

Date: October 2022



OMG Systems Modeling Language™ (SysML®)

Version 2.0

Release 2022-09

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 Partners

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

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

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

Table of Contents

0 Submission Introduction	1
0.1 Submission Overview	1
0.2 Submission Submitters.....	1
0.3 Submission - Issues to be discussed.....	1
0.4 Language Requirements Tables	2
0.4.1 Mandatory Language Requirements Table	2
0.4.2 Non-Mandatory Language Requirements Table.....	53
0.4.3 Mandatory Language Requirements - Satisfied-by Table	70
0.4.4 Non-Mandatory Language Requirements - Satisfied-by Table	88
0.4.5 Changed Language Requirements Table	94
1 Scope.....	101
2 Conformance.....	103
3 Normative References	105
4 Terms and Definitions	107
5 Symbols	109
6 Introduction.....	111
6.1 Document Overview	111
6.2 Document Organization	112
6.3 Document Conventions.....	113
6.4 Acknowledgements	114
7 Language Description	117
7.1 Language Overview	117
7.2 Elements and Relationships	118
7.2.1 Overview	118
7.2.2 Abstract Syntax	119
7.2.3 Notation.....	119
7.3 Annotations	120
7.3.1 Overview	120
7.3.2 Abstract Syntax	121
7.3.3 Notation.....	121
7.4 Namespaces and Packages	124
7.4.1 Overview	124
7.4.2 Abstract Syntax	125
7.4.3 Notation.....	126
7.5 Dependencies	134
7.5.1 Overview	134
7.5.2 Abstract Syntax	134
7.5.3 Notation.....	134
7.6 Definition and Usage	135
7.6.1 Overview	135
7.6.2 Abstract Syntax	138
7.6.3 Notation.....	141
7.7 Attributes.....	153
7.7.1 Overview	153
7.7.2 Abstract Syntax	153
7.7.3 Notation.....	154
7.8 Enumerations	155
7.8.1 Overview	155
7.8.2 Abstract Syntax	156
7.8.3 Notation.....	156

7.9 Occurrences.....	158
7.9.1 Overview	158
7.9.2 Abstract Syntax	160
7.9.3 Notation.....	161
7.10 Items.....	166
7.10.1 Overview	166
7.10.2 Abstract Syntax	167
7.10.3 Notation.....	167
7.11 Parts.....	169
7.11.1 Overview	169
7.11.2 Abstract Syntax	169
7.11.3 Notation.....	170
7.12 Ports	173
7.12.1 Overview	173
7.12.2 Abstract Syntax	173
7.12.3 Notation.....	174
7.13 Connections.....	177
7.13.1 Overview	177
7.13.2 Abstract Syntax	179
7.13.3 Notation.....	180
7.14 Interfaces.....	191
7.14.1 Overview	191
7.14.2 Abstract Syntax	191
7.14.3 Notation.....	192
7.15 Allocations	196
7.15.1 Overview	196
7.15.2 Abstract Syntax	196
7.15.3 Notation.....	197
7.16 Actions	199
7.16.1 Overview	199
7.16.2 Abstract Syntax	201
7.16.3 Notation.....	203
7.17 States	219
7.17.1 Overview	220
7.17.2 Abstract Syntax	221
7.17.3 Notation.....	222
7.18 Calculations.....	225
7.18.1 Overview	225
7.18.2 Abstract Syntax	226
7.18.3 Notation.....	226
7.19 Constraints	227
7.19.1 Overview	227
7.19.2 Abstract Syntax	228
7.19.3 Notation.....	229
7.20 Requirements	230
7.20.1 Overview	230
7.20.2 Abstract Syntax	232
7.20.3 Notation.....	235
7.21 Cases	237
7.21.1 Overview	237
7.21.2 Abstract Syntax	237
7.21.3 Notation.....	239
7.22 Analysis Cases	239
7.22.1 Overview	239

7.22.2 Abstract Syntax	240
7.22.3 Notation	240
7.23 Verification Cases	242
7.23.1 Overview	242
7.23.2 Abstract Syntax	242
7.23.3 Notation	244
7.24 Use Cases	245
7.24.1 Overview	246
7.24.2 Abstract Syntax	246
7.24.3 Notation	247
7.25 Views and Viewpoints	249
7.25.1 Overview	249
7.25.2 Abstract Syntax	250
7.25.3 Notation	252
7.26 Metadata	256
7.26.1 Overview	257
7.26.2 Abstract Syntax	257
7.26.3 Notation	257
8 Metamodel	263
8.1 Metamodel Overview	263
8.2 Concrete Syntax	263
8.2.1 Concrete Syntax Overview	263
8.2.2 Textual Notation	263
8.2.2.1 Textual Notation Overview	263
8.2.2.1.1 EBNF Conventions	263
8.2.2.1.2 Lexical Structure	265
8.2.2.2 Elements and RelationshipsTextual Notation	265
8.2.2.3 Annotations Textual Notation	265
8.2.2.3.1 Annotations	265
8.2.2.3.2 Comments and Documentation	266
8.2.2.3.3 Textual Representation	266
8.2.2.4 Namespaces and Packages Textual Notation	266
8.2.2.4.1 Packages	267
8.2.2.4.2 Package Elements	270
8.2.2.5 Dependencies Textual Notation	271
8.2.2.6 Definition and Usage Textual Notation	271
8.2.2.6.1 Definitions	272
8.2.2.6.2 Usages	273
8.2.2.6.3 Reference Usages	273
8.2.2.6.4 Body Elements	274
8.2.2.6.5 Specialization	277
8.2.2.6.6 Multiplicity	279
8.2.2.7 Attributes Textual Notation	279
8.2.2.8 Enumerations Textual Notation	280
8.2.2.9 Occurrences Textual Notation	280
8.2.2.9.1 Occurrence Definitions	280
8.2.2.9.2 Occurrence Usages	281
8.2.2.9.3 Occurrence Successions	281
8.2.2.10 Items Textual Notation	281
8.2.2.11 Parts Textual Notation	282
8.2.2.12 Ports Textual Notation	282
8.2.2.13 Connections Textual Notation	282
8.2.2.13.1 Connection Definition and Usage	283
8.2.2.13.2 Binding Connectors	283

8.2.2.13.3 Successions.....	283
8.2.2.13.4 Messages and Flow Connections	284
8.2.2.14 Interfaces Textual Notation	286
8.2.2.14.1 Interface Definitions.....	287
8.2.2.14.2 Interface Usages	288
8.2.2.15 Allocations Textual Notation	288
8.2.2.16 Actions Textual Notation	288
8.2.2.16.1 Action Definitions.....	289
8.2.2.16.2 Action Usages.....	290
8.2.2.16.3 Action Nodes.....	291
8.2.2.16.4 Action Successions.....	295
8.2.2.17 States Textual Notation	295
8.2.2.17.1 State Definitions.....	296
8.2.2.17.2 State Usages	298
8.2.2.17.3 Transition Usages.....	299
8.2.2.18 Calculations Textual Notation.....	301
8.2.2.18.1 Calculation Definitions.....	301
8.2.2.18.2 Calculation Usages	301
8.2.2.19 Constraints Textual Notation	302
8.2.2.20 Requirements Textual Notation	302
8.2.2.20.1 Requirement Definitions	303
8.2.2.20.2 Requirement Usages.....	304
8.2.2.20.3 Concerns	304
8.2.2.21 Cases Textual Notation	305
8.2.2.22 Analysis Cases Textual Notation	305
8.2.2.23 Verification Cases Textual Notation.....	306
8.2.2.24 Use Cases Textual Notation	306
8.2.2.25 Views and Viewpoints Textual Notation	306
8.2.2.25.1 View Definitions	307
8.2.2.25.2 View Usages.....	307
8.2.2.25.3 Viewpoints.....	307
8.2.2.25.4 Renderings.....	308
8.2.2.26 Metadata Textual Notation	309
8.2.3 Graphical Notation	309
8.2.3.1 Graphical Notation Overview	310
8.2.3.2 Elements and Relationships Graphical Notation.....	311
8.2.3.3 Annotations Graphical Notation.....	311
8.2.3.3.1 Nodes.....	311
8.2.3.3.2 Relationships	312
8.2.3.4 Namespaces and Packages Graphical Notation	313
8.2.3.4.1 Nodes.....	313
8.2.3.4.2 Relationships	314
8.2.3.5 Dependencies Graphical Notation.....	315
8.2.3.6 Definition and Usage Graphical Notation	315
8.2.3.6.1 Nodes.....	315
8.2.3.6.2 Relationships	317
8.2.3.7 Attributes Graphical Notation	318
8.2.3.8 Enumerations Graphical Notation.....	318
8.2.3.9 Occurrences Graphical Notation	318
8.2.3.10 Items Graphical Notation	319
8.2.3.11 Parts Graphical Notation	319
8.2.3.12 Ports Graphical Notation	319
8.2.3.13 Connections Graphical Notation	319
8.2.3.14 Interfaces Graphical Notation	320

8.2.3.15 Allocations Graphical Notation.....	320
8.2.3.16 Actions Graphical Notation.....	320
8.2.3.17 States Graphical Notation.....	321
8.2.3.18 Calculations Graphical Notation	321
8.2.3.19 Constraints Graphical Notation	321
8.2.3.20 Requirements Graphical Notation.....	321
8.2.3.21 Cases Graphical Notation.....	322
8.2.3.22 Analysis Cases Graphical Notation.....	322
8.2.3.23 Verification Cases Graphical Notation	322
8.2.3.24 Use Cases Graphical Notation.....	323
8.2.3.25 Views and Viewpoints Graphical Notation.....	323
8.2.3.26 Metadata Graphical Notation	324
8.3 Abstract Syntax	324
8.3.1 Abstract Syntax Overview	324
8.3.2 Elements and Relationships Abstract Syntax.....	324
8.3.3 Annotations Abstract Syntax.....	325
8.3.4 Namespaces and Packages Abstract Syntax	326
8.3.5 Dependencies Abstract Syntax.....	327
8.3.5.1 Overview	328
8.3.5.2 Dependency	328
8.3.6 Definition and Usage Abstract Syntax	328
8.3.6.1 Overview	329
8.3.6.2 Definition	329
8.3.6.3 ReferenceUsage.....	333
8.3.6.4 Usage	333
8.3.6.5 VariantMembership.....	337
8.3.7 Attributes Abstract Syntax	338
8.3.7.1 Overview	338
8.3.7.2 AttributeUsage	338
8.3.7.3 AttributeDefinition	339
8.3.8 Enumerations Abstract Syntax	339
8.3.8.1 Overview	340
8.3.8.2 EnumerationDefinition	340
8.3.8.3 EnumerationUsage	341
8.3.9 Occurrences Abstract Syntax	341
8.3.9.1 Overview	342
8.3.9.2 EventOccurrenceUsage	342
8.3.9.3 LifeClass.....	343
8.3.9.4 OccurrenceDefinition	343
8.3.9.5 OccurrenceUsage	344
8.3.9.6 PortioningFeature	346
8.3.9.7 PortionKind	346
8.3.10 Items Abstract Syntax	347
8.3.10.1 Overview	347
8.3.10.2 ItemDefinition	347
8.3.10.3 ItemUsage	348
8.3.11 Parts Abstract Syntax	348
8.3.11.1 Overview	348
8.3.11.2 PartDefinition	349
8.3.11.3 PartUsage	349
8.3.12 Ports Abstract Syntax	349
8.3.12.1 Overview	350
8.3.12.2 ConjugatedPortDefinition	350
8.3.12.3 ConjugatedPortTyping	351

8.3.12.4 PortConjugation	352
8.3.12.5 PortDefinition.....	352
8.3.12.6 PortUsage	353
8.3.13 Connections Abstract Syntax	354
8.3.13.1 Overview	354
8.3.13.2 BindingConnectorAsUsage	355
8.3.13.3 ConnectionDefinition	355
8.3.13.4 ConnectionUsage	356
8.3.13.5 ConnectorAsUsage.....	356
8.3.13.6 FlowConnectionDefinition.....	357
8.3.13.7 FlowConnectionUsage	357
8.3.13.8 SuccessionAsUsage.....	358
8.3.13.9 SuccessionFlowConnectionUsage	358
8.3.14 Interfaces Abstract Syntax	359
8.3.14.1 Overview	359
8.3.14.2 InterfaceDefinition	360
8.3.14.3 InterfaceUsage.....	360
8.3.15 Allocations Abstract Syntax.....	361
8.3.15.1 Overview	361
8.3.15.2 AllocationDefinition.....	361
8.3.15.3 AllocationUsage	362
8.3.16 Actions Abstract Syntax.....	362
8.3.16.1 Overview	363
8.3.16.2 AcceptActionUsage.....	365
8.3.16.3 ActionDefinition.....	366
8.3.16.4 ActionUsage	367
8.3.16.5 AssignmentActionUsage.....	368
8.3.16.6 ControlNode	368
8.3.16.7 DecisionNode	369
8.3.16.8 ForkNode.....	369
8.3.16.9 ForLoopActionUsage	370
8.3.16.10 IfActionUsage	371
8.3.16.11 JoinNode.....	371
8.3.16.12 LoopActionUsage.....	372
8.3.16.13 MergeNode.....	372
8.3.16.14 PerformActionUsage	373
8.3.16.15 SendActionUsage	374
8.3.16.16 TriggerInvocationExpression	375
8.3.16.17 TriggerKind	375
8.3.16.18 WhileLoopsActionusage	376
8.3.17 States Abstract Syntax.....	376
8.3.17.1 Overview	377
8.3.17.2 ExhibitStateUsage	378
8.3.17.3 StateSubactionKind	379
8.3.17.4 StateSubactionMembership	379
8.3.17.5 StateDefinition	380
8.3.17.6 StateUsage	381
8.3.17.7 TransitionFeatureKind	382
8.3.17.8 TransitionFeatureMembership	383
8.3.17.9 TransitionUsage	383
8.3.18 Calculations Abstract Syntax	384
8.3.18.1 Overview	385
8.3.18.2 CalculationDefinition	385
8.3.18.3 CalculationUsage	385

8.3.19 Constraints Abstract Syntax	386
8.3.19.1 Overview	386
8.3.19.2 AssertConstraintUsage	387
8.3.19.3 ConstraintDefinition	388
8.3.19.4 ConstraintUsage	388
8.3.20 Requirements Abstract Syntax	389
8.3.20.1 Overview	389
8.3.20.2 ActorMembership	391
8.3.20.3 ConcernDefinition	392
8.3.20.4 ConcernUsage	392
8.3.20.5 FramedConcernMembership	393
8.3.20.6 RequirementConstraintKind	393
8.3.20.7 RequirementConstraintMembership	394
8.3.20.8 RequirementDefinition	394
8.3.20.9 RequirementUsage	396
8.3.20.10 SatisfyRequirementUsage	397
8.3.20.11 SubjectMembership	398
8.3.20.12 StakeholderMembership	398
8.3.21 Cases Abstract Syntax	399
8.3.21.1 Overview	399
8.3.21.2 CaseDefinition	400
8.3.21.3 CaseUsage	401
8.3.21.4 ObjectiveMembership	401
8.3.22 Analysis Cases Abstract Syntax	402
8.3.22.1 Overview	402
8.3.22.2 AnalysisCaseDefinition	402
8.3.22.3 AnalysisCaseUsage	403
8.3.23 Verification Cases Abstract Syntax	404
8.3.23.1 Overview	404
8.3.23.2 RequirementVerificationMembership	405
8.3.23.3 VerificationCaseDefinition	406
8.3.23.4 VerificationCaseUsage	406
8.3.24 Use Cases Abstract Syntax	407
8.3.24.1 Overview	407
8.3.24.2 IncludeUseCaseUsage	408
8.3.24.3 UseCaseDefinition	408
8.3.24.4 UseCaseUsage	409
8.3.25 Views Abstract Syntax	409
8.3.25.1 Overview	410
8.3.25.2 Expose	412
8.3.25.3 RenderingDefinition	412
8.3.25.4 RenderingUsage	413
8.3.25.5 ViewDefinition	413
8.3.25.6 ViewpointDefinition	414
8.3.25.7 ViewpointUsage	415
8.3.25.8 ViewRenderingMembership	415
8.3.25.9 ViewUsage	416
8.3.26 Metadata Abstract Syntax	417
8.3.26.1 Overview	417
8.3.26.2 MetadataDefinition	417
8.3.26.3 MetadataUsage	418
8.4 Semantics	418
8.4.1 Semantics Overview	418
8.4.2 Definition and Usage Semantics	418

8.4.3 Attributes Semantics	418
8.4.4 Enumerations Semantics	419
8.4.5 Occurrences Semantics	419
8.4.6 Items Semantics	419
8.4.7 Parts Semantics	419
8.4.8 Ports Semantics	419
8.4.9 Connections Semantics	419
8.4.10 Interfaces Semantics	419
8.4.11 Allocations Semantics	419
8.4.12 Actions Semantics	419
8.4.13 States Semantics	419
8.4.14 Calculations Semantics	419
8.4.15 Constraints Semantics	419
8.4.16 Requirements Semantics	419
8.4.17 Cases Semantics	419
8.4.18 Analysis Cases Semantics	419
8.4.19 Verification Cases Semantics	419
8.4.20 Use Cases Semantics	419
8.4.21 View Semantics	419
8.4.22 Metadata Semantics	419
9 Model Libraries	421
9.1 Model Libraries Overview	421
9.2 Systems Model Library	421
9.2.1 Systems Model Library Overview	421
9.2.2 Attributes	421
9.2.2.1 Attributes Overview	422
9.2.2.2 Elements	422
9.2.2.2.1 AttributeValue	422
9.2.2.2.2 attributeValues	422
9.2.2.3 Items	422
9.2.3.1 Items Overview	422
9.2.3.2 Elements	423
9.2.3.2.1 Item	423
9.2.3.2.2 items	424
9.2.3.2.3 Touches	424
9.2.4 Parts	424
9.2.4.1 Parts Overview	425
9.2.4.2 Elements	425
9.2.4.2.1 Part	425
9.2.4.2.2 parts	425
9.2.5 Ports	426
9.2.5.1 Ports Overview	426
9.2.5.2 Elements	426
9.2.5.2.1 Port	426
9.2.5.2.2 ports	426
9.2.6 Connections	427
9.2.6.1 Connections Overview	427
9.2.6.2 Elements	427
9.2.6.2.1 BinaryConnection	427
9.2.6.2.2 binaryConnections	427
9.2.6.2.3 Connection	428
9.2.6.2.4 connections	428
9.2.6.2.5 FlowConnection	429
9.2.6.2.6 flowConnections	429

9.2.6.2.7 SuccessionFlowConnection.....	430
9.2.6.2.8 successionFlowConnections.....	430
9.2.7 Interfaces	431
9.2.7.1 Interfaces Overview	431
9.2.7.2 Elements	431
9.2.7.2.1 Interface.....	431
9.2.7.2.2 interfaces	431
9.2.8 Allocations	432
9.2.8.1 Allocations Overview.....	432
9.2.8.2 Elements	432
9.2.8.2.1 Allocation	432
9.2.8.2.2 allocations.....	432
9.2.9 Actions	433
9.2.9.1 Actions Overview.....	433
9.2.9.2 Elements	433
9.2.9.2.1 AcceptAction.....	433
9.2.9.2.2 acceptActions	434
9.2.9.2.3 Action	434
9.2.9.2.4 actions.....	436
9.2.9.2.5 AssignmentAction	436
9.2.9.2.6 assignmentActions.....	437
9.2.9.2.7 ControlAction	437
9.2.9.2.8 DecisionAction	437
9.2.9.2.9 DecisionTransitionAction	438
9.2.9.2.10 ForkAction.....	438
9.2.9.2.11 ForLoopAction	439
9.2.9.2.12 forLoopActions	440
9.2.9.2.13 IfThenAction	440
9.2.9.2.14 ifThenActions.....	441
9.2.9.2.15 IfThenElseAction	441
9.2.9.2.16 ifThenElseActions	441
9.2.9.2.17 JoinAction	442
9.2.9.2.18 LoopAction.....	442
9.2.9.2.19 loopActions	443
9.2.9.2.20 MergeAction.....	443
9.2.9.2.21 SendAction	444
9.2.9.2.22 sendActions	444
9.2.9.2.23 TransitionAction	445
9.2.9.2.24 transitionActions	445
9.2.9.2.25 WhileLoopAction	446
9.2.9.2.26 whileLoopActions	446
9.2.10 States	447
9.2.10.1 States Overview.....	447
9.2.10.2 Elements	447
9.2.10.2.1 StateAction	447
9.2.10.2.2 stateActions	447
9.2.10.2.3 StateTransitionAction	448
9.2.11 Calculations.....	448
9.2.11.1 Calculations Overview	448
9.2.11.2 Elements	448
9.2.11.2.1 Calculation.....	448
9.2.11.2.2 calculations	449
9.2.12 Constraints	449
9.2.12.1 Constraints Overview	449

9.2.12.2 Elements	449
9.2.12.2.1 ConstraintCheck	449
9.2.12.2.2 constraintChecks	450
9.2.13 Requirements.....	450
9.2.13.1 Requirements Overview	450
9.2.13.2 Elements	450
9.2.13.2.1 ConcernCheck	450
9.2.13.2.2 concernChecks.....	451
9.2.13.2.3 DesignConstraintCheck	451
9.2.13.2.4 FunctionalRequirementCheck	452
9.2.13.2.5 InterfaceRequirementCheck	452
9.2.13.2.6 PerformanceRequirementCheck	453
9.2.13.2.7 PhysicalRequirementCheck	453
9.2.13.2.8 RequirementCheck	453
9.2.13.2.9 requirementChecks.....	454
9.2.14 Cases	455
9.2.14.1 Cases Overview.....	455
9.2.14.2 Elements	455
9.2.14.2.1 Case	455
9.2.14.2.2 cases.....	455
9.2.15 Analysis Cases	456
9.2.15.1 Analysis Cases Overview	456
9.2.15.2 Elements	456
9.2.15.2.1 AnalysisAction	456
9.2.15.2.2 AnalysisCase	456
9.2.15.2.3 analysisCases.....	457
9.2.15.2.4 AnalysisAction	458
9.2.15.2.5 AnalysisCase	458
9.2.15.2.6 analysisCases	459
9.2.16 Verification Cases	459
9.2.16.1 Verification Cases Overview	459
9.2.16.2 Elements	459
9.2.16.2.1 PassIf	459
9.2.16.2.2 VerificationCase	460
9.2.16.2.3 verificationCases	460
9.2.16.2.4 VerificationCheck	461
9.2.16.2.5 VerificationMethod	461
9.2.17 Use Cases	462
9.2.17.1 Use Cases Overview	462
9.2.17.2 Elements	462
9.2.17.2.1 UseCase	462
9.2.17.2.2 useCases	462
9.2.18 Views.....	463
9.2.18.1 Views Overview	463
9.2.18.2 Elements	463
9.2.18.2.1 asElementTable	463
9.2.18.2.2 asInterconnectionDiagram	463
9.2.18.2.3 asTextualNotation	464
9.2.18.2.4 asTreeDiagram	464
9.2.18.2.5 GraphicalRendering	464
9.2.18.2.6 Rendering	465
9.2.18.2.7 renderings	465
9.2.18.2.8 TabularRendering	466
9.2.18.2.9 TextualRendering	466

9.2.18.2.10 View	467
9.2.18.2.11 ViewpointCheck	467
9.2.18.2.12 viewpointChecks	468
9.2.18.2.13 viewpointConformance	468
9.2.18.2.14 views	469
9.2.19 Standard View Definitions	469
9.2.19.1 Standard View Definitions Overview	469
9.2.19.2 Elements	473
9.2.19.2.1 ActionFlowView	473
9.2.19.2.2 AnalysisCaseView	474
9.2.19.2.3 ContainmentView	475
9.2.19.2.4 DefinitionAndUsageView	475
9.2.19.2.5 GeometryView	476
9.2.19.2.6 GridView	476
9.2.19.2.7 InterconnectionView	477
9.2.19.2.8 LanguageExtensionView	477
9.2.19.2.9 MemberView	478
9.2.19.2.10 PackageView	478
9.2.19.2.11 RequirementView	479
9.2.19.2.12 SequenceView	479
9.2.19.2.13 StateTransitionView	480
9.2.19.2.14 TreeView	480
9.2.19.2.15 UseCaseView	481
9.2.19.2.16 VerificationCaseView	481
9.2.19.2.17 ViewAndViewpointView	482
9.2.20 Metadata	483
9.2.20.1 Metadata Overview	483
9.2.20.2 Elements	483
9.2.20.2.1 MetadataItem	483
9.2.20.2.2 metadataItems	483
9.2.21 SysML	484
9.3 Metadata Domain Library	484
9.3.1 Metadata Domain Library Overview	485
9.3.2 Modeling Metadata	485
9.3.2.1 Modeling Metadata Overview	485
9.3.2.2 Elements	485
9.3.2.2.1 Issue	485
9.3.2.2.2 Rationale	485
9.3.2.2.3 Refinement	486
9.3.2.2.4 StatusInfo	486
9.3.2.2.5 StatusKind	487
9.3.2.3 Risk Metadata	488
9.3.3.1 Risk Metadata Overview	488
9.3.3.2 Elements	488
9.3.3.2.1 Level	488
9.3.3.2.2 LevelEnum	488
9.3.3.2.3 Risk	489
9.3.3.2.4 RiskLevel	489
9.3.3.2.5 RiskLevelEnum	490
9.3.3.3 Parameters of Interest Metadata	490
9.3.4.1 Parameters of Interest Metadata Overview	491
9.3.4.2 Elements	491
9.3.4.2.1 MeasureOfEffectiveness	491
9.3.4.2.2 MeasureOfPerformance	491

9.3.4.2.3 measuresOfEffectiveness	492
9.3.4.2.4 measuresOfPerformance	492
9.3.5 Image Metadata	492
9.3.5.1 Image Metadata Overview	493
9.3.5.2 Elements	493
9.3.5.2.1 Icon	493
9.3.5.2.2 Image	493
9.4 Analysis Domain Library	494
9.4.1 Analysis Domain Library Overview	494
9.4.2 Analysis Tooling	494
9.4.2.1 Analysis Tooling Overview	494
9.4.2.2 Elements	494
9.4.2.2.1 ToolExecution	494
9.4.2.2.2 ToolVariable	495
9.4.3 Sampled Functions	495
9.4.3.1 Sampled Functions Overview	495
9.4.3.2 Elements	495
9.4.3.2.1 Domain	495
9.4.3.2.2 Interpolate	496
9.4.3.2.3 interpolateLinear	496
9.4.3.2.4 Range	497
9.4.3.2.5 Sample	498
9.4.3.2.6 SampleFunction	498
9.4.3.2.7 SamplePair	499
9.4.4 State Space Representation	499
9.4.4.1 State Space Representation Overview	499
9.4.4.2 Elements	500
9.4.5 Trade Studies	501
9.4.5.1 Trade Studies Overview	501
9.4.5.2 Elements	501
9.4.5.2.1 EvaluationFunction	501
9.4.5.2.2 MaximizeObjective	501
9.4.5.2.3 MinimizeObjective	502
9.4.5.2.4 TradeStudy	502
9.4.5.2.5 TradeStudyObjective	503
9.5 Cause and Effect Domain Library	504
9.5.1 Cause and Effect Domain Library Overview	504
9.5.2 Causation Connections	504
9.5.2.1 Causation Connections Overview	504
9.5.2.2 Elements	504
9.5.2.2.1 causes	504
9.5.2.2.2 effects	505
9.5.2.2.3 Multicausation	505
9.5.2.2.4 multicausations	506
9.5.3 Cause and Effect	506
9.5.3.1 Cause and Effect Overview	506
9.5.3.2 Elements	506
9.5.3.2.1 CausationMetadata	506
9.5.3.2.2 CausationSemanticMetadata	507
9.5.3.2.3 CauseMetadata	507
9.5.3.2.4 EffectMetadata	508
9.5.3.2.5 MulticausationSemanticMetadata	508
9.6 Requirement Derivation Domain Library	509
9.6.1 Requirement Derivation Domain Library Overview	509

9.6.2 Derivation Connections.....	509
9.6.2.1 Derivation Connections Overview	509
9.6.2.2 Elements	509
9.6.2.2.1 Derivation.....	509
9.6.2.2.2 derivations	510
9.6.2.2.3 derivedRequirements.....	510
9.6.2.2.4 originalRequirements	511
9.6.3 Requirement Derivation	511
9.6.3.1 Requirement Derivation Overview	511
9.6.3.2 Elements	511
9.6.3.2.1 DerivationMetadata	511
9.6.3.2.2 DerivedRequirementMetadata	512
9.6.3.2.3 OriginalRequirementMetadata	512
9.7 Geometry Domain Library	513
9.7.1 Geometry Domain Library Overview	513
9.7.2 Spatial Items	513
9.7.2.1 Spatial Items Overview	513
9.7.2.2 Elements	513
9.7.2.2.1 CompoundSpatialItem.....	513
9.7.2.2.2 CurrentDisplacementOf.....	514
9.7.2.2.3 CurrentPositionOf	514
9.7.2.2.4 DisplacementOf.....	515
9.7.2.2.5 PositionOf.....	515
9.7.2.2.6 SpatialItem.....	516
9.7.3 Shape Items	517
9.7.3.1 Shape Items Overview	517
9.7.3.2 Elements	517
9.7.3.2.1 Circle	517
9.7.3.2.2 CircularCone	518
9.7.3.2.3 CircularCylinder.....	518
9.7.3.2.4 CircularDisc.....	519
9.7.3.2.5 Cone.....	519
9.7.3.2.6 ConeOrCylinder	520
9.7.3.2.7 ConicSection	520
9.7.3.2.8 ConicSurface	521
9.7.3.2.9 Cuboid	521
9.7.3.2.10 CuboidOrTriangularPrism.....	522
9.7.3.2.11 Cylinder	523
9.7.3.2.12 Disc	524
9.7.3.2.13 EccentricCone	524
9.7.3.2.14 EccentricCylinder.....	525
9.7.3.2.15 Ellipse	525
9.7.3.2.16 Ellipsoid	526
9.7.3.2.17 Hyperbola	526
9.7.3.2.18 Hyperboloid.....	527
9.7.3.2.19 Line	527
9.7.3.2.20 Parabola	527
9.7.3.2.21 Paraboloid.....	528
9.7.3.2.22 Path	528
9.7.3.2.23 PlanarCurve	529
9.7.3.2.24 PlanarSurface.....	529
9.7.3.2.25 Polygon	530
9.7.3.2.26 Polyhedron	530
9.7.3.2.27 Pyramid	531

9.7.3.2.28 Quadrilateral	531
9.7.3.2.29 Rectangle	532
9.7.3.2.30 RectangularCuboid	533
9.7.3.2.31 RectangularPyramid	533
9.7.3.2.32 RectangularToroid	534
9.7.3.2.33 RightCircularCone	534
9.7.3.2.34 RightCircularCylinder	535
9.7.3.2.35 RightTriangle	535
9.7.3.2.36 RightTriangularPrism	535
9.7.3.2.37 Shell	536
9.7.3.2.38 Sphere	537
9.7.3.2.39 Tetrahedron	537
9.7.3.2.40 Toriod	537
9.7.3.2.41 Torus	538
9.7.3.2.42 Triangle	539
9.7.3.2.43 TriangularPrism	539
9.8 Quantities and Units Domain Library	540
9.8.1 Quantities and Units Domain Library Overview	540
9.8.2 Quantities	541
9.8.2.1 Quantities Overview	541
9.8.2.2 Elements	542
9.8.2.2.1 scalarQuantities	542
9.8.2.2.2 ScalarQuantityValue	542
9.8.2.2.3 tensorQuantities	543
9.8.2.2.4 TensorQuantityValue	543
9.8.2.2.5 vectorQuantities	545
9.8.2.2.6 VectorQuantityValue	545
9.8.2.3 Measurement References	546
9.8.3.1 Measurement References Overview	546
9.8.3.2 Elements	546
9.8.3.2.1 AffineTransformationMatrix3d	546
9.8.3.2.2 ConversionByConvention	547
9.8.3.2.3 ConversionByPrefix	547
9.8.3.2.4 CoordinateFrame	548
9.8.3.2.5 CoordinateFramePlacement	549
9.8.3.2.6 CoordinateTransformation	549
9.8.3.2.7 CyclicRatioScale	550
9.8.3.2.8 DefinitionalQuantityValue	551
9.8.3.2.9 DerivedUnit	551
9.8.3.2.10 IntervalScale	552
9.8.3.2.11 LogarithmicScale	552
9.8.3.2.12 MeasurementScale	553
9.8.3.2.13 MeasurementUnit	554
9.8.3.2.14 OrdinalScale	554
9.8.3.2.15 Rotation	555
9.8.3.2.16 ScalarMeasurementReference	555
9.8.3.2.17 ScaleValueDefinition	556
9.8.3.2.18 ScaleValueMapping	557
9.8.3.2.19 SimpleUnit	557
9.8.3.2.20 TensorMeasurementReference	558
9.8.3.2.21 Translation	559
9.8.3.2.22 TranslationOrRotation	559
9.8.3.2.23 TranslationRotationSequence	560
9.8.3.2.24 UnitConversion	560

9.8.3.2.25 UnitPowerFactor	561
9.8.3.2.26 UnitPrefix	561
9.8.3.2.27 VectorMeasurementReference	562
9.8.3.2.28 VectorQuantityValue[1]	563
9.8.4 ISQ	563
9.8.4.1 ISQ Overview	563
9.8.4.2 Elements	563
9.8.4.2.1 amountOfSubstance	563
9.8.4.2.2 AmountOfSubstanceUnit	564
9.8.4.2.3 AmountOfSubstanceValue	564
9.8.4.2.4 AngularMeasureValue	565
9.8.4.2.5 Cartesian3dSpatialCoordinateSystem	565
9.8.4.2.6 duration	565
9.8.4.2.7 DurationUnit	566
9.8.4.2.8 DurationValue	566
9.8.4.2.9 electricCurrent	567
9.8.4.2.10 ElectricCurrentUnit	567
9.8.4.2.11 ElectricCurrentValue	567
9.8.4.2.12 length	568
9.8.4.2.13 LengthUnit	568
9.8.4.2.14 LengthValue	568
9.8.4.2.15 luminousIntensity	569
9.8.4.2.16 LuminousIntensityUnit	569
9.8.4.2.17 LuminousIntensityValue	570
9.8.4.2.18 mass	570
9.8.4.2.19 MassUnit	570
9.8.4.2.20 MassValue	571
9.8.4.2.21 Position3dVector	571
9.8.4.2.22 thermodynamicTemperature	571
9.8.4.2.23 ThermodynamicTemperatureUnit	572
9.8.4.2.24 ThermodynamicTemperatureValue	572
9.8.5 SI Prefixes	573
9.8.5.1 SI Prefixes Overview	573
9.8.5.2 Elements	574
9.8.6 SI	574
9.8.6.1 SI Overview	574
9.8.6.2 Elements	574
9.8.7 US Customary Units	574
9.8.7.1 US Customary Units Overview	574
9.8.7.2 Elements	574
9.8.8 Time	574
9.8.8.1 Time Overview	574
9.8.8.2 Elements	574
9.8.8.2.1 Clock	574
9.8.8.2.2 Date	575
9.8.8.2.3 DateTime	575
9.8.8.2.4 DurationOf	576
9.8.8.2.5 Iso8601DateTime	576
9.8.8.2.6 Iso8601DateTimeEncoding	577
9.8.8.2.7 Iso8601DateTimeStructure	577
9.8.8.2.8 timeInstant	578
9.8.8.2.9 TimeInstantValue	578
9.8.8.2.10 TimeOf	579
9.8.8.2.11 TimeOfDay	579

9.8.8.2.12 TimeScale	580
9.8.8.2.13 universalClock	580
9.8.8.2.14 UTC	581
9.8.8.2.15 utcTimeInstant	581
9.8.8.2.16 UtcTimeInstantValue	582
A Annex: Conformance Test Suite	583
B Annex: Example Model	585
B.1 Introduction	585
B.2 Model Organization	585
B.3 Definitions	586
B.4 Parts	589
B.5 Parts Interconnection	592
B.6 Actions	595
B.7 States	598
B.8 Requirements	600
B.9 Analysis	602
B.10 Verification	603
B.11 View and Viewpoint	605
B.12 Variability	606
B.13 Individuals	608

List of Tables

1. Mandatory Language Requirements Table	2
2. Non-Mandatory Language Requirements Table	53
3. Mandatory Language Requirements - Satisfied-by Table	70
4. Non-Mandatory Language Requirements - Satisfied-by Table	88
5. Changed Language Requirements Table	94
6. Standard Language Names	123
7. Annotations - Representative Notation	123
8. Packages - Representative Notation	131
9. Dependencies - Representative Notation	135
10. Definition and Usage - Representative Notation	146
11. Representative Compartment Matrix	151
12. Attributes - Representative Notation	155
13. Enumerations - Representative Notation	158
14. Occurrences - Representative Notation	164
15. Items - Representative Notation	168
16. Parts - Representative Notation	171
17. Ports - Representative Notation	176
18. Connections - Representative Notation	186
19. Interfaces - Representative Notation	193
20. Allocations - Representative Notation	198
21. Control Node Definitions	205
22. Actions - Representative Notation	212
23. States - Representative Notation	223
24. Calculations - Representative Notation	227
25. Constraints - Representative Notation	229
26. Requirements - Representative Notation	235
27. Analysis Cases - Representative Notation	241
28. Verification Cases - Representative Notation	244
29. Use Cases - Representative Notation	248
30. Views and Viewpoints - Representative Notation	253
31. Metadata - Representative Notation	260
32. EBNF Notation Conventions	263
33. Abstract Syntax Synthesis Notation	264
34. Grammar Production Definitions	264
35. Graphical BNF Conventions	310
36. Standard View Definitions	470

List of Figures

1. SysML Language Architecture	112
2. Elements.....	119
3. Annotation.....	121
4. Namespaces.....	125
5. Packages.....	126
6. Dependencies	134
7. Definition and Usage	139
8. Multiplicities	139
9. Classifiers.....	140
10. Subsetting.....	140
11. Variant Membership	141
12. Attribute Definition and Usage	154
13. Enumeration Definition and Usage	156
14. Occurrence Definition and Usage	161
15. Event Occurrences	161
16. Item Definition and Usage	167
17. Part Definition and Usage	170
18. Port Definition and Usage	174
19. Port Conjugation	174
20. Connectors as Usages	179
21. Connection Definition and Usage	179
22. Flow Connections	180
23. Feature Values	180
24. Interface Definition and Usage	192
25. Allocation Definition and Usage	197
26. Action Definition and Usage	201
27. Control Nodes	202
28. Action Performance	202
29. Send and Accept Actions	202
30. Assignment Actions	203
31. State Definition and Usage	221
32. State Membership	221
33. State Exhibition.....	222
34. Transition Usage	222
35. Calculation Definition and Usage	226
36. Constraint Definition and Usage	228
37. Constraint Assertion.....	229
38. Requirement Definition and Usage	233
39. Requirement Satisfaction	233
40. Concern Definition and Usage	234
41. Requirement Constraint Membership	234
42. Requirement Parameter Memberships.....	235
43. Case Definition and Usage	238
44. Case Membership.....	238
45. Analysis Case Definition and Usage	240
46. Verification Case Definition and Usage	243
47. Verification Membership	243
48. Use Case Definition and Usage	247
49. Use Case Inclusion.....	247
50. View Definition and Usage	250
51. Viewpoint Definition and Usage	251

52. Rendering Definition and Usage	251
53. Expose Relationship.....	252
54. View Rendering Membership	252
55. Metadata Definition and Usage	257
56. Elements.....	325
57. Annotation.....	326
58. Namespaces.....	327
59. Packages.....	327
60. Dependencies	328
61. Definition and Usage	329
62. Variant Membership	329
63. Attribute Definition and Usage.....	338
64. Enumeration Definition and Usage.....	340
65. Occurrence Definition and Usage	342
66. Event Occurrences	342
67. Item Definition and Usage	347
68. Part Definition and Usage.....	348
69. Port Definition and Usage.....	350
70. Port Conjugation	350
71. Connectors as Usages	354
72. Connection Definition and Usage.....	354
73. Flow Connections	355
74. Interface Definition and Usage	359
75. Allocation Definition and Usage	361
76. Action Definition and Usage	363
77. Control Nodes	363
78. Action Performance	364
79. Send and Accept Actions.....	364
80. Assignment Actions	364
81. Structured Control Actions	365
82. State Definition and Usage	377
83. State Membership	377
84. State Exhibition.....	378
85. Transition Usage	378
86. Calculation Definition and Usage	385
87. Constraint Definition and Usage	386
88. Constraint Assertion.....	387
89. Requirement Definition and Usage	389
90. Requirement Satisfaction	390
91. Concern Definition and Usage.....	390
92. Requirement Constraint Membership	391
93. Requirement Parameter Memberships	391
94. Case Definition and Usage	399
95. Case Membership.....	400
96. Analysis Case Definition and Usage	402
97. Verification Case Definition and Usage	404
98. Verification Membership	405
99. Use Case Definition and Usage	407
100. Use Case Inclusion.....	407
101. View Definition and Usage	410
102. Viewpoint Definition and Usage	410
103. Rendering Definition and Usage	411
104. Expose Relationship.....	411
105. View Rendering Membership	412

106. Metadata Definition and Usage	417
107. State Space Representation action and calculation definitions	500
108. Model Organization for SimpleVehicleModel	585
109. Part Definition for Vehicle.....	586
110. Part Definition for FuelTank Referencing Fuel it Stores.....	587
111. Axle and its Subclass FrontAxe.....	588
112. Example Definition Elements	589
113. Part Usage for vehicle_b.....	590
114. Parts Tree for vehicle_b.....	591
115. Variant engine4Cyl	592
116. Parts Interconnection for vehicle_b	593
117. Action providePower	595
118. Action flow for providePower	596
119. Action flow for transportPassenger	597
120. Vehicle States.....	598
121. Requirement Definition MassRequirement	601
122. Requirements Group vehicleSpecification	601
123. Analysis Case fuelEconomyAnalysis	602
124. Vehicle Mass Verification Test	604
125. Vehicle Safety View	605
126. Rendering of view vehiclePartsTree_Safety.....	606
127. Variability Model for vehicleFamily	607
128. Vehicle Individuals and Snapshots	609

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>.

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

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 submission includes a model of a transformation from SysML v1.7 (expected to be the last version of SysML v1) to SysML v2 (see view link doesn't refer to a view). 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).

Reqt. ID	Reqt. Name	Text
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.
ANL 1.03	Parameters of Interest	Proposals for SysML v2 shall include the capability to identify the key parameters of interest including measures-of-effectiveness (MoE) and other key measures of performance (MoP).
ANL 1.04	Analysis Case	<p>Proposals for SysML v2 shall include the capability to model the analysis case to specify the analysis scenarios and associated analysis methods needed to produce an analysis result that achieves the analysis objectives.</p> <p>Supporting Information: This is intended to be a specialization of Case.</p>
ANL 1.05	Analysis Objectives	Proposals for SysML v2 shall include the capability to model the objective of the analysis being performed in text or as a mathematical formalism, e.g. math expression, so that it can be evaluated.
ANL 1.06	Analysis Scenarios	Proposals for SysML v2 shall include the capability to model the scenarios that identify the analysis models to be executed, the conditions and assumptions, and the configurations of the subject of the analysis and the related infrastructure to perform the analysis.

Reqt. ID	Reqt. Name	Text
ANL 1.07	Analysis Assumption	Proposals for SysML v2 shall include the capability to model the assumptions of the analyses in a text or mathematical form, e.g. constraints and boundary conditions.
ANL 1.08	Analysis Decomposition	Proposals for SysML v2 shall include the capability to decompose an analysis into constituent analyses.
ANL 1.09	Analysis Model	<p>Proposals for SysML v2 shall include the capability to specify an analysis model.</p> <p>Supporting Information: Analysis models can be defined natively in SysML (e.g. parametric model or behavior model) or externally (e.g. equation-based math models, finite element analysis models, or computational fluid dynamics models). The level of fidelity of the specification of the analysis model can vary from an abstract specification that defines the intent of the analysis including its input and output parameters, to a detailed specification that a particular solver can execute.</p>
ANL 1.11	Analysis Result	<p>Proposals for SysML v2 shall include the capability to relate the results of executing analysis models to the analysis.</p> <p>Supporting Information: The results may be stored in the SysML v2 model itself or in an external store (e.g. CSV file or database). The results can be used to evaluate how well the analysis objectives are satisfied, and to obtain the supporting rationale for decisions taken based on the analysis.</p>

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	<p>Proposals for SysML v2 shall include a capability to represent a decision as one or more selections among alternatives.</p> <p>Supporting Information: This Decision and Rationale can be related through an Explanation relationship. The Rationale can refer to the supporting analysis.</p>
ANL 1.14.5	Criteria	Proposals for SysML v2 shall include a capability to represent criteria that is used as a basis for a decision or evaluation.
ANL 1.14.6	Rationale	Proposals for SysML v2 shall include a capability to represent rationale for a decision or other conclusion.
BHV 1	Behavior Requirements Group	

Reqt. ID	Reqt. Name	Text
BHV 1.01	Behavior	<p>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.</p>
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	

Reqt. ID	Reqt. Name	Text
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>

Reqt. ID	Reqt. Name	Text
BHV 1.03.4	Opaque Behavior	Proposals for SysML v2 shall include the capability to represent a behavior that embeds the definition in a language such as a programming language.
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	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.

