InterfaceBlock_Mapping	hasStereotypeApplied(Stereotype, 'SysML::Requirements::Requirement' hasStereotypeApplied(Stereotype, 'SysML::Ports&Flows::InterfaceBlock')

C.2.4.9.2 Mapping Specifications

C.2.4.9.2.1 ElementImport_Mapping

Description

*** not specified yet ***

General Mappings

GenericToMembership_Mapping
DirectedRelationship_Mapping

Mapping Source

ElementImport

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 149. Table ElementImport_Mapping Rules

Target Property	Target Value
aliases	result = from.alias->asSet()
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
memberElement	result = ElementMain_Mapping.getMapped(from.importedElement)

Target Property	Target Value
memberName	result = from.importedElement.name
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)
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Relationship::source	result = Set{}
Relationship::target	result = Set{}
visibility	result = Helper.getKerMLVisibilityKind(from.visibility)

C.2.4.9.2.2 Package_Mapping

Description

*** not specified yet ***

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 150. Table Package_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	

Target Property	Target Value
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Package::ownedRelationship	result = from.ownedElement->reject(e e.oclIsKindOf(UML::ProfileApplication))->collect(e ElementOwningMembership_Mapping.getMapped(e))

C.2.4.9.2.3 PackageImport_Mapping

Description

*** not specified yet ***

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 151. Table PackageImport_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
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)
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.2.4.10 SimpleClassifiers

C.2.4.10.1 Overview

This chapter specifies the mapping of the UML4SysML metaclasses defined in the UML specification in the SimpleClassifiers chapter.

Table 152. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
BehavioredClassifier	PerformActionUsage PartDefinition	BehavioredClassifierToPerformActionUsage_Mapping Block_Mapping BehavioredClassifierToFeatureTyping_Mapping ClassifierBehaviorMembership_Mapping	hasStereotypeApplied(B' SysML::Blocks::Block')
DataType	AttributeDefinition AttributeDefinition	DataType_Mapping ValueType_Mapping	hasStereotypeApplied(D' SysML::Blocks::ValueT')
Enumeration	EnumerationDefinition	Enumeration_Mapping	
EnumerationLiteral	EnumerationUsage VariantMembership	EnumerationLiteral_Mapping EnumerationVariantMembership_Mapping	
Interface	PortConjugation ConjugatedPortDefinition PortDefinition	InterfacePortConjugation_Mapping InterfaceConjugatedPortDefinitionMembership_Mapping InterfaceConjugatedPortDefinition_Mapping Interface_Mapping	
InterfaceRealization	Subclassification	InterfaceRealization_Mapping	
PrimitiveType	AttributeDefinition AttributeDefinition	DataType_Mapping ValueType_Mapping	hasStereotypeApplied(P' SysML::Blocks::ValueT')
Reception	AttributeUsage	Reception_Mapping	
Signal	AttributeDefinition	Signal_Mapping	

C.2.4.10.2 Mapping Specifications

C.2.4.10.2.1 Attribute_Mapping

Description

An UML4SysML attribute (UML4SysML::Property) is mapped to a SysMLv2::AttributeUsage.

General Mappings

StructuralFeature_Mapping

Mapping Source

Property

Mapping Target

AttributeUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = src.oclAsType(UML::Property).owner.ocllsKindOf(UML::DataType)`

Table 153. Table Attribute_Mapping Rules

Target Property	Target Value
Element::aliasId	<code>result = Set{}</code>
Feature::direction	<code>result = null</code>
Element::documentation	<code>result = Set{}</code>
Element::humanId	<code>result = null</code>
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	<code>result = false</code>
Feature::isComposite	<code>result = false</code>
Feature::isDerived	<code>result = false</code>
Feature::isEnd	<code>result = false</code>
Feature::isOrdered	<code>result = false</code>
Feature::isPortion	<code>result = false</code>
Feature::isReadOnly	<code>result = false</code>
Type::isSufficient	<code>result = false</code>
Feature::isUnique	<code>result = true</code>
Element::ownedAnnotation	<code>result = Set{}</code>
Type::ownedFeatureMembership	<code>result = Set{}</code>
Namespace::ownedImport	<code>result = Set{}</code>
Namespace::ownedMembership	
Feature::ownedMembership	<code>result = Set{}</code>
Element::ownedRelationship	<code>result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)</code>
Feature::owningFeatureMembership	<code>result = null</code>