Reqt. ID	Reqt. Name	Text
BHV 1.04.3	Integration of Constraints with Finite State Behavior	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.
BHV 1.05	Discrete and Continuous Time Behavior	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.
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: The ordering of actions (i.e., functions) is accomplished through ordering of their start and completion events. Events can trigger a change from one finite-state to another. Events should be able to be explicitly represented in both function-based behavior and finite-state behavior. Events can be defined and used in different contexts.</p>
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>

Reqt. ID	Reqt. Name	Text
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	

Reqt. ID	Reqt. Name	Text
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.</p> <p>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	

Reqt. ID	Reqt. Name	Text
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>

Reqt. ID	Reqt. Name	Text
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>
CNF 1.3	Model Interoperability Conformance	<p>Proposals for SysML v2 shall provide test cases to assess conformance of a SysML v2 implementation with the SysML v2 model interoperability specification.</p>

Reqt. ID	Reqt. Name	Text
CNF 1.4	Traceability Matrix	Proposals for SysML v2 shall include a traceability matrix (include reference) that demonstrates how each language feature is verified by the conformance test suite.
CRC 1	Cross-cutting Requirements Group	The following specify the requirements that apply to all model elements.
CRC 1.1	Model and Model Library Group	
CRC 1.1.1	Model	<p>Proposals for SysML v2 shall include a capability to represent a Model (aka system model) that contains a set of uniquely identifiable model elements.</p> <p>Supporting Information: This is intended to be a kind of Container or Namespace.</p>
CRC 1.1.2	Model Library	<p>Proposals for SysML v2 shall include a capability to represent a Model Library that contains a set of model elements that are intended to support reuse.</p> <p>Supporting Information: This is intended to be a kind of Container or Namespace.</p>
CRC 1.1.3	Container	<p>Proposals for SysML v2 shall include the capability to represent a Container that is a model element that contains other model elements. Model elements within a container shall be distinguishable from one another.</p> <p>Supporting Information: This provides a way to organize the model. Containers can contain other containers.</p>
CRC 1.2	Model Element Group	

Reqt. ID	Reqt. Name	Text
CRC 1.2.2	Unique Identifier	<p>Proposals for SysML v2 shall include a capability to represent a single universally unique identifier for each model element that cannot be changed.</p> <p>Supporting Information: The unique identifier should enable assignment of URIs.</p>
CRC 1.2.3	Name and Aliases	<p>Proposals for SysML v2 shall include a capability to represent a name and one or more aliases for any named model element.</p> <p>Supporting Information:</p> <p>Selected kinds of model elements may not require a name (e.g. dependency), or the name may be optional, but still should be distinguishable within a namespace.</p> <p>Aliases enable users to assign more than one name for the same element, such as a shortened name. A common use of aliases is the use of an abbreviated or shortened name.</p>
CRC 1.2.4	Definition / Description	<p>Proposals for SysML v2 shall include a capability to represent one or more definitions and/or descriptions for each model element.</p>

Reqt. ID	Reqt. Name	Text
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>

Reqt. ID	Reqt. Name	Text
CRC 1.3.03	Dependency Relationship	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.
CRC 1.3.04	Cause-Effect Relationship	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.
CRC 1.3.05	Explanation Relationship	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.
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.

Reqt. ID	Reqt. Name	Text
CRC 1.3.09	Element Group Relationship	<p>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.</p>
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>

Reqt. ID	Reqt. Name	Text
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.
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.

Reqt. ID	Reqt. Name	Text
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.</p> <p>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>

Reqt. ID	Reqt. Name	Text
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>
CRC 1.6	Metadata Group	<p>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.</p>
CRC 1.6.1	Version	<p>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.</p>
CRC 1.6.2	Time Stamp	<p>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.</p>

Reqt. ID	Reqt. Name	Text
CRC 1.6.3	Data Protection Controls	<p>Proposals for SysML v2 shall include a capability to represent Data Protection Controls for one or more model elements, or of another element that refers to one or more elements.</p> <p>Supporting Information: This can include markings such as ITAR, proprietary or security classifications</p>
INF 1	Interface Requirements Group	<p>SysML v2 is intended to provide a robust capability to model interfaces that constrain the physical and functional interaction between structural elements. An interface in SysML v2 includes two (2) interface ends, the connection between them, and any constraints on the interaction.</p> <p>Supporting Information:</p> <p>An interface should support the following:</p> <ol style="list-style-type: none"> 1. Different levels of abstraction that include logical, functional, and physical interfaces, nested interfaces, and interface layers; 2. Diverse domains that include a combination of electrical, mechanical, software, and user interfaces; 3. Reuse of interfaces in different contexts; 4. Generation of interface control documents and interface specifications <p>A Port is also used to refer to an Interface End.</p>

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>

Reqt. ID	Reqt. Name	Text
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	

Reqt. ID	Reqt. Name	Text
INF 1.07.1	Item Exchange Constraints	<p>Proposals for SysML v2 shall provide the capability to constrain the interaction between the interface ends that includes constraints on the items to be exchanged, the allowable sequences and directions of those items, timing of the exchange and other characteristics. The items exchanged shall be consistent with the type and direction of the items specified in the connected Interface Ends.</p>
INF 1.07.2	Property Constraints	<p>Proposals for SysML v2 shall provide the capability to constrain the interaction between the interface ends that include mathematical constraints on the properties exposed by the Interface Ends.</p> <p>Supporting Information: The value properties may further be identified as Across or Through variables consistent with standard usage of the terms (e.g. specify properties that are constrained by conservation laws).</p>

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>

Reqt. ID	Reqt. Name	Text
INF 1.10	Allocating Functional Exchange to Interfaces	<p>Proposals for SysML v2 shall provide the capability to allocate or bind the outputs and inputs of a function to interface ends (or nested interface ends).</p> <p>Supporting Information:</p> <p>It is expected that there are validation rules to ensure consistency between the inputs and outputs of a function and the interface ends.</p> <p>This allocate or binding should be inherited by the Component subclasses.</p>
LNG 1	Language Architecture and Formalism Requirements Group	<p>This group specifies how the language is structured and defined.</p> <p>Supporting Information: Some concepts may be implemented as user-level model libraries.</p>
LNG 1.1	Metamodel and Profile Group	
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:</p> <p>Equivalent functional capability can be demonstrated by mapping the UML metaclasses and SysML stereotypes between SysML v2 and SysML v1.</p>
LNG 1.1.2	SysML Metamodel	<p>Proposals for SysML v2 shall be specified using a metamodel that includes abstract syntax, concrete syntax, semantics, and the relationships between them.</p>

Reqt. ID	Reqt. Name	Text
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: 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	<p>Proposals for SysML v2 shall provide a standard graphical concrete syntax.</p>
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	

Reqt. ID	Reqt. Name	Text
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	<p>Proposals for SysML v2 shall provide the capability to map shared concepts between SysML and UML.</p>
OTR 1	Interoperability Requirements Group	<p>Other requirements from other topic areas that also relate to interoperability include API 01, LNG 1.6, and the Interoperability Services Group, SVC 6.</p>
OTR 2	Usability Group	<p>An objective for SysML v2 is to address SysML v1 usability issues, and enable systems engineers and others to perform MBSE more effectively. The following usability goals apply to a diverse class of SysML v2 users:</p> <ol style="list-style-type: none"> <li data-bbox="1090 1353 1421 1444">1. User understanding when creating or interpreting models <li data-bbox="1090 1444 1421 1607">2. User engagement when creating or interpreting models (this particularly applies to consumers of the model data) <li data-bbox="1090 1607 1421 1712">3. User productivity when creating models across the lifecycle

Reqt. ID	Reqt. Name	Text
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	<p>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.</p>

Reqt. ID	Reqt. Name	Text
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	<p>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.</p>
PRP 1.03	Value Expression	<p>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.</p>

Reqt. ID	Reqt. Name	Text
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>

Reqt. ID	Reqt. Name	Text
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>
PRP 1.11	Variable Length Collection Value Types	<p>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).</p>

Reqt. ID	Reqt. Name	Text
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	<p>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.</p>

Reqt. ID	Reqt. Name	Text
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.
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.

Reqt. ID	Reqt. Name	Text
RQT 1.1.2	Requirement Identifier	<p>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.</p>
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	<p>Proposals for SysML v2 shall include a capability to represent a requirement definition that contains an optional textual requirement statement.</p>
RQT 1.1.5	Restricted Requirement Statement Group	<pre><html></pre> <p>Supporting Information: Refer to Restricted Use Case Modeling (RUCM) [36] as an example of a restricted requirement statement.</p>

Reqt. ID	Reqt. Name	Text
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.
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>

Reqt. ID	Reqt. Name	Text
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	<p>Proposals for SysML v2 shall include a capability to represent the order of each requirement in a requirement group that is not constrained by its requirement identifier.</p> <p>Supporting Information: This primarily allows the user to further organize the requirements, but it does not impact the meaning of the requirements. For example, there may be a requirement group with one requirement to open a valve and another requirement to close a valve. The user may want to order the open requirement as the first requirement in the group.</p>
RQT 1.3	Requirement Relationships Group	
RQT 1.3.1	Requirement Specialization	Proposals for SysML v2 shall include a capability to represent a generalization relationship that relates a specialized requirement definition to a more general requirement definition.
RQT 1.3.2	Requirement Satisfaction	<p>Proposals for SysML v2 shall include a capability to represent a satisfy relationship that relates a requirement to a model element that is asserted to satisfy it.</p> <p>Supporting Information: This is intended to be a specialization of the Conform Relationship.</p>

Reqt. ID	Reqt. Name	Text
RQT 1.3.3	Requirement Verification	Proposals for SysML v2 shall include a capability to represent a verify relationship that relates a verification case to the requirement it is intended to verify.
RQT 1.3.4	Requirement Derivation	Proposals for SysML v2 shall include a capability to represent a derive relationship that relates a derived requirement to a source requirement.
RQT 1.3.5	Requirement Group Relationship	<p>Proposals for SysML v2 shall include a capability to represent a relationship between a requirement group and the members of the group that can include either a requirement or another requirement group.</p> <p>Supporting Information: This relationship groups requirements into a shared context.</p>
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>

Reqt. ID	Reqt. Name	Text
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</p>

Reqt. ID	Reqt. Name	Text
		and Item are introduced in the Interface requirements section.
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.

Reqt. ID	Reqt. Name	Text
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, subclasses, and different values of a value property.</p>
STC 1.10	Structure of an Individual	<p>Proposals for SysML v2 shall include a capability to represent a (de)composition of an Individual Element that is uniquely identifiable, and that can conform to an associated Structure resolved to a Single Variant and/or a Structure with Variability.</p> <p>Supporting Information: Such a digital representation of a real-world system is sometimes called a 'digital twin'. The elements in a Structure of an Individual are typically designated by a unique serial number, a batch number or an effectivity code.</p>

Reqt. ID	Reqt. Name	Text
STC 1.11	Usage Specific Localized Type	<p>Proposals for SysML v2 shall include a capability to represent local override, redefinition, or addition of features with respect to the features defined by its more general type at any level of nesting.</p> <p>Supporting Information: The more-general to more-specific type chain is: Definition Element - direct Usage Feature - deeply nested Usage Feature - Configuration Element - Individual Element.</p> <p>The localized usage should support capabilities equivalent to redefinition and sub-setting for usage elements at any level of nesting.</p>
VRF 1	Verification and Validation Requirements Group	<p>The requirements in this group represent how to evaluate whether systems satisfy their requirements using verification methods.</p> <p>Supporting Information: The requirements for validation are not called out explicitly, but are intended to be supported in a similar way as the requirements for verification.</p>
VRF 1.1	Verification Context	<p>Proposals for SysML v2 shall include the capability to model a Verification Context that includes the unit-under-verification, the verification case, and the verification system and associated environment that performs the verification.</p>
VRF 1.2	Verification Case Group	

Reqt. ID	Reqt. Name	Text
VRF 1.2.1	Verification Case	<p>Proposals for SysML v2 shall include the capability to model a verification case to evaluate whether one or more requirements are satisfied by a unit under verification.</p> <p>Supporting Information: This is intended to be a specialization of Case.</p>
VRF 1.2.2	Verification Objectives	<p>The verification case shall include verification objectives to be implemented by the verification activities.</p>
VRF 1.2.3	Verification Success Criteria	<p>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.</p>
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>

Reqt. ID	Reqt. Name	Text
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). Refer to the requirement for Model Mappings and Transformations under the Language Architecture and Formalism Requirements.</p>
ANL 1.12	Analysis Infrastructure	<p>Proposals for SysML v2 may include the capability to represent the hardware, software, and the personnel (analysis experts) required for performing the analysis.</p>
ANL 1.14	Decision Group	
ANL 1.14.1	Trade-off	<p>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.</p>
ANL 1.14.3	Decision Expression	<p>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)).</p>
BHV 1	Behavior Requirements Group	

Reqt. ID	Reqt. Name	Text
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>
BHV 1.03.5	Behavior Library	<p>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.</p>

Reqt. ID	Reqt. Name	Text
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	<p>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.</p>
CRC 1	Cross-cutting Requirements Group	
CRC 1.2	Model Element Group	

Reqt. ID	Reqt. Name	Text
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	
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	

Reqt. ID	Reqt. Name	Text
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	

Reqt. ID	Reqt. Name	Text
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: The SysML v2 Specification includes syntax, semantics, and vocabulary, so extending the language requires all of these to be extensible.</p>
LNG 1.6	Model Interchange, Model Mapping, and Transformations Group	

Reqt. ID	Reqt. Name	Text
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>

Reqt. ID	Reqt. Name	Text
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>
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>

Reqt. ID	Reqt. Name	Text
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>

Reqt. ID	Reqt. Name	Text
PRP 1.17	Across and Through Value Properties	<p><html></p> <p>Proposals for SysML v2 may include a capability to define across and through properties of flows on Interface Ends that participate in representing physical interactions in lumped parameter models.</p> <p>Supporting Information: Typically, the across and through properties are defined together as a pair, where the across property does not conserve energy and the through property does. For example, in a lumped parameter model of an electric circuit, the across and through properties are voltage and current respectively. See [SysPISF] for details.</p>

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. Consider references to standard formats (e.g., ISO 10303 (STEP), IGES)</p>

Reqt. ID	Reqt. Name	Text
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	

Reqt. ID	Reqt. Name	Text
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>

Reqt. ID	Reqt. Name	Text
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	refer to analysis subject.
ANL 1.02	Analysis	Partial	AnalysisCases	Refer to analysis case and its subject. The analysis models and related infrastructure can be modeled if desired as parts that perform the analysis case, or as metadata that refers to the analysis model.
ANL 1.03	Parameters of Interest	Yes	AnalysisCases	add provisions to identify an attribute as a moe, mop using key word /metadata.
ANL 1.04	Analysis Case	Yes	AnalysisCases	refer to analysis case and analysis result.
ANL 1.05	Analysis Objectives	Yes	AnalysisCases	refer to analysis objective.

ID	Name	Satisfied?	Satisfied-by	Comment
ANL 1.06	Analysis Scenarios	Yes	AnalysisCases	an analysis case is a kind of behavior that can include a set of actions which can be performed by infrastructure elements. The analysis models to be executed can be modeled explicitly if desired, or referred to externally via a url.
ANL 1.07	Analysis Assumption	Yes	AnalysisCases	assumptions can be modeled as constraints. They can also be included in the analysis objective.
ANL 1.08	Analysis Decomposition	Yes	AnalysisCases	analysis case can be decomposed into actions and/or other analyses cases.
ANL 1.09	Analysis Model	Yes		An analysis case can be used to specify an analysis model.
ANL 1.11	Analysis Result	Partial	AnalysisCases	This is intended to be supported by the API. For example, an analysis tool may execute and generate an output file. The API could then be used to input the data from the file and set values of attributes, or alternatively, just provide have the attribute refer to the file for the values. An execution trace is another form of analysis result that is specified by the execution semantics.

ID	Name	Satisfied?	Satisfied-by	Comment
ANL 1.13	Analysis Metadata	Yes		Metadata can be specified for an analysis including identification of input (independent) and output (dependent) parameters, and the location of an analysis tool (e.g., a url).
ANL 1.14	Decision Group			
ANL 1.14.2	Alternative	Yes	TradeStudies	Alternatives can be represented as variant design inputs to the trade study.
ANL 1.14.4	Decision	Yes	TradeStudies	The selection rationale can refer to an analysis.
ANL 1.14.5	Criteria	Yes	TradeStudies	A criteria can be specified as an attribute of a design alternative that is used in the evaluation. For example, an engine tradeoff analysis may include criteria for peak horsepower, fuel economy, weight, and cost.
ANL 1.14.6	Rationale	Yes	ModelingMetadata	Refer to rationale.
BHV 1	Behavior Requirements Group			
BHV 1.01	Behavior	Yes	Actions	Structural elements (e.g., parts) interact with other parts by performing actions that can accept inputs and produce outputs that can be input to other actions that are performed by other parts.

ID	Name	Satisfied?	Satisfied-by	Comment
BHV 1.02	Behavior Decomposition	Yes	Actions	Bahviors, such as actions and states, can be decomposed in the same way as structural elements, and nested usages can be redefined to specify localized usages.
BHV 1.03	Function-based Behavior Group			
BHV 1.03.1	Function-based Behavior	Yes	Actions	Refer to actions and interactions.
BHV 1.03.3	Function-based Behavior Constraints	Partial	Actions Constraints	There is no explicit notation for pre and post concition, but constraints can be specified on the start and done snapshots of an action, or throughout an action's execution.
BHV 1.03.4	Opaque Behavior	Yes	Actions Annotations	Refer to a textual representation.
BHV 1.03.6	Structure Modification Behavior	Yes	Actions	Use assignment to set the value of a feature at a specific point in time to null. Connectionsusages and partusages are kinds of features.
BHV 1.04	State-based Behavior Group			
BHV 1.04.1	Regions, States, and Transitions	Yes	States	States can be parallel or not.
BHV 1.04.2	Integration of Function-based Behavior with Finite State Behavior	Yes	States	Refer to actions in states and on transitions.
BHV 1.04.3	Integration of Constraints with Finite State Behavior	Yes	States Constraints	Refer to states.

ID	Name	Satisfied?	Satisfied-by	Comment
BHV 1.05	Discrete and Continuous Time Behavior	Yes	Actions	Inputs and outputs can vary at discrete points in time or continuously over time.
BHV 1.06	Events	Yes	Actions	Signal events result from send and accept actions. Change and time events are specified as triggers on accept actions.
BHV 1.07	Control Nodes	Partial	Actions	A guard on a flow specify the logical conditions to enable the flow.
BHV 1.08	Time Constraints	Yes	Time	The time model includes the ability to define a reference clock that specifies current time, and can be used as a basis for specifying absolute and relative time values.
BHV 1.10	Behavior Execution	Partial		The semantics of actions specify valid traces for the history of an execution (work in progress on documenting an execution interpretation of the declarative semantics.)
BHV 1.11	Integration between Structure and Behavior			
BHV 1.11.1	Allocation of Behavior to Structure	Yes	Actions PerformActionUsage States Allocations Occurrences	Refer to actions, states, and interactions. An action that requires more than one performer can be allocated to two parts.

ID	Name	Satisfied?	Satisfied-by	Comment
BHV 1.11.2	Integration of Control Flow and Input/Output Flow	Yes	Connections	The transfer of inputs to outputs are based on a common KerML concept of transfer. SysML flow connections are kinds of transfers.
BHV 1.11.3	Storing Items in Storage Elements Requirements Group			
BHV 1.11.3.1	Storage Element and Stored Item Usages	Partial		This requirements was not in the original RFP. Although there is not explicit storage element, an item can be stored as a reference feature, and conservation constraints can be specified. (Note: This requirements was not in the original RFP)
BHV 1.11.3.2	Create, Modify, and Consume Stored Items	Not Planned		This requirements was not in the original RFP.
BHV 1.12	Case	Yes	Cases	Refer to case.
CNF 1	Conformance Requirements Group			
CNF 1.1	Metamodel Conformance	No		
CNF 1.2	Profile Conformance	Not Planned		The requirement for a profile for SysML v2 has been superceded by the subset of elements that map from SysML v2 to SysML v1.
CNF 1.3	Model Interoperability Conformance	Not Planned		Interoperability with UML is done via the SysML v1 to v2 transformation.
CNF 1.4	Traceability Matrix	No		

ID	Name	Satisfied?	Satisfied-by	Comment
CRC 1	Cross-cutting Requirements Group			
CRC 1.1	Model and Model Library Group			
CRC 1.1.1	Model	Yes	Packages	Although the concept of a model is not explicitly defined, a similar concept can be represented as a root package. Can add a metadata tag if needed.
CRC 1.1.2	Model Library	Yes	Packages	Although the concept of a model library is not explicitly defined, a similar concept can be represented as a package that is designated to contain reusable elements. This package or its content can be imported into other packages to facilitate its reuse. Can add a metadata tag if needed.
CRC 1.1.3	Container	Yes	Packages	Refer to namespaces and packages.
CRC 1.2	Model Element Group			
CRC 1.2.2	Unique Identifier	Yes	Elements	Refer to Element, where each Element contains a unique id.
CRC 1.2.3	Name and Aliases	Yes	Namespaces	Elements can contain names and aliases relative to their namespace.
CRC 1.2.4	Definition / Description	Yes	Annotations	A documentation is a kind of comment on an element.

ID	Name	Satisfied?	Satisfied-by	Comment
CRC 1.2.5	Annotation	Yes	Annotations	An annotating element can annotate 0 to many other elements. A comment can include text with a hyperlink, which acts like a navigation relationship.
CRC 1.2.6	Element Group	Partial		An element group can be accomplished by a definition or usage element that contains reference features, or by a package that imports other elements based on filter criteria. Number 3 is not supported.
CRC 1.2.7	Additional Cross-Cutting Concepts Group			
CRC 1.2.7.1	Problem	Yes		The cause-effect relationship supports the identification of a problem as a cause of an effect.
CRC 1.2.7.2	Risk	Yes	RiskMetadata	The model library includes the ability to represent a risk..
CRC 1.3	Model Element Relationships Requirements Group			
CRC 1.3.01	Relationship	Yes	Elements	There is a KerML concept of relationship, but SysML includes different kinds of relationships dependency, membership, and connection.
CRC 1.3.02	Derived Relationship	Not Planned		Unable to determine a straightforward design approach,

ID	Name	Satisfied?	Satisfied-by	Comment
CRC 1.3.03	Dependency Relationship	Yes	Dependencies	Refer to dependency.
CRC 1.3.04	Cause-Effect Relationship	Yes	CauseAndEffect	Refer to causation relationship.
CRC 1.3.05	Explanation Relationship	Yes	ModelingMetadata	A rationale can refer to another element.
CRC 1.3.06	Conform Relationship	Not Planned		This relationship was evaluated to be of little value. It is often achieved through some form of specialization, e.g., subclassification, subset, redefinition.
CRC 1.3.07	Refine Relationship	Yes	ModelingMetadata	Refer to refinement relationship.
CRC 1.3.08	Allocation Relationship	Yes	Allocations	Refer to allocation.
CRC 1.3.09	Element Group Relationship	Partial		This is generally represented as a referential feature membership or an import relationship.
CRC 1.3.10	Navigation Relationship	Not Planned		This is addressed by API external relationship service, or more informally via a hyperlink in an annotating element.
CRC 1.4	Variability Modeling Group			
CRC 1.4.1	Variation Point	Yes	DefinitionAndUsage	Refer to General. Variation point applies to any definition and usage element.
CRC 1.4.2	Variant	Yes	DefinitionAndUsage	Refer to General. Variant can apply to any usage.
CRC 1.4.3	Variability Expression and Constraints	Yes	Constraints	Refer to constraints, which can constrain a selection of a variant at a variation point.

ID	Name	Satisfied?	Satisfied-by	Comment
CRC 1.4.4	Variant Binding	Not Planned		It is assumed that an external variability modeling tool can access the needed information through the API without an explicitly variant binding.
CRC 1.5	View and Viewpoint Group			
CRC 1.5.1	View Definition	Yes	Views	Refer to View Definition and Usage.
CRC 1.5.2	Viewpoint	Yes	Views	Refer to Viewpoint, stakeholder, and concerns.
CRC 1.6	Metadata Group			
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		Metadata can be defined to classify model elements to enable access control. Examples could include proprietary markings.
INF 1	Interface Requirements Group			
INF 1.01	Interface Definition and Reuse	Yes	Interfaces	Refer to Interfaces.
INF 1.02	Interface Usage	Yes	Interfaces	Refer to Interfaces.
INF 1.03	Interface Decomposition	Yes	Interfaces	Refer to Interfaces. Interfaces can be decomposed.
INF 1.04	Interface End Definitions	Yes	Ports	Refer to Ports.
INF 1.05	Conjugate Interface Ends	Yes	Ports	Refer to Conjugate Port Definitions.

ID	Name	Satisfied?	Satisfied-by	Comment
INF 1.06	Item Definition	Yes	Items	Refer to Items. Update rate is not supported explicitly. Requires a domain library.
INF 1.07	Interface Agreement Group			
INF 1.07.1	Item Exchange Constraints	Yes	Occurrences Constraints Connections	Allowable sequences and time ordering of exchanges is specified by interactions that are specified by occurrences. Constraints can be applied to an exchange.
INF 1.07.2	Property Constraints	Partial	Constraints Interfaces	Refer to Interfaces and Constraints.
INF 1.08	Interface Medium	Partial	Interfaces	An interface medium can be modeled as a part either referenced or owned by an interface. No special syntax is provided.
INF 1.09	Interface Layers	Yes	Interfaces	A layer is a kind of part with ports. An interface between layers can be modeled using interfaces.
INF 1.10	Allocating Functional Exchange to Interfaces	Yes	Connections	inputs and outputs of actions can be bound to directed features of ports. A flow can be performed by a connection.
LNG 1	Language Architecture and Formalism Requirements Group			
LNG 1.1	Metamodel and Profile Group			

ID	Name	Satisfied?	Satisfied-by	Comment
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	Refer to Clause 8 of the SysML Specification.
LNG 1.1.3	Metamodel Specification	Yes	1. Abstract Syntax	Refer to Clause 8 of the SysML Specification.
LNG 1.3	Abstract Syntax Group			
LNG 1.3.1	Syntax Specification	Yes	1. Abstract Syntax	The abstract syntax is specified in SMOF. The concrete syntax is specified in BNF, and maps to the abstract syntax.
LNG 1.3.2	View Independent Abstract Syntax	Yes	1. Abstract Syntax	The abstract syntax is specified independent of the concrete syntax.
LNG 1.4	Concrete Syntax Group			
LNG 1.4.1	Concrete Syntax to Abstract Syntax Mapping	Yes		The textual syntax is mapped to the abstract syntax. The graphical syntax is in progress. User defined element images are supported using metadata.
LNG 1.4.2	Graphical Concrete Syntax	Yes		The standard textual syntax is defined. The standard graphical syntax is specified to be consistent with the textual syntax.

ID	Name	Satisfied?	Satisfied-by	Comment
LNG 1.4.3	Syntax Examples	Yes		Every example model is defined using the textual syntax that maps to the abstract syntax.
LNG 1.5	Extensibility Group			
LNG 1.5.1	Extension Mechanisms	Yes	Metadata	The semantic metadata provides a means to extend the language syntax and semantics.
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	Partial		The primary usability criteria is the understandability of each language concept, which was assessed through the initial stakeholder review. The pilot implementation is used for on-going user validation of the language including both understandability and ease of use.
PRP 1	Properties, Values and Expressions Requirements Group			
PRP 1.01	Unified Representation of Values	Yes	Attributes Expressions FeatureValues	Refer to attribute, expressions, and feature values (e.g., constant is bound to a value).

ID	Name	Satisfied?	Satisfied-by	Comment
PRP 1.02	Value Type	Yes	Attributes	Refer to Attributes.
PRP 1.03	Value Expression	Yes	Expressions Annotations	Refer to Expressions. An 'opaque expression' can be represented as a kind of annoating element called a Textual Representation.
PRP 1.05	Unification of Expression and Constraint Definition	Yes	Constraints	Refer to Constraints.
PRP 1.06	System of Quantities	Yes	Quantities	Refer to the Quantities and Units Domain Library.
PRP 1.07	System of Units and Scales	Yes	MeasurementReferences ISQ SI USCustomaryUnits	Refer to the Quantities and Units Domain Library.
PRP 1.08	Range Restriction for Numerical Values	Yes	Constraints	Refer to Constraints. A constraint can be applied to an attribute or attribute definitioon.
PRP 1.10	Primitive Data Types	Yes	ScalarValues	Refer to KerML Scalar Values Library.
PRP 1.11	Variable Length Collection Value Types	Yes	Collections	Refer to KerML Collections Library.
PRP 1.12	Compound Value Type	Yes	Attributes	An attribute definition can be a structured data type.
PRP 1.15	Probabilistic Value Distributions	Not Planned		Did not have time to address this properly.
PRP 1.19	Materials with Properties	Not Planned		Did not have time and can be done as a future library.
RML 1	Example Model and Model Libraries Group			
RML 1.1	Example Model	Yes		Refer to Annex B of the SysML v2 Specification.

ID	Name	Satisfied?	Satisfied-by	Comment
RQT 1	Requirement Group			
RQT 1.1	Requirement Definition Group			
RQT 1.1.1	Requirement Definition Name	Yes	Requirements	Refer to Requirements.
RQT 1.1.2	Requirement Identifier	Yes	Requirements	Refer to Requirements. A requirement id is provided that can be set by a user defined numbering scheme.
RQT 1.1.3	Requirement Attributes	Yes	ModelingMetadata RiskMetadata	Refer to Metadata library that includes status, risk, etc.
RQT 1.1.4	Textual Requirement Statement	Yes	Requirements	Refer to Requirements.
RQT 1.1.5	Restricted Requirement Statement Group			
RQT 1.1.5.1	Restricted Requirement Statement	Not Planned		This was determined to be out of scope for in this specification.
RQT 1.1.5.2	Restricted Requirement Statement Extensibility	Not Planned		This was determined to be out of scope for in this specification.
RQT 1.1.5.3	Restricted Requirement Statement Transformation	Not Planned		This was determined to be out of scope for in this specification.
RQT 1.1.6	Formal Requirement Statement Group			
RQT 1.1.6.1	Formal Requirement Statement	Yes	Requirements	Refer to Requirements.
RQT 1.1.6.2	Assumptions	Yes	Requirements	Refer to Requirements.
RQT 1.2	Groups of Requirements			

ID	Name	Satisfied?	Satisfied-by	Comment
RQT 1.2.1	Requirement Group	Partial	Requirements	Refer to Requirements. Treated as a composite requirement. Did not include special syntax.
RQT 1.2.2	Requirement Usage (localized)	Yes	Requirements	Refer to Requirements.
RQT 1.2.3	Requirement Usage Identifier	Yes	Requirements	Refer to Requirements.
RQT 1.2.4	Requirement Usage Ordering	Yes	Types	This is supported by feature ordering.
RQT 1.3	Requirement Relationships Group			
RQT 1.3.1	Requirement Specialization	Yes	Requirements	Refer to Requirements.
RQT 1.3.2	Requirement Satisfaction	Partial	Requirements	Refer to Requirements. Can only satisfy a requirement by a feature.
RQT 1.3.3	Requirement Verification	Yes	VerificationCases	Refer to Verification Case.
RQT 1.3.4	Requirement Derivation	Yes	RequirementDerivation	Refer to derivation connection.
RQT 1.3.5	Requirement Group Relationship	Yes	Requirements	This is supported by the feature membership relationship between the requirement and its owner.

ID	Name	Satisfied?	Satisfied-by	Comment
RQT 1.3.6	Relationships to a Requirement Group	Yes	Requirements	A requirement group is a composite requirement. So, the meaning of a relationship to a requirement group is the same as the meaning for a composite requirement. For example, in order to satisfy a parent requirement, each child requirement must be satisfied.
RQT 1.4	Requirement Supporting Information	Yes	Annotations	Refer to Annotations.
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	Refer to Items, Parts.
STC 1.02	Usage Element	Yes	DefinitionAndUsage	Refer to General.
STC 1.03	Generic Hierarchical Structure	Yes	DefinitionAndUsage	Refer to General.
STC 1.04	Reference Element	Yes	DefinitionAndUsage	Refer to General.
STC 1.05	Multiplicity of Usage	Yes	Features	Refer to Features.
STC 1.06	Definition Element Specialization	Yes	Types	Refer to Types.
STC 1.07	Unambiguous Deeply Nested Structure	Yes	DefinitionAndUsage	Refer to General.
STC 1.08	Structure With Variability	Yes	DefinitionAndUsage	Refer to General.
STC 1.10	Structure of an Individual	Yes	Occurrences	Refer to Occurrences.
STC 1.11	Usage Specific Localized Type	Yes	DefinitionAndUsage	Refer to General.

ID	Name	Satisfied?	Satisfied-by	Comment
VRF 1	Verification and Validation Requirements Group			
VRF 1.1	Verification Context	Yes	Parts VerificationCases	The verification context can be modeled as a part or part def.
VRF 1.2	Verification Case Group			
VRF 1.2.1	Verification Case	Yes	VerificationCases	Refer to Verification Case.
VRF 1.2.2	Verification Objectives	Yes	VerificationCases	Refer to Verification Case.
VRF 1.2.3	Verification Success Criteria	Partial	VerificationCases	The verification objective can include criteria.
VRF 1.2.4	Verification Methods	Yes	VerificationCases	Refer to Verification Cases Library.
VRF 1.3	Verification System	Yes	VerificationCases Parts	A verification system and its components can perform the verification case and its nested actions and cases.
VRF 1.4	Verification Relationships Group			
VRF 1.4.1	Verification Objectives to Verification Cases	Yes	VerificationCases	The verification objective and associated verify relationship is part of the verification case.

ID	Name	Satisfied?	Satisfied-by	Comment
VRF 1.4.2	Validate Relationship	Not Planned		A requirement is validated when the requirement is determined to be the correct requirement that reflects the stakeholder needs. The way this is accomplished is subject to different methodologies. For example, a user prototype can be used to demonstrate certain functionality, and the user can validate that this is the correct functionality. The specific method of validation may impose different kinds of relationships. The SysML v2 concept of stakeholder concern can be used to create a specific relationship to note whether the concern is satisfactorily addressed or not.

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	Not Planned		defer to mapping service for API. Mapping of input and output parameters can be handled by metadata.

ID	Name	Satisfied?	Satisfied-by	Comment
ANL 1.12	Analysis Infrastructure	Not Planned		This was evaluated to not be that useful. If desired, these elements can be included in a model library or as metadata.
ANL 1.14	Decision Group			
ANL 1.14.1	Trade-off	Yes		Refer to trade off pattern in model library.
ANL 1.14.3	Decision Expression	Yes		The objective function is used to select a preferred alternative based on some criteria. Current objective functions include returning the alternative that results in the maximum value of the evaluation, or the alternative that results in a minimum value of the evaluation. The set of alternatives are input, the evaluation function is computed for each alternative, and the alternative with the maximum or minimum value is returned. Other objective functions can be defined.
BHV 1	Behavior Requirements Group			
BHV 1.03	Function-based Behavior Group			
BHV 1.03.2	Composite Input and Output	Yes	Actions	Inputs and outputs can be decomposed and bound to the inputs and outputs of nested actions.
BHV 1.03.5	Behavior Library	Not Planned		

ID	Name	Satisfied?	Satisfied-by	Comment
BHV 1.09	State History	Yes	Occurrences	Refer to snapshots that can be connected via successions. Also, attribute values can have default values.
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	Elements	Refer to Eement.
CRC 1.3	Model Element Relationships Requirements Group			
CRC 1.3.11	Copy Relationship	Not Planned		This was determined to not provide value, and potentially have an adverse impact.
INF 1	Interface Requirements Group			
INF 1.07	Interface Agreement Group			
INF 1.07.3	Geometric Constraints	Partial	ShapeItems	Refer to shapes and relationships. Can include shape features on ports.
LNG 1	Language Architecture and Formalism Requirements Group			
LNG 1.2	Semantics Group			
LNG 1.2.1	Semantic Model Libraries	Yes	Kernel Semantic Library	Refer to the KerML Semantics and their extension to SysML semantics in Clause 8 of the SysML Specification.

ID	Name	Satisfied?	Satisfied-by	Comment
LNG 1.2.2	Declarative Semantics	Yes		Refer to the KerML Core Semantics expressed in mathematical logic.
LNG 1.2.3	Reasoning Capability	Partial		There is a subset of KerML that can be mapped to OWL DL as a decideable subset. However, the complete and decideable subset has not been identified yet.
LNG 1.4	Concrete Syntax Group			
LNG 1.4.4	Textual Concrete Syntax	Yes		Refer to the SysML v2 textual syntax.
LNG 1.5	Extensibility Group			
LNG 1.5.2	Extensibility Consistency	Yes		Concrete syntax extensions include the ability to define a key word with an associated image.
LNG 1.6	Model Interchange, Model Mapping, and Transformations Group			
LNG 1.6.1	Model Interchange	Partial		XMI interchange is supported. JSON interchange format is in process. Can interchange the textual notation but without maintaining the id's.
LNG 1.6.2	Model Mappings and Transformations	Not Planned		This capability is delegated to a future version of the API.
PRP 1	Properties, Values and Expressions Requirements Group			
PRP 1.04	Logical Expressions	Yes	ControlFunctions BooleanFunctions	Refer to Control Functions.

ID	Name	Satisfied?	Satisfied-by	Comment
PRP 1.09	Automated Quantity Value Conversion	Yes	MeasurementReferenceQuantities and Units Domain Library.	Refer to the Quantities and Units Domain Library.
PRP 1.13	Discretely Sampled Function Value Type	Yes	SampledFunctions	Refer to Sampled Function.
PRP 1.14	Discretely Sampled Function Interpolation	Yes	SampledFunctions	Refer to Sampled Function interpolate calc def.
PRP 1.16	System Simulation Models	Not Planned		Did not have time to address this. Consider an equivalent library for SysPhys.
PRP 1.17	Across and Through Value Properties	Partial		Did not include specific syntax for across and through variables, but can represent them as features of ports.
PRP 1.18	Basic Geometry	Yes	ShapeItems SpatialItems MeasurementReferences	Refer to spatial items and shape items for representing geomerty, and refer to measurement refrences for coordinate frames and transformations.
PRP 1.20	Equivalent Element	Not Planned		This requirement was not in the original RFP.
RML 1	Example Model and Model Libraries Group			
RML 1.2	Model Libraries	Partial	ScalarValues Quantities MeasurementReferences Interfaces Actions Requirements VerificationCases AnalysisCases ShapeItems Views ImageMetadata Parts	Missing natural environments and materials in particular. Other libraries can be specialized from the libraries included.

ID	Name	Satisfied?	Satisfied-by	Comment
RQT 1	Requirement Group			
RQT 1.3	Requirement Relationships Group			
RQT 1.3.7	Relationship Logical Constraint	Yes	Constraints Requirements	Logical constraints can be applied to requirements such as and, xor, ...
STC 1	Structure Requirements Group			
STC 1.09	Structure Resolved to a Single Variant	Yes	DefinitionAndUsage	This is accomplished through subsetting a superset or product family which contains all possible valid variations.
VRF 1	Verification and Validation Requirements Group			
VRF 1.2	Verification Case Group			
VRF 1.2.5	Verification Activity	Yes	VerificationCases	This can be represented as actions in a verification case.
VRF 1.2.6	Verification Evaluation Activity	Yes	VerificationCases	This is accomplished by the verify requirement and the verdict.

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	<p>16 June 2018 - Added to mandatory requirements based on Creating and Accessing Stored Items Use Case 9</p> <p>Sept 2021 - Changed to Non-mandatory. Change was reviewed at SST Track Leads Meeting on 9 Sept 2021. All attendees agreed on the change.</p>

ID	Name	Requirement Text	Change Status	Change Description
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	<p>16 June 2018 - Added to mandatory requirements based on Creating and Accessing Stored Items Use Case 9</p> <p>Sept 2021 - Changed to Non-mandatory. Change was reviewed at SST Track Leads Meeting on 9 Sept 2021. All attendees agreed on the change.</p>

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</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).

ID	Name	Requirement Text	Change Status	Change Description
		<p>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>		
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. This includes representing the requirements, structure, and behavior of the system, and the specification of analysis cases and verification cases used to analyze and verify the system. 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](#).)
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 view link doesn't refer to a view. 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>

[SysML v1] *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 [9.8](#)):

[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>

[ISO 8601-1] ISO 8601-1:2019 (First edition) Date and time — Representations for information interchange — Part 1: Basic rules
<https://www.iso.org/standard/70907.html>

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 SysML Version 2 (SysML v2). SysML v2 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 extending the Kernel metamodel from the Kernel Modeling Language [KerML]. 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 v2 abstract syntax (see [8.3](#)), which extends the Kernel abstract syntax, providing specialized constructs for modeling systems (as shown in [Fig. 1](#)). Further, the Systems Library (see [9.2](#)) is a model library that extends the Kernel Library to provide the semantic specification for SysML v2 (see [8.4](#); see also [KerML] on the use of model libraries for semantic specification). Finally, SysML v2 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).

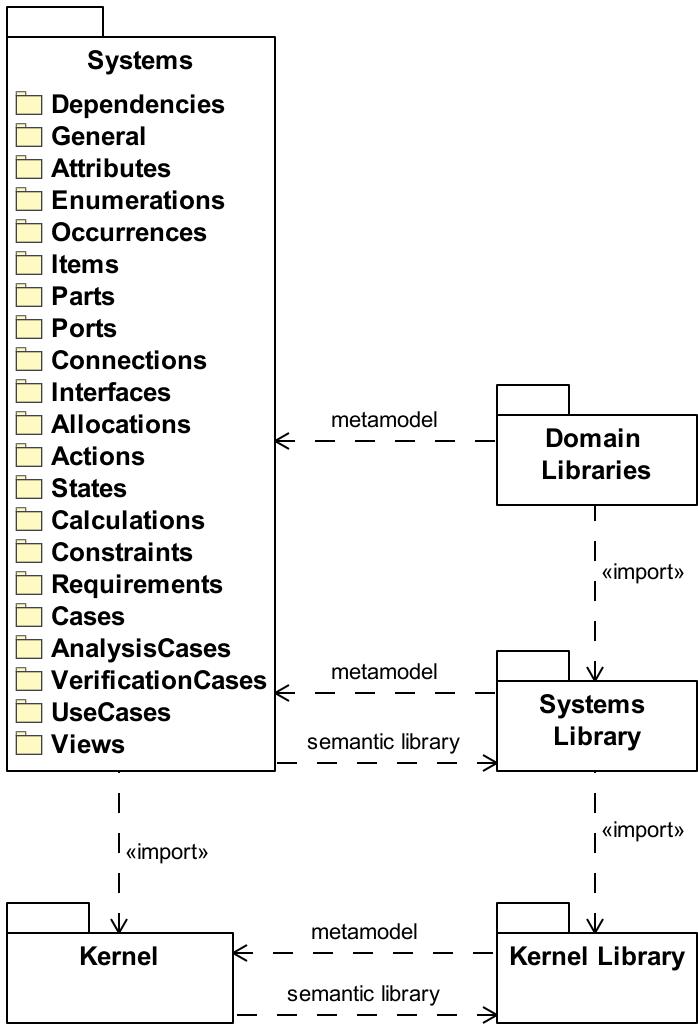


Figure 1. SysML Language Architecture

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 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 can 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 the a formal transformation from SysML v1 models to SysML v2 models that, to the greatest extent possible, preserves the semantics of conformant SysML v1 models. (This annex has been moved to a separate document.)

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 abstract syntax 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 an attribute of the Definition metaclass."
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 [8.2.2.1.1](#). For the graphical notation, this EBNF notation is further extended to allow the use of graphical symbols within productions (see [8.2.3.1](#)).

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

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
- Roger Burkhart, Thematix
- Tim Weilkiens, oose
- Hans Peter de Koning, DEKonsult

Other contributors include:

- Eran Gery, IBM
- Oystein Haugen, Østfold University College
- Tomas Juknevicius, Dassault Systèmes
- Charles Krueger, BigLever Software

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 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 (v1 to v2 transformation co-lead)
- Bjorn Cole, Lockheed Martin Corporation (metamodel development co-lead)

- Sanford Friedenthal, SAF Consulting (SST co-lead, requirements V&V lead)
- Charles Galey, Lockheed Martin Corporation (metamodel development co-lead)
- Karen Ryan, Siemens (metamodel development co-lead)
- Ed Seidewitz, Model Driven Solutions (SST co-lead, pilot implementation lead)
- Tim Weilkiens, oose (v1 to v2 transformation co-lead)

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

- Tyler Anderson, No Magic/Dassault Systèmes
- Christopher Delp, Jet Propulsion Laboratory
- Ivan Gomes, Twingineer
- Doris Lam, 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, Twingineer
- Hisashi Miyashita, Mgnite
- Miyako Wilson, Georgia Institute of Technology
- Santiago Leon, Tom Sawyer
- William Piers, Obeo
- Tilo Schreiber, Siemens
- 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 and textual representations (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 [7.26](#), 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 temporal extent, 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, to form 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 and ports are interconnected.
- *Allocations* that assign responsibility for realizing the features of one element by another element (see [7.15](#)).
- The modeling of *behavior*, which specifies how parts interact and includes:
 - *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 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* 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](#)).
- The modeling of user-defined *metadata* that allows for both simple tagging of elements with additional model-level information and more sophisticated semantic extension of the SysML language. In a similar way that SysML extends KerML, modelers can use this metadata capability 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 a short name. It can also have any number of alias names relative to one or more namespaces (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

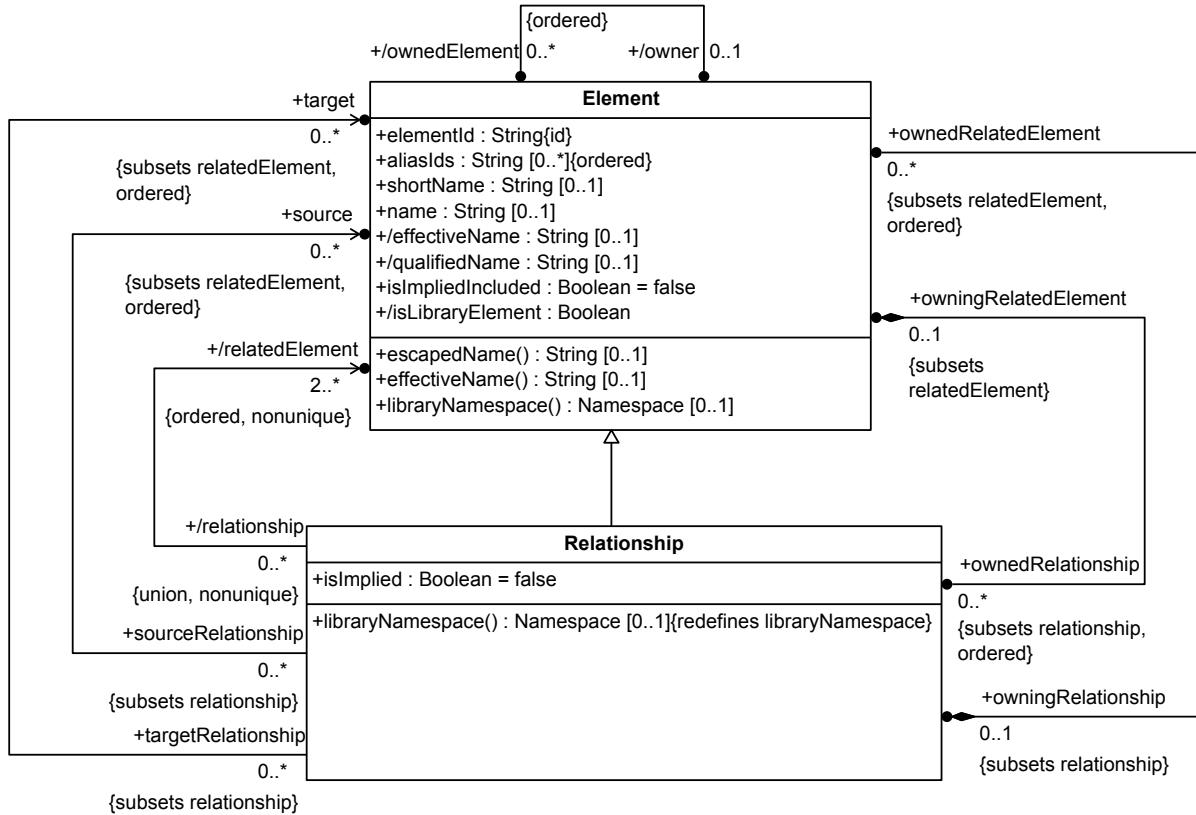


Figure 2. Elements

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

7.2.3 Notation

Textual Notation

For Element and Relationships Textual Notation BNF, see [8.2.2.2](#).

While Element and Relationship metaclasses are not abstract in the Kernel 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 `elementId` 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 `elementId` 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 `elementId` 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 `shortName` and/or `name` for it, in that order. Both the `shortName` and the `name` are have the same lexical structure, but the `shortName` is distinguished by being surrounded by the delimiting characters < and >.

```
part <'1.2.4'> MyName;
```

Note that it is not required to specify either a `shortName` 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.

Graphical Notation

For Elements and Relationships Graphical Notation BNF, see [8.2.3.2](#).

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 annotated 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 distinguished kind of comment used to document the annotated element. Documentation comments always annotate a single element, which is their owning element.

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.

It is also possible to annotate elements with user-defined *metadata*, allowing both syntactic and semantic extension of SysML. This capability is described in [7.26](#).

7.3.2 Abstract Syntax

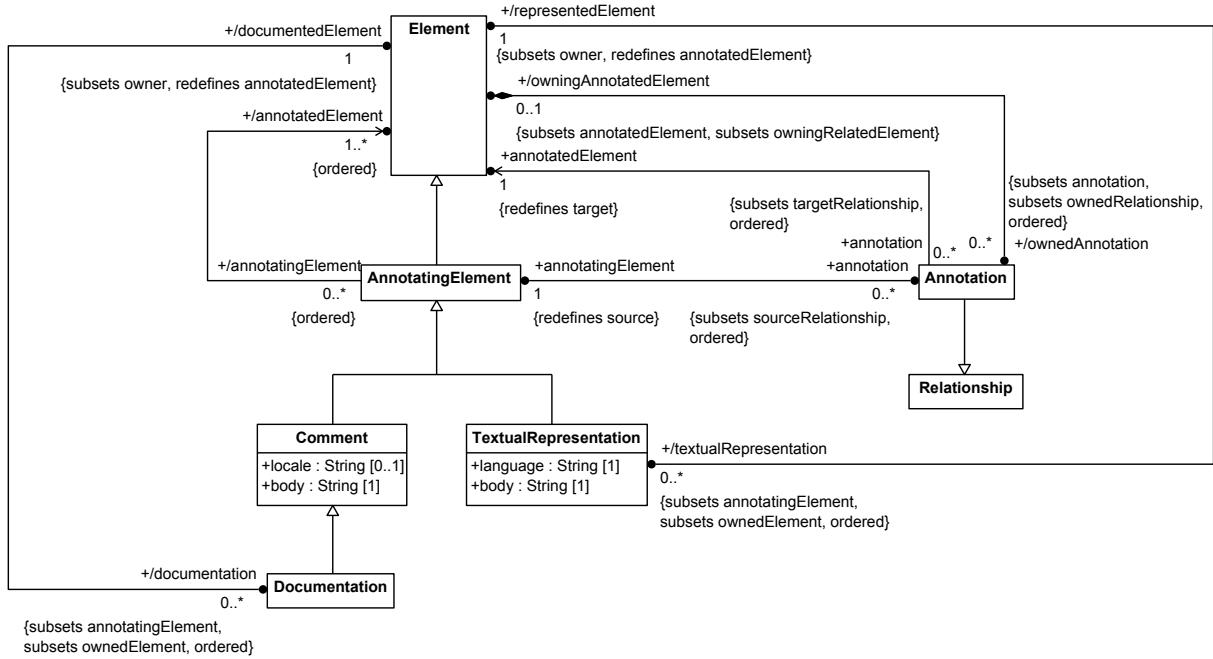


Figure 3. Annotation

7.3.3 Notation

For Annotations Textual Notation BNF, see [8.2.2.3](#).

Comments

The full declaration of a Comment begins with the keyword `comment`, optionally followed by a `shortName` 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 `shortName` 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. */
}
  
```

A Documentation Comment is notated similarly to a regular Comment, but using the keyword `doc` rather than `comment`. The `documentingElement` of a Documentation is always the owning Element of the Documentation.

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

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]. 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 referenced above.

Textual Representation

A TextualRepresentation is notated similarly to a regular Comment, but with the keyword `rep` used instead of `comment`. As for Documentation, a TextualRepresentation is always owned by its `representedElement`. In particular, if the TextualRepresentation is an `ownedMember` of a Namespace (see [7.4](#)), then, if the `representedElement` shall 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 `shortName` or `name`, then the `rep` keyword can also be omitted.

```
part def C {
    attribute x: Real;
    assert x_constraint {
        rep inOCL 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

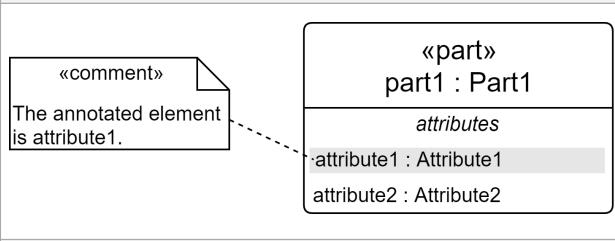
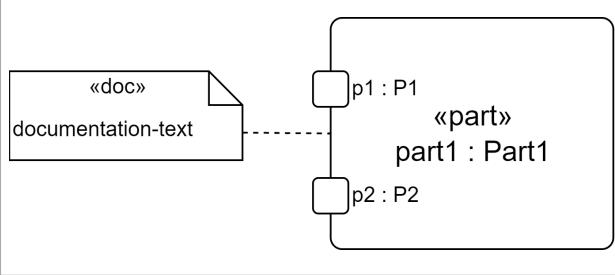
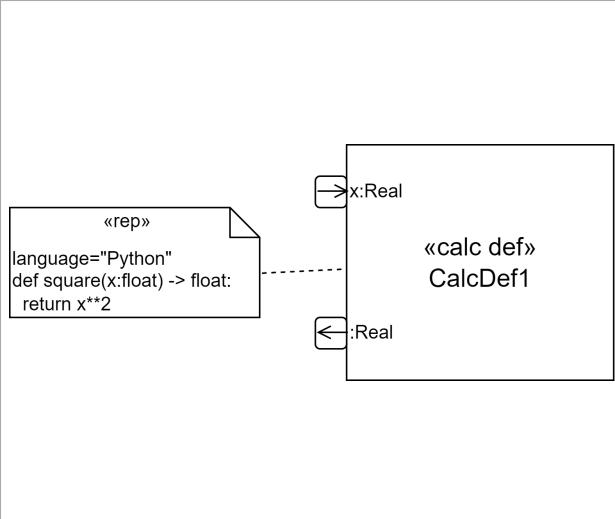
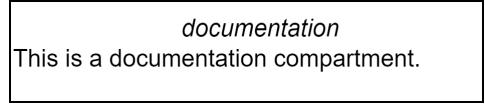
Language Name	Specification
sysml	Systems Modeling Language (this specification)
kerml	Kernel Modeling Language [KerML]
ocl	Object Constraint Language [OCL]
alf	Action Language for fUML [Alf]

Graphical Notation

For Annotations Graphical Notation BNF, see [8.2.3.3](#).

Table 7. Annotations - Representative Notation

Element	Graphical Notation	Textual Notation
Comment		<code>/*This is a comment.*/</code>
Comment		<code>comment Comment1</code> <code>/*This is a comment.*/</code>
Documentation		<code>doc /*This is documentation.*/</code>
Documentation		<code>doc <Document1></code> <code>/*This is documentation.*/</code>
Textual Representation		<code>rep language</code> <code>"language1" /* body1 */</code> or <code>language "language1" /*</code> <code>body1 */</code>

Element	Graphical Notation	Textual Notation
Annotation	 <pre data-bbox="1073 304 1421 418"> comment about part1::attribute1 /* The annotated element * is attribute1. */ </pre>	
Annotation-Documentation	 <pre data-bbox="1073 536 1372 692"> part part1 : Part1 { doc /* documentation-text */ port p1 : P1; port p2 : P2; } </pre>	
Annotation-Textual Representation	 <pre data-bbox="1073 798 1405 1254"> calc def square(x); rep about square language "Python" /* def square(x:float) ->float: * return x**2 */ or calc def square(x) { language "Python" /* def square(x) : * return x**2 */ } </pre>	
Documentation Compartment	 <pre data-bbox="1073 1326 1307 1438"> doc /*This is a documentation *compartment.*/ </pre>	

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

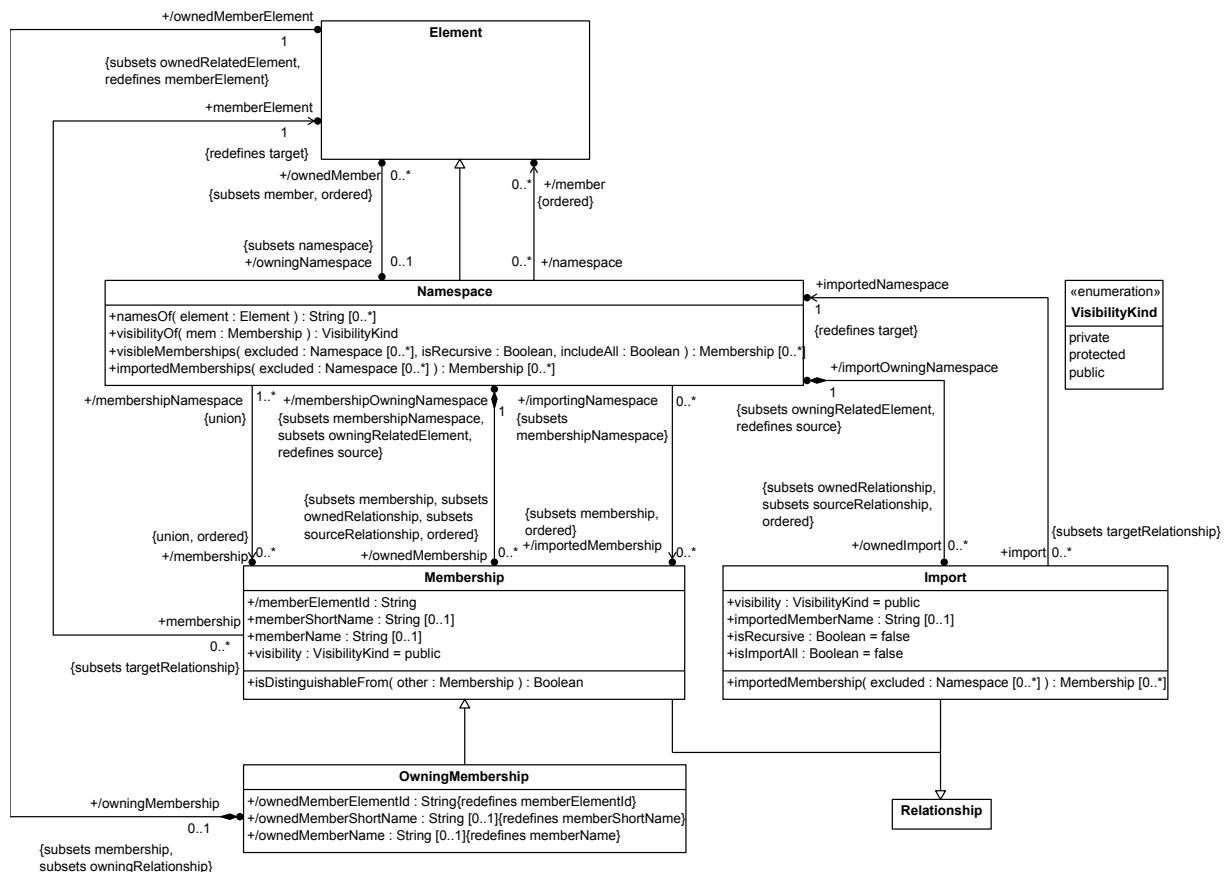


Figure 4. Namespaces

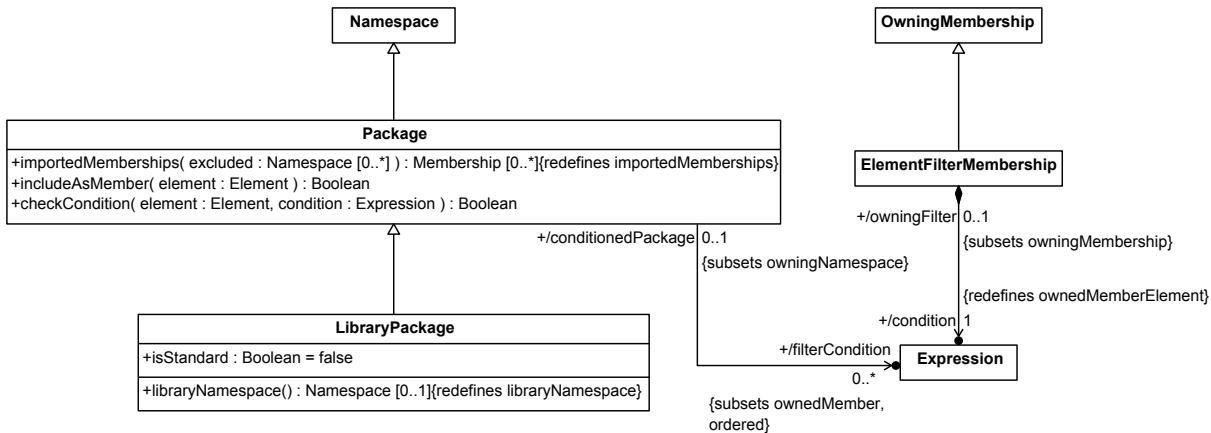


Figure 5. Packages

7.4.3 Notation

For Namespaces and Packages Textual Notation BNF, see [8.2.2.4](#).

Declarations and Bodies

Namespaces in SysML include Packages and all kinds of SysML Definitions and Usages (see [7.6](#) and following subclauses). All rules discussed below generically for Namespaces shall apply generally to Packages, Definitions and Usages (even though the examples in this subclause are given in terms of Packages). In addition, the KerML rules for name resolution shall also apply for the SysML textual concrete syntax (see [KerML, 8.2.3.3.4]).

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.

Namespace Members and Qualified Names

A Namespace may identify one or more names for some or all of its `member` Elements. The names of an `ownedMember` of a Namespace always include the `shortName` and `name` (or `effectiveName`, see [7.6.3](#)) of the Element, if any. A Namespace may also declare one or more *alias* names for its own `members`, or for `members` of other Namespaces. Therefore, an Element may have different names in different Namespaces, and the same name may identify different Elements relative to different Namespaces.

In general, then, to unambiguously identify an Element, the Element name must be *qualified* by the Namespace relative to which the Element name is to be resolved. Such a *qualified name* is notated by specifying a name to identify the Namespace, followed by the symbol `::`, followed by the Element name. Since the Namespace name may also be qualified, a qualified name is most generally a sequence of Element names separated by `::` punctuation, of which all but the last must identify Namespaces. An unqualified name can be considered the degenerate case of a qualified name with just one Element name in its sequence, for which the Namespace to be used is implicit.

Note that qualified names do not appear in the abstract syntax. Instead, the abstract syntax representation contains actual references to the identified Elements. *Name resolution* is the process of determining the Element that is identified by a qualified name. An unqualified name used within the body of a Namespace is resolved in the context of that Namespace and, potentially, other Namespaces in which the first Namespace is lexically nested, taking into account imported (see below) and inherited (see [7.6](#)) Memberships. A qualified name with more than one segment is

resolved by recursively resolving the name of the qualifying Namespace and then resolving the Element name in that context. The full name resolution process is specified in [KerML, 8.2.3.3.4].

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.

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 is given simply by the list of representations of its top-level Elements. A single modeling "project" may contain one or more root Namespaces (though the concept of a "project" is not formally defined in the SysML syntax).

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

While a root Namespace has no explicit owner, it is considered to be within the scope of a single *global namespace*. This global namespace may contain several root Namespaces (such as those being managed as a "project"), and always contains at least all of the KerML and SysML model libraries (see [KerML, Clause 9] and [Clause 9](#)). Any root Namespace within the global Namespace may refer to the name of a top-level Element of any other root Namespace using an unqualified name (since root Namespaces are themselves never named).

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 and `shortName` given for the Element (if any) becomes the `ownedMemberName` and `ownedMemberShortName` of the `OwningMembership`, respectively. 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 `memberShortName` and/or `memberName`, with a qualified name identifying the Element given after the keyword `for`. This denotes an `ownedMembership` of the containing Namespace that is *not* an `OwningMembership`, with the identified Element as its `memberElement`. The `visibility` of the Membership can be specified as for an `ownedMember`. An alias declaration may also optionally have a body containing `AnnotatingElements` owned by the `OwningMembership` for the alias via Annotation Relationships (see [8.2.2.3.1](#)).

```
package P1 {
    item A;
    item B;
    alias <C> CCC for B {
        doc /* Documentation of the alias. */
    }
    private alias D for B;
}
```

Imports

An ownedImport of a Namespace is denoted using the keyword **import** followed by a qualified name, optionally suffixed by "`::*`" and/or "`::**`".

If the qualified name in an **import** does *not* have any suffix, then this specifies an Import whose `importedNamespace` is the qualification part of the qualified name and whose `importedMemberName` is given by 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 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 suffixed 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 suffixed 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**. An Import declaration may also optionally have a body containing AnnotatingElements owned by the Import via Annotation Relationships (see [8.2.2.3.1](#)).

```

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

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

```

An Import may also be declared with one or more `filterConditions`, given as model-level evaluable Boolean Expressions (see [KerML, 7.4.9]), 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 Comment (see [7.3](#)), including Documentation, 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. */

    doc P9_Doc
        /* This is documentation about package P9. */
}

```

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 element 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 Kernel 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 and
        @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.

In general, a *library package* is a package that is expected to be commonly available and reused across many user models. A package can be explicitly identified as a library package using the keyword **library**. This allows tooling to identify any element contained directly in a library package as being a *library element* from that specific library package.

```
library package Configurations {
    ...
}
```

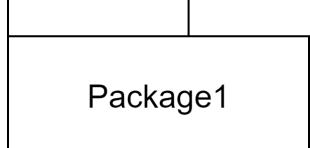
The *standard library packages* in the Systems Model Library and Domain Model Libraries (see [Clause](#)) are further identified using the keyword **standard**. However, only library packages from these libraries, the Kernel Model Libraries [KerML], or from other recognized standard model libraries, should be identified as standard library packages.

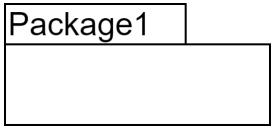
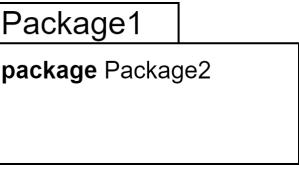
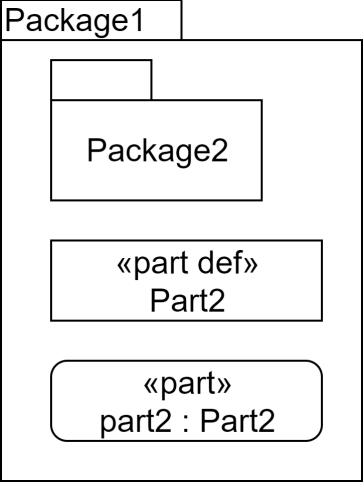
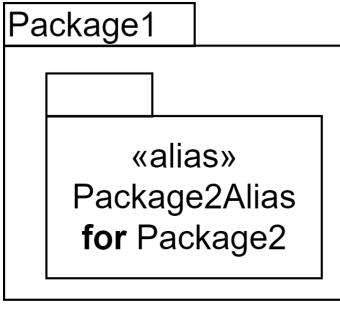
Graphical Notation

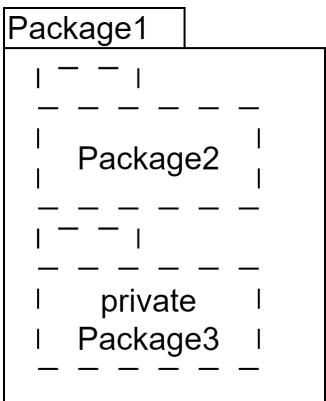
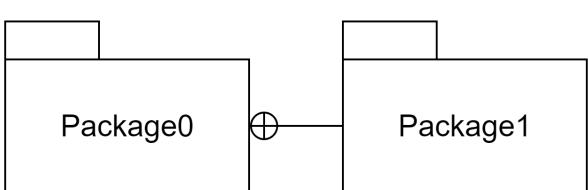
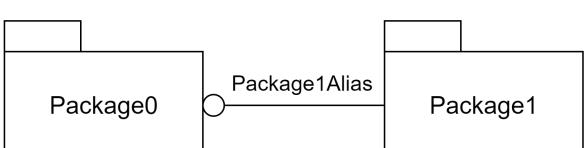
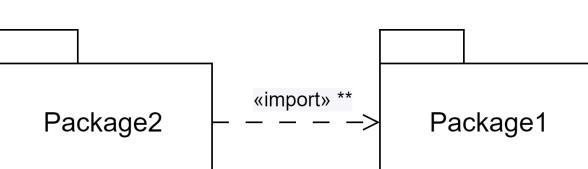
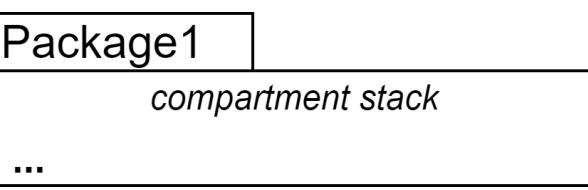
For Namespaces and Packages Graphical Notation BNF, see [8.2.3.4](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 8. Packages - Representative Notation

Element	Graphical Notation	Textual Notation
Package (name in body)		package Package1;

Element	Graphical Notation	Textual Notation
Package (name in tab)		<pre>package Package1;</pre>
Package with owned package		<pre>package Package1 { package Package2; }</pre>
Package with owned members		<pre>package Package1 { package Package2; part def Part2; part part2 : Part2; }</pre>
Package with alias member (unowned)		<pre>package Package1 { package Package2; alias Package2Alias for Package2; }</pre>
Package with alias member (owned)		<pre>package Package1 { package Package2; alias Package2Alias for Package2; }</pre>

Element	Graphical Notation	Textual Notation
Package with imported package (nested notation)	 <pre> classDiagram package Package1 { package Package2 { private package Package3 } } </pre>	<pre> package Package1 { import Package2::*; private import Package3::*; } </pre>
Membership (owned member)		<pre> package Package0 { package Package1; } </pre>
Membership (unowned member with alias name)		<pre> package Package0 { package Package1; alias Package1Alias for Package1; } </pre>
Import (recursive) Note: - no star is element import - single star is package import (content of package) - double star is recursive including outer package		<pre> package Package2 { import Package0::Package1::*; } </pre>
Package with compartment		<pre> package Package1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Package with members compartment	<p>Package1</p> <p>members</p> <p>part def PartDef1</p> <p>part def PartDef2</p> <p>part part1 : PartDef1</p> <p>part part2 : PartDef2</p>	<pre>package Package1 { part def PartDef1; part def PartDef2; part part1 : PartDef1; part part2 : PartDef2; }</pre>

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

For Dependencies Abstract Syntax metaclass descriptions, see [8.3.5](#).

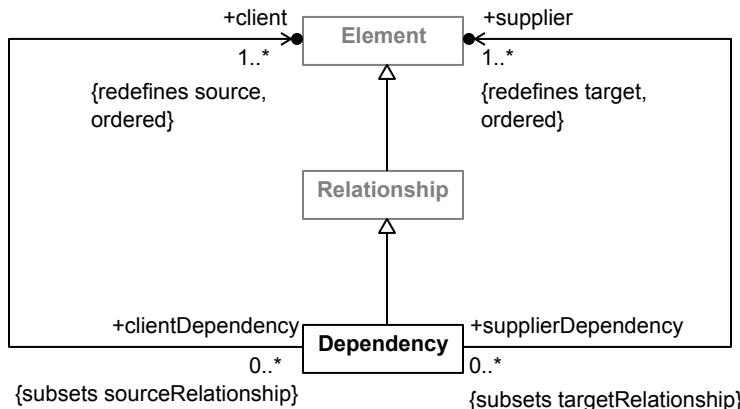


Figure 6. Dependencies

7.5.3 Notation

Textual Notation

For Dependencies Textual Notation BNF, see [8.2.2.5](#).

A Dependency is declared using the keyword **dependency**, optionally followed by a `shortName` and/or `name` (see [7.2](#)). The `client` Elements of the Dependency are then given as a comma-separated list of qualified names following the keyword `from`, followed by a similar list of the `supplier` Elements after the keyword `to`. If no `shortName` or `name` is given for the Dependency, then the keyword `from` may be omitted. An Dependency declaration may also optionally have a body containing AnnotatingElements owned by the Import via Annotation Relationships (see [8.2.2.3.1](#)).

```
dependency Use
  from 'Application Layer' to 'Service Layer';

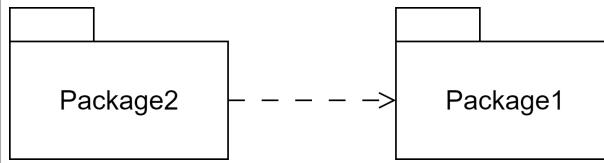
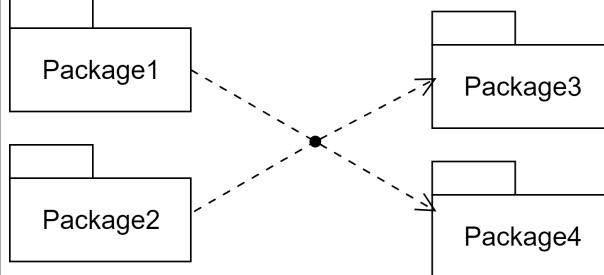
dependency 'Service Layer'
  to 'Data Layer', 'External Interface Layer' {

    /* 'Service Layer' is the client of this
     * Dependency, not its name. */
  }
```

Graphical Notation

For Dependencies Graphical Notation BNF, see [8.2.3.5](#).

Table 9. Dependencies - Representative Notation

Element	Graphical Notation	Textual Notation
Dependency		dependency Package1 to Package2;
Dependency - nary		dependency Package1, Package2 to Package3, Package4;

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, concerns, cases, analysis cases, verification cases, use cases, views, 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.

Release 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

For Definition and Usage Abstract Syntax class descriptions, see [8.3.6](#).

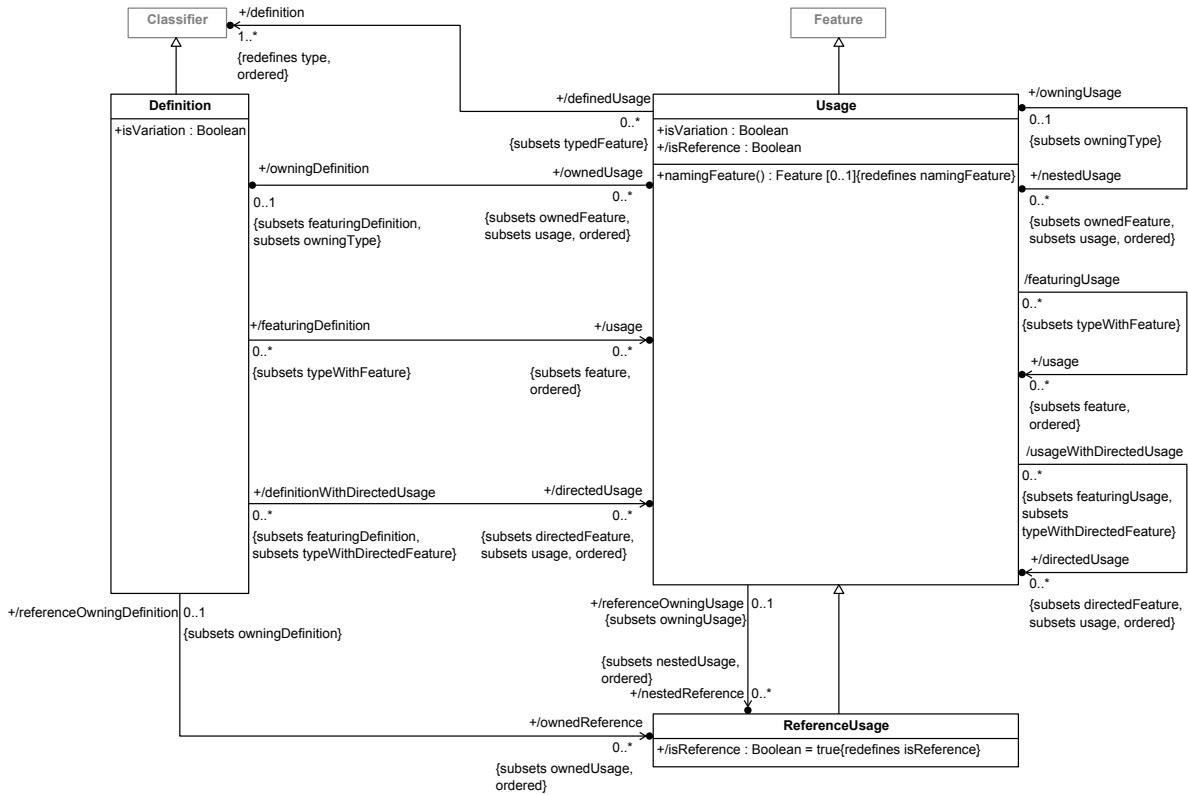


Figure 7. Definition and Usage

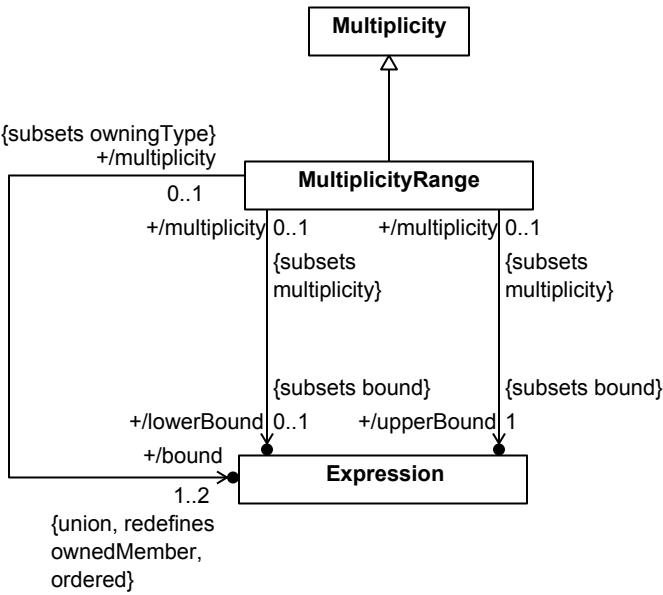


Figure 8. Multiplicities

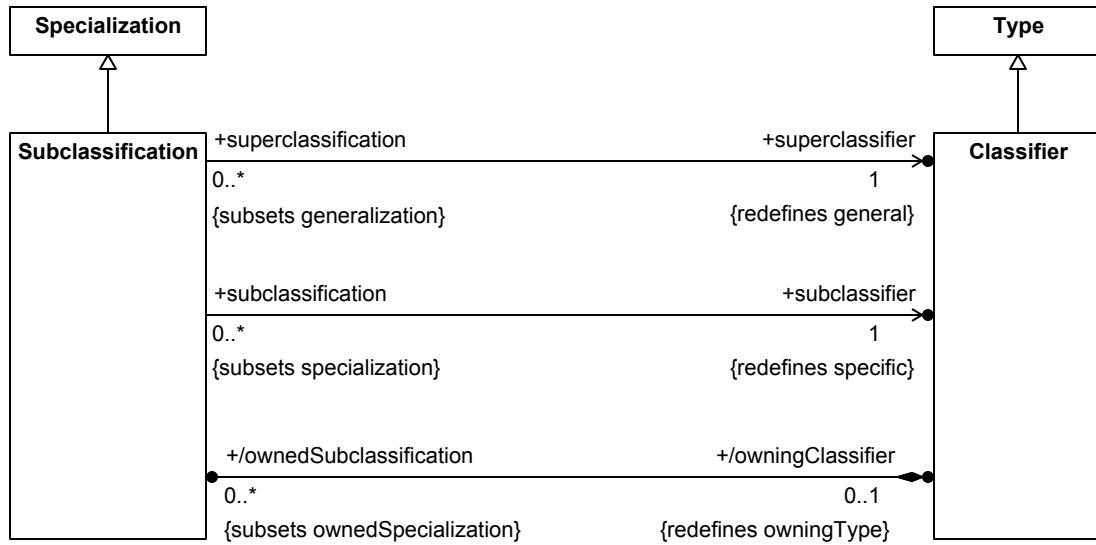


Figure 9. Classifiers

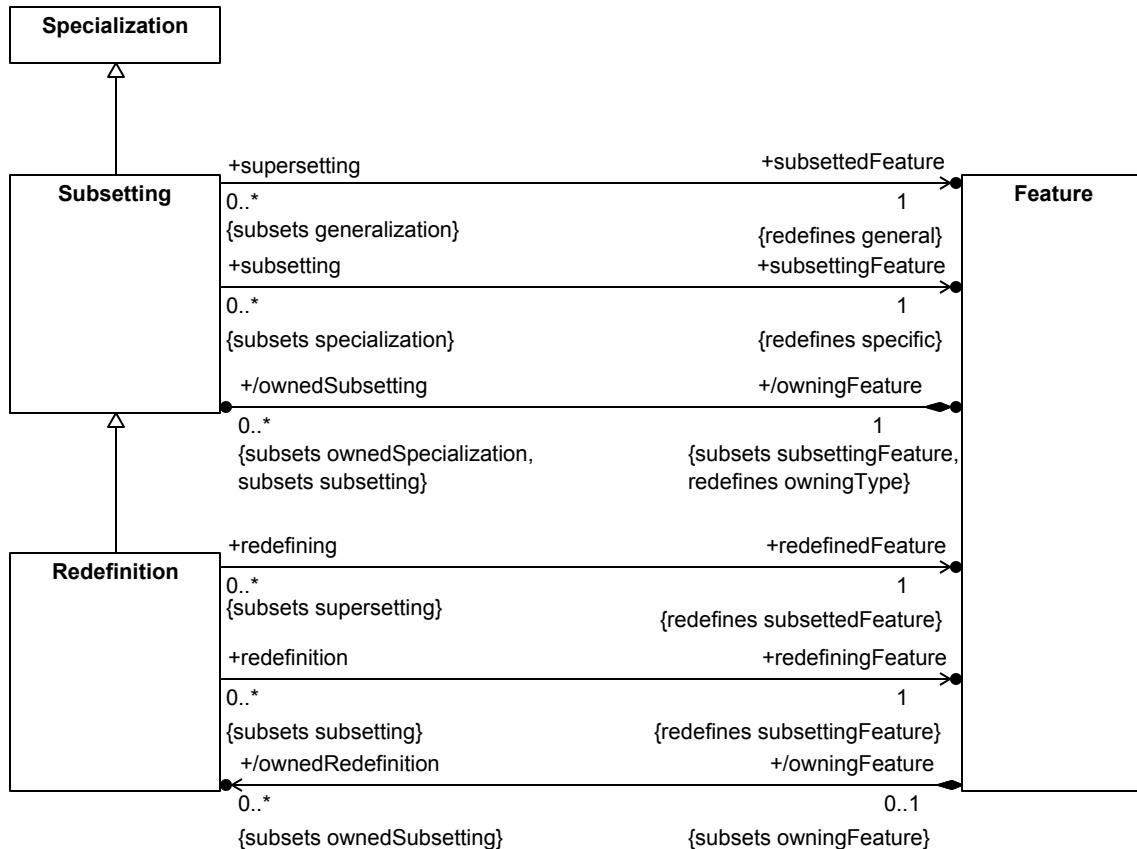


Figure 10. Subsetting

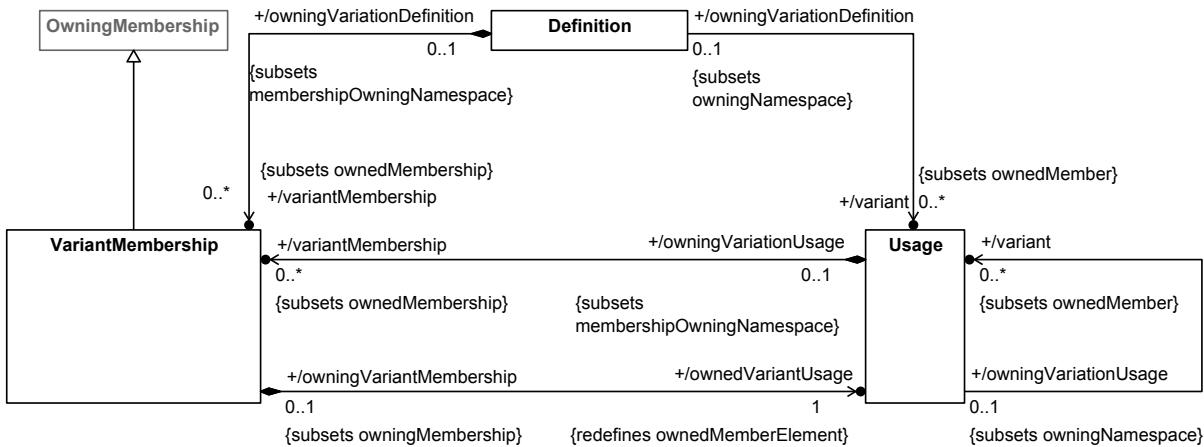


Figure 11. Variant Membership

7.6.3 Notation

For Definition and Usage Textual Notation BNF, see [8.2.2.6](#).

Definitions

Definition is an abstract Element in the SysML syntax, and only specific specialized kinds of Definition can be instantiated in the language. However, there is a basic common notation for all kinds of Definitions, as described here in this subclause, with additional special notations for certain kinds of Definitions described in the later subclauses specifically related to those kinds of Definitions.

As a kind of Namespace (see [7.4](#)), the representation of a Definition includes a *declaration* and a *body*.

A Definition is declared using a keyword specific to the kind of Definition (e.g., **item**, **part**, **action**, etc.) followed by the keyword **def**, optionally followed by a `shortName` and/or `name`. One or more `ownedSubclassifications` may also optionally be included in the declaration of a Definition by giving a comma-separated list of qualified names of the general Definitions after the keyword **specializes** (or the symbol `:>`). A Definition is specified as abstract (`isAbstract = true`) by placing the keyword **abstract** before its kind keyword.

```
abstract part def Vehicle;
part def Automobile specializes Vehicle;
part def Truck :> Vehicle;
```

The body of a Definition is specified generically as for any Namespace, by listing the members and imports between curly braces `{ ... }` (see [7.4](#)), or alternatively by a semicolon `;` if it has no members or imports. Usages declared within the body of a Definition are `ownedUsages` that specify *features* of the Definition.

```
item def Super {
    private package N {
        item def Sub specializes Super;
    }
    item f : N::Sub;
}
```

Usages

Usage is an abstract Element in the SysML syntax, and only specific specialized kinds of Usage can be instantiated in the language. However, there is a basic common notation for all kinds of Usages, as described here in this subclause, with additional special notations for certain kinds of Usage described in the later subclauses specifically related to those kinds of Usages.

As a kind of Namespace (see [7.4](#)), the representation of a Usage includes a *declaration* and a *body*.

A Usage is declared using a keyword specific to the kind of Usage (e.g., **item**, **part**, **action**, etc.), optionally followed by a `shortName` and/or `name`. One or more `ownedSpecializations` may also optionally be included in the declaration of a Usage. There are three kinds `ownedSpecializations` for Usages:

1. `ownedFeatureTypings` specify the `definitions` of a Usage (also known as its *types*). They are declared by giving a comma-separated list of the qualified names of the Definition elements after the keyword **defined by** (or the symbol `:`). The Definitions given must be consistent with the kind of Usage being defined.
2. `ownedSubsettings` specify other Usages subsetted by the owning Usage. They are declared by giving a comma-separated list of the qualified names of the subsetted Usages after the keyword **subsets** (or the symbol `:>`).
3. `ownedRedefinitions` specify other Usages redefined by the owning Usage. They are declared by giving a common-separated list of the qualified names of the redefined Usages after the keyword **redefines** (or the symbol `:>>`). (Note Redefinition is a kind of Subsetting, so the `ownedRedefinitions` of a Usage will be a subset of the `ownedSubsettings` in the abstract syntax.)

```
item x : A, B :> f :>> g;  
  
// Equivalent declaration:  
item x redefines g defined by A subsets f defined by B;
```

The `multiplicity` of a Usage can be given in square brackets `[...]` after the identification part of the declaration of a Usage (if any) or after one of the `ownedSpecialization` clauses in the declaration (but, in all cases, only one multiplicity may be specified). (This allows for a notation style consistent with the of previous modeling languages, in which the multiplicity is always placed after the declared type.)

A MultiplicityRange is written in the form `[lowerBound..upperBound]`, where each of `lowerBound` and `upperBound` is either a literal or the identification of a Usage (by qualified name or feature chain). Literals can be used to specify a MultiplicityRange with fixed lower and/or upper bounds. The values of the bounds of a MultiplicityRange shall be natural numbers. If only a single bound is given, then the value of that bound is used as both the lower and upper bound of the range, unless the result is the infinite value `*`, in which case the lower bound is taken to be 0. If two bounds are given, and the value of the first bound is `*`, then the meaning of the MultiplicityRange is not defined.

```
item def Person {  
    ref item parent[2] : Person;  
    ref item mother : Person[1..1] subsets parent;  
    attribute numberOfChildren : Natural;  
    ref item children[0..numberOfChildren] : Person;  
}  
item def ChildlessPerson specializes Person {  
    ref item redefines children[0];  
}
```

A MultiplicityRange may be optionally followed by one or both of the following keywords (in either order), or they can be used without giving an explicit MultiplicityRange, at any place in the declaration a multiplicity would be allowed.

- **nonunique** – Specifies that no two values of the Usage may be the same (`isUnique = false`, the default is `true`).
- **ordered** – Specifies that the values of the Usage are ordered (`isOrdered = true`, the default is `false`). The values of an ordered Usage may index by integers starting from 1.

If a MultiplicityRange is not declared for a Usage, then the Usage inherits the multiplicity constraints of any other Usages it subsets or redefines. If no tighter constraint is inherited, the effective default is the most general multiplicity `[0..*`] (the multiplicity of the most general feature *things* from the *Base Kernel Library* model). However, a tighter default of `[1..1]` is implicitly declared for the Usage if all of the following conditions hold:

1. The Usage is an AttributeUsage, an ItemUsage, a PartUsage (but not a ConnectionUsage), or a PortUsage.
2. The Usage is owned by a Definition or another Usage (not a Package).
3. The Usage does not have any *explicit* ownedSubsettings or ownedRedefinitions.

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

1. **in, out or inout** – Specifies that the Usage is a *directed feature* with the indicated direction.
2. **abstract** – Specifies that the Usage is *abstract* (`isAbstract = true`).
3. **readonly** – Specifies that the Usage is *read only* (`isReadOnly = true`).
4. **derived** – Specifies that the Usage is *derived* (`isDerived = true`).
5. **end** – Specifies that the Feature is an *end feature* (`isEnd = true`).
6. **ref** – Specifies that the Feature is a *referential (non-composite) feature* (`isReference = true`, `isComposite = false`).

Note. A directed feature is always considered referential, whether or not the keyword **ref** is also given implicitly in its declaration.