C.2.4.10.2.2 BehavioredClassifier_Mapping

Description

*** not specified yet ***

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 154. Table BehavioredClassifier_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{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Sequence(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = ((src.ownedElement-toFeatureMS) - Set{from.classifierBehavior}) in let relationships: Sequence(UML::Element) = toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e))) ->union(toFeatureMS->collect(e thisModule.resolve_PropertyMembership_Mapping((e)))) in if from.classifierBehavior.oclIsUndefined() then relationships else relationships->append(thisModule.resolve_ClassifierBehaviorMembership_Mapping(from.classifierBehavior)) endif
Namespace::ownedRelationship	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e))

C.2.4.10.2.3 BehavioredClassifierToFeatureMembership_Mapping**Description**

*** not specified yet ***

General Mappings

GenericToFeatureMembership_Mapping

Mapping Source

BehavioredClassifier

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 155. Table BehavioredClassifierToFeatureMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Membership::memberElement	(abstract rule)
memberFeature	result = performActionUsageMapping
memberName	result = 'classifierBehavior'
Membership::membershipOwningNamespace	(abstract rule)
Element::ownedAnnotation	result = Set{}
Membership::ownedMemberElement	result = null
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Membership::visibility	result = KerML::VisibilityKind::public

C.2.4.10.2.4 BehavioredClassifierToFeatureTyping_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeatureTyping_Mapping

Mapping Source

BehavioredClassifier

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 156. Table BehavioredClassifierToFeatureTyping_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Specialization::general	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
Element::ownedAnnotation	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Specialization::specific	(abstract rule)
Relationship::target	result = Set{}
type	result = from

C.2.4.10.2.5 BehavioredClassifierToPerformActionUsage_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeature_Mapping

Mapping Source

BehavioredClassifier

Mapping Target

PerformActionUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 157. Table BehavioredClassifierToPerformActionUsage_Mapping 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)
Type::isAbstract	result = false
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
PerformActionUsage::ownedRelationship	result = Set{featureTypingMapping}

C.2.4.10.2.6 DataType_Mapping

Description

A UML4SysML::DataType is mapped to a SysMLv2::AttributeDefinition. The mapping also cover the transformation of UML4SysML::PrimitiveType elements.

General Mappings

Classifier_Mapping

ElementMain_Mapping

Mapping Source

DataType

Mapping Target

AttributeDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

(none)

Table 158. Table DataType_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

Target Property	Target Value
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
AttributeDefinition::ownedRelationship	result = let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = (from.ownedElement - toFeatureMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toFeatureMS->collect(e AttributeMembership_Mapping.getMapped(e)))
Namespace::ownedRelationship	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e))

C.2.4.10.2.7 Enumeration_Mapping

Description

A UML4SysML::Enumeration is mapped to a SysMLv2::EnumerationDefinition.

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 159. Table Enumeration_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false

Target Property	Target Value
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
EnumerationDefinition::ownedRelationship	result = let toVariantMS: Set(UML::Element) = from.ownedElement->select(e e.oclIsKindOf(UML::EnumerationLiteral)) in let toElementOMS: Set(UML::Element) = (from.ownedElement - toVariantMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toVariantMS->collect(e EnumerationVariantMembership_Mapping.getMapped(e)))

C.2.4.10.2.8 EnumerationLiteral_Mapping

Description

A UML4SysML::EnumerationLiteral is mapped to a SysMLv2::EnumerationUsage.

General Mappings

ElementMain_Mapping
 GenericToFeature_Mapping
 InstanceSpecification_Mapping

Mapping Source

EnumerationLiteral

Mapping Target

EnumerationUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 160. Table EnumerationLiteral_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false

Target Property	Target Value
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
EnumerationUsage::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.4.10.2.9 Interface_Mapping

Description

A UML4SysML::Interface is mapped to a SysMLv2::PortDefinition. The mapping also includes the generation of an appropriate ConjugatedPortDefinition. That mappings is performed by the mapping classes InterfaceConjugatedPortDefinitionMembership_Mapping, InterfacePortConjugation_Mapping, and InterfaceConjugatedPortDefinition_Mapping.