```
abstract part def Container {
    abstract ref item content;
}
part def Tank :> Container {
    in item fuelFlow : Fuel;
    ref item fuel : Fuel :>> content;
}
```

A ReferenceUsage is a Usage that is declared without any kind keyword. Unlike other kinds of Usages, the definitions (types) of a ReferenceUsage are not restricted to be of a particular kind. The declaration of a ReferenceUsage may, but is not required, to include the **ref** keyword. However, a ReferenceUsage is always, by definition, referential. A ReferenceUsage is otherwise declared like any other Usage, as given above.

```
abstract part def Container {
    abstract ref content : Base::Anything;
}
```

The body of a Usage is specified generically as for any Namespace, by listing the members and imports between curly braces `{...}` (see [7.4](#)), or alternatively by a semicolon ; if it has no members or imports. Usages declared within the body of another Usage are `nestedUsages` that specify *features* of the owning Usage.

```

part vehicle : Vehicle {
    part wheelAssembly[2] {
        part axle : Axle;
        part wheel : Wheel;
    }
}

```

Effective Names

It is actually the `effectiveName` of an Element that is used as its `ownedMemberName` (see [7.4.3](#)) and, therefore, in the name resolution process (see [KerML, 8.2.3.3.4]). By default, the `effectiveName` of an Element is the same as its name. However, if a name is not given in the declaration of a Usage with an `ownedRedefinition`, then, its `effectiveName` is implicitly set to the `effectiveName` of the `redefiningFeature` of its first `ownedRedefinition` (which may itself be an implicit name, if the `redefinedFeature` is itself a `redefiningFeature`). This is useful for constraining a `redefinedFeature`, while maintaining the same naming.

```

part def Engine {
    part cylinders : Cylinder[2...*];
}

part def FourCylinderEngine {
    // This redefines Engine::cylinders with a
    // new Usage of the, restricting the
    // multiplicity to 4. It's name is empty,
    // but its effective name is "cylinders".
    part redefines cylinders[4];
}

part def SixCylinderEngine {
    part redefines cylinders[6];
}

```

Certain other kinds of Usages (such as `PerformActionUsage` and its specializations) specify an alternate `effectiveName` rule, as described in the subclauses relevant to those Usages (e.g., [7.16.3](#) on action notation includes the rule for `PerformActionUsage`).

Variations and Variants

A Definition or Usage is specified as a *variation* (`isVariation = true`) by placing the keyword **`variation`** before its `kind` keyword. A variation is always abstract anyway, so the **`abstract`** keyword is shall not be used on a variation.

All Usages declared within the body of a variation Definition or Usage shall be declared as *variant* Usages by placing the keyword **`variant`** at the beginning of its declaration. Variant Usages shall only be declared within a variation. The kind of a variant Usage shall be consistent with the kind of its owning variation.

```

variation part def TransmissionChoices :> Transmission {
    variant part manual : ManualTransmission;
    variant part automatic : AutomaticTransmission;
}

```

A non-variant Usage can also be declared to act as a variant of a variation by not including a `kind` keyword in the variant declaration and, instead, following the **`variant`** keyword with the identification of a separately declared Usage (by qualified name or feature chain). Such a variant declaration may also optionally further constraint the variant Usage by including a multiplicity and/or further specializations.

```

part smallEngine : FourCylinderEngine;
part bigEngine : SixCylinderEngine;

```

```

part def Vehicle {
    part engine : Engine {
        variant smallEngine;
        variant bigEngine;
    }
}

```

Feature Chains

A *feature chain* is a sequence of two or more qualified names separated by dot (.) symbols. Each qualified name in a feature chain shall resolve to a Feature (generally a Usage in SysML). The first qualified name in a feature chain shall be resolved in the local Namespace as usual (see [7.4.3](#)). Subsequent qualified names shall then be resolved using the previously resolved Feature as the context Namespace, but considering only public Memberships.

A feature chain is similar to a qualified name but, unlike a qualified name, the path of features in the chain is recorded in the abstract syntax, not just the reference to the final Feature. This means that different paths to the same Feature can be distinguished in the abstract syntax representation of a model. Feature chains can be used to specify the target for most kinds of relationships involving Features, including Subsetting and Redefinition. However, their use is particularly important when specifying `relatedFeatures` of a `ConnectionUsage` that are more deeply nested than the `ConnectionUsage` itself (see [7.13](#)). (See also [KerML, 7.3.4.6].)

```

item uncles subsets parents.siblings;
item cousins redefines parents.siblings.children;
connect vehicle.wheelAssembly.wheels to vehicle.road;

```

In following subclauses, when the notation calls for the *identification* of a Usage, this can be done by using a qualified name or a feature chain.

Implicit Specializations

Every Definition shall directly or indirectly specialize the most general Classifier *Anything* from the *Base Kernel Library* model (see [KerML]), and every Usage shall directly or indirectly specialize the most general Feature *things* from the same library model. However, specific kinds of Definition and Usage have more specific requirements for what library Elements they must specialize, known as their *base* Definitions and Usages. If a Definition or Usage, as explicitly declared, does not directly or indirectly specialize its base Definition or Usage, then the declaration shall be considered to include an *implicit* Subclassification or Subsetting to the appropriate base library Element.

The base Element for a `ReferenceUsage` is the most general Feature *things* from the *Base Kernel Library* model (see [KerML]). The base Elements for other kinds of Definitions and Usages are identified in the following subclauses describing those various kinds.

```

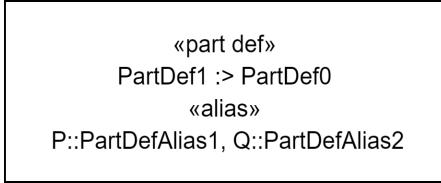
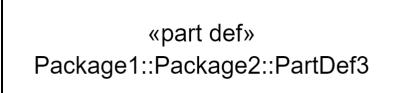
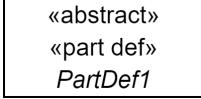
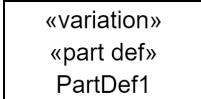
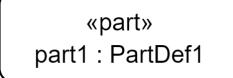
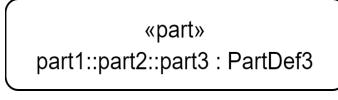
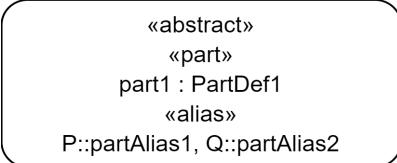
part def P {      // Implicitly specializes Parts::Part.
    ref x;        // Implicitly specializes Base::things.
    part p;        // Implicitly specializes Parts::parts.
    part q :> p; // No implicit specialization.
}
part def Q :> P; // No implicit specialization.

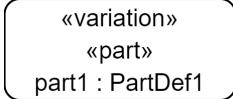
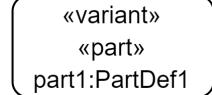
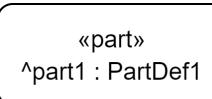
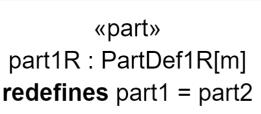
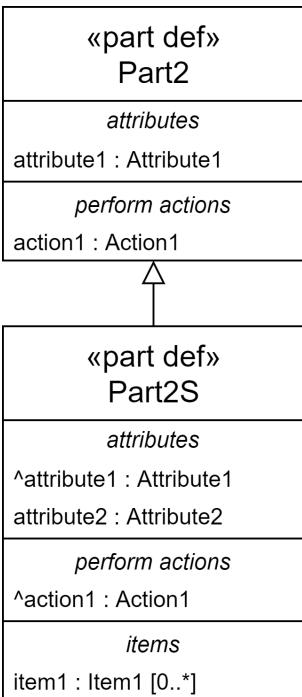
```

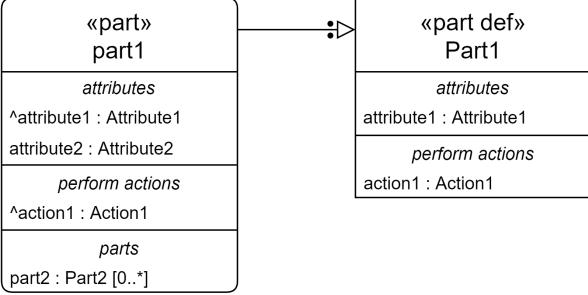
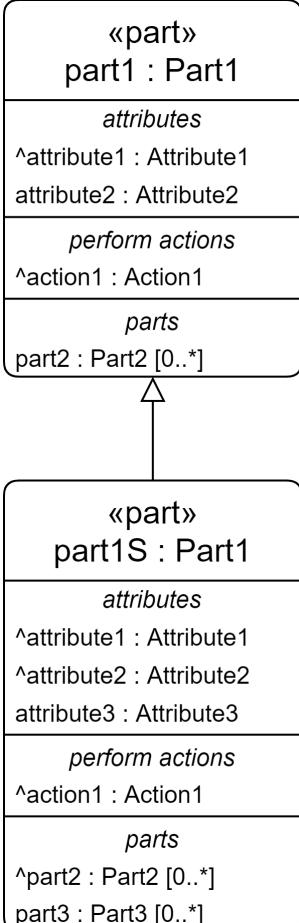
Graphical Notation

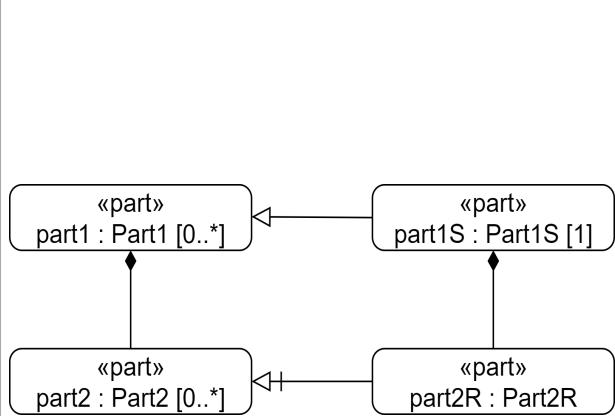
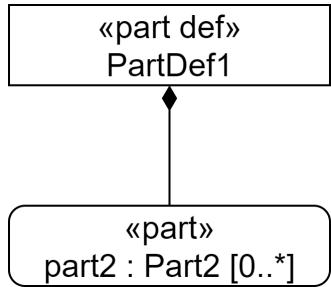
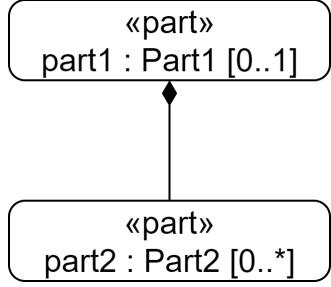
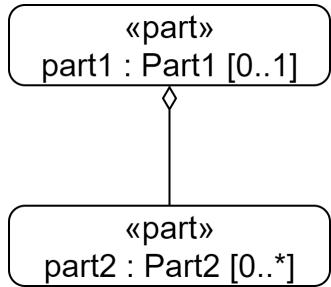
For Definition and Usage Graphical Notation BNF, see [8.2.3.6](#).

Table 10. Definition and Usage - Representative Notation

Element	Graphical Notation	Textual Notation
Name Compartment - Definition		part def PartDef1;
Name Compartment - Definition		abstract part def PartDef1 :> PartDef0; package P { alias PartDefAlias1 for PartDef1; } package Q { alias PartDefAlias2 for PartDef1; }
Name Compartment - Definition (qualified name)		package Package1 { package Package2 { part def PartDef3; } }
Name Compartment - Definition (abstract)		abstract part def PartDef1;
Name Compartment - Definition (variation)		variation part def PartDef1;
Name Compartment - Usage		part part1 : PartDef1;
Name Compartment - Usage		part part1 { part part2 { part part3 : PartDef3; } }
Name Compartment - Usage (abstract)		abstract part part1 : PartDef1;

Element	Graphical Notation	Textual Notation
Name Compartment - Usage (variation)		variation part part1 : PartDef1;
Name Compartment - Usage (variant)		variant part part1 : PartDef1;
Name Compartment - Inherited Usage		No textual notation.
Name Compartment - Subsetted Usage		part part1S : PartDef1S [m] subsets part1;
Name Compartment - Redefined Usage		part part1R : PartDef1R [m] redefines part1 = part2;
Subclassification	 <pre> «part def» Part2 ----- attributes attribute1 : Attribute1 ----- perform actions action1 : Action1 ----- ↑ «part def» Part2S ----- attributes ^attribute1 : Attribute1 attribute2 : Attribute2 ----- perform actions ^action1 : Action1 ----- items item1 : Item1 [0..*] </pre>	<pre> part def Part2 { attribute attribute1 : Attribute1; perform action action1 : Action1; } part def Part2S :> Part2 { attribute attribute2 : Attribute2; item item1 : Item1 [0..*]; } or part def Part2S specializes Part2 { ... } </pre>

Element	Graphical Notation	Textual Notation
Usage Definition	 <pre> <<part>> part1 attributes ^attribute1 : Attribute1 attribute2 : Attribute2 perform actions ^action1 : Action1 parts part2 : Part2 [0..*] </pre> <pre> <<part def>> Part1 attributes attribute1 : Attribute1 perform actions action1 : Action1 </pre>	<pre> part def Part1 { attribute attribute1 : Attribute1; perform action action1 : Action1; } part part1 : Part1 { attribute attribute2 : Attribute2; part part2 : Part2 [0..*]; } or part part1 defined by Part1 { ... } </pre>
Subsetting	 <pre> <<part>> part1 : Part1 attributes ^attribute1 : Attribute1 attribute2 : Attribute2 perform actions ^action1 : Action1 parts part2 : Part2 [0..*] </pre> <pre> <<part>> part1S : Part1 attributes ^attribute1 : Attribute1 ^attribute2 : Attribute2 attribute3 : Attribute3 perform actions ^action1 : Action1 parts ^part2 : Part2 [0..*] part3 : Part3 [0..*] </pre>	<pre> part part1 : Part1 { attribute attribute1 : Attribute1; part part2 : Part2 [0..*]; } part part1S :> part1 { attribute attribute3 : Attribute3; part part3 : Part3 [0..*]; } or part part2S subsets part1 { ... } </pre>

Element	Graphical Notation	Textual Notation
Redefinition	 <pre> graph TD part1["<<part>> part1 : Part1 [0..*]"] <--> part1S["<<part>> part1S : Part1S [1]"] part2["<<part>> part2 : Part2 [0..*]"] <--> part2R["<<part>> part2R : Part2R"] </pre>	<pre> part part1 : Part1 [0..*] { part part2 : Part2 [0..*]; } part part1S : Part1S [1] >: part1 { part part2R : Part2R :>> part2; } or part part1S : Part1S [1] subsets part1 { part part2R : Part2R redefines part2; } </pre>
Feature Membership (isComposite=true)	 <pre> graph TD PartDef1["<<part def>> PartDef1"] --> part2["<<part>> part2 : Part2 [0..*]"] </pre>	<pre> part def PartDef1 { part part2 : Part2 [0..*]; } </pre>
Feature Membership (isComposite=true)	 <pre> graph TD part2["<<part>> part2 : Part2 [0..*]"] --> part1["<<part>> part1 : Part1 [0..1]"] </pre>	<pre> part part1 : Part1 [0..1] { part part2 : Part2 [0..*]; } </pre>
Feature Membership (isComposite=false)	 <pre> graph TD part2["<<part>> part2 : Part2 [0..*]"] --> part1["<<part>> part1 : Part1 [0..1]"] </pre>	<pre> part part1 : Part1 [0..1] { ref part part2 : Part2 [0..*]; } </pre>

Element	Graphical Notation	Textual Notation
Variant Membership	<pre> «variation» «part» part1 : Part1 + «variant» «part» part1a : Part1 + «variant» «part» part1b : Part1 </pre>	<pre> variation part part1 : Part1 { variant part part1a : Part1a; variant part part1b : Part1b; } </pre>
Variants Compartment	<pre> variants part part1 : PartDef1 ref item item1 : ItemDef1 ... </pre>	<pre> variation item itemChoice : ItemDef { variant part1; variant item1; } </pre>
Variant Parts Compartment	<pre> variant parts part1 : PartDef1 ref part2 : PartDef2 ... </pre>	<pre> variation part partChoice : PartDef { variant part part1 : PartDef1; variant part part2 : PartDef2; } </pre>
Relationships Compartment	<pre> relationships defined by PartDef1 subsets part1 connect to part3 ... </pre> <pre> relationships defines part1 specializes PartDef1 connect to part3 ... </pre>	No specific textual notation

Compartments

The graphical notation for a definition or usage may include one or more compartments, which shows member elements (if any) using textual or graphical notation.

[Table 11](#) identifies representative compartments for each kinds of definition or usage element. For example, an action definition or usage can contain compartments for its nested actions, performed actions, allocations, analysis, and many other compartments. In additions, member and packages compartments are also allow in package symbols (see [7.4](#) on packages).

In the graphical symbols in all Representative Notation tables in [Clause 7](#), the *compartment stack* is a placeholder for any valid compartment for the model element that is described in the respective row from [Table 11](#).

Table 11. Representative Compartment Matrix

Compartment	Definition/Usage Element																				Package	
	Action	Allocation	Analysis	Attribute	Calc	Connection	Constraint	Enumeration	Individual	Interface	Item	Occurrence	Part	Port	Requirement	Snapshot	State	Timeslice	Use Case	Verification	View	Viewpoint
actions	X		X	X	X	X			X	X		X	X			X	X	X	X			
perform actions						X						X			X		X					
state actions																	X					
include use cases																			X			
actors			X												X			X	X		?	
allocations	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
analyses	X		X		X												X					
attributes	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
calcs																						
connections					X				X	X	X					X		X				
constraints	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
assert constraints	X												X			X	X	X				
directed features												X	X									
documentation	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
ends		X			X				X							X		X				
enums								X														
exhibited by																	X					
exposes																			X			
filters																			X			
frames																				X		
individuals	X	X	X		X	X	X		X	X	X	X	X	X	X		X	X	X	X	X	
interfaces										X		X	X				X		X			
items	?									X	X	X				X		X				
flows											X						X		X			
members	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
metadata	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	
objective					X													X	X			
occurrences											X				X		X					

		Definition/Usage Element																						
Compartments		Action	Allocation	Analysis	Attribute	Calc	Connection	Constraint	Enumeration	Individual	Interface	Item	Occurrence	Part	Port	Requirement	Snapshot	State	Timeslice	Use Case	Verification	View	Viewpoint	Package
packages																							x	
parameters	x	x	x	x	x							x			x	x	x			x				
parts												x	x		x		x							
performed by	u	u	u	u	u							x	x	x						u	u			
ports												x	x	x										
relationships	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
requirements														x										
result		x	x	x														x	x					
satisfies	x											x	x			x					x			
shapes												x	x			x		x						
snapshots	x	x	x		x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x		
stakeholders														x				x				?		
states									x	?	x	x			x	x	x							
exhibit states									?	x	x			x		x								
exhibited by															u									
subject		x												x			x		x	x				
successions																								
timeslices	x	x	x		x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x		
variants	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
variant element usages	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x		
verifications														x				x						
verifies																		x						
verification methods																		x			x			
views														x		x		x		x		x		
viewpoints														x		x			x	x				

Legend

- x – compartment is valid for both Definition and Usage Element, or for Package
- d – compartment is valid for Definition Element only
- u – compartment is valid for Usage Element only
- ? – to be determined

Release Note. All "to be determined" cases in the matrix will be resolved in the final submission.

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 [9.8](#)). 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

For Attributes Abstract Syntax class descriptions, see [8.3.7](#).

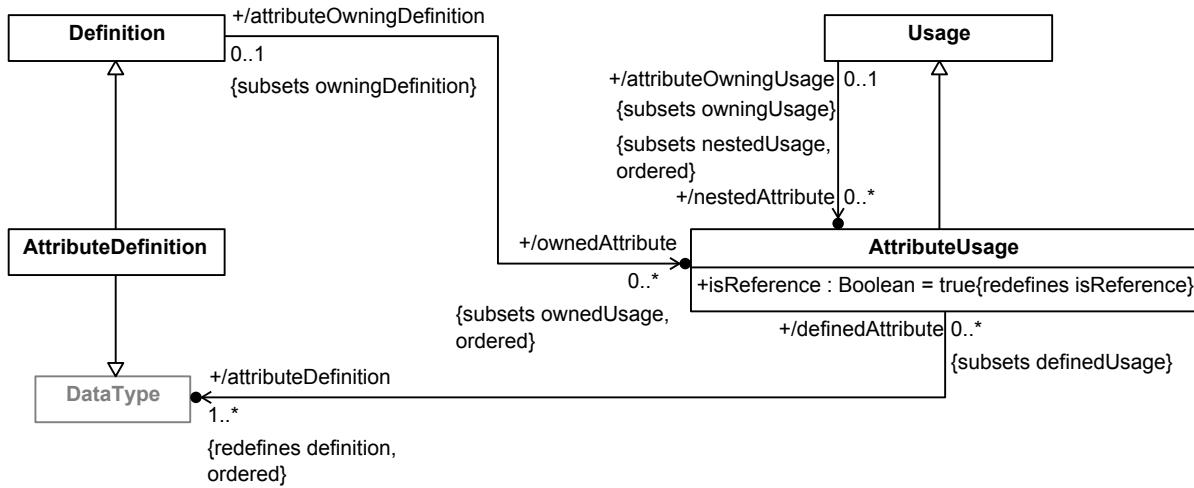


Figure 12. Attribute Definition and Usage

7.7.3 Notation

Textual Notation

For Attributes Textual Notation BNF, see [8.2.2.7](#)

AttributeDefinitions and AttributeUsages may be declared as described in [7.6.3](#), using the kind keyword **attribute**. An AttributeDefinition shall only specialize other AttributeDefinitions (including EnumerationDefinitions) or KerML DataTypes (see [KerML]). An AttributeUsage shall only be defined by AttributeDefinitions or KerML DataTypes. (See also [7.8](#) on EnumerationDefinitions and EnumerationUsages.)

The **ref** keyword may be used in the declaration of an AttributeUsage, but an AttributeUsage is always referential, whether or not **ref** is included in its declaration. The default multiplicity of an AttributeUsage shall be **[1..1]**, under the conditions described in [7.6.3](#). In general, any kind of Usage may be declared in the body of an AttributeDefinition or AttributeUsage, but all such Usages shall be referential. (There are further restrictions on what can be nested in an EnumerationDefinition – see [7.8](#).)

```

attribute def SensorRecord {
    ref part sensor : Sensor;
    attribute reading : Real;
}

```

The base AttributeDefinition is *AttributeValue* from the *Attributes* library model (see [9.2.2](#)), which is simply an alias for the DataType *DataValue* from the *Base Kernel Library* model (see [KerML]). The base AttributeUsage is *attributeValues* from the *Attributes* library model (see [9.2.2](#)), which is simply an alias for the DataValue *dataValues* from the *Base Kernel Library* model (see [KerML]).

Graphical Notation

For Attributes Graphical Notation BNF, see [8.2.3.7](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 12. Attributes - Representative Notation

Element	Graphical Notation	Textual Notation
Attribute Definition		<pre>attribute def AttributeDef1; attribute def AttributeDef1 { /* members */ }</pre>
Attribute		<pre>attribute attribute1 : AttributeDef1; attribute attribute1 : AttributeDef1 { /* members */ }</pre>
Attributes Compartment	<pre> attributes ^attribute2 : AttributeDef2 attribute1 : AttributeDef1 [1..*] ordered nonunique attribute3R : AttributeDef3R redefines attribute3 attribute4R : AttributeDef4R >> attribute5 :>> attribute5 attribute6S : AttributeDef6S [m] subsets attribute6 attribute7S : AttributeDef7S [m] > attribute7 attribute8 : AttributeDef8 = expression1 attribute10 : AttributeDef10 := expression2 /attribute12 readonly attribute13 abstract attribute14 ... </pre>	<pre>{ 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.

Release Note. The restriction of enumerations to just attribute definitions may be removed in the final submission.

7.8.2 Abstract Syntax

For Enumerations Abstract Syntax class descriptions, see [8.3.8](#).

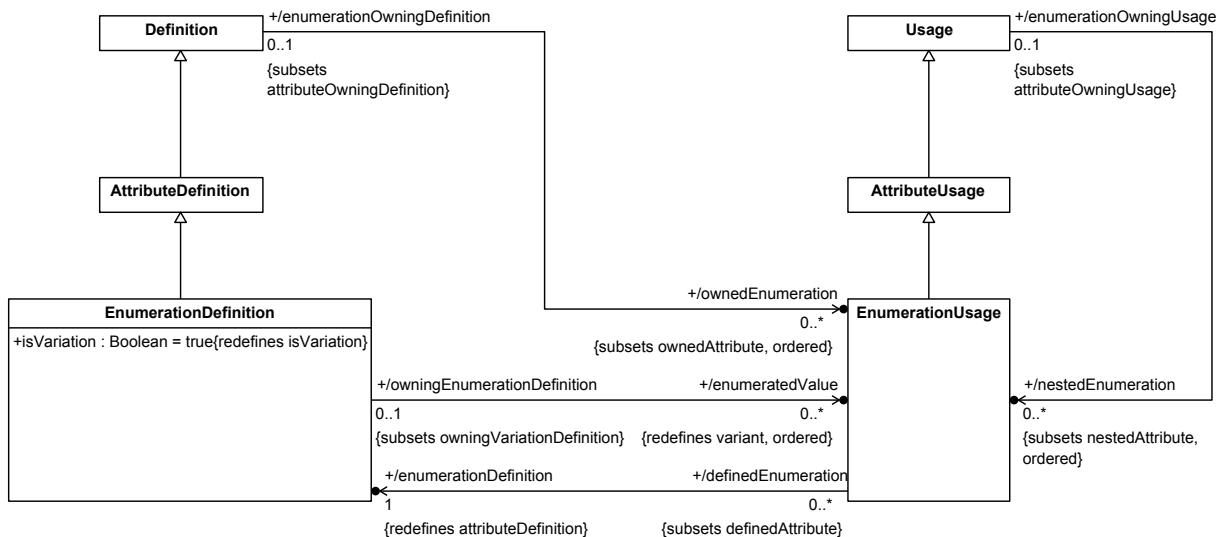


Figure 13. Enumeration Definition and Usage

7.8.3 Notation

Textual Notation

For Enumerations Textual Notation BNF, see [8.2.2.8](#).

EnumerationDefinitions and EnumerationUsages are declared as described in [7.6.3](#), using the kind keyword **enum**, with the additional restrictions described below. As a kind of AttributeDefinition, an EnumerationDefinition may generally subclassify other AttributeDefinitions or KerML DataTypes. However, an EnumerationDefinition shall not subclassify another EnumerationDefinition.

Any `ownedMembers` declared in the body of an EnumerationDefinition shall be EnumerationUsages, which are the `enumeratedValues` of the EnumerationDefinition. An EnumerationDefinition is always considered to be a variation and its `enumeratedValues` are its variants. The keywords **abstract** and **variation** shall not be used with an EnumerationDefinition. The declaration of an EnumerationUsage as an `enumeratedValue` of an EnumerationDefinition shall not include the keyword **variant** nor any of the other property keywords given in [7.6.3](#) (i.e, the direction keywords, **abstract**, **variation**, etc.).

Since the body of an EnumerationDefinition may only declare EnumerationUsages, the declaration of an `enumeratedValue` may omit the **enum** keyword. An EnumeratedUsage declared as an `enumeratedValue` shall be considered to be implicitly defined by its owning EnumerationDefinition and, therefore, shall not have any explicitly declared definition other than that EnumerationDefinition (and need not have any explicitly declared definition at all).

```
enum def ConditionColor { red; green; yellow; }
attribute def ConditionLevel {
    attribute color : ConditionColor;
}
enum def RiskLevel {
    import ConditionColor::.*;
    enum low { color = green; }
    enum medium { color = yellow; }
    enum high { color = red; }
}
```

There are no special restrictions on the declaration of an EnumerationUsage outside the body of an EnumerationDefinition, other than as for an AttributeUsage in general (see [9.2.2](#)), except that such an EnumerationUsage must be explicitly defined by a single EnumerationDefinition. As a kind of AttributeUsage, an EnumerationUsage is always referential, whether or not **ref** is included in its declaration, and all the `nestedUsages` of an EnumerationUsage shall be referential.

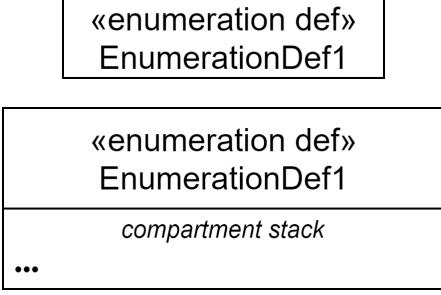
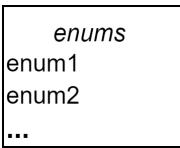
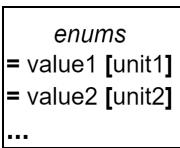
```
enum assessedRisk : RiskLevel {
    // The following feature is added for this usage.
    // It is legal, since all attribute usages are
    // referential.
    attribute assessment : String;
}
```

Graphical Notation

For Enumerations Graphical Notation BNF, see [8.2.3.8](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 13. Enumerations - Representative Notation

Element	Graphical Notation	Textual Notation
Enumeration Definition		<pre>enum def EnumerationDef1; enum def EnumerationDef1 { /* members */ }</pre>
Enums Compartment		<pre>enum def EnumerationDef1 { enum1; enum2; } or enum def EnumerationDef1 { enum1; enum2; }</pre>
Enums Compartment		<pre>enum def EnumerationDef1 { enum = value1 [unit1]; enum = value2 [unit2]; } or enum def EnumerationDef1 { = 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

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, which happens at the same time as the end snapshot of the previous ownership (or inventory) time slice.

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*, *Wheel*, 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

For Occurrences Abstract Syntax class descriptions, see [8.3.9](#).

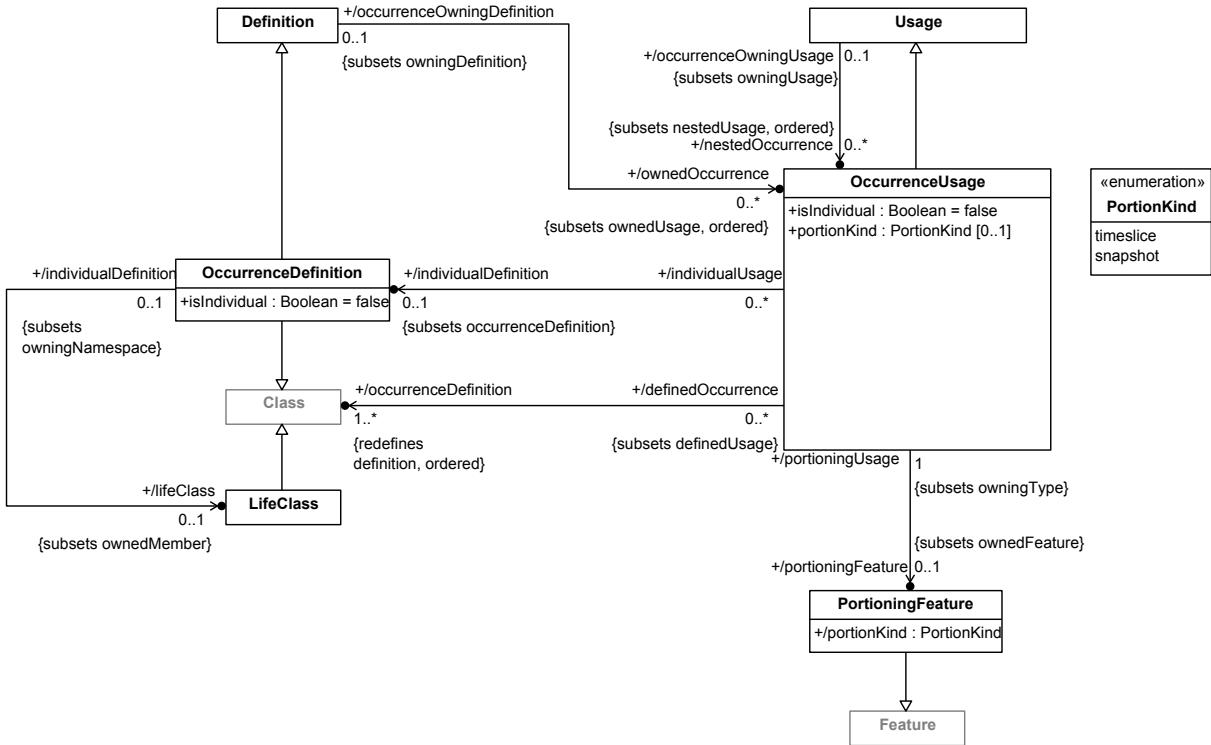


Figure 14. Occurrence Definition and Usage

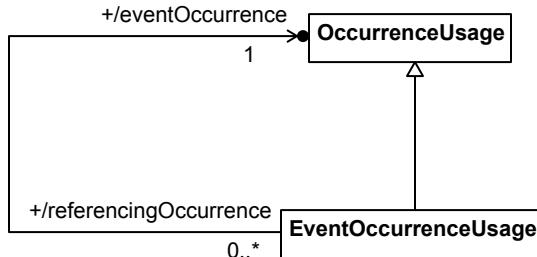


Figure 15. Event Occurrences

7.9.3 Notation

For Occurrences Textual Notation BNF, see [8.2.2.9](#).

Occurrences

An OccurrenceDefinition or OccurrenceUsage (that is not of a more specialized kind) can be declared as described in [7.6.3](#), using the kind keyword `occurrence`. An OccurrenceUsage shall only be defined by OccurrenceDefinitions (of any kind) or KerML Classes (see [KerML]).

The base Element for OccurrenceDefinitions is the Class `Occurrence` from the `Occurrences` Kernel Library model (see [KerML]). The base Element for OccurrenceUsages is the Feature `occurrences` from the `Occurrences` Kernel Library model (see [KerML]).

Time Slices and Snapshots

An OccurrenceUsage (of any kind) can be declared as a *time slice* (`portionKind = timeslice`) or a *snapshot* (`portionKind = snapshot`) using the keyword `timeslice` or `snapshot`, respectively, placed immediately before the kind keyword of the declaration (after any of the other Usage property keywords described in [7.6.3](#)). Alternatively, `timeslice` or `snapshot` may be used in place of the kind keyword, in which case the declaration is equivalent to `timeslice occurrence OR snapshot occurrence` (that is, an OccurrenceUsage not of a more specialized kind, but with the given `portionKind`).

```
occurrence def Flight {
    ref part aircraft : Aircraft;
}
// The following are time slices of Flight.
timeslice preflight : Flight;
timeslice inflight : Flight;
timeslice postflight : Flight;

part aircraft : Aircraft;
// The following are snapshots of the part aircraft.
snapshot part aircraftTakeOff :> aircraft;
snapshot part aircraftLanding :> aircraft;
```

An OccurrenceUsage with a non-null `portionKind` is normally considered to represent portions of instances of its defining OccurrenceDefinition(s). However, if such an OccurrenceUsage has no explicitly declared definition, but is declared in the body of an OccurrenceDefinition, then it is considered to implicitly subclassify the owning OccurrenceDefinition and, so, represent portions of instances of that OccurrenceDefinition. If it is declared in the body of another OccurrenceUsage, then it is considered to implicitly subset the owning OccurrenceUsage and, so, represent portions of the definition(s) of that OccurrenceUsage.

```
occurrence def Flight {
    ref part aircraft : Aircraft;
    // The following are time slices of Flight.
    timeslice preflight;
    timeslice inflight;
    timeslice postflight;
}

part aircraft : Aircraft {
    // The following are snapshots of the part vehicle.
    snapshot part aircraftTakeOff;
    snapshot part aircraftLanding;
}
```

Individuals

An OccurrenceDefinition (of any kind) can be declared as an *individual definition* (`isIndividual = true`) using the keyword `individual`, placed immediately before the kind keyword of the declaration. Alternatively, `individual` may be used in place of the kind keyword, in which case the declaration is equivalent to `individual occurrence` (that is, an OccurrenceUsage not of a more specialized kind, but with `isIndividual = true`).

```
individual def Flight_248 :> Flight;
individual part def TestPlane_1 :> Aircraft;
```

An OccurrenceUsage (of any kind) is considered to be an *individual usage* if it has a definition that is an individual definition. An OccurrenceUsage shall not have more than one definition that is an individual definition. An OccurrenceUsage may also be explicitly declared to be an individual usage using the keyword

individual, placed after any of the other Usage property keywords described in [7.6.3](#), but before a **timeslice** or **snapshot** keyword (if any). In this case, the OccurrenceUsage must have exactly one definition that is an individual definition. If the declaration of an OccurrenceUsage includes the the keyword **individual** (and, possibly, **timeslice** or **snapshot**), but no kind keyword, then this shall be equivalent to having used the **occurrence** keyword (that is, an OccurrenceUsage not of a more specialized kind, but with `isIndividual = true` and, possibly, a given `portionKind`).

```
individual flightRecord : Flight_248 {
    individual part redefines aircraft : TestPlane_1;
    individual timeslice redefines preflight;
    individual timeslice redefines inflight;
    individual timeslice redefines postflight;
}
```

Events

An EventOccurrenceUsage can be declared like an OccurrenceUsage, as described above (including possibly being an individual or a portion), but using the kind keyword **event occurrence** instead of just **occurrence**. The `eventOccurrence` of the EventOccurrenceUsage is given by the `ownedReferenceSubsetting` of the EventOccurrenceUsage, which is specified using the keyword **references** or the symbol `::>`, and which must be an OccurrenceUsage. If the EventOccurrenceUsage has no `ownedReferenceSubsetting`, then the `eventOccurrence` is the EventOccurrenceUsage itself.

```
part client {
    event occurrence request[1] references subscriptionMessage.source;
    event occurrence delivery[*] ::> publicationMessage.target;
}
```

An EventOccurrenceUsage may also be declared using just the keyword **event** instead of **event occurrence**. In this case, the declaration shall not include either a `shortName` or `name`. Instead, the `eventOccurrence` of the EventOccurrenceUsage is identified by giving a qualified name or feature chain immediately after the **event** keyword.

```
part client {
    event subscriptionMessage.source[1];
    event publicationMessage.target[*];
}
```

The **ref** keyword may be used in the declaration of an EventOccurrenceUsage, but an EventOccurrenceUsage is always referential, whether or not **ref** is included in its declaration.

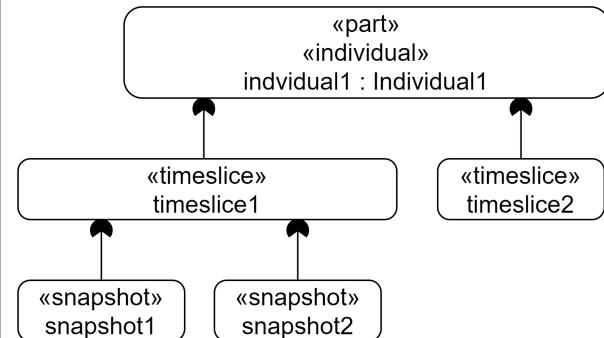
Graphical Notation

For Occurrences Graphical Notation BNF, see [8.2.3.9](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 14. Occurrences - Representative Notation

Element	Graphical Notation	Textual Notation
Occurrence Definition		<pre> occurrence def OccurrenceDef1; occurrence def OccurrenceDef1 { /* members */ } </pre>
Occurrence		<pre> occurrence occurrence1 : OccurrenceDef1; occurrence occurrence1 : OccurrenceDef1 { /* members */ } </pre>
Individual Occurrence Definition		<pre> individual def 'OccurrenceDef1-1' :> OccurrenceDef1; </pre>
Individual Occurrence		<pre> individual occurrence1 : 'OccurrenceDef1-1' :> occurrence1; </pre>
Timeslice		<pre> timeslice timeslice1 : OccurrenceDef1; </pre>
Snapshot		<pre> snapshot snapshot1 : OccurrenceDef1; </pre>

Element	Graphical Notation	Textual Notation
Portion Membership	 <pre> <<part>> <<individual>> individual1 : Individual1 +--> <<timeslice>> timeslice1 +--> <<timeslice>> timeslice2 +--> <<snapshot>> snapshot1 +--> <<snapshot>> snapshot2 </pre>	<pre> individual part individual1 : Individual1 { timeslice timeslice1 { snapshot snapshot1; snapshot snapshot2; } timeslice timeslice2; } </pre>
Occurrences Compartment	<pre> occurrences ^occur2 : OccurDef2 occur1 : OccurDef1 [1..*] ordered nonunique occur3R : OccurDef3R redefines occur3 occur4R : OccurDef4R :>> occur4 :>> occur5 occur6S : OccurDef6S [m] subsets occur6 occur7S : OccurDef7S [m] :> occur7 occur8R = occur8 ref occur9 : OccurDef9 occur 10 occur 10.1 occur 10.2 /occur11 readonly occur12 ... </pre>	<pre> { occurrence occur1 : OccurDef [1..*] ordered nonunique; /* ... */ } </pre>
Individuals Compartment (parts)	<pre> individual parts ^part2 : PartDef2_1 part1 : PartDef1_1 [1..*] ordered nonunique part3R : PartDef3R_1 redefines part3 part4R : PartDef4R_1 :>> part4 :>> part5 part6S : PartDef6S_1 [m] subsets part6 part7S : PartDef7S_1 [m] :> part7 part8R_1 = part8 ref part9 : PartDef9_1 part10 part10.1 part10.2 ... </pre>	<pre> { individual part part1 : PartDef1_1 [1..*] ordered nonunique; /* ... */ } </pre>

Element	Graphical Notation	Textual Notation
Timeslices Compartment	<p style="text-align: center;"><i>timeslices</i></p> state state1 ^part2_timeslice2 : PartDef2 part1_timeslice1 : PartDef1 [1..*] ordered nonunique part3R_timeslice3R : PartDef3R redefines part3 part4R_timeslice4R : PartDef4R >> part4 :>> part5 part6S_timeslice6S : PartDef6S [m] subsets part6 part7S_timelice7S : PartDef7S [m] >> part7 part8R_timeslice8R = part8 ref part9_timelice9 : PartDef9 part10_timslice_10 part10.1_timeslice10 part10.2_timeslice10 ... 	<pre>{ timeslice state state1; timeslice part1_timeslice1 : PartDef1 [1..*] ordered nonunique; /* ... */ }</pre>
Snapshots Compartment	<p style="text-align: center;"><i>snapshots</i></p> state state1 ^part2_snapshot2 : PartDef2 part1_snapshot1 : PartDef1 [1..*] ordered nonunique part3R_snapshot3R : PartDef3R redefines part3 part4R_snapshot4R : PartDef4R >> part4 :>> part5 part6S_snapshot6S : PartDef6S [m] subsets part6 part7S_snapshot7S : PartDef7S [m] >> part7 part8R_snapshot8R = part8 ref part9_snapshot9 : PartDef9 part10_snaphsot10 part10.1_snapshot10 part10.2_snapshot10 ...	<pre>{ snapshot state state1; snapshot part1_snapshot1 : PartDef1 [1..*] ordered nonunique; /* ... */ }</pre>

7.10 Items

7.10.1 Overview

Items

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.

Space Boundaries

Items can have boundaries in space, identified as their *shape*. Items without shapes are *closed*, such as spheres, enabling them to be boundaries of other Items. Items can identify others as *enveloping* shapes, which are closed items that are the boundary of a hypothetical item occupying the same or larger space as the item. Some of these can be *bounding* shapes, which overlap the item on all sides.

The spatial boundaries of items can break into separate closed items, such as the inside and outside of an egg shell. These inner boundaries can be the boundary of a hypothetical item, the interior of which is a *void* of the Item. Items with no voids are *solid*.

7.10.2 Abstract Syntax

For Item Abstract Syntax class descriptions, see [8.3.10](#).

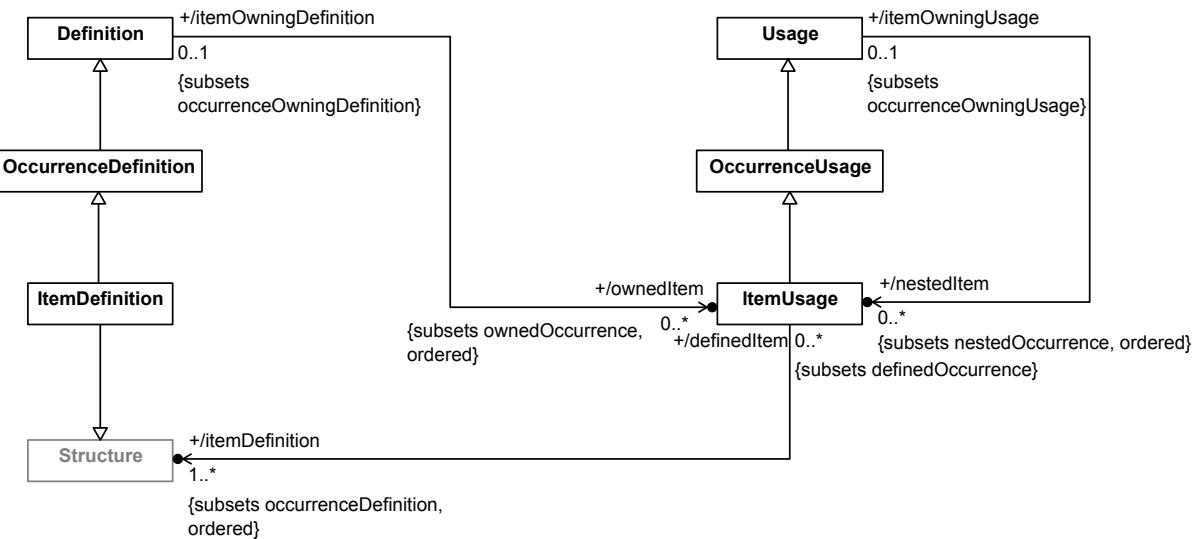


Figure 16. Item Definition and Usage

7.10.3 Notation

Textual Notation

For Items Textual Notation BNF, see [8.2.2.10](#)

An ItemDefinition or ItemUsage (that is not of a more specialized kind) can be declared as a kind of OccurrenceDefinition or OccurrenceUsage (see [7.9.3](#)) (see [7.9.3](#)), using the kind keyword `item`. An ItemUsage shall

only be defined by ItemDefinitions (of any kind) or KerML Structures (see [KerML]). The default multiplicity of an ItemUsage shall be [1..1], under the conditions described in [7.6.3](#).

```
item def LinkedList {
    ref values[0..*];
    ref item rest[0..1] : LinkedList;
}
```

The base ItemDefinition is *Item* from the *Items* library model (see [9.2.3](#)). The base Element for an ItemUsage is one of the following ItemUsages from the *Items* library model (see [9.2.3](#)):

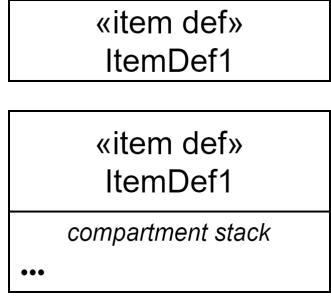
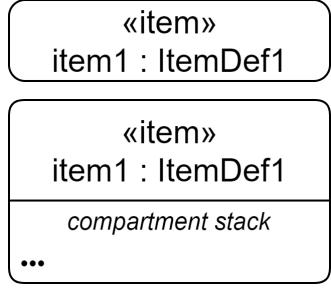
1. If the declared ItemUsage is a feature of an ItemDefinition or ItemUsage, then *Item*::*subitems*.
2. Otherwise, *items*.

Graphical Notation

For Items Graphical Notation BNF, see [8.2.3.10](#)

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 15. Items - Representative Notation

Element	Graphical Notation	Textual Notation
Item Definition		<pre>item def ItemDef1; item def ItemDef1 { /* members */ }</pre>
Item		<pre>item item1 : ItemDef1; item item1 : ItemDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
Items Compartment	<pre> <i>items</i> ^item2 : ItemDef2 item1 : ItemDef1 [1..*] ordered nonunique item3R : ItemDef3R redefines item3 item4R : ItemDef4R :>> item4 :>> item5 item6S : ItemDef6S [m] subsets item6 item7S : ItemDef7S [m] :> item7 item8R = item8 ref item9 : itemDef9 item10 item10.1 item10.2 ... </pre>	<pre> { item item1 : ItemDef1 [1..*] ordered nonunique; /* ... */ } </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

For Parts Abstract Syntax class descriptions, see [8.3.11](#).

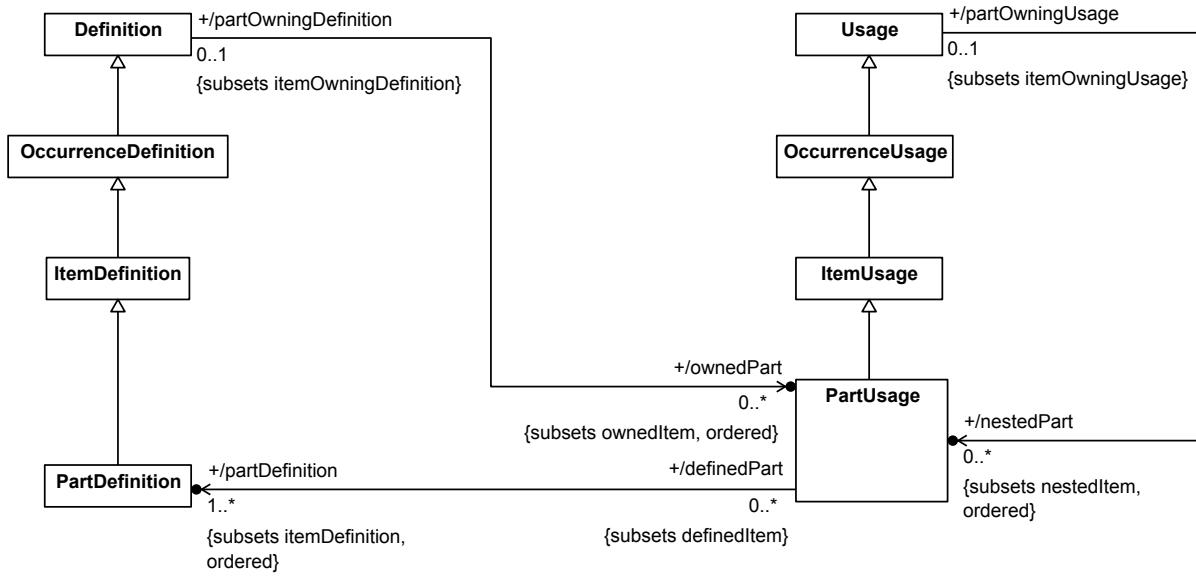


Figure 17. Part Definition and Usage

7.11.3 Notation

Textual Notation

For Parts Textual Notation BNF, see [8.2.2.11](#).

A PartDefinition or PartUsage (that is not of a more specialized kind) can be declared as a kind of OccurrenceDefinition or OccurrenceUsage (see [7.9.3](#)), using the kind keyword **part**. As a kind of ItemUsage (see [7.10](#)), a PartUsage shall only be defined by ItemDefinitions (including PartDefinitions) or KerML Structures (see [KerML]). The default multiplicity of a PartUsage shall be **[1..1]**, under the conditions described in [7.6.3](#).

```

item def Person;
part def Vehicle {
    ref part driver[0..1] : Person;
    part engine : Engine;
    part wheels[4] : Wheel;
}
  
```

The base PartDefinition is **Part** from the **Parts** library model (see [9.2.4](#)). The base Element for a PartUsage is one of the following PartUsages:

1. If the declared PartUsage is a stakeholder of a RequirementDefinition or RequirementUsage (see [7.20](#)), then **RequirementCheck::stakeholders** from the **Requirements** library model (see [9.2.13](#)).
2. If the declared PartUsage is an actor of a RequirementDefinition or RequirementUsage (see [7.20](#)), then **RequirementCheck::actors** from the **Requirements** library model (see [9.2.13](#)).
3. If the declared PartUsage is an actor of a CaseDefinition or CaseUsage (see [7.21](#)), then **Case:::actors** from the **Cases** library model (see [9.2.14](#)).
4. If the declared PartUsage is a feature of an ItemDefinition or ItemUsage, then **Item:::subparts** from the **Items** library model (see [9.2.3](#)).
5. Otherwise, **parts** from the **Parts** library model (see [9.2.4](#)).

Note. Because the base Element of a PartUsage is a PartUsage that is defined by a PartDefinition, every PartUsage is always directly or indirectly defined by at least one PartDefinition, implicitly if not explicitly.

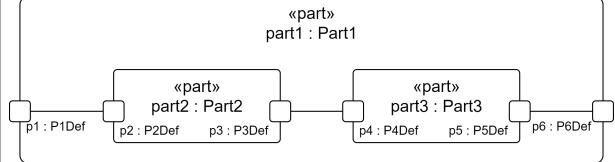
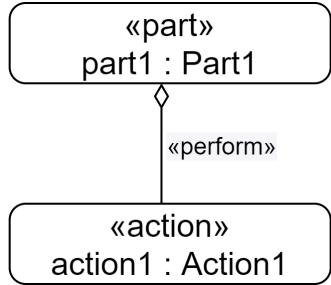
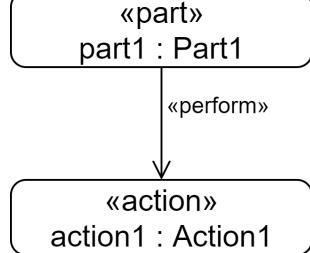
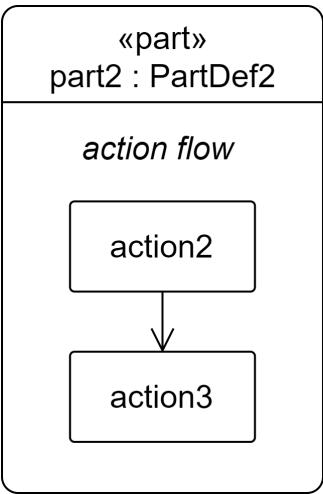
Graphical Notation

For Parts Graphical Notation BNF, see [8.2.3.11](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 16. Parts - Representative Notation

Element	Graphical Notation	Textual Notation
Part Definition	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> <p>«part def»</p> <p>PartDef1</p> </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> <p>«part def»</p> <p>PartDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>part def PartDef1; part def PartDef1 { /* members */ }</pre>
Part	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> <p>«part»</p> <p>part1 : PartDef1</p> </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> <p>«part»</p> <p>part1 : PartDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>part part1 : PartDef1; part part1 : PartDef1 { /* members */ }</pre>
Part with Ports		<pre>part part1 { port p0 : NestedPortDef0 { port p1 : PortDef1; port p2 : PortDef2; port p3 : PortDef3; } port p4 : PortDef4; port p5 : PortDef5; }</pre>

Element	Graphical Notation	Textual Notation
Part with Graphical Compartment showing a standard interconnection view of part1.	 <pre data-bbox="1065 291 1392 682"> part part1 : Part1{ port p1 : P1Def; port p6 : P6Def; part part2:Part2; part part3:Part3; connect p1 = part2.p2; connect part2.p3 to part3.p4; connect part2.p5 = p6; } </pre>	
Part performs a reference action that can subset or bind to another action	 <pre data-bbox="1065 819 1359 925"> part part1 : Part1 { perform action action1 : Action1; } </pre>	
Part performs an anonymous reference action that subsets another action	 <pre data-bbox="1065 1083 1359 1252"> action action1 : Action1; part part1 : Part1 { perform action1; } </pre>	
Part with Graphical Compartment with perform actions and flows between them. This is informally referred to as a swim lane.	 <pre data-bbox="1065 1484 1408 1643"> part part2 : PartDef2 { perform action action2; then perform action action3; } </pre>	

Element	Graphical Notation	Textual Notation
Parts Compartment	<pre> <i>parts</i> ^part2 : PartDef2 part1 : PartDef1 [1..*] ordered nonunique part3R : PartDef3R redefines part3 part4R : PartDef4R >> part4 :>> part5 part6S : PartDef6S [m] subsets part6 part7S : PartDef7S [m] >> part7 part8R = part8 ref part9 : PartDef9 part10 part10.1 part10.2 ... </pre>	<pre> { part part1 : PartDef1 [1..*] ordered nonunique; /* ... */ } </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

For Ports Abstract Syntax class descriptions, see [8.3.12](#).

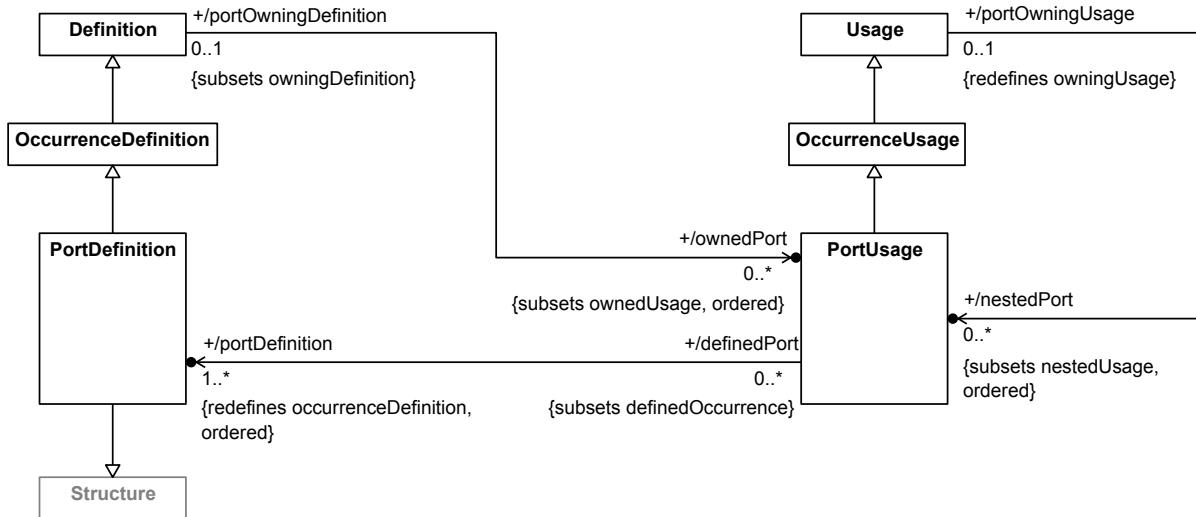


Figure 18. Port Definition and Usage

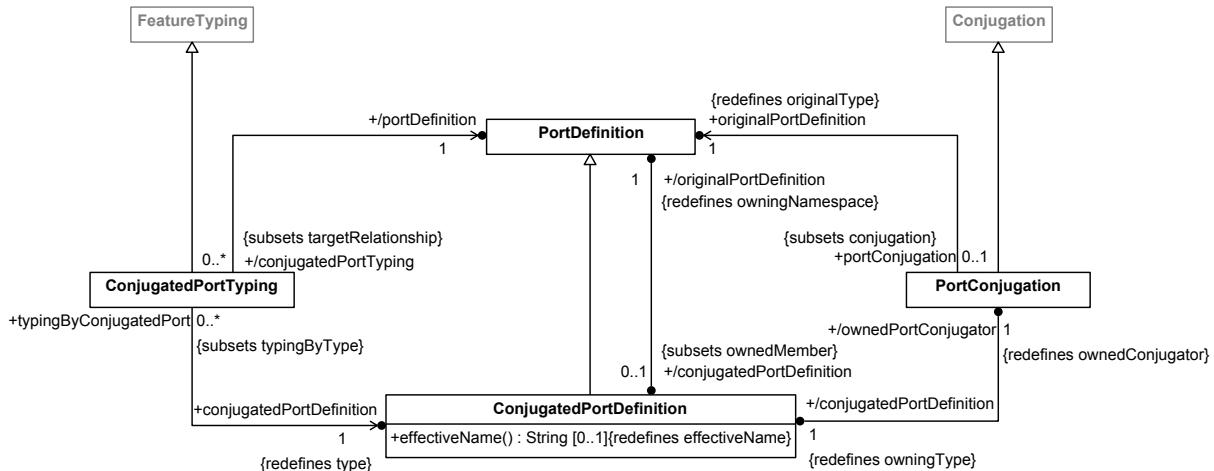


Figure 19. Port Conjugation

7.12.3 Notation

For Ports Textual Notation BNF, see [8.2.2.12](#).

Ports

A PortDefinition or PortUsage can be declared as a kind of OccurrenceDefinition or OccurrenceUsage (see [7.9.3](#)), using the keyword **port**. A PortUsage shall only be defined by PortDefinitions. The default multiplicity of a PortUsage shall be $[1..1]$, under the conditions described in [7.6.3](#). All the features of a PortDefinition or PortUsage, other than any nested PortUsages, shall be referential (non-composite).

```

port def FuelingPort {
    attribute flowRate : Real;
    out fuelOut : Fuel;
    in fuelReturn : Fuel;
}
part def FuelTank {
  
```

```

port fuelOutPort : FuelingPort;
}

```

The base PortDefinition is *Port* from the *Ports* library model (see [9.2.5](#)). The base Element for a PortUsage is one of the following PortUsages:

1. If the declared PortUsage is a feature of a PartDefinition or PartUsage, then *Part*::*portsOnPart* from the *Parts* library model (see [9.2.4](#)).
2. If the declared PortUsage is a feature of a PortDefinition or PortUsage, then *Port*::*subports* from the *Ports* library model (see [9.2.5](#)).
3. Otherwise, *ports* from the *Ports* library model (see [9.2.5](#)).

Conjugated Ports

Every PortDefinition also implicitly declares a single, nested ConjugatedPortDefinition which has the same features as its *originalPortDefinition*, except that any directed features have conjugated directions (i.e., **in** and **out** are reversed, with **inout** unchanged). The name of the ConjugatedPortDefinition shall be the name of the *originalPortDefinition* with the character ~ prepended, in the namespace of the *originalPortDefinition*. For example, if a PortDefinition has the name *P*, then its ConjugatedPortDefinition has the name *P*::'~*P*'.

A ConjugatedPortTyping is a shorthand for using a ConjugatedPortDefinition to define a PortUsage. With this shorthand, rather than using the actual name of the ConjugatedPortDefinition, the name of the *originalPortDefinition* can be used, preceded by the symbol ~. Note, however, that since the symbol ~ is *not* considered part of a name in this case, it does not have to be placed within quotes. For example,

```
port p : ~P;
```

is equivalent to

```
port p : P::'~P';
```

Note that, if a PortDefinition has the name '*P-1*', then its ConjugatedPortDefinition has the (qualified) name '*P-1*::'~*P-1*', but the corresponding ConjugatedPortTyping is

```
port p1 : '~P-1'
```

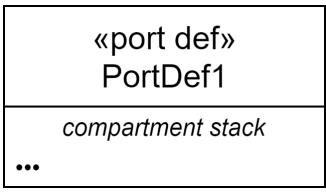
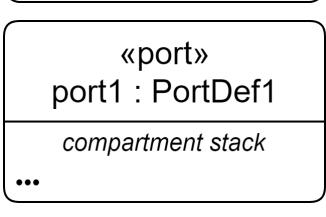
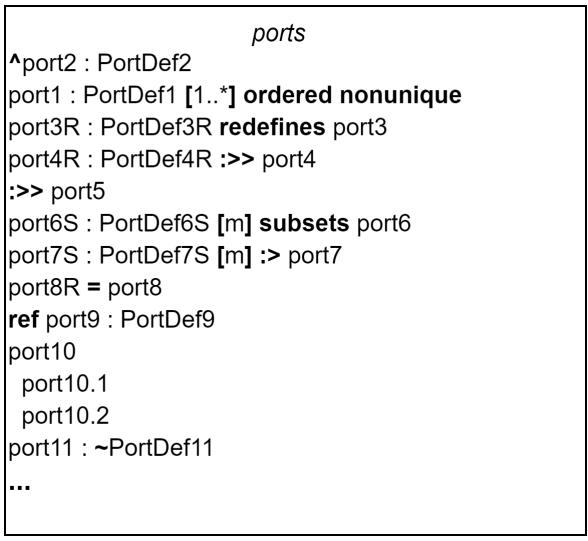
where the ~ is *not* placed inside the quotes.

Graphical Notation

For Ports Graphical Notation BNF, see [8.2.3.12](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 17. Ports - Representative Notation

Element	Graphical Notation	Textual Notation
Port Definition	 	<pre>port def PortDef1; port def PortDef1 { /* members */ }</pre>
Port	 	<pre>port port1 : PortDef1; port port1 : PortDef1 { /* members */ }</pre>
Ports Compartment	 <pre> ports ^port2 : PortDef2 port1 : PortDef1 [1..*] ordered nonunique port3R : PortDef3R redefines port3 port4R : PortDef4R :>> port4 :>> port5 port6S : PortDef6S [m] subsets port6 port7S : PortDef7S [m] :> port7 port8R = port8 ref port9 : PortDef9 port10 port10.1 port10.2 port11 : ~PortDef11 ...</pre>	<pre>{ port port1 : PortDef1 [1..*] ordered nonunique; /* ... */ }</pre>

Element	Graphical Notation	Textual Notation
Directed Features Compartment	<div style="border: 1px solid black; padding: 10px;"> <p style="text-align: center;"><i>directed features</i></p> <pre> in attribute1 : AttributeDef1 out attribute2 : AttributeDef2 inout attribute3 : AttributeDef3 in item1 : ItemDef1 out item2 : ItemDef2 inout item3 : ItemDef3 ... </pre> </div>	<pre> { in attribute1 : AttributeDef1; out attribute2 : AttributeDef2; inout attribute3 : AttributeDef3; in item1 : ItemDef1; out item2 : ItemDef2; inout item3 : ItemDef3; } </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 initializing or binding a usage to the result of an *expression* (see [7.18](#)) as part of the declaration of the usage. Using a feature value for initialization is described in [7.16](#). Using a feature value for binding is described here.

There are two types of feature value binding.

- 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

For Connections Abstract Syntax class descriptions, see [8.3.13](#).

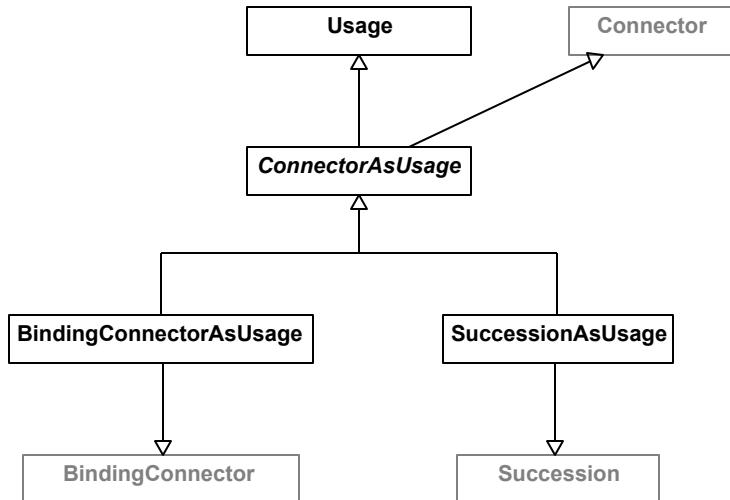


Figure 20. Connectors as Usages

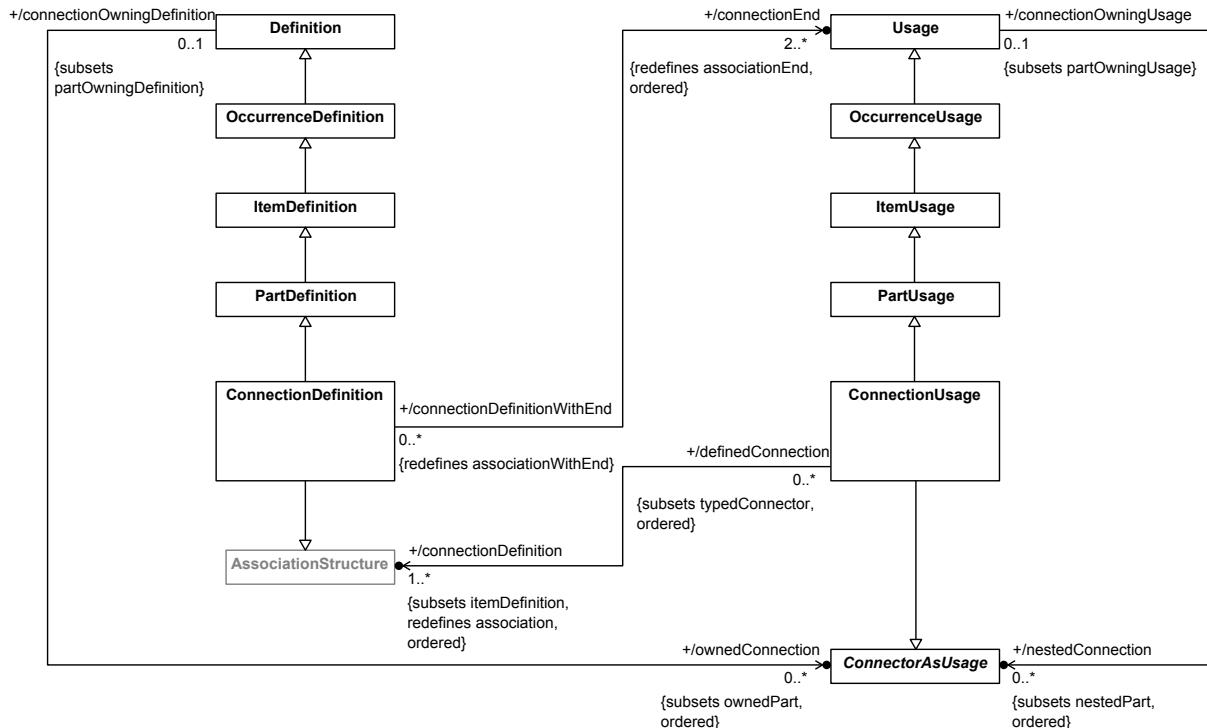


Figure 21. Connection Definition and Usage

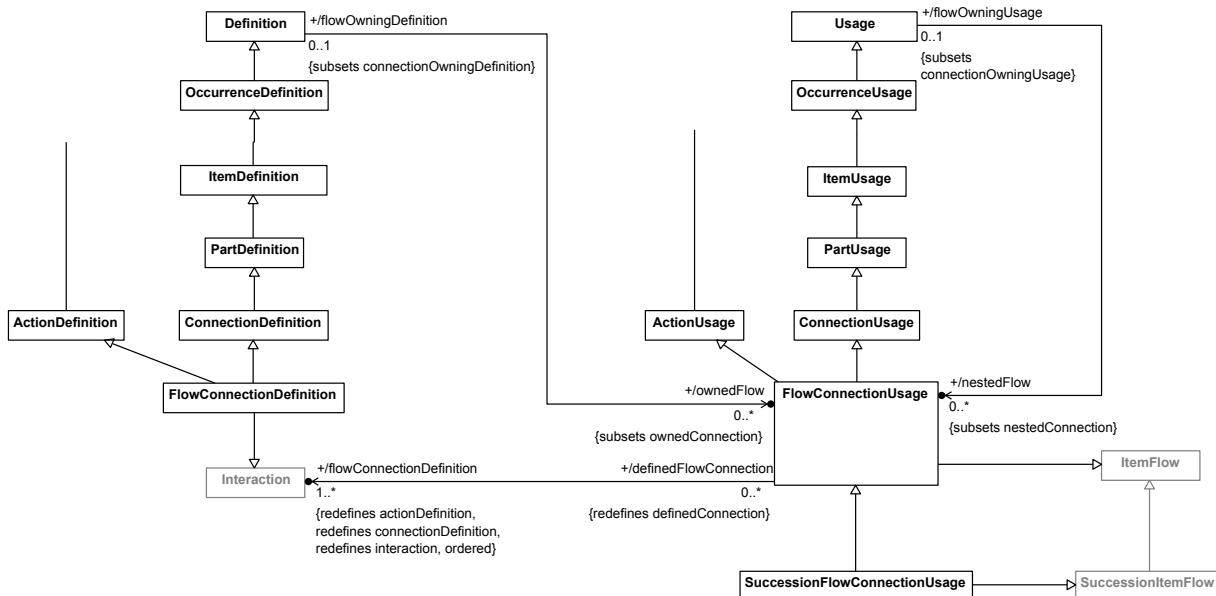


Figure 22. Flow Connections

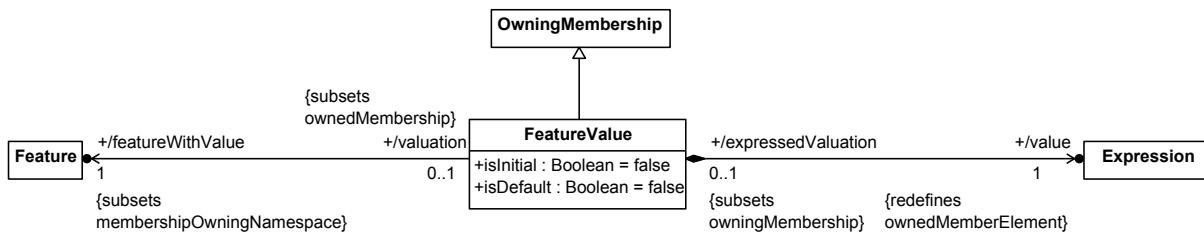


Figure 23. Feature Values

7.13.3 Notation

For Connections Textual Notation BNF, see [8.2.2.13](#).

Connections

A ConnectionDefinition or ConnectionUsage (that is not of a more specialized kind) can be declared as a kind of OccurrenceDefinition or OccurrenceUsage (see [7.9.3](#)), using the kind keyword **connection**. A ConnectionUsage shall only be defined by ConnectionDefinitions (of any kind) or KerML AssociationStructures (see [KerML]).

Unless it is abstract, a ConnectionDefinition or ConnectionUsage shall have at least two end features (which may be owned or inherited). A *binary* ConnectionDefinition or ConnectionUsage is one that has exactly two end features. The end features of a ConnectionDefinition or ConnectionUsage shall always be considered referential (non-composite), whether or not their declaration explicitly includes the **ref** keyword. (See also [7.6.3](#) on the notation for end features.)

For a ConnectionDefinition, the **definitions** of the end features are the **relatedElements** of the ConnectionDefinition considered as a Relationship (known as its **relatedTypes**). For a ConnectionUsage, each end feature shall have an **ownedReferenceSubsetting**, specified using the keyword **references** or the equivalent symbol **::>**, which specify the **relatedElements** of the ConnectionUsage as a Relationship (known as its **relatedFeatures**). (See also [7.2](#) on Relationships.)

```

// The related Elements of this connection definition
// are the part definitions Hub and Device.
connection def DeviceConnection {
    end part hub : Hub[1];
    end part devices : Device[0..*];

    // This is a non-end feature of the connection.
    attribute bandwidth : Real;
}

// The related Elements of this connection usage
// are the part usages mainSwitch and sensorFeed.
connection connection1 : DeviceConnection {
    end part hub ::> mainSwitch[1];
    end part device ::> sensorFeed[1];
}

```

Rather than being declared with end features in its body, the `relatedFeatures` of a ConnectionUsage may be identified in a comma-separated list, between parentheses (...), preceded by the keyword `connect`, placed after the ConnectionUsage declaration and before its body. The identification of a `relatedFeature` may optionally be preceded by an end feature name followed by the keyword `references` or the symbol `::>`, and/or followed by a MultiplicityRange. If the declaration part of the ConnectionUsage is empty when using this notation, then the keyword `connection` may be omitted.

```

connection connection1 : DeviceConnection connect (
    hub ::> mainSwitch[1], device ::> sensorFeed[1]
);

// This is a ternary connection.
// It is equivalent to "connection connect (item1, item2, item3);"
connect (axle, wheel1, wheel2);

```

If the ConnectionUsage is binary, then a special notation may be used in which the source `relatedFeature` is identified directly after the keyword `connect` and the target `relatedFeature` is identified after the keyword `to`. As above, if the declaration part of the ConnectionUsage is empty, then the keyword `connection` may be omitted.

```

connection connection1 : DeviceConnection
    connect hub ::> mainSwitch[1] to device ::> sensorFeed[1];

    connect leftWheel to leftHalfAxe;

```

The base Element for a ConnectionDefinition is one of the following ConnectionDefinitions from the *Connections* library model (see [7.13](#)):

1. If the declared ConnectionDefinition is binary, then *BinaryConnection*.
2. Otherwise, *Connection*.

The base Element for a ConnectionUsage is one of the following ConnectionUsages from the *Connections* library model (see [7.13](#)):

1. If the declared ConnectionUsage is binary, then *binaryConnections*.
2. Otherwise, *connections*.

If a ConnectionDefinition has a single `ownedSuperclassification` with a `superclassifier` that is a ConnectionDefinition, it may inherit end features from this general ConnectionDefinition. However, if it declares any owned end features, then each of these shall redefine an end feature of the general ConnectionDefinition, in

order, up to the number of end features of the general ConnectionDefinition. If no redefinition is given explicitly for an owned end feature, then it shall be considered to implicitly redefine the end feature at the same position, in order, of the general ConnectionDefinition, if any.

```
// Specializes Connections::BinaryConnection by default.
connection def Ownership {
    attribute valuationOnPurchase : MonetaryValue;
    end item owner[1..*] : LegalEntity; // Redefines BinaryConnection::source.
    end item ownedAsset[*] : Asset;      // Redefines BinaryConnection::target.
}
connection def SoleOwnership specializes Ownership {
    end item owner[1]; // Redefines Ownership::owner.
    // ownedAsset is inherited.
}
```

If a ConnectionDefinition has more than one ownedSuperclassification with `superClassifiers` that are ConnectionDefinitions, then the ConnectionDefinition *must* declare a number of owned end features at least equal to the maximum number of end features of any of the general ConnectionDefinitions. Each of these owned end features shall then redefine the corresponding end feature (if any) at the same position, in order, of each of the general ConnectionDefinitions.

Similar rules hold for the end features of a ConnectionUsage, relative to any specialization by FeatureTyping (definition), Subsetting or Redefinition.

```
connection connection1 : DeviceConnection {
    end part hub ::> mainSwitch[1]; // Redefines DeviceConnection::hub.
    end part device ::> sensorFeed[1]; // Redefines DeviceConnection::device.
}
```

Bindings and Feature Values

A BindingConnectorAsUsage can be declared as described in [7.6.3](#), using the kind keyword `binding`. However, a BindingConnectorAsUsage must be declared with exactly two `relatedFeatures` using the special notation described below. Note also that a BindingConnectorAsUsage is not a kind of OccurrenceUsage (unlike a true ConnectionUsages as described above), so the notations for time slices, snapshots and individuals (described in [7.9.3](#)) do *not* apply to it.

The base Usage for a BindConnectorAsUsage is the Feature `selfLinks` from the *Links* Kernel Library model (see [KerML]).

The two `relatedFeatures` of a BindingConnectorAsUsage are identified after its declaration part and before its body, following the keyword `bind` and separated by the symbol =. If the declaration part is empty, then the keyword `binding` may be omitted. The end features of a BindingConnectorAsUsage always have multiplicity [1..1].

```
part def WheelAssembly {
    part fuelTank {
        out fuelFlowOut : Fuel;
    }
    part engine {
        in fuelFlowIn : Fuel;
    }
    binding fuelFlowBinding
        bind fuelTank.fuelFlowOut = engine.fuelFlowIn;

    // The following is equivalent to the above, but
```

```

    // without the name.
    bind fuelTank.fuelFlowOut = engine.fuelFlowIn;
}

```

A FeatureValue with `isInitial = false` can be used to specify the binding of a Usage to the result of an Expression. If it also has `isDefault = false`, then it is declared using the symbol `=` followed by a representation of its value Expression (using the Expression notation from [KerML]). This notation is appended to the declaration of the Usage that is the `featureWithValue` for the FeatureValue.

```

attribute monthsInYear : Natural = 12;
item def TestRecord {
    attribute scores[1..*] : Integer;
    derived attribute averageScore[1] : Rational = sum(scores)/size(scores);
    attribute cutoff : Integer default = 0.75 * averageScore;
}

```

Note. The semantics of binding mean that a FeatureValue with `isInitial = false` and `isDefault = false` asserts that a Usage is *equivalent* to the result of the `value` Expression. To highlight this, a Usage with such a FeatureValue can be flagged as `derived` (though this is not required, nor is it required that the value of a `derived` Usage be computed using a FeatureValue – see also [7.6.3](#)).

A FeatureValue with `isInitial = false` and `isDefault = true` is declared similarly to the above, but with the keyword `default` either preceding or instead of the symbol `=`.

```

part def Vehicle {
    attribute mass : Real default 1500.0;
}

item def TestWithCutoff :> TestRecord {
    attribute cutoff : Rational default = 0.75 * averageScore;
}

```

A FeatureValue with `isInitial = true` is used to specify an initial value for a Feature, rather than binding it. This case is described in [7.16](#).

Successions

A SuccessionAsUsage can be declared as described in [7.6.3](#), using the kind keyword `succession`. However, a SuccessionAsUsage must be declared with exactly two `relatedFeatures` using the special notation described below. Note also that a SuccessionAsUsage is not a kind of OccurrenceUsage (unlike a true ConnectionUsages as described above), so the notations for time slices, snapshots and individuals (described in [7.9.3](#)) do *not* apply to it.

The base Usage for a SuccessionAsUsage is the Feature `happensBeforeLinks` from the *Occurrences* Kernel Library model (see [KerML]).

The two `relatedFeatures` of a SuccessionAsUsage are identified after its declaration part and before its body, following the keyword `first` and separated by the keyword `then`. If the declaration part is empty, then the keyword `succession` may be omitted. The `relatedFeatures` of a SuccessionAsUsage must be OccurrenceUsages. As for ConnectionUsages, constraining multiplicities can also be defined on the end features of a SuccessionAsUsage.

```

part def Camera {
    action focus[*] : Focus;
    action shoot[*] : Shoot;
    // A focus may be preceded by a previous focus.
    succession multiFocusing
}

```

```

        first focus[0..1] then focus[0..1];
        // A shoot must follow a focus.
        first focus[1] then shoot[0..1];
        // The Camera can be focused after shooting.
        first shoot[0..1] then focus;
    }
}

```

If a SuccessionAsUsage is placed lexically directly between the two to OccurrenceUsages that are its relatedElements, then the declaration of the SuccessionAsUsage can be shortened to just the keyword **then**, prepended to the declaration of the second OccurrenceUsage. A MultiplicityRange for the source end of the SuccessionAsUsage can optionally be placed directly after the **then** keyword.

```

occurrence def Flight {
    timeslice preflight[1];
    then timeslice inflight[1];
    then timeslice postflight[1];
}

// The above is equivalent to the following.
occurrence def Flight {
    timeslice preflight[1];
    first preflight then inflight;
    timeslice inflight[1];
    first inflight then postflight;
    timeslice postflight[1];
}

```

Note. There are additional shorthands for the use of SuccessionAsUsages within the bodies of ActionDefinitions and ActionsUsages (see [7.16.3](#)).

Messages and Flows

A FlowConnectionUsage may be declared as either a message, a complete flow or as a further specialized SuccessionFlowConnectionUsage. The base Usage for a FlowConnectionUsage is the FlowConnectionUsage *flowConnections* from the *Connections* library model (see [7.13](#)), unless it is a SuccessionFlowConnectionUsage, in which case the base Usage is *successionFlowConnections*, also from the *Connections* library model.

A FlowConnectionUsage is declared as a message similarly to a ConnectionUsage (see above), but with the kind keyword **message**. In addition, the declaration of a message may also optionally include an explicit specification of the name, type (definition) and/or multiplicity of the payload of the message, after the keyword **of**. The payload name is followed the keyword **defined by** (or the symbol `:`), but this keyword (or the symbol) is omitted if the name is. In the absence of a payload specification, the message declaration does not constrain what kinds of values may be transferred between the source and target of the message.

A message FlowConnectionUsage is always abstract (whether or not the **abstract** keyword is included explicitly in its declaration), so its declaration may or may not include identification of source and target *relatedFeatures*. If they are included, then they follow the payload specification (if any), with the source *relatedFeature* identified after the keyword **from**, followed by the target *relatedFeature* after the keyword **to**. Alternative, if the source and target identification is *not* included, then the message declaration may include a FeatureValue to provide a value for the message.

```

part def Vehicle {
    attribute def ControlSignal;

    part controller;
}

```

```

part engine;

message of ControlSignal from controller to engine;
}

```

A complete FlowConnectionUsage is declared similarly to a message declaration, but with the kind keyword **flow**. However, instead of identifying the source and target relatedElements of the flow, such a flow declaration must identify (after the keyword **from**) the output feature of the source from which the flow receives its payload and (after the keyword **to**) the input feature of the target to which the flow delivers the payload. This is done by giving a feature chain with at least two features, the last of which identifies the output or input feature, with the preceding part of the chain identifying the source or target relatedFeature of the flow. If no declaration part or payload specification is included in the flow declaration, then the **from** keyword may also be omitted.

```

part def Vehicle {
    part fuelTank {
        out fuelOut : Fuel;
    }
    part engine {
        in fuelIn : Fuel;
    }
    // This FlowConnectionUsage actually connects the fuelTank to the
    // engine. The transfer moves Fuel from fuelOut to fuelIn.
    flow fuelFlow of flowingFuel : Fuel
        from fuelTank.fuelOut to engine.fuelIn;

    // The following is equivalent to the above, except without
    // the name and leaving the payload implicit.
    flow fuelTank.fuelOut to engine.fuelIn;
}

```

A SuccessionFlowConnectionUsage is declared like a flow declaration above, but using the keyword **succession flow**.