General Mappings

GenericToPortDefinition_Mapping
 ElementMain_Mapping
 Classifier_Mapping

Mapping Source

Interface

Mapping Target

PortDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 161. Table Interface_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
Definition::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
PortDefinition::ownedRelationship	result = let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.ocIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = (from.ownedElement - toFeatureMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e, from))->union(toFeatureMS->collect(e PropertyMembership_Mapping.getMapped(e, from)))->append(conjugatedPortDefinitionMembership)
Namespace::ownedRelationship	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e))
Definition::variantMembership	result = Set{}

C.2.4.10.2.10 InterfaceConjugatedPortDefinition_Mapping

Description

As part of the mapping from a UML4SysML::Interface to a SysMLv2::PortDefinition, this mapping class is used to create the appropriate ConjugatedPortDefinition.

General Mappings

GenericToPortDefinition_Mapping

Mapping Source

Interface

Mapping Target

ConjugatedPortDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 162. Table InterfaceConjugatedPortDefinition_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
Definition::isVariation	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
ConjugatedPortDefinition::ownedRelationship	result = Set{portConjugation}
Definition::variantMembership	result = Set{}

C.2.4.10.2.11 InterfaceConjugatedPortDefinitionMembership_Mapping

Description

As part of the mapping from a UML4SysML::Interface to a SysMLv2::PortDefinition, this mapping class is used to create the membership relationship for the ConjugatedPortDefinition.

General Mappings

GenericToMembership_Mapping

Mapping Source

Interface

Mapping Target

No target element.

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 163. Table InterfaceConjugatedPortDefinitionMembership_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
memberName	result = '~'.concat(from.name)
Element::ownedAnnotation	result = Set{}
ownedRelatedElement	result = Set{conjugatedPortDefinitionMapping}
ownedRelationship	result = Set{portConjugation}
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.4.10.2.12 InterfacePortConjugation_Mapping**Description**

As part of the mapping from a UML4SysML::Interface to a SysMLv2::PortDefinition, this mapping class is used to create the appropriate PortConjugation relationship.

General Mappings

GenericToRelationship_Mapping

Mapping Source

Interface

Mapping Target

PortConjugation

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 164. Table InterfacePortConjugation_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
PortConjugation::conjugatedType	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)

Target Property	Target Value
PortConjugation::originalPortDefinition	result = from
Element::ownedAnnotation	result = Set{}
Element::ownedRelationship	result = Set{}

C.2.4.10.2.13 InterfaceRealization_Mapping

Description

A UML4SysML::InterfaceRealization is mapped to a SysMLv2::Superclassing.

General Mappings

Realization_Mapping

Generalization_Mapping

Mapping Source

InterfaceRealization

Mapping Target

Subclassification

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 165. Table InterfaceRealization_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{}
Specialization::general	(abstract rule)
Element::humanId	result = null
Element::identifier	(abstract rule)
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)
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Specialization::specific	(abstract rule)
Dependency::supplier	result = from.target->collect(e ElementMain_Mapping.getMapped(e))

C.2.4.10.2.14 Reception_Mapping

Description

A UML4SysML::Reception is mapped to a SysMLv2::AttributeUsage with feature direction "in".

General Mappings

GenericToUsage_Mapping
ElementMain_Mapping

Mapping Source

Reception

Mapping Target

AttributeUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 166. Table Reception_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	

Target Property	Target Value
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null

C.2.4.10.2.15 Signal_Mapping

Description

A UML4SysML::Signal is mapped to a SysMLv2::AttributeDefinition.