```

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

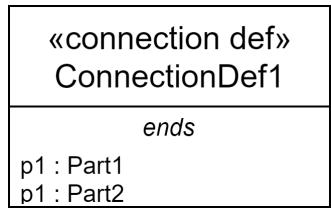
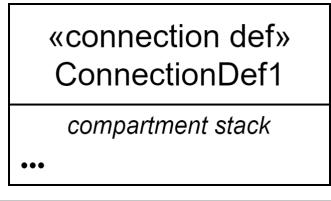
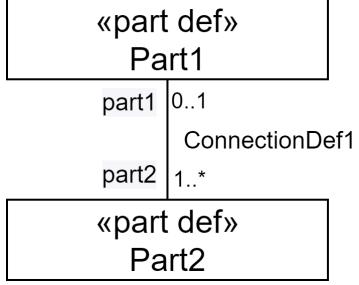
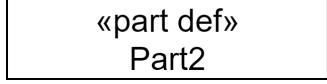
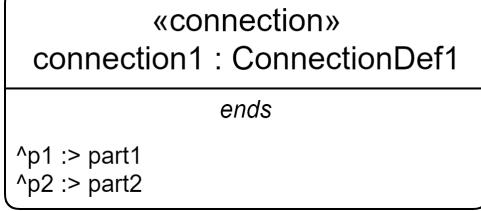
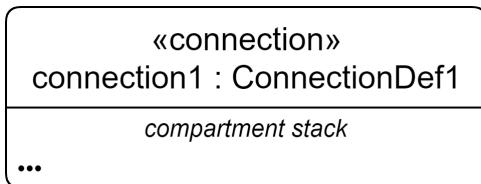
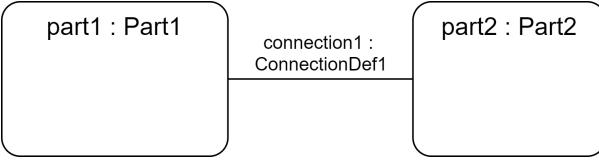
```

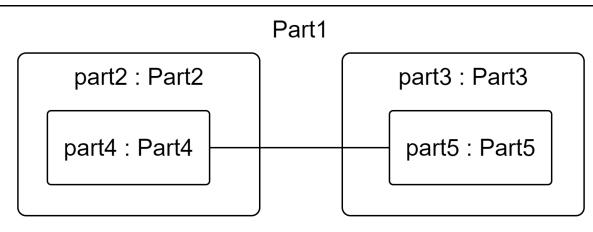
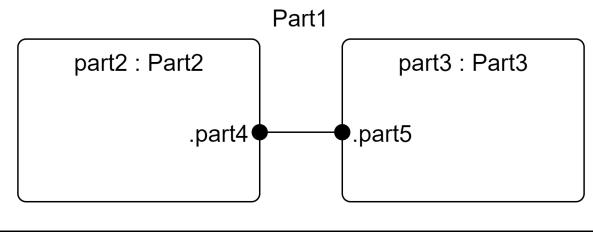
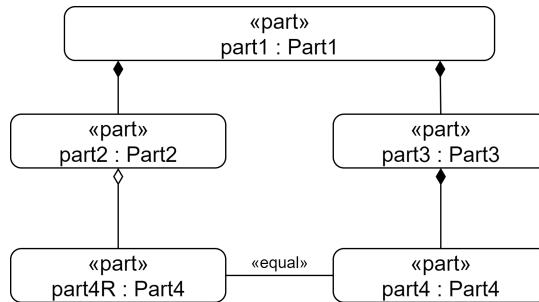
Graphical Notation

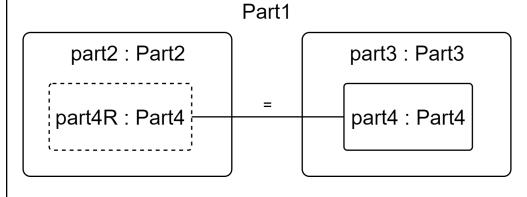
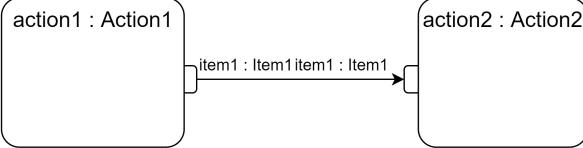
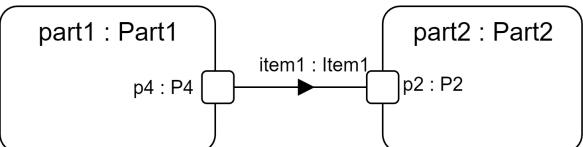
For Connections Graphical Notation BNF, see [8.2.3.13](#).

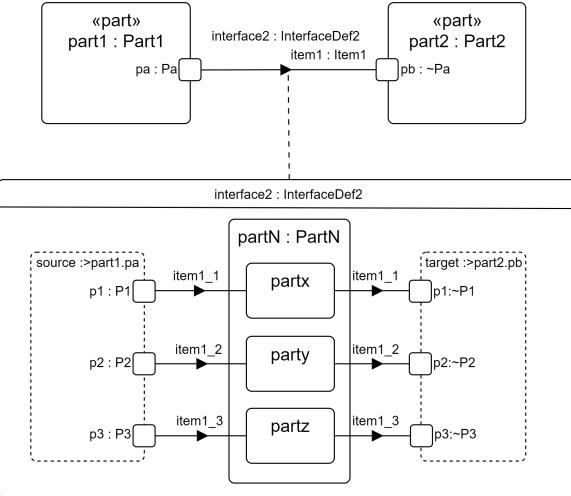
For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

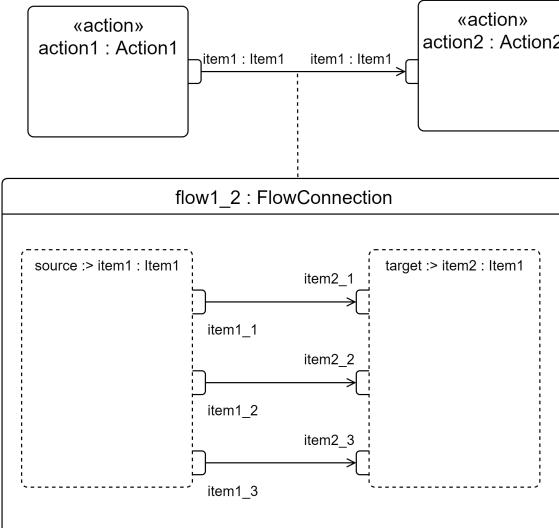
Table 18. Connections - Representative Notation

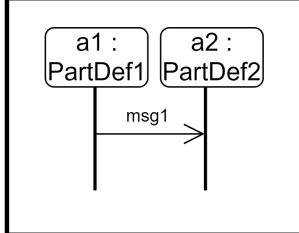
Element	Graphical Notation	Textual Notation
Connection Definition	 	<pre>connection def ConnectionDef1 { end p1 : Part1; end p2 : Part2; }</pre> <pre>connection def ConnectionDef1 { /* members */ }</pre>
Connection Definition	 	<pre>connection def ConnectionDef1 { end part1 : Part1 [0..1]; end part2 : Part2 [1..*]; }</pre>
Connection	 	<pre>connection connection1 : ConnectionDef1 { end p1 :> part1; end p2 :> part2; }</pre> <pre>connection connection1 : ConnectionDef1 { /* members */ }</pre>
Connection		<pre>connection connection1 : ConnectionDef1 connect part1 to part2;</pre>

Element	Graphical Notation	Textual Notation
Nested Connection		<pre> part def Part1 { part part2 : Part2 { part part4 : Part4; } part part3 : Part3 { part part5 : Part5; } connection connection1 : ConnectionDef1 connect part2.part4 to part3.part5; } } </pre>
Proxy Connection		<pre> part def Part1 { part part2 : Part2 { part part4 : Part4; } part part3 : Part3 { part part5 : Part5; } connection connection1 : ConnectionDef1 connect part2.part4 to part3.part5; } } </pre>
Connections Compartment		
Binding Connection		<pre> part part1 : Part1 { part part2 : Part2 { ref part part4R : Part4; } part part3 : Part3 { part part4 : Part4; } bind part2.part4R = part3.part4; } </pre>

Element	Graphical Notation	Textual Notation
Binding Connection		<pre> part def Part1 { part part2:part2 { ref part } part4R:Part4; } part part3:part3 { part part4:Part4; } bind part2.part4R = part3.part4; } </pre>
Flow		<pre> action action1:Action1 { out item1:Item1; } action action2:Action2 { in item1:Item1; } flow action1.item1 to action2.item1; </pre>
Message		<pre> part part1:Part1 { port p4:P4; } part part2:Part2 { port p2:P2; } message of item1:Item1 from part1.p4 to part2.p2; </pre>

Element	Graphical Notation	Textual Notation
Interface as Node	 <pre> graph LR subgraph "interface2 : InterfaceDef2" direction TB subgraph source [source :>part1.pa] p1((p1 : P1)) p1 -- item1_1 --> partx[partx] end subgraph partN [partN : PartN] partx party[party] partz[partz] partx -- item1_1 --> target["target :>part2.pb"] party -- item1_2 --> target partz -- item1_3 --> target end subgraph target [target :>part2.pb] p2((p2 : P2)) p3((p3 : P3)) p1 --> p2 p2 --> p3 end end </pre>	<pre> port def Pa { out item item1 : Item1; port p1 : P1 { out item item1_1 } port p2 : P2 { out item item1_2 } port p3 : P3 { out item item1_3 } } part def Part1 { port pa : Pa; } part def Part2 { port pb : ~Pa; } interface def InterfaceDef2 { end source : Pa; end target : ~Pa; } part part1 : Part1; part part2 : Part2; interface interface2 : connect source :> part1.pa to target :> part2.pb; /*...*/ </pre>

Element	Graphical Notation	Textual Notation
Flow as Node		<pre> action action1 : Action1 { out item1 : Item1; } action action2 : Action2 { in item2 : Item1; } flow flow1_2 from source :> action1.item1 to target :> action2.item2 { flow source.item1.item1_1 to target.item2.item2_1; flow source.item1.item1_2 to target.item2.item2_2; flow source.item1.item1_3 to target.item2.item2_3; } </pre>
Flows Compartment		<pre>{ flow action1.output to action2.input; succession flow action1.output to action2.input; /* ... */ }</pre>

Element	Graphical Notation	Textual Notation
Message		<pre> occurrence def { part a1 : PartDef1 { event occurrence msg1_source; } part a2 : PartDef2 { event occurrence msg1_target; } message msg1 from a1.msg1_source to a2.msg1_target; } </pre>

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. An interface may be defined that can then be specialized to represent more specific 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

For Interfaces Abstract Syntax class descriptions, see [8.3.14](#).

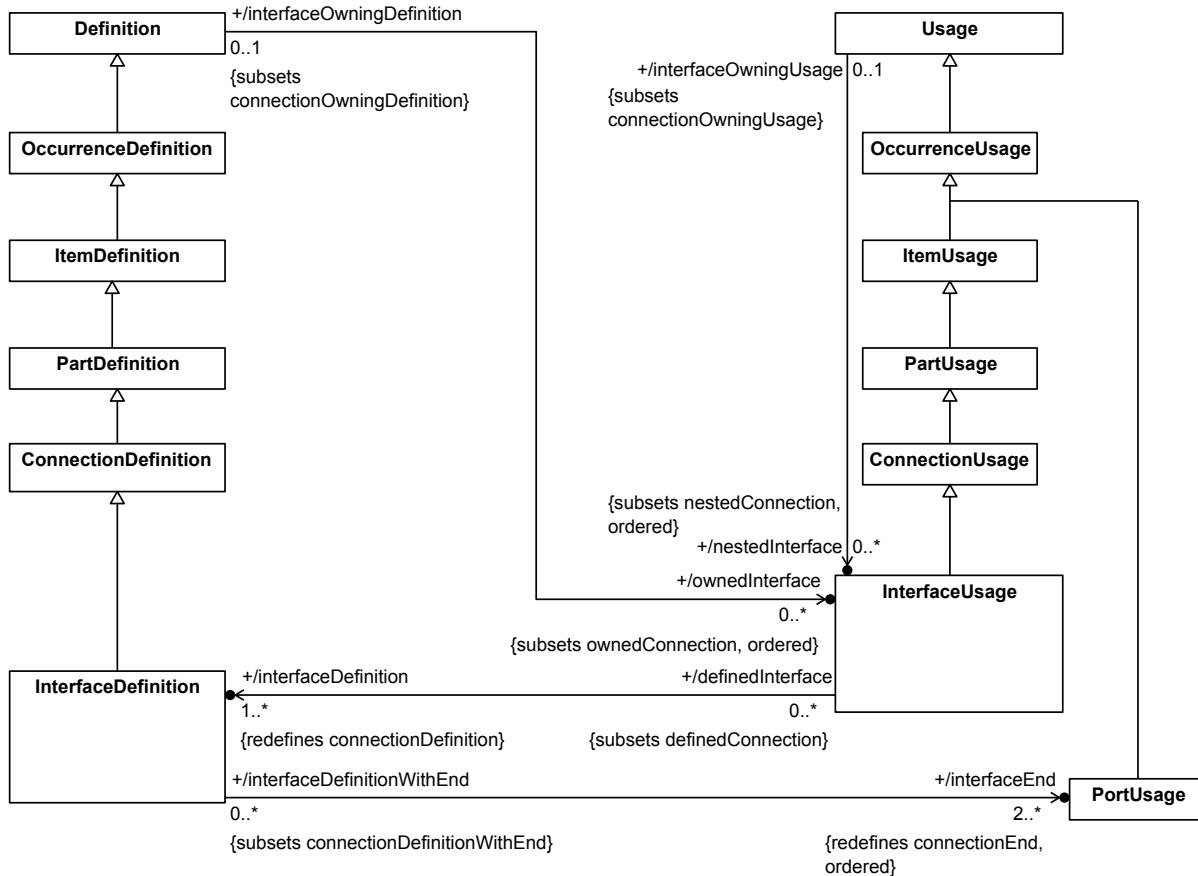


Figure 24. Interface Definition and Usage

7.14.3 Notation

Textual Notation

For Interfaces Textual Notation, see [8.2.2.14](#).

An **InterfaceDefinition** or **InterfaceUsage** is declared like a **ConnectionDefinition** or **ConnectionUsage** (see [7.13.3](#)), but using the **kind** keyword **interface**. An **InterfaceUsage** shall only be defined by **InterfaceDefinitions**. All the end features of an **InterfaceDefinition** or **InterfaceUsage** shall be **PortUsages**, so the use of the **port** keyword is optional on such end features.

The shorthand notations for **ConnectionUsages** described in (see [7.13.3](#)) may also be used for **InterfaceUsages**. However, if the declaration part of an **InterfaceUsage** is empty, then the **interface** keyword is still included, but the **connect** keyword may be omitted.

```

port def FuelingPort {
    out fuel : Fuel;
}
interface def FuelingInterface {
    end fuelOutPort : FuelingPort;
    end fuelInPort : ~FuelingPort;
}
interface fuelLine : FuelingInterface
    connect fuelTank.fuelingPort to engine.fuelingPort;

```

```
// The following is equivalent to the above, except
// for not using a specialized interface definition.
interface fuelTank.fuelingPort to engine.fuelingPort;
```

The base Element for an InterfaceDefinition is one of the following InterfaceDefinitions from the *Interfaces* library model (see [9.2.7](#)):

1. If the declared InterfaceDefinition is binary, then *BinaryInterface*.
2. Otherwise, *Interface*.

The base Element for an InterfaceUsage is one of the following InterfaceUsages from the *Interfaces* library model (see [9.2.7](#)):

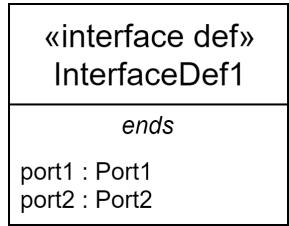
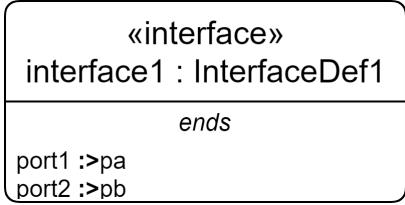
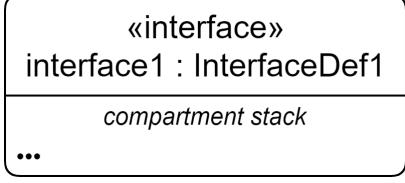
1. If the declared InterfaceUsage is binary, then *binaryInterfaces*.
2. Otherwise, *interfaces*.

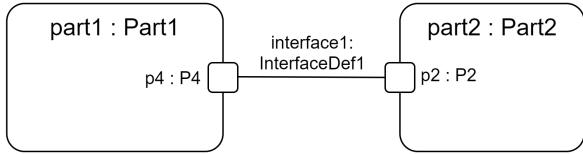
Graphical Notation

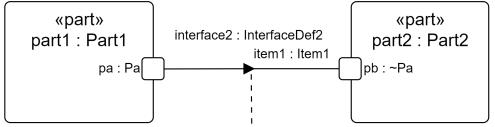
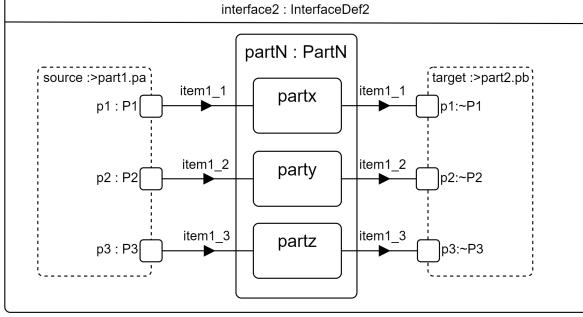
For the Interfaces Graphical BNF, see [8.2.3.14](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 19. Interfaces - Representative Notation

Element	Graphical Notation	Textual Notation
Interface Definition	 	<pre>interface def InterfaceDef1 { end port1:Port1; end port2:Port2; }</pre> <pre>interface def InterfaceDef1 { /* members */ }</pre>
Interface	 	<pre>interface interface1 : InterfaceDef1 { end port1 :> pa; end port2 :> pb; }</pre> <pre>interface interface1 : InterfaceDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
Interfaces Compartment	<pre data-bbox="561 283 943 460"> interfaces interface1 : InterfaceDef1 [1..*] interface2 : InterfaceDef2 ... </pre>	<pre data-bbox="1067 283 1405 481"> { interface interface1 : InterfaceDef1 [1..*]; interface interface2 : InterfaceDef2; /* ... */ } </pre>
Interface	 <p>The diagram shows two rectangular compartments labeled "part1 : Part1" and "part2 : Part2". Each compartment contains a port symbol (a small square with a line) and a port name: "p4 : P4" for part1 and "p2 : P2" for part2. A horizontal line connects the two ports, with the label "interface1: InterfaceDef1" written above the connection line.</p>	<pre data-bbox="1067 557 1388 840"> part part1:Part1 { port p4:P4; } part part2:Part2 { port p2:P2; } interface interface1 : InterfaceDef1 connect part1.p4 to part2.p2; </pre>

Element	Graphical Notation	Textual Notation
Interface as Node (with flow)	  <pre> graph LR subgraph interface2 [interface2 : InterfaceDef2] direction TB p1[p1 : P1] -- item1_1 --> partx p2[p2 : P2] -- item1_2 --> party p3[p3 : P3] -- item1_3 --> partz partx -- item1_1 --> p1 party -- item1_2 --> p2 partz -- item1_3 --> p3 end </pre>	<pre> part part1 : Part1 { port pa : Pa; } part part2 : Part2 { port pb : ~Pa; } interface interface2 : InterfaceDef2 connect source :> part1.pa to target :> part2.pb { part partN:PartN { part partx; part party; part partz; } message of item1 : Item1 from source to target { message of 'item1.1' : 'Item1.1' from source.p1 to partN.partx; message of 'item1.2' : 'Item1.2' from source.p2 to partN.party; message of 'item1.3' : 'Item1.3' from source.p3 to partN.partz; message of 'item1.1':'Item1.1' from partN.partx to target.p1; message of 'item1.2':'Item1.2' from partN.party to target.p2; message of 'item1.3':'Item1.3' from partN.partz to target.p3; </pre>

Element	Graphical Notation	Textual Notation
		}

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.

Release Note. The library model for allocations currently does not provide any specializations of the most general definition of an *Allocation*. Consideration will be given to including specializations of *Allocation* in 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

For Allocations Abstract Syntax class descriptions, see [8.3.15](#).

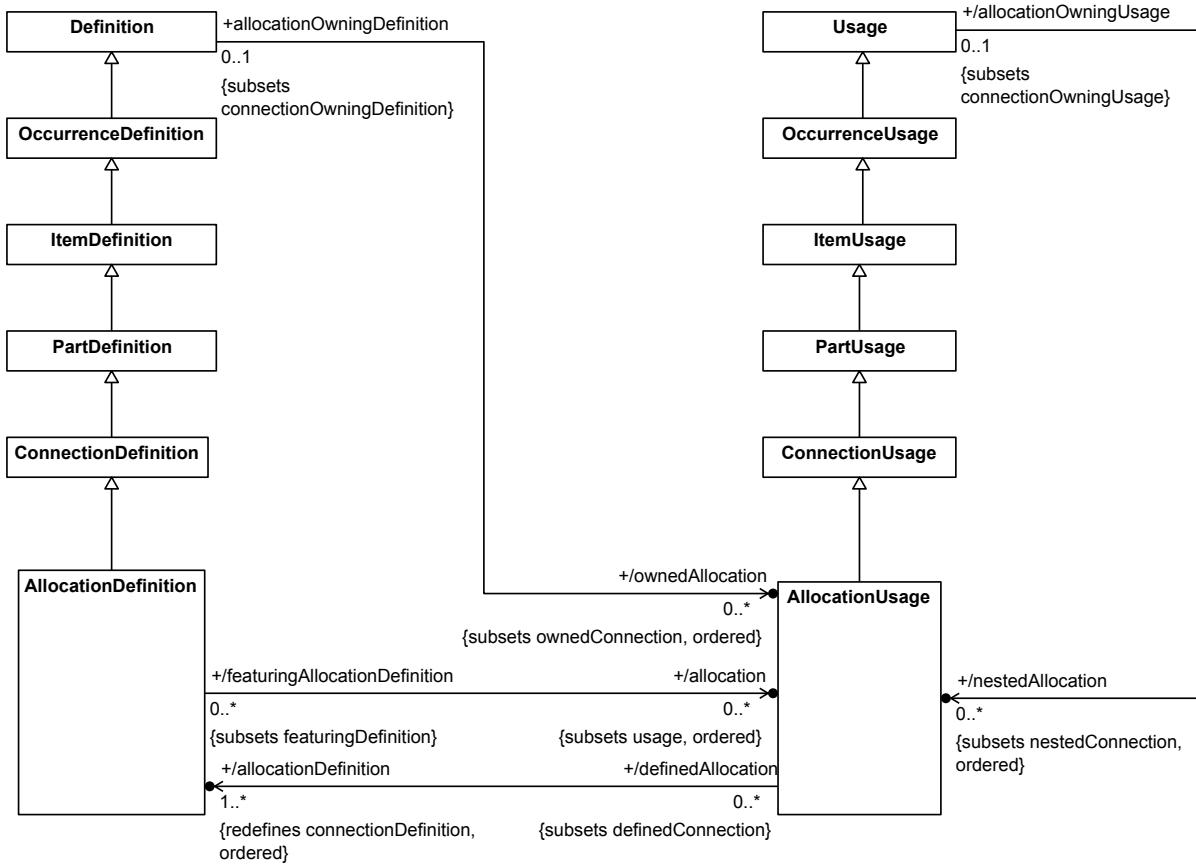


Figure 25. Allocation Definition and Usage

7.15.3 Notation

Textual Notation

For Allocations Textual Notation BNF, see [8.2.2.15](#).

An AllocationDefinition or AllocationUsage is declared like a ConnectionDefinition or ConnectionUsage (see [7.13.3](#)), but using the kind keyword **allocation**. An AllocationUsage shall only be defined by AllocationDefinitions. AllocationDefinitions and AllocationUsages are typically binary and always have at least two end features, even if abstract.

Shorthand notations similar to those for ConnectionUsages, as described in see [7.13.3](#), may also be used for AllocationUsages, but using the keyword **allocate** instead of **connect**. If the declaration part of the AllocationUsage is empty when using this notation, then the keyword **allocation** may be omitted.

```

part def LogicalSystem {
    part component : LogicalComponent;
}
part def PhysicalDevice {
    part assembly : PhysicalAssembly;
}
allocation def LogicalToPhysicalAllocation {
    end part logical : LogicalSystem;
    end part physical : PhysicalDevice;
}

```

```

    // This is a nested sub-allocation.
    allocate logical.component to physical.assembly;
}

part system : LogicalSystem;
part device : PhysicalDevice;
allocation systemToDevice : LogicalToPhysicalAllocation
    allocate logical :> system to physical :> device;

```

The base AllocationDefinition is the AllocationDefinition *Allocation*, and the base AllocationUsage is the AllocationUsage *allocations*, both from the *Allocations* library model (see [9.2.8](#)).

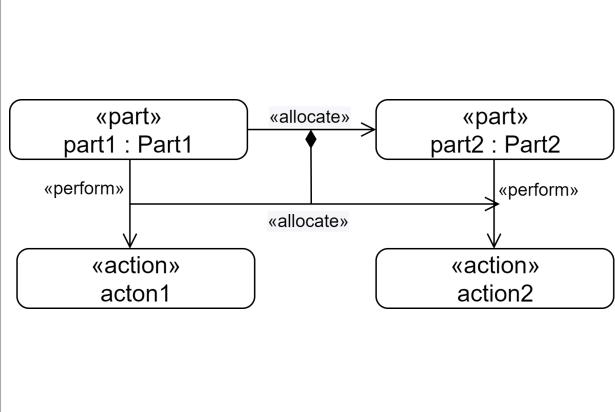
Graphical Notation

For Allocations Graphical Notation BNF, see [8.2.3.15](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 20. Allocations - Representative Notation

Element	Graphical Notation	Textual Notation
Allocation Definition		<pre> allocation def AllocationDef1; allocation def AllocationDef1 { /* members */ } </pre>
Allocation		<pre> allocation allocation1 : AllocationDef1; allocation allocation1 : AllocationDef1 { /* members */ } </pre>
Allocated Compartment		<pre> part part3 { allocate part1 to part3; allocate part3 to part2; } </pre>
Allocation		<pre> part part1 : Part1; part part2 : Part2; allocate part1 to part2; </pre>

Element	Graphical Notation	Textual Notation
Allocation (with sub allocation)	 <pre> graph LR part1["«part» part1 : Part1"] part2["«part» part2 : Part2"] action1["«action» action1"] action2["«action» action2"] part1 -- «allocate» --> part2 part1 -- «perform» --> action1 part2 -- «allocate» --> action2 part2 -- «perform» --> action2 </pre>	<pre> part part1 : Part1 { perform action1; } part part2 : Part2 { perform action2; } allocate part1 to part2 { allocate part1.action1 to part2.action2; } </pre>

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.

Bindings and 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.

Assignment Actions

An *assignment action usage* is used to change the value of a *referent* feature of a *target* occurrence. The target is specified as the result of an expression and the referent is specified as a feature chain relative to that target. The new value for the feature is determined as the result of a different expression. When the assignment action usage completes, the referent feature has the new assigned value for the target occurrence.

Note that the target must be an occurrence, because the values of the features of attributes do not change over time (see also [7.7](#) on attributes and [7.9](#) on occurrences). If the referent feature has a multiplicity upper bound other than 1, then an assignment action can assign multiple values to it, consistent with the multiplicity of the feature. The values are all assigned atomically, at the same time.

A *initializing feature value* can be used a shorthand for assigning an initial value to a usage as part of the declaration of the usage that is a feature of an occurrence definition or usage. Unlike when feature is a bound using a feature value (as described in [7.13](#)), the initial value of a feature can be later assigned a different value.

As for a binding feature value, there are two types of initializing feature value.

- A *fixed* feature value establishes the assigns the result of evaluating the given expression to a usage at the point of declaration of the usage. Such a binding cannot be overridden in a redefinition of the usage because, once asserted, because it would be indeterminate which initialization is to be used.
- A *default* feature value also includes an initial-value expression, but it does not immediately assign an initial value to the usage. Instead, the evaluation of the expression and the assignment of its result to 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 initializing the redefining usage.

7.16.2 Abstract Syntax

For Actions Abstract Syntax class descriptions, see [8.3.16](#).

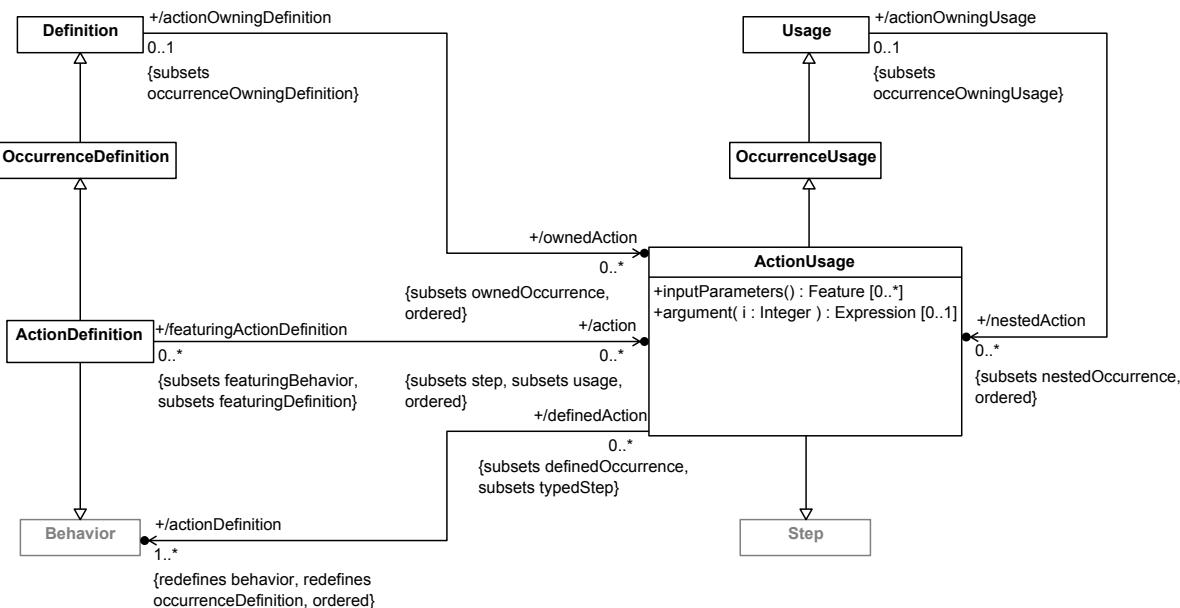


Figure 26. Action Definition and Usage

Return the owned `input` Features of an `ActionUsage`.

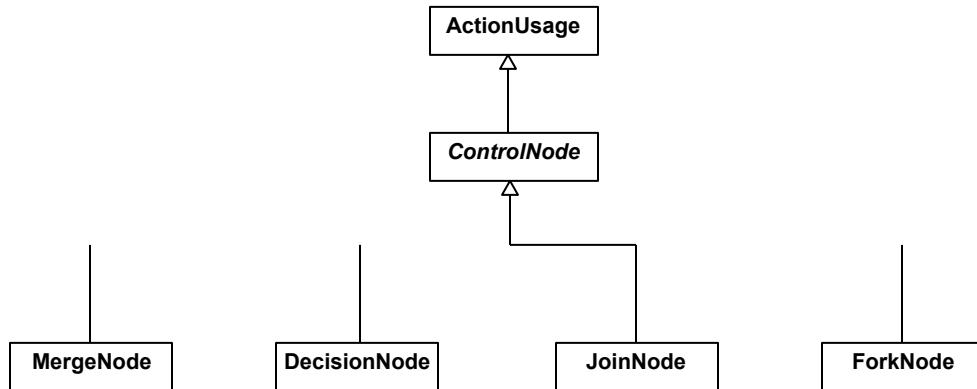


Figure 27. Control Nodes

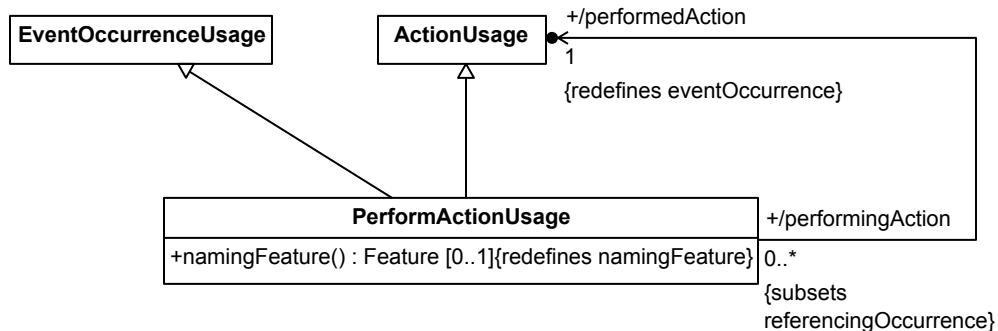


Figure 28. Action Performance

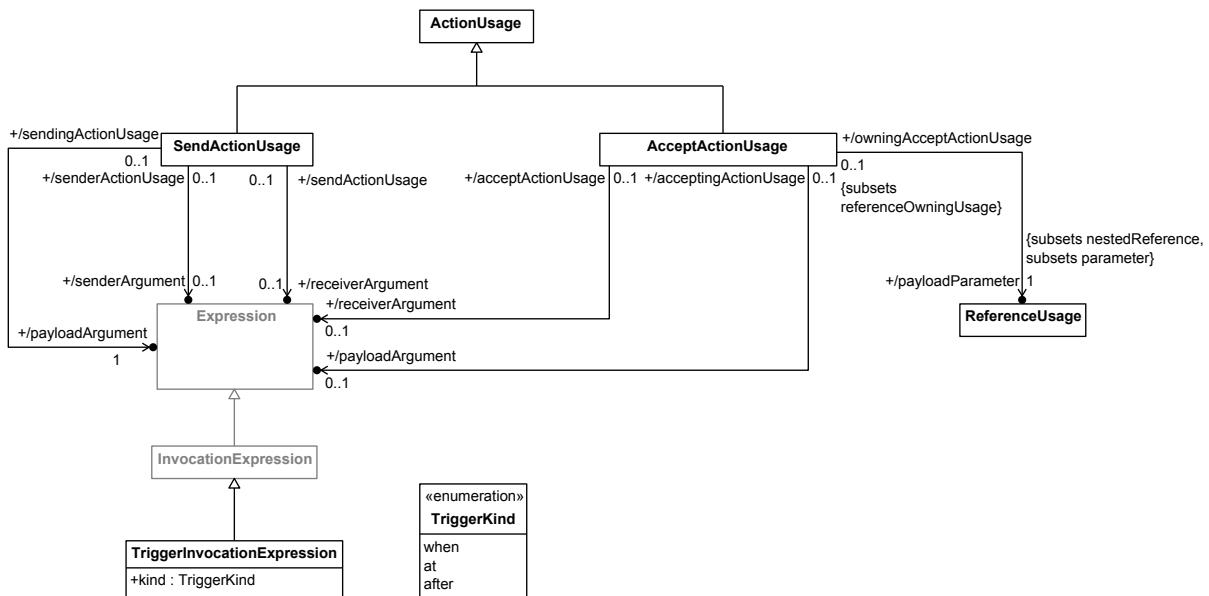


Figure 29. Send and Accept Actions

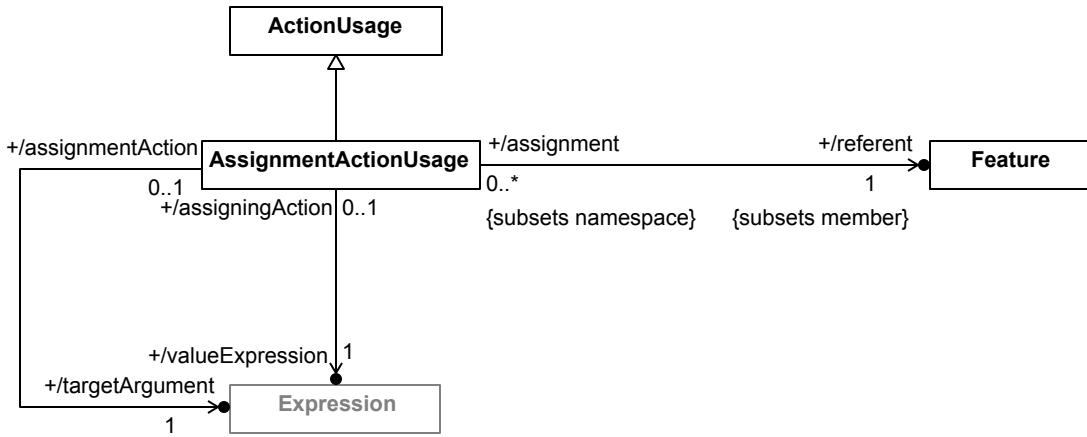


Figure 30. Assignment Actions

7.16.3 Notation

For Actions Textual Notation BNF, see [8.2.2.16](#).

Actions

An ActionDefinition or ActionUsage (that is not of a more specialized kind) can be declared as a kind of OccurrenceDefinition or OccurrenceUsage (see [7.9.3](#)) (see [7.9.3](#)), using the kind keyword **action**. An ActionUsage shall only be defined by ActionDefinitions (of any kind) or KerML Behaviors (see [KerML]).

The base ActionDefinition is *Action* from the *Actions* library model (see [9.2.9](#)). The base Element for an ActionUsage is one of the following ActionUsages from the *Actions* library model (see [9.2.9](#)):

1. If the declared ActionUsage is a **feature** of an ActionDefinition or ActionUsage, then
 $\text{Action}::\text{subactions}$.
2. Otherwise, *actions*.

Parameters

Any directed features declared in the body of an ActionDefinition or ActionUsage are considered to be owned parameters of the action. Features with direction **in** are input parameters, those with direction **out** are output parameters, and those with direction **inout** are both input and output parameters.

```

action def TakePicture {
    // The following two features are considered parameters.
    in scene : Scene;
    out picture : Picture;

    bind focus.scene = scene;
    action focus : Focus (in scene, out image);
    first focus then shoot;
    flow focus.image to shoot.image;
    action shoot : Shoot (in image, out picture);
    bind picture = focus.picture;
}
  
```

If an ActionDefinition has ownedSuperclassifications whose superclassifiers are ActionDefinitions, then each of the owned parameters of the subclassifier ActionDefinition shall, in order, redefine the

parameter at the same position of each of the `superclassifier` ActionDefinitions. The redefining parameters shall have the same direction as the redefined parameters.

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

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

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

If an ActionDefinition has `ownedGeneralizations` (including all FeatureTypings, Subsettings and Redefinitions) whose general Type is an ActionDefinition or ActionUsage, then the rules for the redefinition of the parameters of those ActionDefinitions and ActionUsages shall be the same as for the redefinition of the parameters of `superclassifier` ActionDefinitions by a `subclassifer` Behavior, as given above.

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

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

Performed Actions

A PerformActionUsage can be declared like an ActionUsage, as described above, but using the kind keyword `perform action` instead of just `action`. The `performedAction` of the PerformActionUsage is given by the first explicitly declared subsetted Feature of the PerformActionUsage, which must be an ActionUsage, or, if there is no such Feature, then the PerformActionUsage itself.

```
part def Vehicle {
    perform action powerVehicle :> VehicleActions::providePower;
    abstract perform action moveVehicle; // Performed action is itself.
}
```

A PerformActionUsage may also be declared using just the keyword `perform` instead of `perform action`. In this case, the declaration shall not include either a `shortName` or `name`. Instead, the `performedAction` of the PerformActionUsage is identified by giving a qualified name or feature chain immediately after the `perform` keyword.

```
part vehicle : Vehicle {
    // The performed action is VehicleActions::move.
```

```

perform ActionTree::move :>> Vehicle::moveVehicle;
}

```

If a PerformActionUsage is a feature of a PartDefinition or PartUsage, then it shall (explicitly or implicitly) subset the ActionUsage *Part::performedActions* from the *Parts* library model (see [9.2.4](#)). A PerformActionUsage may also be used in the body of another ActionDefinition or ActionUsage, in which case it acts like a referential "call" of the performedActionUsage by the containing action.

```

action initialization {
    in item device;
    perform Utility::startUpCheck {
        in component = device;
        out status;
    }
    ...
}

```

The **ref** keyword may be used in the declaration of a PerformActionUsage, but a PerformActionUsage is always referential, whether or not **ref** is included in its declaration.

Control Nodes

A *control node* is a special syntactic notation for an ActionUsage whose *definition* is a concrete specialization of the abstract ActionUsage *ControlAction* from the *Actions* library model (see [9.2.9](#)). A control node is declared like a normal ActionUsage (as described above), but using one of the keywords shown in [Table 21](#) instead of the keyword **action**. A control node shall only be declared in the body of an ActionDefinition or ActionUsage and shall implicitly subset the ActionUsage shown in the table corresponding to its keyword, thereby inheriting the corresponding definition. A control node is always composite. The **ref** keyword shall not be used in a control node declaration. A control node declaration can have a body, but only containing AnnotatedElements, which are owned by the control node ActionUsage via Annotation relationships (see [8.2.2.3.1](#)).

Table 21. Control Node Definitions

Keyword	Subsetting	Definition
merge	Actions::Action::merges	Actions::MergeAction
decide	Actions::Action::decisions	Actions::DecisionAction
join	Actions::Action::joins	Actions::JoinAction
fork	Actions::Action::forks	Actions::ForkAction

Control nodes are used to control the sequencing of other ActionUsages connected to them via Successions. The following rules apply to these connections.

1. Incoming Successions to a **merge** node shall have source multiplicity 0..1 and subset the *incomingHBLLink* Feature inherited by *MergeAction* from the Kernel Library Behavior *ControlPerformances::MergePerformance* (see [KerML]).
2. Outgoing succession connectors from a **decide** node shall have target multiplicity 0..1 and subset the *outgoingHBLLink* Feature inherited from the Kernel Library Behavior *ControlPerformances::DecisionPerformance* (see [KerML]).
3. Incoming Successions to a **join** node shall have source multiplicity 1..1.
4. Outgoing Succession from a **fork** node shall have target multiplicity 1..1.

These rules shall be enforced in the abstract syntax, even if not shown explicitly in the concrete syntax notation for a model.

```
// Both action1 and action2 will proceed concurrently
// after fork1.
fork fork1;
first fork1 then action1;
first fork1 then action2;

action action1;
action action2;

// join1 will be performed after both action1
// and action2 have completed.
first action1 then join1;
first action2 then join1;
join join1;

first join1 then decision1;

// One of action3 or action4 will be chosen
// (non-deterministically) to be performed after decision1.
decide decision1;
first decision1 then action3;
first decision1 then action4;

action action3;
action action4;

// mergel will be performed after either action3
// or action4 have completed.
first action3 then mergel;
first action4 then mergel;
merge mergel;
```

Succession Shorthands

The basic notation for SuccessionAsUsages (see [7.13.3](#)) may be used to specify the sequencing of ActionUsages within the body of an ActionDefinition or ActionUsage. There are also addition shorthands that may be used *only* within the body of an ActionDefinition or ActionUsage, as described below. Further, every action inherits the features *start* and *done* from the base ActionDefinition *Actions::Action*, which represent the start and end snapshots of the action.

The source of a SuccessionAsUsage may be specified separately from the target by using the keyword **first** followed by a qualified name or feature chain for the source ActionUsage. Similarly, the target of a SuccessionAsUsage may be specified separately from the source by using the keyword **then** followed by a qualified name or feature chain for the target ActionUsage.

```
first action1;
then action2;

// The above two declarations are together
// equivalent to the following single succession.
first action1 then action2;
```

The **then** keyword may also be followed by a complete ActionUsage declaration, rather than just the name.

```

first action1;
then action2;

// The above two declarations are together
// equivalent to the following.
first action1 then action2;
action action2;

```

The **then** shorthand can be used lexically following any ActionUsage, not just following a **start** declaration, with the preceding ActionUsage becoming the source of the SuccessionAsUsage. This is particularly useful when a sequence of actions to be performed successively or in a loop.

```

first start;
then merge loop;
then action initialize;
then action monitor;
then action finalize;
then loop;

```

The source of a SuccessionAsUsage must be an OccurrenceUsage. Therefore, the source of a SuccessionAsUsage represented using the **then** shorthand is actually determined as the nearest OccurrenceUsage lexically previous to the **then**, skipping over any intervening non-OccurrenceUsages. Since a SuccessionAsUsage is not an OccurrenceUsage, this allows several **then** successions to be placed in a sequence after a common source ActionUsage. This is particularly useful for specifying multiple successions outgoing from **fork** and **decide** nodes.

```

// The two successions following fork1 both have
// fork1 as their source.
fork fork1;
    then action1;
    then action2;

action action1;
    then join1;

action action2;
    then join1;

join join1;

// The two successions following decision1 both have
// decision1 as their source.
then decide decision1;
    then action3;
    then action4;

action action3;
    then merge1;

action action4;
    then merge1;

merge merge1;

```

Conditional Successions

A SuccessionAsUsage within the body of an ActionDefinition or ActionUsage may be given a *guard* condition. A guard given as a Boolean-valued Expression preceded by the keyword **if**. It is placed in the declaration of the

SuccessionAsUsage (see [7.13.3](#)) after the specification of the source of the succession and before the specification of the target.

```
succession conditionalOnActive
    first initialize if isActive then monitor;
```

Such a conditional succession actually declares a TransitionUsage defined by the ActionDefinition *DecisionTransitionAction* and subsetting the ActionUsage *Actions::transitions* (both from the *Actions* model library; see Actions). The succession itself redefines the *transitionLink* for the *DecisionTransitionAction* and the guard Expression redefines the guard of the TransitionUsage (*transitionLink* and *guard* are inherited from the Kernel Library Behavior *TransitionPerformances::NonStateTransitionPerformance* [KerML]).

As usual, if the declaration part is empty, the keyword **succession** may be omitted. The source for the succession may then be further omitted, in which case the source is identified from a lexically previous ActionUsage, as for the **then** shorthand described above. Further, the keyword **else** may be used in place of a guard Expression to indicate a succession to be taken if the guards evaluate to false on all of an immediately preceding set of conditional successions. However, the target of a conditional succession *must* be specified as a qualified name or feature chain and cannot be a full ActionUsage declaration, even when the shorthand notation is used.

The conditional succession shorthand notation is particularly useful for notating several conditional successions outgoing from a **decide** node.

```
merge loop;
action checkLevel { out level; }

decide;
if level <= refillLevel then refill;
if level >= maxLevel then drain;
else continue;

action refill;
then loop;

action drain;
then loop;

action continue;
```

Bindings and Flows between Actions

The regular notation for BindingConnectorAsUsages and FlowConnectionUsages (see [7.13.3](#)) can be used for them in the body of an ActionDefinition or ActionUsage. In addition, the FeatureValue shorthand for binding can be used with parameters, as for any directed feature.

```
action providePower : ProvidePower {
    in fuelCmd : FuelCmd;
    action generatePower : GeneratePower {
        in fuelCmd : FuelCmd = providePower::fuelCmd;
        out generatedTorque : Torque;
    }

    flow generatePower.generatedTorque
        to transmitPower.generatedTorque;

    action transmitPower : TransmitPower {
        in generatedTorque : Torque;
```

```

        out transmittedTorque;
    // ...
}

```

Send and Accept Actions

A SendActionUsage is declared like a regular ActionUsage (as described above), except that it also includes a payload Expression placed following the action declaration and before the action body (if any), after the keyword **send**, followed by a sender Expression, after the keyword **via**, and/or a receiver Expression, after the keyword **to**. If the declaration part is empty, then the **action** keyword may be omitted.

```

action sendReadingTo {
    in part destination;

    perform getReading { out reading : SensorReading; }
    action sendReading
        send getReading.reading to destination;

    // The following send action is equivalent to the
    // one above, but without a name.
    send getReading.reading to destination;
}

```

A SendAction can have both a sender (**via**) and receiver (**to**) parts, but it will generally have one or the other. When sending through a port (see [7.12](#) on ports), the PortUsage will usually be the sender (**via**), with the actual receiver determined by interface connections having the PortUsage as their source (see [7.14](#)).

```

part def MonitorDevice {
    port readingPort;
    action Monitoring {
        perform getReading { out reading : SensorReading; }
        send getReading.reading via readingPort;
    }
}

```

The base Element for a SendActionUsage is one of the following ActionUsages from the *Actions* library model (see [9.2.9](#)):

1. If the declared SendActionUsage is a feature of an ActionDefinition or ActionUsage, then
Action::sendSubactions.
2. Otherwise, *sendActions*.

An AcceptActionUsage is declared like a regular ActionUsage (as described above), except that it also includes an *accept parameter* placed following the action declaration and before the action body (if any), after the keyword **accept**, optionally followed by an Expression giving the transfer receiver, after the keyword **via**. If the declaration part is empty, then the **action** keyword may be omitted.

An accept parameter identifies the type of values accept any an AcceptActionUsage. It is declared as a ReferenceUsage (see [7.6](#)), but without the **ref** keyword, and shall not have a FeatureValue or body, but shall have at least one ownedSpecialization (FeatureTyping, Subsetting or Redefinition). If the accept parameter declaration has the form of a single qualified name (and, optionally, a multiplicity), then the qualified name shall be interpreted as the type of the accept parameter.

```

part controller {
    perform action {
        // ...
}

```

```

accept reading : SensorReading via controller;

// The following accept action is equivalent to the
// one above, but it does not name the accept parameter.
accept SensorReading via controller;

// ...
}

}

```

An accept parameter declaration can also include a FeatureValue or a flow source. In this case, the AcceptActionUsage will only accept exactly the value that is the result of the FeatureValue Expression or obtained from the incoming flow. The following special notations can also be used for the FeatureValue of an accept parameter:

- *Change trigger*. A change trigger is notated using the keyword **when** followed by an Expression whose result must be a *Boolean* value. A change trigger evaluates to a *ChangeSignal* (as defined in the Kernel Library [KerML]) that is sent when the result of the given Expression changes from *false* to *true* (or if it is *true* when first evaluated).
- *Absolute time trigger*. An absolute time trigger is notated using the keyword **at** followed by an Expression whose result must be a *TimeInstantValue* (see [9.8.8](#)). An absolute time trigger evaluates to a *TimeSignal* (as defined in the Kernel Library [KerML]) that is sent when the current time (relative to the *defaultClock*, see [9.8.8](#)) reaches the *TimeInstantValue* that is the result of the given Expression.
- *Relative time trigger*. A relative time trigger is notated using the keyword **after** followed by an Expression whose result must be a *DurationValue* (see [9.8.8](#)). An absolute time trigger evaluates to a *TimeSignal* (as defined in the Kernel Library [KerML]) that is sent when the current time (relative to the *defaultClock*, see [9.8.8](#)) reaches the *TimeInstantValue* that is computed as the result of the given Expression added to the time at which the time trigger is evaluated.

```

part controller {
    in level : Real;
    attribute threshold : Real;

    perform action {
        // Both of the following accept actions trigger (once) when the
        // given expression becomes true.
        accept : ChangeSignal = Triggers::triggerWhen({ level > threshold });
        accept when level > threshold;

        // The following accept action triggers at the given date and time.
        accept at Iso8601DateTime("2024-02-01T00:00:00Z");

        // The following accept action triggers 30 seconds after the evaluation
        // of its time trigger.
        accept after 30 [s];
    }
}

```

The base Element for an AcceptActionUsage is one of the following ActionUsages from the *Actions* library model (see [9.2.9](#)):

1. If the declared AcceptActionUsage is a feature of an ActionDefinition or ActionUsage, then *Action::acceptSubactions*.
2. Otherwise, *acceptActions*.

Every SendActionUsage or an AcceptActionUsage shall be one of the following:

1. A `ownedFeature` of an `ActionDefinition` or `ActionUsage`.
2. The owned `entry`, `do` or `exit` action of a `StateDefinition` or `StateUsage` (see [7.17](#)).
3. The owned `effect` action or `accept` action (`AcceptActionUsage` only) of a `TransitionUsage` (see [7.17](#)).

Assignment Actions

An `AssignmentActionUsage` is declared like a regular `ActionUsage` (as described above), except that it has an *assignment part* between the usual action declaration part and the action body (if any). An assignment part consists of the keyword `assign` followed by the target Expression and the referent Feature chain, separated by a dot (.), followed by the symbol `:=` and the value Expression. If the declaration part is empty, then the `action` keyword may be omitted.

```
action def UpdateVehiclePosition {
    in part sim : Simulation;
    in attribute deltaT : TimeDurationValue;

    // The target of the assignment below is "sim".
    // The referent feature chain is "vehicle.position".
    assign sim.vehicle.position :=
        sim.vehicle.position + sim.vehicle.velocity * deltaT;

    // After the above assignment "sim.vehicle.position" has the
    // value of the result of the assigned value expression,
    // evaluated at the time of the assignment.
}

action def RecordNames {
    in item record : Record;
    in item entries : Entry[1..*];

    // "entries.name" evaluates to the names of all entries.
    // These values are assigned to the "names" feature of "record".
    assign record.names := entries.name;
}
```

The base Element for an `AssignmentActionUsage` is one of the following `ActionUsages` from the *Actions* library model (see [9.2.9](#)):

1. If the declared `AssignmentActionUsage` is a `feature` of an `ActionDefinition` or `ActionUsage`, then `Action::assignments`.
2. Otherwise, `assignmentActions`.

Every `AssignmentActionUsage` shall be one of the following:

1. An `ownedFeature` of an `ActionDefinition` or `ActionUsage`.
2. The owned `entry`, `do` or `exit` action of a `StateDefinition` or `StateUsage` (see [7.17](#)).
3. The owned `effect` of a `TransitionUsage` (see [7.17](#)).

If the target Expression of an `AssignmentActionUsage` is omitted, then the target is implicitly the occurrence owning the `AssignmentActionUsage`. This is the case, in particular, for an assignment of an initial value to a Usage as specified by an initializing FeatureValue, that is, one with `isInitial = true`. An initializing FeatureValue is specified similarly to a binding FeatureValue (with `isInitial = false`), but using the symbol `:=` instead of `=`. The keyword `default` is included for `isDefault = true` and omitted for `isDefault = false`.

```
action {
    attribute count : Natural := 0;
```

```

    // ...
assign count := count + 1;
    // ...
}

part def User {
    attribute email : String;
    attribute userName : String default := email;
}