General Mappings

DataType_Mapping

ElementMain_Mapping

Mapping Source

Signal

Mapping Target

AttributeDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 167. Table Signal_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.ocellsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = from.ownedElement.excluding(toFeatureMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toFeatureMS->collect(e PropertyMembership_Mapping.getMapped(e)))

C.2.4.11 StructuredClassifiers

C.2.4.11.1 Overview

Table 168. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	File
Association	FeatureMembership MetadataFeatureValue Annotation AnnotatingFeature MetadataFeature ConnectionDefinition	AssociationToMetadataFeatureMembership_Mapping AssociationToMetadataFeatureValue_Mapping AssociationIsDerived_Mapping AssociationToAnnotatingFeature_Mapping AssociationToMetadataFeature_Mapping Association_Mapping	
AssociationClass	Association	AssociationClass_Mapping	
Class	RequirementDefinition OccurrenceDefinition PortDefinition	Requirement_Mapping Class_Mapping InterfaceBlock_Mapping	hasStereotypeApplied(Class, 'SysML::Requirements::Requirement') hasStereotypeApplied(Class, 'SysML::Ports&Flows::InterfaceBlock')
ConnectableElement	Feature Relationship	TypedElementToFeatureTyping_Mapping ElementOwnership_Mapping	not ConnectableElement.type. ConnectableElement.oclIsKind('ConnectableElement')
Connector	BindingConnector	BindingConnector_Mapping	hasStereotypeApplied(Connector, 'SysML::Blocks::BindingConnector')
ConnectorEnd		ConnectorEnd_Mapping	
EncapsulatedClassifier	PartDefinition ConcernUsage PartDefinition	Stakeholder_Mapping StakeholderToConcern_Mapping Block_Mapping	hasStereotypeApplied(EncapsulatedClassifier, 'SysML::ModelElements::Stakeholder') hasStereotypeApplied(EncapsulatedClassifier, 'SysML::ModelElements::Stakeholder') hasStereotypeApplied(EncapsulatedClassifier, 'SysML::Blocks::Block')
Port	Feature PortUsage	PortPart_Mapping Port_Mapping	
StructuredClassifier	PartDefinition ConcernUsage PartDefinition	Stakeholder_Mapping StakeholderToConcern_Mapping Block_Mapping	hasStereotypeApplied(StructuredClassifier, 'SysML::ModelElements::Stakeholder') hasStereotypeApplied(StructuredClassifier, 'SysML::ModelElements::Stakeholder') hasStereotypeApplied(StructuredClassifier, 'SysML::Blocks::Block')

C.2.4.11.2 Mapping Specifications

C.2.4.11.2.1 Association_Mapping

Description

A UML4SysML::Association is mapped to a SysMLv2::ConnectionDefinition. The UML4SysML::Association::isDerived property is not supported in SysML v2. To preserve the information, it is stored in a metadata annotation.

General Mappings

Classifier_Mapping
Relationship_Mapping

Mapping Source

Association

Mapping Target

ConnectionDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 169. 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
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Relationship::ownedRelatedElement	result = Set{}
ConnectionDefinition::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))->append(annotationFeatureMapping)
Namespace::ownedRelationship	result = from.ownedElement->collect(e ElementOwningMembership_Mapping.getMapped(e))
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.4.11.2.2 AssociationClass_Mapping

Description

*** not specified yet ***

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 170. Table AssociationClass_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Relationship::ownedRelatedElement	result = from.relatedElement->select(e from.ownedElement->includes(e))->collect(e ElementMain_Mapping.getMapped(e))
Classifier::ownedRelationship	result = let toFeatureMS: Sequence(UML::Element) = src.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = ((src.ownedElement-toFeatureMS) - Set{from.classifierBehavior}) in let relationships: Sequence(UML::Element) = toElementOMS->collect(e thisModule.resolve_ElementOwningMembership_Mapping((e))) ->union(toFeatureMS->collect(e thisModule.resolve_PropertyMembership_Mapping((e)))) in if from.classifierBehavior.oclIsUndefined() then relationships else relationships->append(thisModule.resolve_ClassifierBehaviorMembership_Mapping(from.classifierBehavior)) endif
Relationship::owningRelatedElement	result = ElementMain_Mapping.getMapped(from.owner)
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.4.11.2.3 AssociationToMetadata_Mapping

Description

*** not specified yet ***

General Mappings

GenericToAnnotatingElement_Mapping
GenericToFeature_Mapping

Mapping Source

Association

Mapping Target

MetadataFeature

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 171. 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
MetadataFeature::metadataFeatureValue	result = featureValue
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Element::ownedRelationship	result = Set{}

C.2.4.11.2.4 Class_Mapping

Description

*** not specified yet ***

General Mappings

GenericFromBehavioredClassifier_Mapping

Mapping Source

Class

Mapping Target

OccurrenceDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 172. Table Class_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.ocllsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = from.ownedElement.excluding(toFeatureMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toFeatureMS->collect(e PropertyMembership_Mapping.getMapped(e)))

C.2.4.11.2.5 ConnectorMapping

Description

*** not specified yet ***

General Mappings

ElementMain_Mapping

GenericToConnector_Mapping

Mapping Source

Connector

Mapping Target

ConnectionUsage

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 173. Table ConnectorMapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
ConnectionUsage::ownedFeatureMembership	result = Set{connectorEndMapping}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Relationship::ownedRelatedElement	result = Set{}
Element::ownedRelationship	result = Set{}
Feature::owningFeatureMembership	result = null
Relationship::owningRelatedElement	result = null
Relationship::source	result = Set{}
Relationship::target	result = Set{}

C.2.4.11.2.6 Port_Mapping

Description

*** not specified yet ***

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 174. Table Port_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.4.12 UseCases

C.2.4.12.1 Overview

Table 175. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
Actor	Class	BehavioredClassifierToPerformActionUsage_Mapping Block_Mapping BehavioredClassifierToFeatureTyping_Mapping ClassifierBehaviorMembership_Mapping	hasStereotypeApplied(Actor, 'SysML::Blocks::Block')
Extend	Relationship	DirectedRelationship_Mapping	
ExtensionPoint	Feature Relationship	ElementOwnership_Mapping	
Include	Relationship	DirectedRelationship_Mapping	
UseCase	UseCaseDefinition	UseCase_Mapping	

C.2.4.12.2 Mapping Specifications

C.2.4.12.2.1 UseCase_Mapping

Description

*** not specified yet ***

General Mappings

GenericFromBehavioredClassifier_Mapping
ElementMain_Mapping

Mapping Source

UseCase

Mapping Target

UseCaseDefinition

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 176. Table UseCase_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Classifier::isAbstract	result = from.isAbstract

Target Property	Target Value
Type::isSufficient	result = false
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Classifier::ownedRelationship	result = let toFeatureMS: Set(UML::Element) = from.ownedElement->select(e e.oclIsKindOf(UML::Property)) in let toElementOMS: Set(UML::Element) = from.ownedElement.excluding(toFeatureMS) in toElementOMS->collect(e ElementOwningMembership_Mapping.getMapped(e)) ->union(toFeatureMS->collect(e PropertyMembership_Mapping.getMapped(e)))

C.2.4.13 Values

C.2.4.13.1 Overview

Table 177. List of all Overview Mapping Specifications

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
Duration	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
DurationConstraint		Constraint_Mapping	
DurationInterval	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
DurationObservation	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
Expression	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
Interval	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
IntervalConstraint		Constraint_Mapping	
LiteralBoolean	LiteralBoolean	LiteralBoolean_Mapping	
LiteralInteger	LiteralInteger	LiteralInteger_Mapping	
LiteralNull	NullExpression	LiteralNull_Mapping	
LiteralReal	LiteralRational	LiteralReal_Mapping	
LiteralSpecification	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
LiteralString	LiteralString	LiteralString_Mapping	

SysML v1 Concept	SysML v2 Concept	Mapping Class	Filter
LiteralUnlimitedNatural	LiteralInfinity LiteralInteger	LiteralUnlimitedToUnbounded_Mapping LiteralUnlimitedToInteger_Mapping	src.oclAsType(UML::LiteralUnlimitedNatural) = -1 src.oclAsType(UML::LiteralUnlimitedNatural) <> -1
Observation	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
OpaqueExpression	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
StringExpression	Feature Relationship FeatureValue Expression	ElementOwnership_Mapping SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
TimeConstraint		Constraint_Mapping	
TimeExpression	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
TimeInterval	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)
TimeObservation	Feature Relationship Feature Relationship	ElementOwnership_Mapping ElementOwnership_Mapping	
ValueSpecification	FeatureValue Expression	SlotValue_Mapping ValueSpecification_Mapping	src.owner.oclIsKindOf(UML::Slot)

C.2.4.13.2 Mapping Specifications

C.2.4.13.2.1 LiteralBoolean_Mapping

Description

*** not specified yet ***

General Mappings

ValueSpecification_Mapping

Mapping Source

LiteralBoolean

Mapping Target

LiteralBoolean

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
 (none)

Table 178. Table LiteralBoolean_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
LiteralBoolean::value	result = from.value

C.2.4.13.2.2 LiteralInteger_Mapping

Description

*** not specified yet ***

General Mappings

ValueSpecification_Mapping

Mapping Source

LiteralInteger

Mapping Target

LiteralInteger

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 179. Table LiteralInteger_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
LiteralInteger::value	result = from.value

C.2.4.13.2.3 LiteralNull_Mapping

Description

*** not specified yet ***

General Mappings

ValueSpecification_Mapping

Mapping Source

LiteralNull

Mapping Target

NullExpression

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 180. Table LiteralNull_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.4.13.2.4 LiteralReal_Mapping

Description

*** not specified yet ***

General Mappings

ValueSpecification_Mapping

Mapping Source

LiteralReal

Mapping Target

LiteralRational

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 181. Table LiteralReal_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

Target Property	Target Value
LiteralRational::value	result = from.value

C.2.4.13.2.5 LiteralString_Mapping

Description

*** not specified yet ***

General Mappings

ValueSpecification_Mapping

Mapping Source

LiteralString

Mapping Target

LiteralString

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 182. Table LiteralString_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	<i>(abstract rule)</i>
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	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{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
LiteralString::value	result = if from.value.oclIsUndefined() then " else from.value endif

C.2.4.13.2.6 LiteralUnlimitedToUnbounded_Mapping

Description

*** not specified yet ***

General Mappings

ValueSpecification_Mapping

Mapping Source

LiteralUnlimitedNatural

Mapping Target

LiteralInfinity

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

`result = src.oclAsType(UML::LiteralUnlimitedNatural).value = -1`

Table 183. Table LiteralUnlimitedToUnbounded_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false

Target Property	Target Value
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null

C.2.4.13.2.7 LiteralUnlimitedToInteger_Mapping

Description

*** not specified yet ***

General Mappings

ValueSpecification_Mapping

Mapping Source

LiteralUnlimitedNatural

Mapping Target

LiteralInteger

Applicable filters

This mapping applies only if the following (OCL) condition is verified:

src.oclAsType(UML::LiteralUnlimitedNatural).value <> -1

Table 184. Table LiteralUnlimitedToInteger_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}
Feature::direction	result = null
Element::documentation	result = Set{}
Element::humanId	result = null
Element::identifier	(abstract rule)
Type::isAbstract	result = false
Feature::isComposite	result = false

Target Property	Target Value
Feature::isDerived	result = false
Feature::isEnd	result = false
Feature::isOrdered	result = false
Feature::isPortion	result = false
Feature::isReadOnly	result = false
Type::isSufficient	result = false
Feature::isUnique	result = true
Element::ownedAnnotation	result = Set{}
Type::ownedFeatureMembership	result = Set{}
Namespace::ownedImport	result = Set{}
Namespace::ownedMembership	
Feature::ownedMembership	result = Set{}
Element::ownedRelationship	result = ElementOwnership_Mapping.getMappedColl(from.ownedElement)
Feature::owningFeatureMembership	result = null
LiteralInteger::value	result = from.value

C.2.4.13.2.8 ValueSpecification_Mapping

Description

*** not specified yet ***

General Mappings

GenericToFeature_Mapping
ElementMain_Mapping

Mapping Source

ValueSpecification

Mapping Target

Expression

Applicable filters

This mapping applies only if the following (OCL) condition is verified:
(none)

Table 185. Table ValueSpecification_Mapping Rules

Target Property	Target Value
Element::aliasId	result = Set{}

Target Property	Target Value
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	
Element::ownedRelationship	result = Set{}
Expression::typing	result = TypedElement_Mapping.getMapped(from)