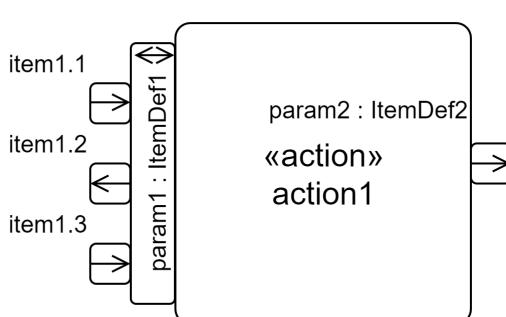
```

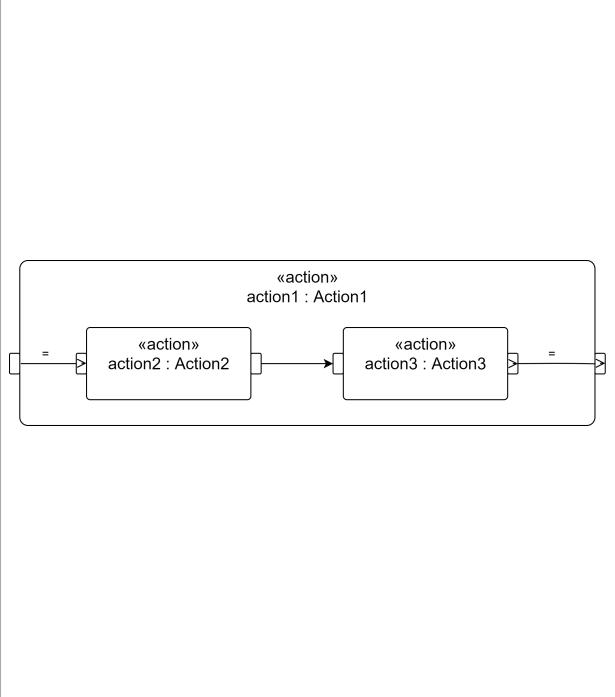
Graphical Notation

For Actions Graphical Notation BNF, see [8.2.3.16](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 22. Actions - Representative Notation

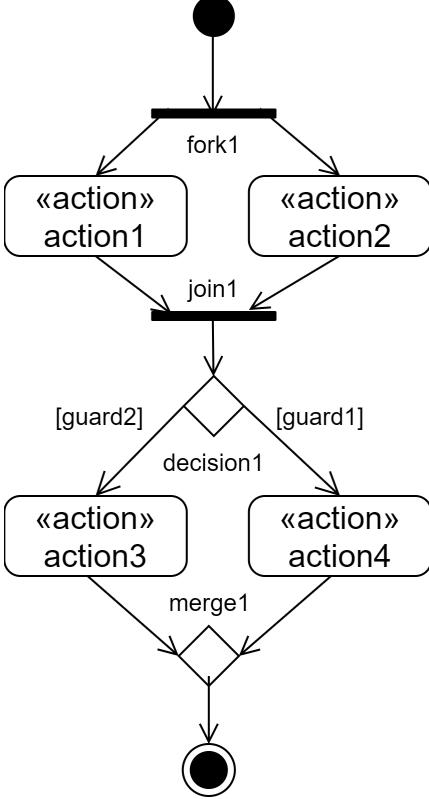
Element	Graphical Notation	Textual Notation
Action Definition	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> «action def» ActionDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> «action def» ActionDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> «compartment stack» ... </div>	action def ActionDef1; action def ActionDef1 { /* members */ }
Action	<div style="border: 1px solid black; padding: 5px; width: fit-content;"> «action» action1 : ActionDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> «action» action1 : ActionDef1 </div> <div style="border: 1px solid black; padding: 5px; width: fit-content;"> «compartment stack» ... </div>	action action1 : ActionDef1; action action1 : ActionDef1 { /* members */ }
Action with Parameters		item def ItemDef1 { in item 'item1.1'; out item 'item1.2'; in item 'item1.3'; } action action1 { inout param1 : ItemDef1; out param2 : ItemDef2; }

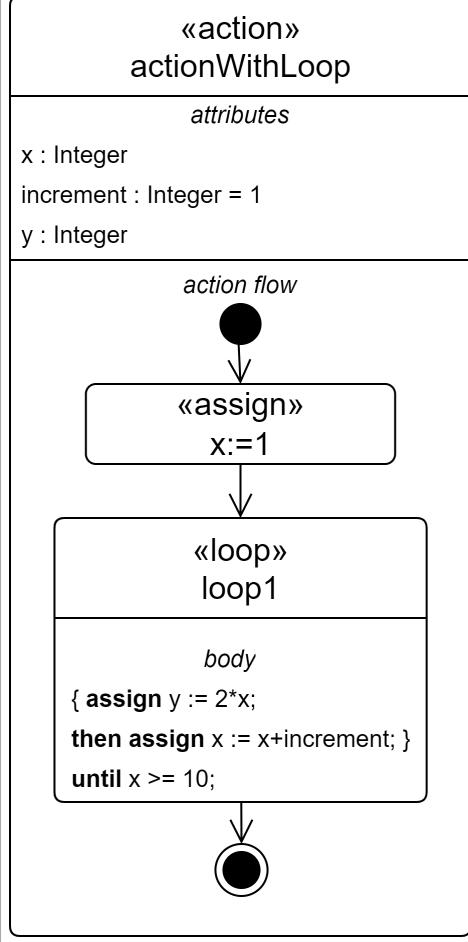
Element	Graphical Notation	Textual Notation
Action with Graphical Compartment showing standard action flow view		<pre> action action1 : Action1 { in input1; bind input1 = action2.input2; action action2 : Action2 { in input2; out output2; } flow action2.output2 to action3.input3; action action3 : Action3 { in input3; out output3; } bind action3.output3 = output1; out output1; } </pre>
Actions Compartment	<pre> <i>actions</i> ^action2 : ActionDef2 (in : ParamDef1, out : ParamDef2) action1 : ActionDef1 [1..*] ordered nonunique action3R : ActionDef3R redefines action3 action4R : ActionDef4R :>> action4 :>> action5 action6S : ActionDef6S [m] subsets action6 action7S : ActionDef7S [m] :> action7 action8R = action8 ref action9 : ActionDef9 perform action10 action11 action11.1 action11.2 ... </pre>	<pre> { action action1 : ActionDef1 [1..*] ordered nonunique; /* ... */ perform action action10; action action11 { action 'action11.1'; action 'action11.2'; } } </pre>

Element	Graphical Notation	Textual Notation
Perform Actions Compartment	<pre> <i>perform actions</i> ^action2 : ActionDef2 (in : ParamDef1, out : ParamDef2) action1 : ActionDef1 [1..*] ordered nonunique action3R : ActionDef3R redefines action3 action4R : ActionDef4R :>> action4 :>> action5 action6S : ActionDef6S [m] subsets action6 action7S : ActionDef7S [m] :> action7 action8R = action8 action11 action11.1 action11.2 ... </pre>	<pre> { perform action action1 : ActionDef1 [1..*] ordered nonunique; /* ... */ } </pre>

Element	Graphical Notation	Textual Notation
Perform Actions Swimlanes	<pre> graph TD Start(()) --> Action0[«action» action0] Action0 --> Part1[«part» part1 : PartDef1] Action0 --> Part2[«part» part2 : PartDef2] Part1 --> Action1[«action» action1] Part2 --> Action2[«action» action2] Action1 --> Action3[«action» action3] Action2 --> Action3 Action3 --> Action4[«action» action4] Action4 --> End(()) </pre>	<pre> package SwimLanes { part def Part0; part def Part1; part def Part2; part part0 : PartDef0 { perform action0; part part1 : PartDef1 { perform action0.action1; perform action0.action4; } part part2 : PartDef2 { perform action0.action2; perform action0.action3; } } action action0 { action action1; action action2; action action3; action action4; first start then action1; first action1 then action2; first action2 then action3; first action3 then action4; first action4 then done; } } </pre>

Element	Graphical Notation	Textual Notation
Parameters Compartment	<pre> parameters ^in param5 : ParamDef5 in param1 : ParamDef1 [1..*] ordered nonunique out param2 : ParamDef2 inout param3 : ParamDef3 return param4 : ParamDef4 in param6R : ParamDef6R redefines param6 in param7R : ParamDef7R >>param7 in >> param8 in param9S : ParamDef9S [m] subsets param9 in param10S : ParamDef10S [m] > param10 in param11 : ParamDef11 = expression1 ... </pre>	<pre> { in param1 : ParamDef [1..*] ordered nonunique; /* ... */ } </pre>
Actions with and without Conditional Succession	<pre> graph TD A["«action» action1 : Action1"] -- "[guard1]" --> B["«action» action2 : Action2"] C["«action» action1 : Action1"] -- "" --> D["«action» action2 : Action2"] </pre>	<pre> action action1 : Action1; action action2 : Action2; succession action1 if guard1 then action2; or action action1 : Action1; if guard1 then action2; action action2 : Action2; </pre>

Element	Graphical Notation	Textual Notation
Actions with Control Nodes	 <pre> graph TD start(()) --> fork1[] fork1 --> action1["«action» action1"] fork1 --> action2["«action» action2"] action1 --> join1[] action2 --> join1 join1 --> decision1{ } decision1 -- "[guard2]" --> action3["«action» action3"] decision1 -- "[guard1]" --> action4["«action» action4"] action3 --> merge1{ } action4 --> merge1 merge1 --> done((())) </pre>	<pre> first start; then fork fork1; then action1; then action2; action action1; then join1; action action2; then join1; join join1; then decide decision1; if guard2 then action3; if guard1 then action4; action action3; then merge1; action action4; then merge1; merge merge1; then done; </pre>
Performed By Compartment		No textual notation

Element	Graphical Notation	Textual Notation
Action with Loop (body in textual notation)	 <pre> <<action>> actionWithLoop attributes x : Integer increment : Integer = 1 y : Integer action flow <<assign>> X:=1 <<loop>> loop1 body { assign y := 2*x; then assign x := x+increment; } until x >= 10; </pre>	<pre> package Loop { action actionWithLoop { attribute x:Integer := 1; attribute increment:Integer = 1; attribute y:Integer; loop action loop1 { assign y := 2*x; then assign x := x+increment; } until x >= 10; then done; } } </pre>

Element	Graphical Notation	Textual Notation
Action with Loop (body in graphical notation)	<p>«action» actionWithLoop</p> <p>attributes</p> <p>x : Integer increment : Integer = 1 y : Integer</p> <pre> graph TD Start(()) --> Assign1["<<assign>> x:=1"] Assign1 --> Loop1["<<loop>> loop1"] Loop1 --> Assign2["<<assign>> y := 2*x"] Assign2 --> Assign3["<<assign>> x := x+increment"] Assign3 --> Decision1{ } Decision1 --> ExitNode(()) Decision1 --> ElseAssign["<<assign>> x := x+increment"] ElseAssign --> Decision2{ } Decision2 --> ExitNode Decision2 --> Condition{x >= 10} Condition --> ExitNode </pre>	<pre> package Loop { action actionWithLoop { attribute x:Integer := 1; attribute increment:Integer = 1; attribute y:Integer; loop action loop1 { assign y := 2*x; then assign x := x+increment; } until x >= 10; then done; } } </pre>

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 subsetted 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.

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

For States Abstract Syntax class descriptions, see [8.3.17](#).

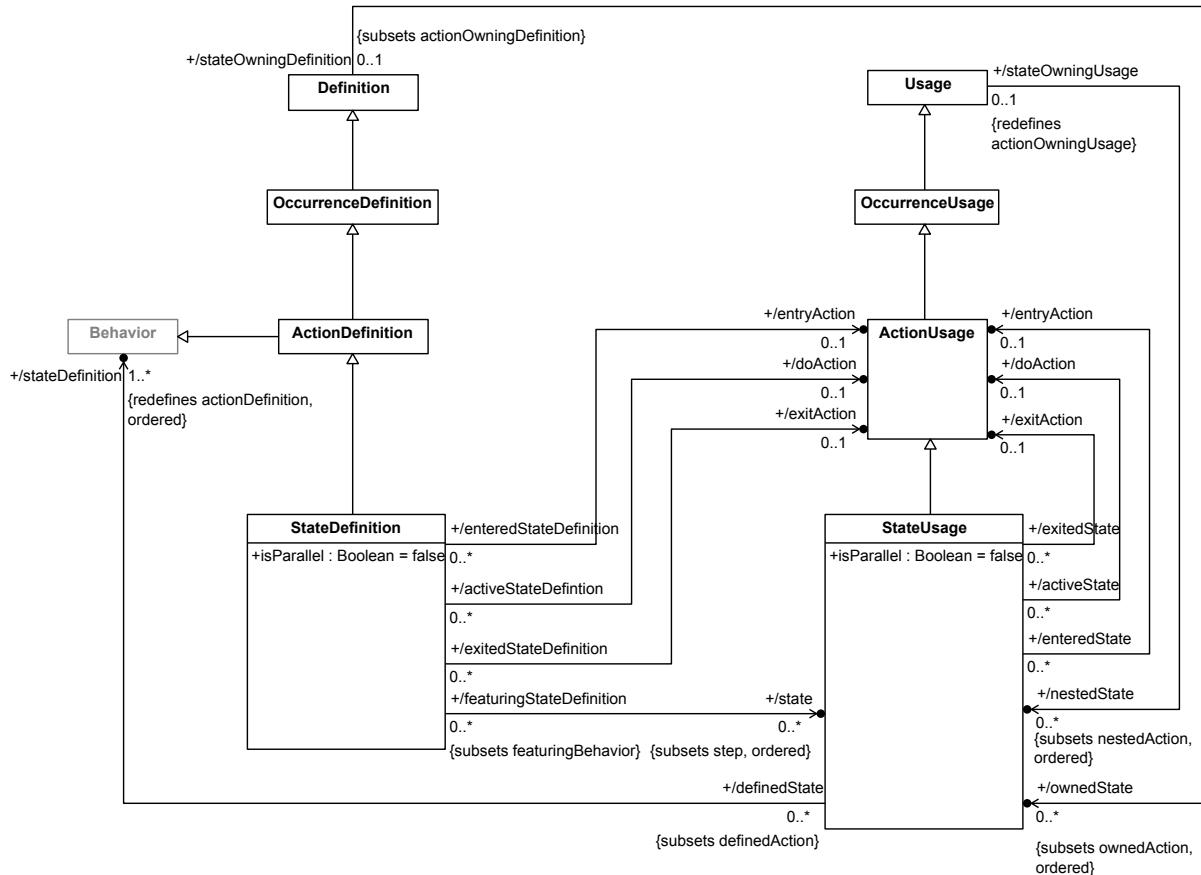


Figure 31. State Definition and Usage

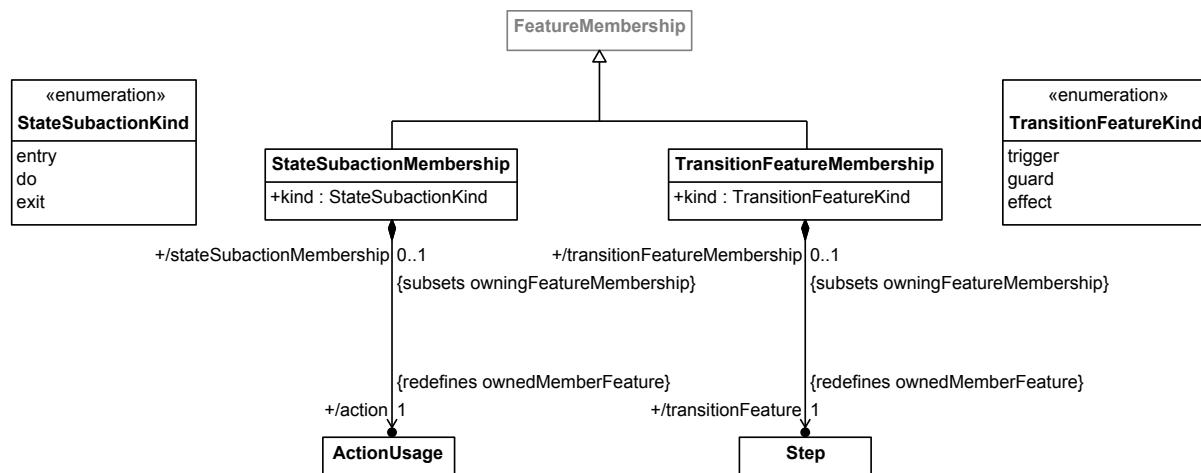


Figure 32. State Membership

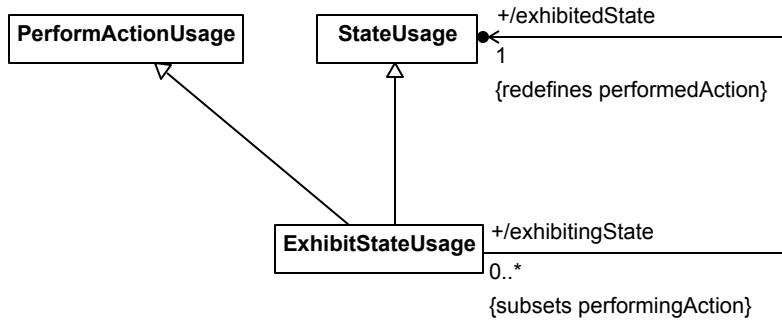


Figure 33. State Exhibition

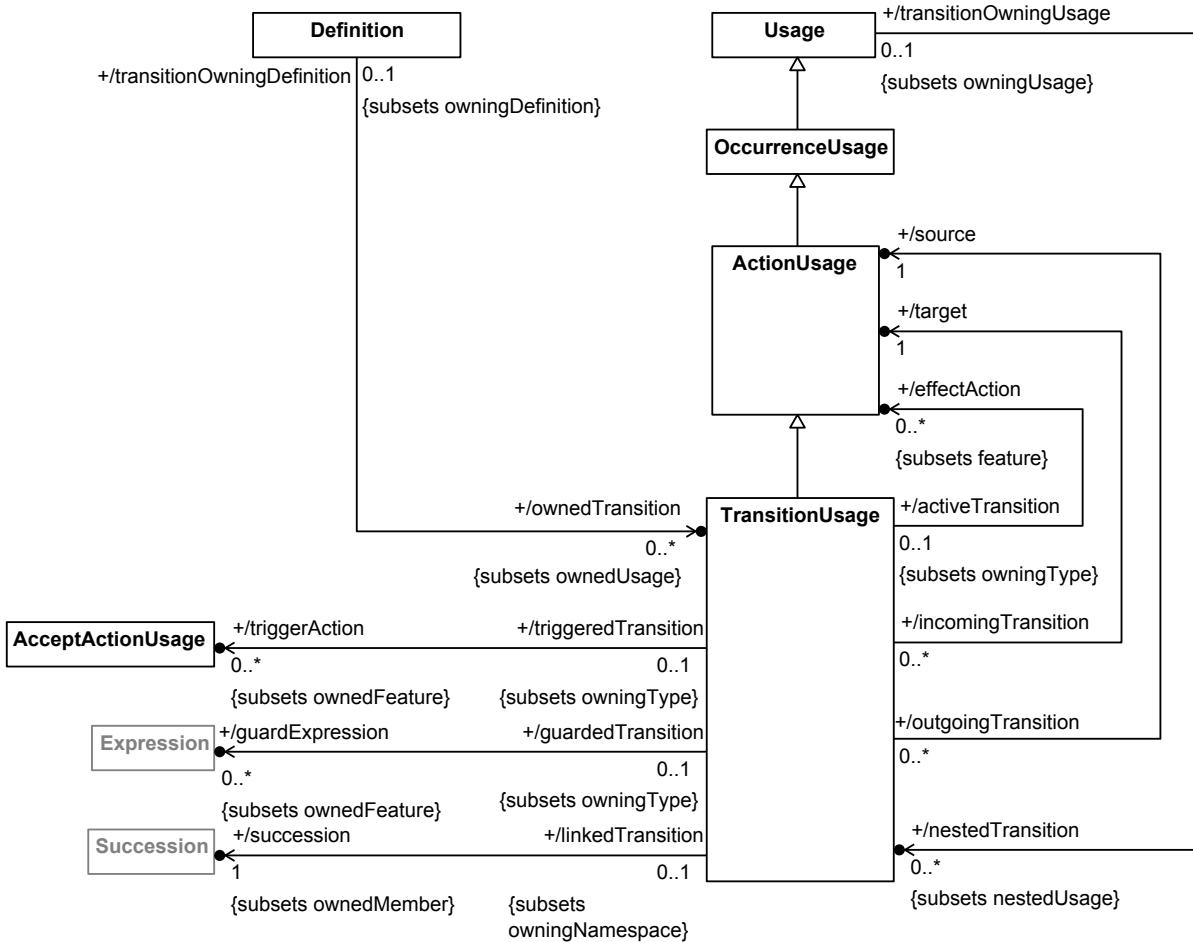


Figure 34. Transition Usage

7.17.3 Notation

Textual Notation

For States Textual Notation BNF, see [8.2.2.17](#).

Release Note. Details to be provided.

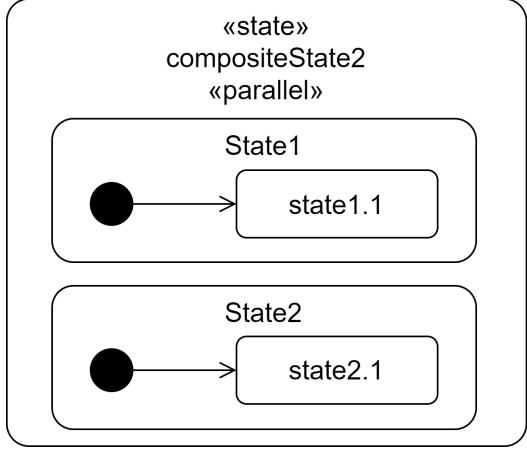
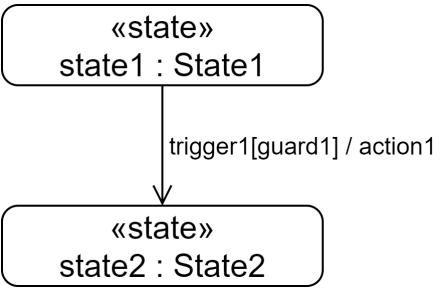
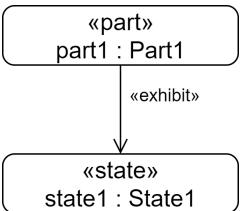
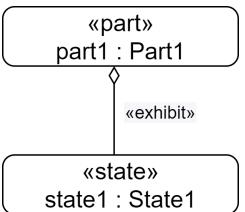
Graphical Notation

For States Graphical Notation BNF, see [8.2.3.17](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 23. States - Representative Notation

Element	Graphical Notation	Textual Notation
State Definition	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«state def»</p> <p>StateDef1</p> </div> <div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«state def»</p> <p>StateDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>state def StateDef1; state def StateDef1 { /* members */ }</pre>
State	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«state»</p> <p>state1 : StateDef1</p> </div> <div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«state»</p> <p>state1 : StateDef1</p> <p>compartment stack</p> <p>...</p> </div>	<pre>state state1 : StateDef1; state state1 : StateDef1 { /* members */ }</pre>
State	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«state»</p> <p>state1 : StateDef1</p> <p>actions</p> <p>entry action1</p> <p>do action2</p> <p>exit action3</p> </div>	<pre>state state1 { entry entryAction; do doAction; exit exitAction; }</pre>
State with Graphical Compartment with standard state transition view (sequential states)	<div style="border: 1px solid black; padding: 10px; width: fit-content;"> <p>«state»</p> <p>compositeState1</p> </div>	<pre>state compositeState1 { entry; then state1; state state1; transition first state1 accept trigger1 if guard1 do action1 then state2; state state2; then done; }</pre>

Element	Graphical Notation	Textual Notation
State with Graphical Compartment with standard state transition view (parallel states)	 <pre> <<state>> compositeState2 <<parallel>> State1 State1 --> state1.1 State2 State2 --> state2.1 </pre>	<pre> state compositeState2 parallel { state state1 { entry; then 'state1.1'; state 'state1.1'; } state state2 { entry; then 'state2.1'; state 'state2.1'; } } </pre>
Transition	 <pre> <<state>> state1 : State1 state1 --> state2 : State2 trigger1[guard1] / action1 </pre>	<pre> state state1 : State1; state state2 : State2; transition first state1 accept trigger1 if guard1 do action1 then state2; </pre> <p>or</p> <pre> state state1 : State1; accept trigger1 if guard1 do action1 then state2; state state2 : State2; </pre>
Exhibit	 <pre> <<part>> part1 : Part1 part1 --> state1 : State1 <<exhibit>> </pre>	<pre> part part1 : Part1 { exhibit state1; } </pre>
Exhibit State	 <pre> <<part>> part1 : Part1 part1 --> state1 : State1 <<exhibit>> </pre>	<pre> part part1 : Part1 { exhibit state state1 : State1; } </pre>

Element	Graphical Notation	Textual Notation
States Compartment	<pre> states ^state2 : StateDef2 state1 : StateDef1 [1..*] ordered nonunique state3R : StateDef3R redefines state3 state4R : StateDef4R :>> state4 :>> state5 state6S : StateDef6S [m] subsets state6 state7S : StateDef7S [m] :> state7 state8R = state8 ref state9 : StateDef9 exhibit state10 state11 state11.1 state11.2 ... </pre>	<pre> { state state1 : StateDef [1..*] ordered nonunique; /* ... */ exhibit state state10; state state11 { state 'state11.1'; state 'state11.2'; } } </pre>
Exhibit States Compartment	<pre> exhibit states ^state2 : StateDef2 state1 : StateDef1 [1..*] ordered nonunique state3R : StateDef3R redefines state3 state4R : StateDef4R :>> state4 :>> state5 state6S : StateDef6S [m] subsets state6 state7S : StateDef7S [m] :> state7 state8R = state8 state11 state11.1 state11.2 ... </pre> <div data-bbox="605 1311 894 1512" style="border: 1px solid black; padding: 5px; width: 150px;"> <p>exhibits</p> ^state2 state1 ... </div>	<pre> { exhibit state state1 : StateDef [1..*] ordered nonunique; /* ... */ } </pre>
Exhibited By Compartment	<div data-bbox="597 1533 902 1638" style="border: 1px solid black; padding: 5px; width: 187px;"> <p>exhibited by</p> item1 : ItemDef1 </div>	No textual notation

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

For Calculations Abstract Syntax class descriptions, see [8.3.18](#).

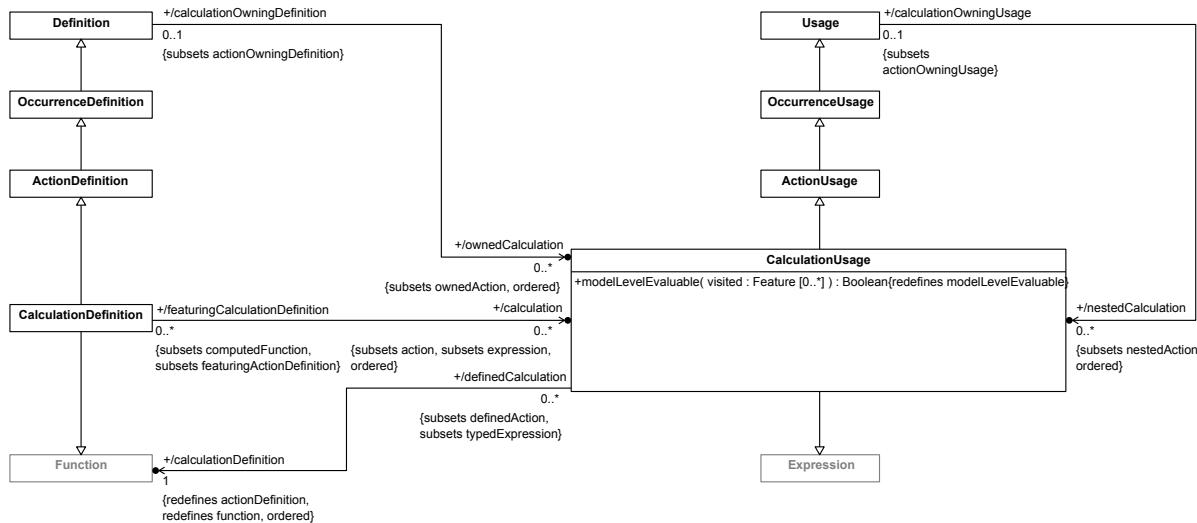


Figure 35. Calculation Definition and Usage

7.18.3 Notation

Textual Notation

For Calculations Textual Notation BNF, see [8.2.2.18](#).

Release Note. Details to be provided.

Graphical Notation

For Calculations Graphical Notation BNF, see [8.2.3.18](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 24. Calculations - Representative Notation

Element	Graphical Notation	Textual Notation
Calc Definition	<div style="border: 1px solid black; padding: 10px;"> <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;"> result expression1 </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;"> compartment stack ... </div> </div>	<pre>calc def CalcDef1 { expression1 }</pre> <pre>calc def CalcDef1 { /* members */ }</pre>
Calc	<div style="border: 1px solid black; padding: 10px;"> <div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> «calc» calc1 : CalcDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> result expression1 </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> «calc» calc1 : CalcDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-bottom: 10px;"> compartment stack ... </div> </div>	<pre>calc calc1 : CalcDef1 { expression1 }</pre> <pre>calc calc1 : CalcDef1 { /* members */ }</pre>

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 constraint 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

For Constraints Abstract Syntax class descriptions, see [8.3.19](#).

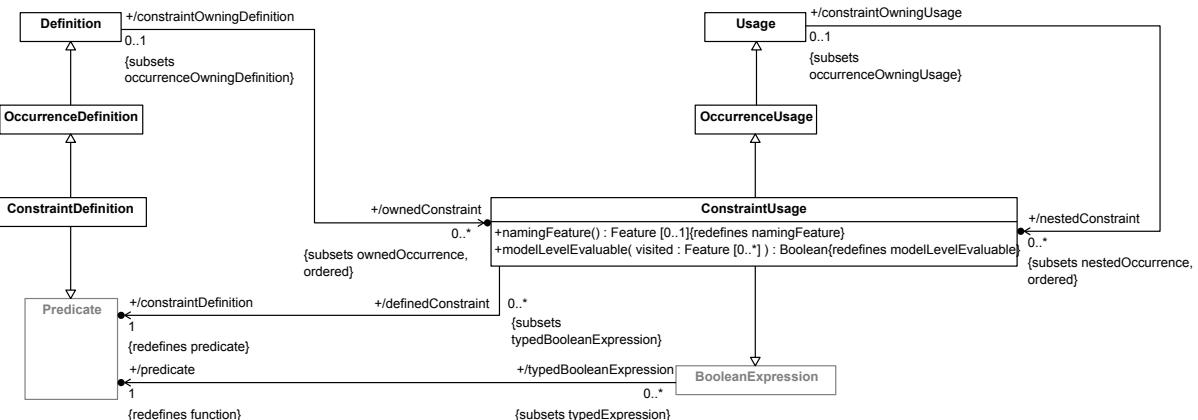


Figure 36. Constraint Definition and Usage

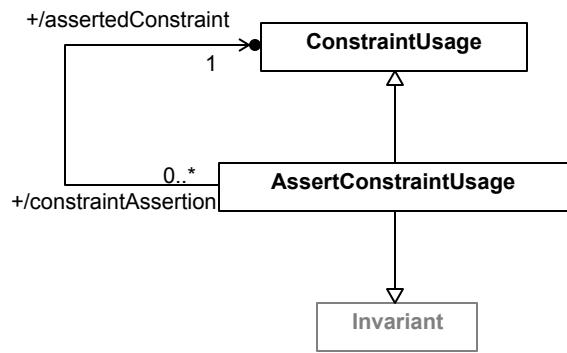


Figure 37. Constraint Assertion

7.19.3 Notation

Textual Notation

For Constraints Textual Notation BNF, see [8.2.2.19](#).

Release Note. Details to be provided.

Graphical Notation

For Constraints Graphical Notation BNF, see [8.2.3.19](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 25. Constraints - Representative Notation

Element	Graphical Notation	Textual Notation
Constraint Definition	<div style="border: 1px solid black; padding: 5px;"> «constraint def» ConstraintDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> «constraint def» ConstraintDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> compartment stack ... </div>	<pre> constraint def ConstraintDef1; constraint def ConstraintDef1 { /* members */ } </pre>
Constraint	<div style="border: 1px solid black; padding: 5px;"> «constraint» constraint1 : ConstraintDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> «constraint» constraint1 : ConstraintDef1 </div> <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> compartment stack ... </div>	<pre> constraint constraint1 : ConstraintDef1; constraint constraint1 : ConstraintDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Constraints Compartment	<pre> constraints ^constraint2 : ConstraintDef2 constraint1 : ConstraintDef1 [1..*] ordered nonunique constraint3R : ConstraintDef3R redefines constraint3 constraint4R : ConstraintDef4R >> constraint4 :>> constraint5 constraint6S : ConstraintDef6S [m] subsets constraint6 constraint7S : ConstraintDef7S [m] >> constraint7 constraint8R = constraint8 ref constraint9 : ConstraintDef9 assert constraint10 assert {boolean_expression1} {boolean_expression1} constraint11 require constraint12 assume constraint13 ... </pre>	<pre> { constraint constraint1 : ConstraintDef1 [1..*] ordered nonunique; /* ... */ assert constraint constraint10; constraint {boolean_expression1} }</pre>
Assert Constraints Compartment	<div style="border: 1px solid black; padding: 5px;"> assert constraints constraint10 {boolean_expression1} ... </div>	<pre> { assert constraint constraint1 : ConstraintDef1 [1..*] ordered nonunique; /* ... */ assert constraint {boolean_expression1} }</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

For Requirements Abstract Syntax class descriptions, see [8.3.20](#).

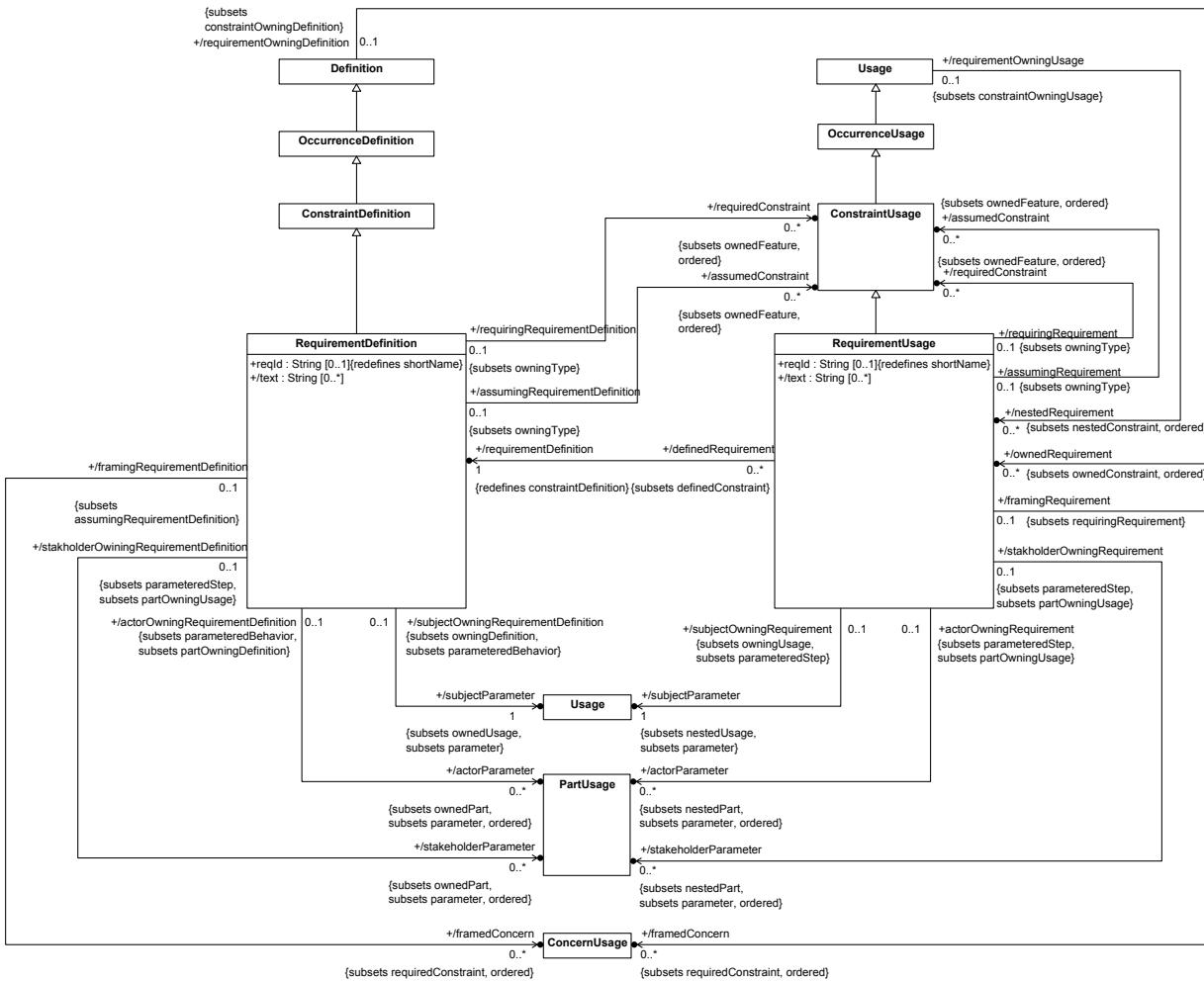


Figure 38. Requirement Definition and Usage

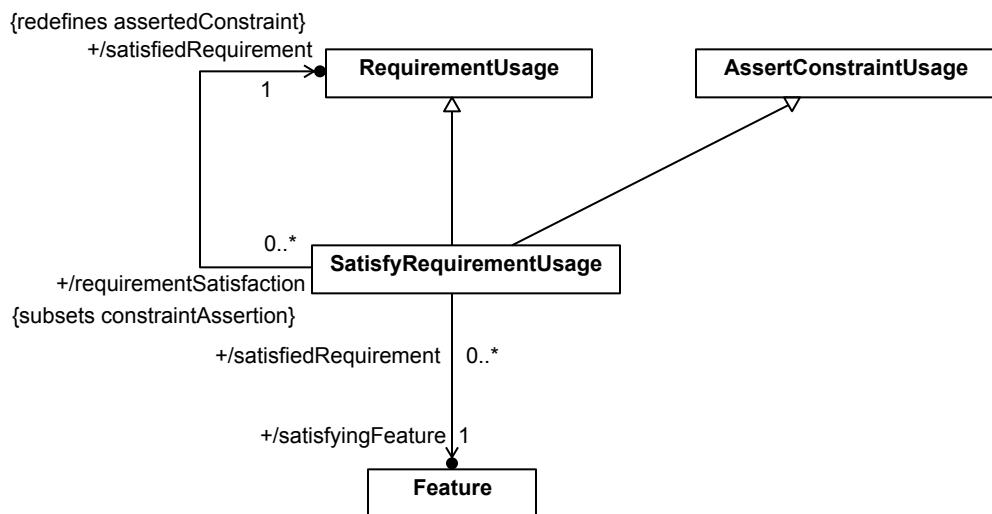


Figure 39. Requirement Satisfaction

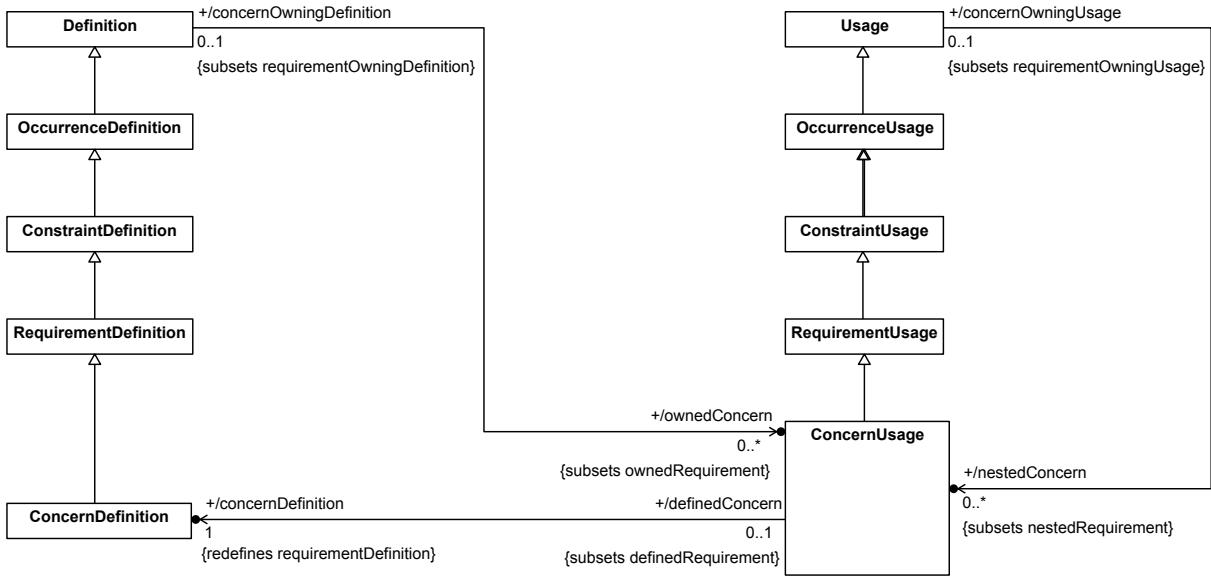


Figure 40. Concern Definition and Usage

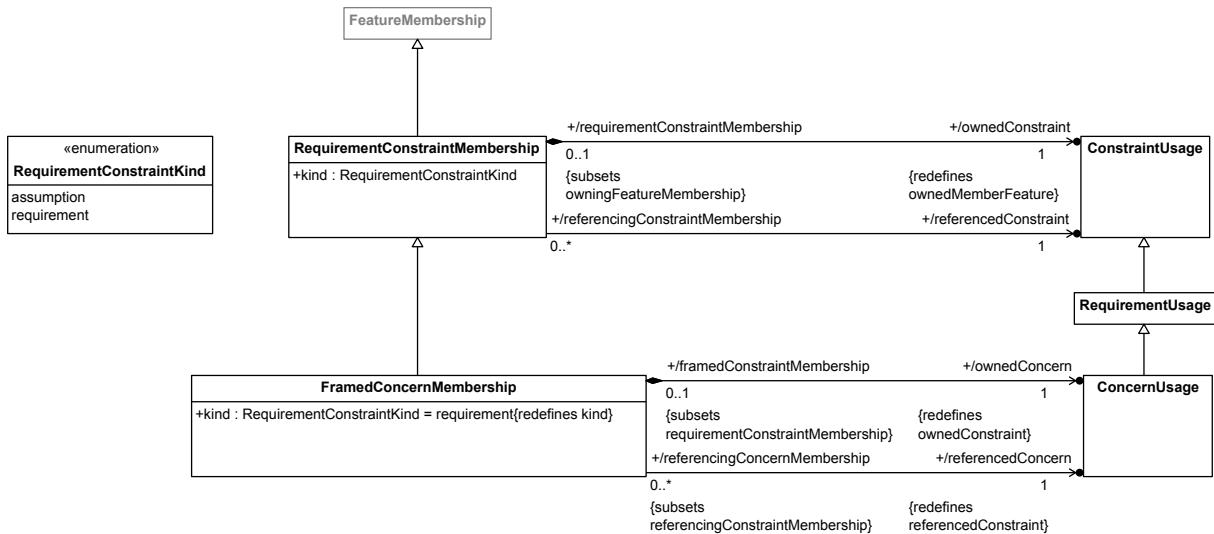


Figure 41. Requirement Constraint Membership

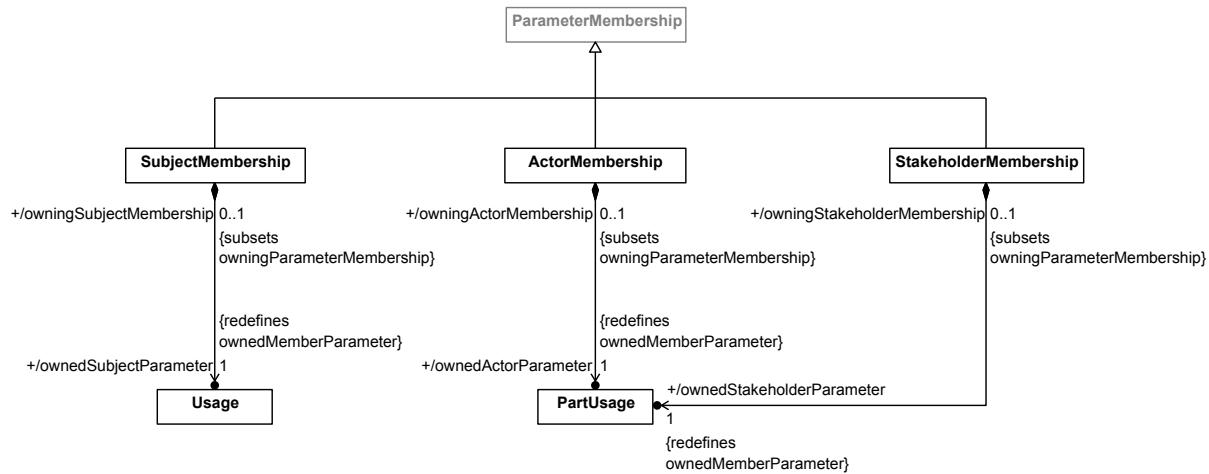


Figure 42. Requirement Parameter Memberships

7.20.3 Notation

Textual Notation

For Requirements Textual Notation BNF, see [8.2.2.20](#).

Release Note. Details to be provided.

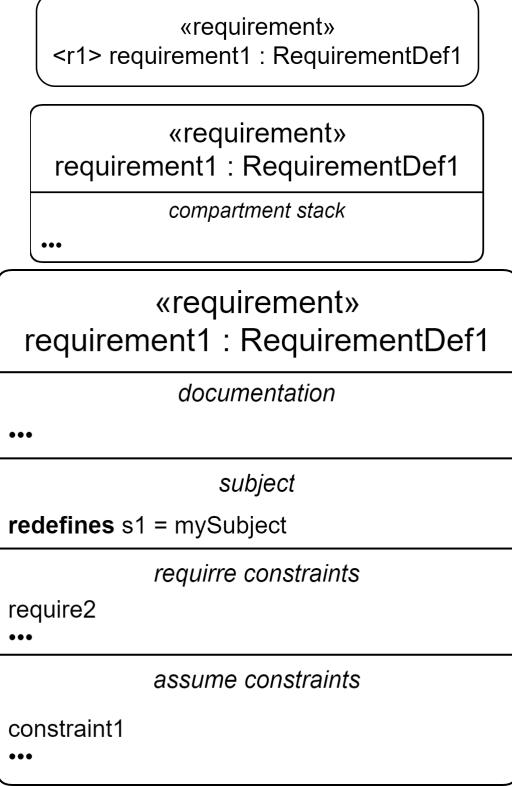
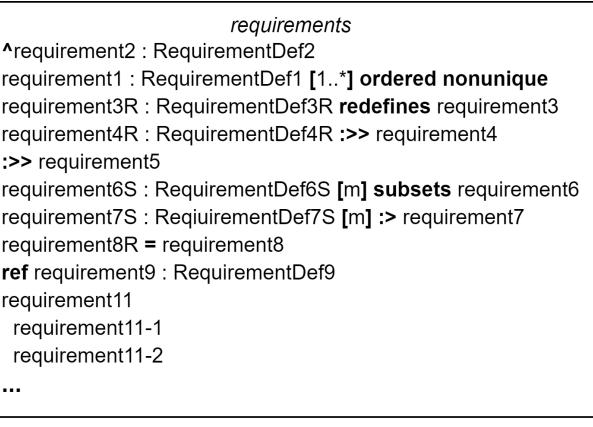
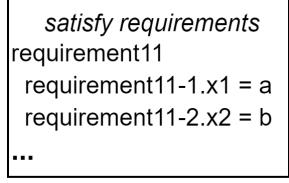
Graphical Notation

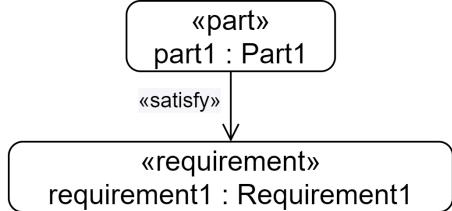
For Requirements Graphical Notation BNF, see [8.2.3.20](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 26. Requirements - Representative Notation

Element	Graphical Notation	Textual Notation
Requirement Definition	<div style="border: 1px solid black; padding: 10px; text-align: center;"> «requirement def» <R1> RequirementDef1 </div> <div style="border: 1px solid black; padding: 10px; text-align: center;"> «requirement def» RequirementDef1 <div style="border: 1px solid black; padding: 5px; margin-top: 5px;"> <i>compartment stack</i> <i>...</i> </div> </div>	<pre> requirement def <R1> RequirementDef1 { subject s1 : Subject1; } requirement def RequirementDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Requirement	 <pre> <> requirement <r1> requirement1 : RequirementDef1 <> requirement requirement1 : RequirementDef1 compartment stack ... <> requirement requirement1 : RequirementDef1 documentation ... subject redefines s1 = mySubject require constraints require2 ... assume constraints constraint1 ... </pre>	<pre> requirement <r1> requirement1 : RequirementDef1 { subject redefines s1 = mySubject; } requirement requirement1 : RequirementDef1 { doc /* ... */ subject redefines s1 = mySubject; require require2; assume constraint1; } </pre>
Requirements Compartment	 <pre> requirements ^requirement2 : RequirementDef2 requirement1 : RequirementDef1 [1..*] ordered nonunique requirement3R : RequirementDef3R redefines requirement3 requirement4R : RequirementDef4R >> requirement4 :>> requirement5 requirement6S : RequirementDef6S [m] subsets requirement6 requirement7S : RequirementDef7S [m] >> requirement7 requirement8R = requirement8 ref requirement9 : RequirementDef9 requirement11 requirement11-1 requirement11-2 ... </pre>	<pre> { requirement requirement1 : RequirementDef1 [1..*] ordered nonunique; /* ... */ } </pre>
Satisfy Requirements Compartment	 <pre> satisfy requirements requirement11 requirement11-1.x1 = a requirement11-2.x2 = b ... </pre>	<pre> part part1 { satisfy requirement11 by part1 { bind requirement11.x1 = a; bind requirement11.x2 = b; } } </pre>

Element	Graphical Notation	Textual Notation
Satisfy	 <pre> graph TD part["<<part>> part1 : Part1"] -- "<<satisfy>>" --> req["<<requirement>> requirement1 : Requirement1"] </pre>	requirement requirement1 : Requirement1; part part1 : Part1; satisfy requirement1 by part1;

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

For Cases Abstract Syntax class descriptions, see [8.3.22](#).

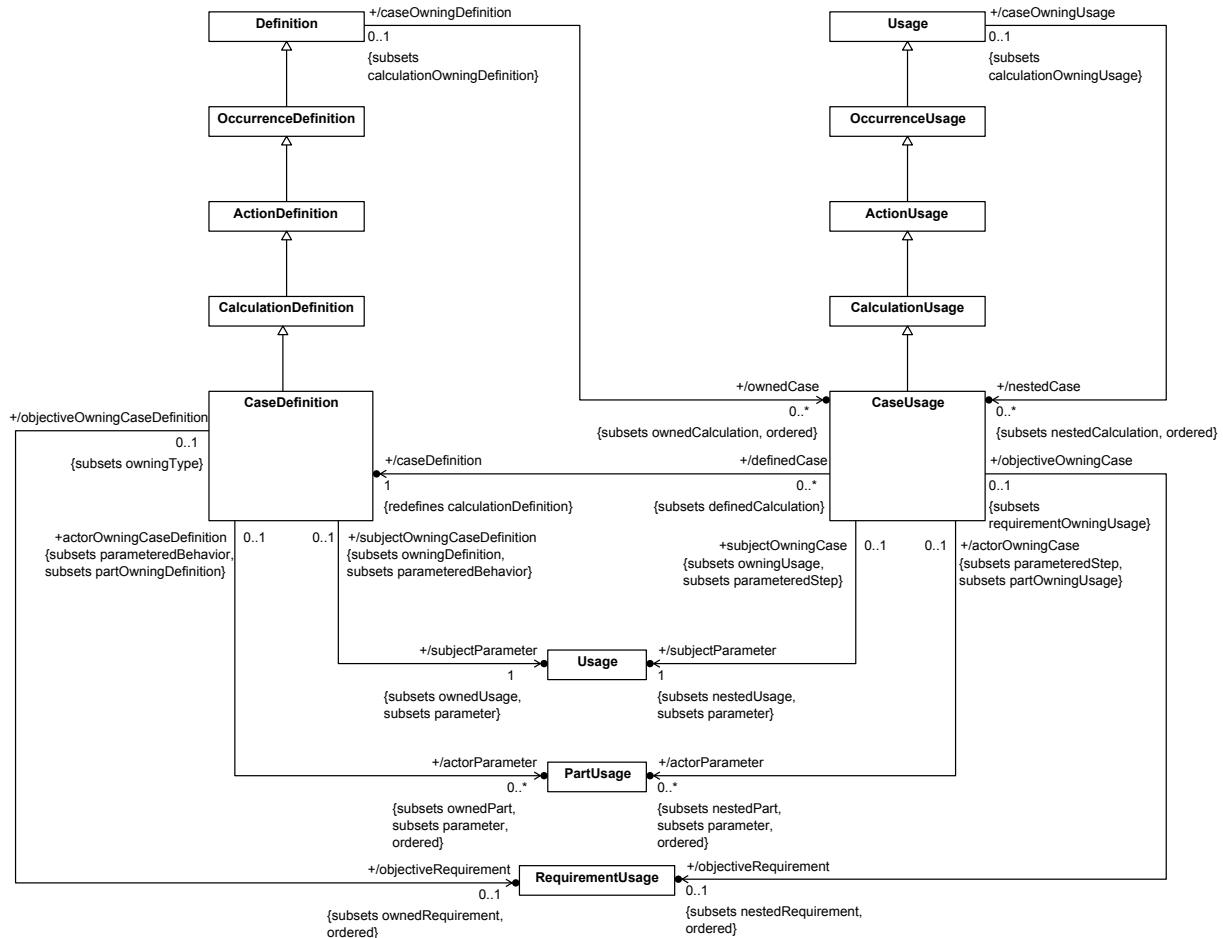


Figure 43. Case Definition and Usage

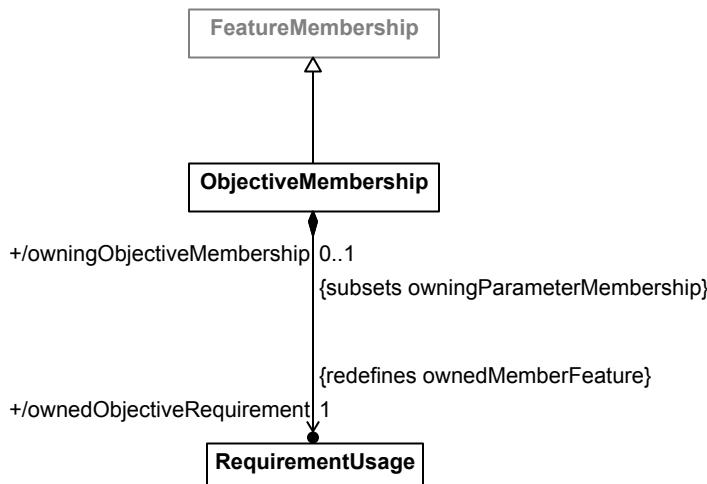


Figure 44. Case Membership

7.21.3 Notation

Textual Notation

For Cases Textual Notation BNF, see [8.2.2.21](#).

Release Note. Details to be provided.

Graphical Notation

For Cases Graphical Notation BNF, see [8.2.3.21](#).

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

For Analysis Cases Abstract Syntax class descriptions, see [8.3.22](#).

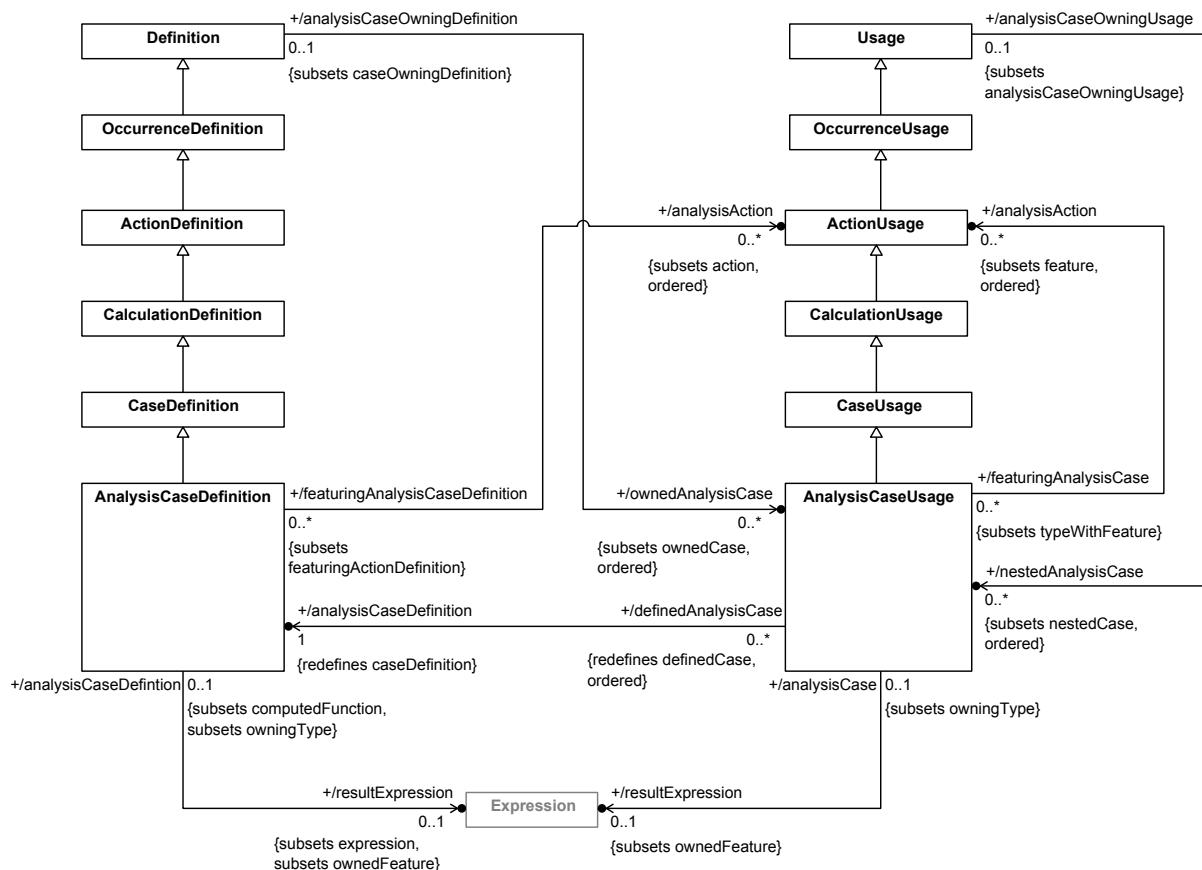


Figure 45. Analysis Case Definition and Usage

7.22.3 Notation

Textual Notation

For Analysis Cases Textual Notation BNF, see [8.2.2.22](#).

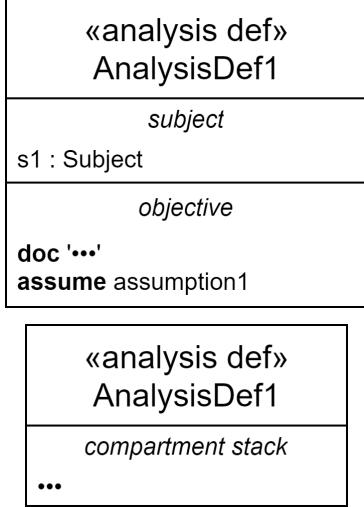
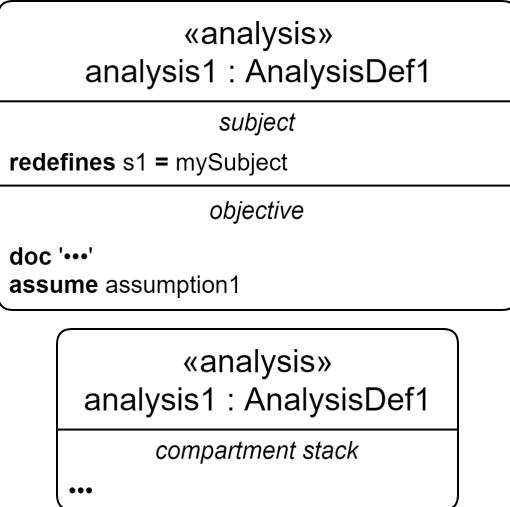
Release Note. Details to be provided.

Graphical Notation

For Analysis Cases Graphical Notation BNF, see [8.2.3.22](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 27. Analysis Cases - Representative Notation

Element	Graphical Notation	Textual Notation
Analysis Case Definition		<pre> analysis def AnalysisDef1 { subject s1 : Subject; objective { doc /* '...' */; assume assumption1; } } analysis def AnalysisDef1 { /* members */ } </pre>
Analysis Case		<pre> analysis analysis1 : AnalysisDef1 { subject redefines s1 = mySubject; objective { doc /* '...' */; assume assumption1; } } analysis analysis1 : AnalysisDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Analyses Compartment	<pre> «analysis» analysis1 : AnalysisDef1 subject redefines s1 = mySubject objective doc '...' assume assumption1 analyses ^analysis3 : AnalysisDef3 analysis4 : AnalysisDef4 </pre>	

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

For Verification Cases Abstract Syntax class descriptions, see [8.3.23](#).

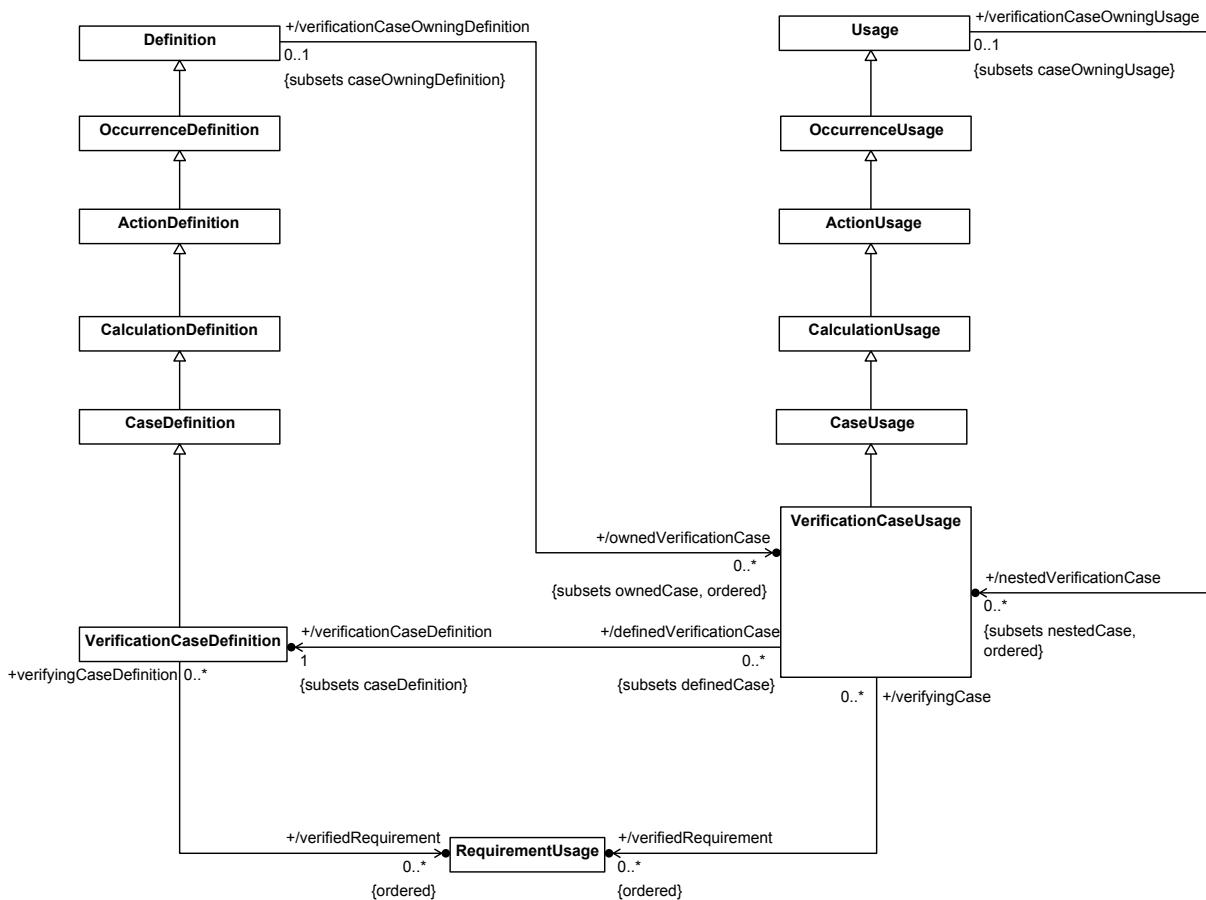


Figure 46. Verification Case Definition and Usage

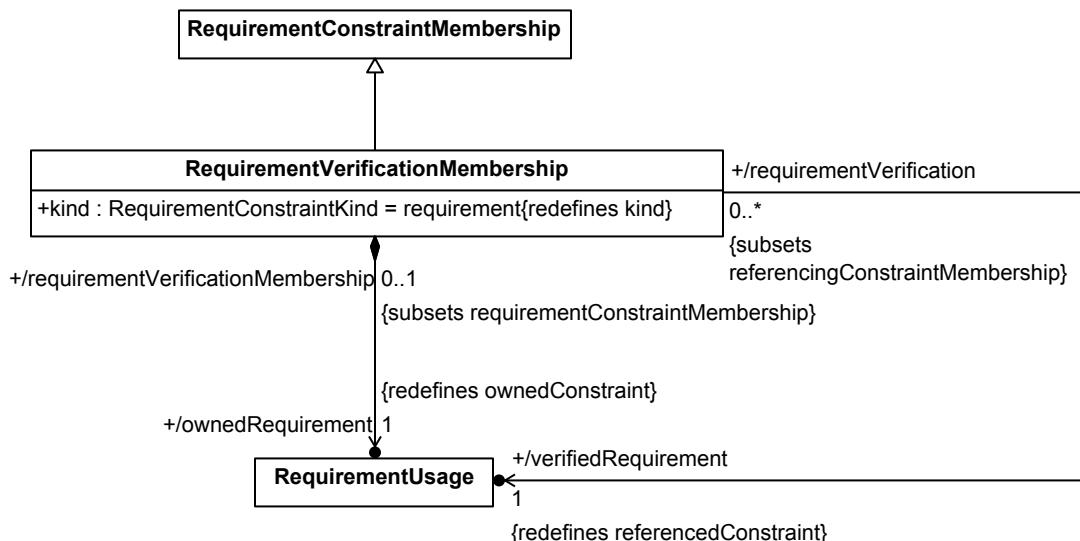


Figure 47. Verification Membership

7.23.3 Notation

Textual Notation

For Verification Cases Textual Notation BNF, see [8.2.2.23](#).

Release Note. Details to be provided.

Graphical Notation

For Verification Cases Graphical Notation BNF, see [8.2.3.23](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 28. Verification Cases - Representative Notation

Element	Graphical Notation	Textual Notation
Verification Case Definition	<div style="border: 1px solid black; padding: 10px;"> <p>«verification def»</p> <p>VerificationDef1</p> <hr/> <p><i>subject</i></p> <p>s1 : Subject1</p> <hr/> <p><i>objective</i></p> <p>doc objective statement</p> <p>verify requirement1</p> <p>...</p> </div> <div style="border: 1px solid black; padding: 10px;"> <p>«verification def»</p> <p>VerificationDef1</p> <hr/> <p><i>compartment stack</i></p> <p>...</p> </div>	<pre> verification def VerificationDef1 { subject s1 : Subject1; objective { doc /* '...' */ verify requirement1; } } verification def VerificationDef1 { /* members */ } </pre>
Verification Case	<div style="border: 1px solid black; padding: 10px;"> <p>«verification»</p> <p>verification1 : VerificationDef1</p> <hr/> <p><i>subject</i></p> <p>redefines s1 = mySubject</p> <hr/> <p><i>objective</i></p> <p>doc objective statement</p> <p>verify requirement2</p> <p>...</p> </div> <div style="border: 1px solid black; padding: 10px;"> <p>«verification»</p> <p>verification1 : VerificationDef1</p> <hr/> <p><i>compartment stack</i></p> <p>...</p> </div>	<pre> verification verification1 : VerificationDef1 { subject redefines s1 = mySubject; objective { doc /* '...' */ verify requirement1; } } verification verification1 : VerificationDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
Verified Requirements Compartment		
Verifications Compartment	<pre> <i>verifications</i> verification1:VerificationDef1 [1..*] ordered nonunique ^verification2:VerificationDef2 (in :ParamDef1, out :ParamDef2) verification3R:VerificationDef3R redefines verification3 verification4R:VerificationDef4R:>>verification4 :>>verification5 verification6S:VerificationDef6S [m] subsets verification6 verification7S:VerificationDef7S [m] :> verification7 verification8R = verification8 ref verification9:VerificationDef9 perform verification10 verification11 verification11.1 verification11.2 ... </pre>	<pre>{ verification verification1 : VerificationDef1 [1..*] ordered nonunique; /* ... */ perform verification verification10; verification verification11 { verification 'verification11.1'; verification 'verification11.2'; } }</pre>
Verification Methods Compartment	<pre> <i>verification methods</i> inspection analysis demonstration test ... </pre>	<pre> metadata VerificationMethod { kind = (inspection, analysis, demonstration, test); } </pre>
Verifies Compartment	<pre> <i>verifies</i> requirement1 requirement2 ... </pre>	<pre> objective { verify requirement1; verify requirement2; } </pre>
Verify	<pre> <i>«verification»</i> verificationCase1 : VerificationCase1 ↓ <i>«verify»</i> ↓ <i>«requirement»</i> requirement1 : Requirement1 </pre>	<pre> requirement requirement1 : Requirement1; verification verificationCase1 : VerificationCase1 { objective { verify requirement1; } } </pre>

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).

Release 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

For Use Cases Abstract Syntax class descriptions, see [8.3.24](#).

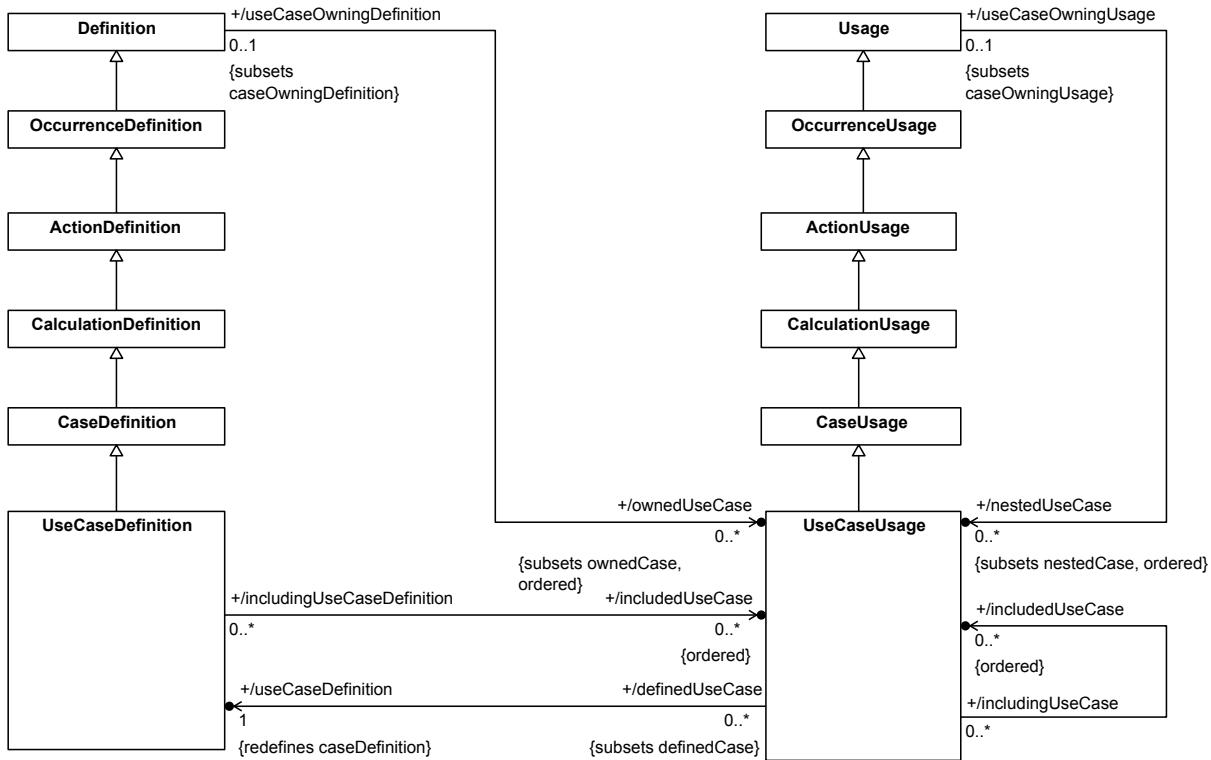


Figure 48. Use Case Definition and Usage

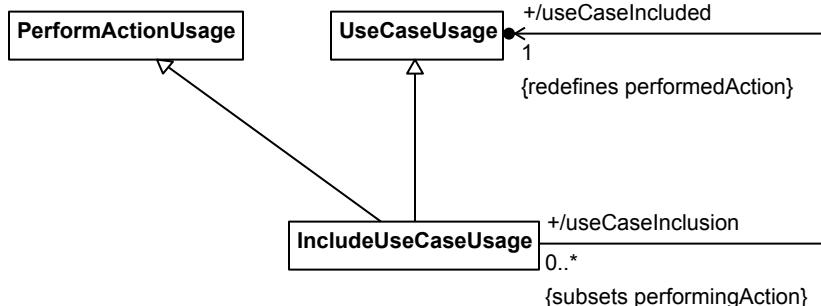


Figure 49. Use Case Inclusion

7.24.3 Notation

Textual Notation

For Use Cases Textual Notation BNF, see [8.2.2.24](#).

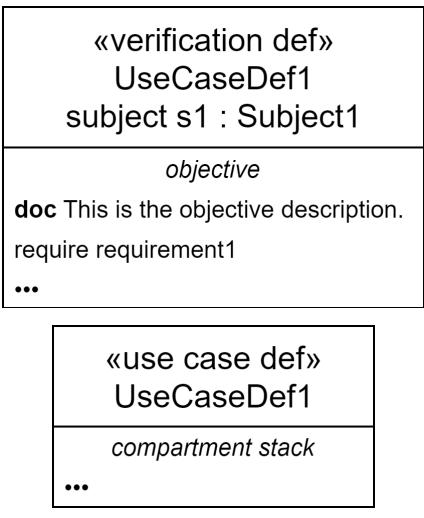
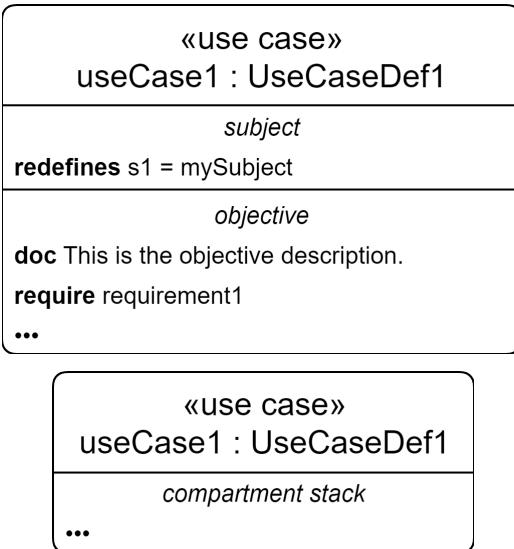
Release Note. Details to be provided.

Graphical Notation

For Use Cases Graphical Notation BNF, see [8.2.3.24](#).

For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 29. Use Cases - Representative Notation

Element	Graphical Notation	Textual Notation
Use Case Definition	 <p>«verification def» UseCaseDef1 subject s1 : Subject1</p> <p><i>objective</i> doc This is the objective description. require requirement1 ...</p> <p>«use case def» UseCaseDef1</p> <p><i>compartment stack</i> ...</p>	<pre>use case def UseCaseDef1 { subject s1:Subject1; objective { doc /* '...' */ require requirement1; } } use case def UseCaseDef1 { /* members */ }</pre>
Use Case	 <p>«use case» useCase1 : UseCaseDef1</p> <p><i>subject</i> redefines s1 = mySubject</p> <p><i>objective</i> doc This is the objective description. require requirement1 ...</p> <p>«use case» useCase1 : UseCaseDef1</p> <p><i>compartment stack</i> ...</p>	<pre>use case useCase1 : UseCaseDef1 { subject redefines s1 = mySubject; objective { doc /* '...' */ require requirement1; } } use case useCase1 : UseCaseDef1 { /* members */ }</pre>

Element	Graphical Notation	Textual Notation
Include Use Case Compartment	<pre> include actions ^action2 : ActionDef2 (in : ParamDef1, out : ParamDef2) action1 : ActionDef1 [1..*] ordered nonunique action3R : ActionDef3R redefines action3 action4R : ActionDef4R :>> action4 :>> action5 action6S : ActionDef6S [m] subsets action6 action7S : ActionDef7S [m] :> action7 action8R = action8 action11 action11.1 action11.2 ... </pre>	<pre> { include use case useCase1 : UseCase1 [1..*] ordered nonunique; /* ... */ } </pre>
Includes Compartment	<pre> includes ^useCase1 useCase2 useCase3 </pre>	
Use Case Graphical Compartment	<pre> subject system=system1 use case useCase1 actor actor1 : Actor1 actor actor2 : Actor2 actor actor3 : Actor3 include useCase2; include useCase3; use case useCase2 use case useCase3 </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 definition and its rendering can preserve a one-to-one correspondence between elements of the model and of the graphical and/or textual elements of the view artifact. The implementation of a rendering can follow this correspondence to propagate changes to a view artifact back to the model from which the view artifact was extracted and rendered.

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.

7.25.2 Abstract Syntax

For Views and Viewpoints Abstract Syntax class descriptions, see [8.3.25](#).

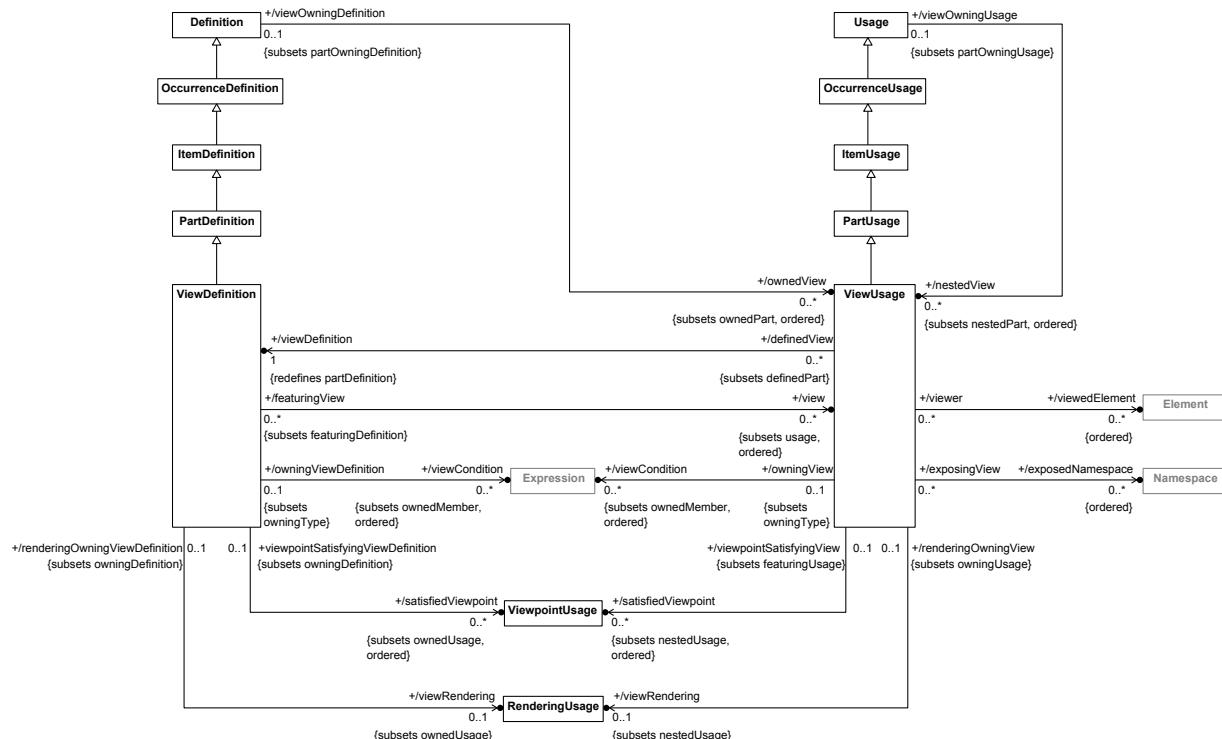


Figure 50. View Definition and Usage

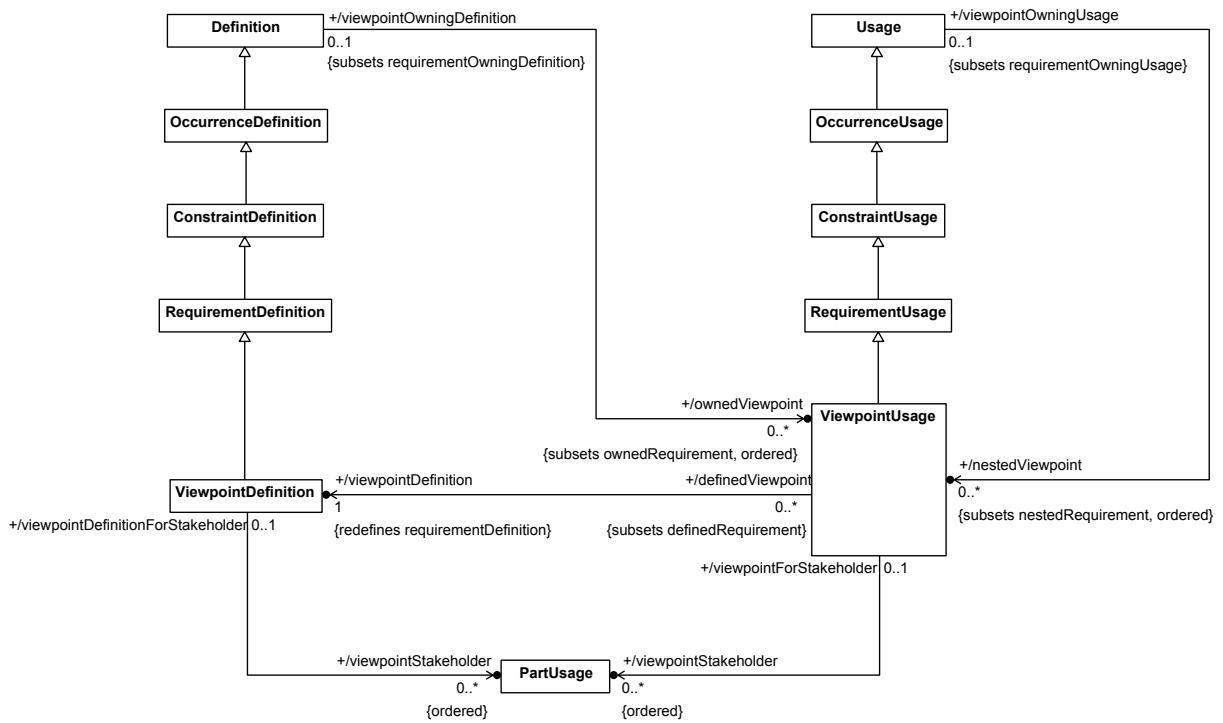


Figure 51. Viewpoint Definition and Usage

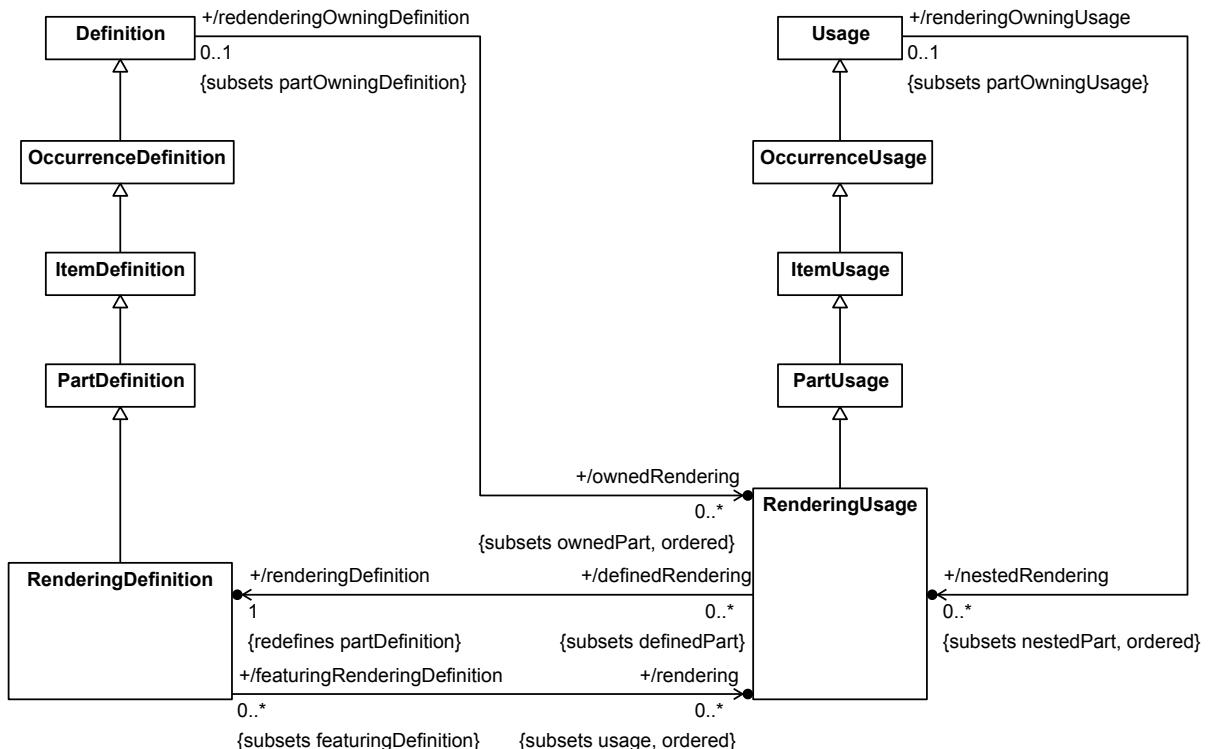


Figure 52. Rendering Definition and Usage

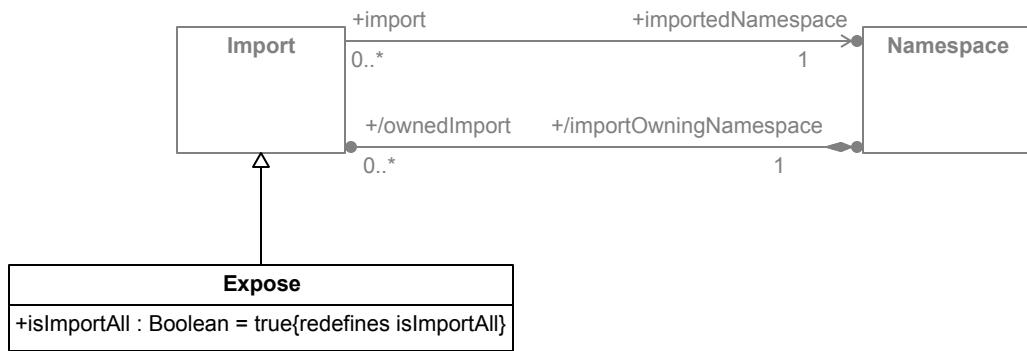


Figure 53. Expose Relationship

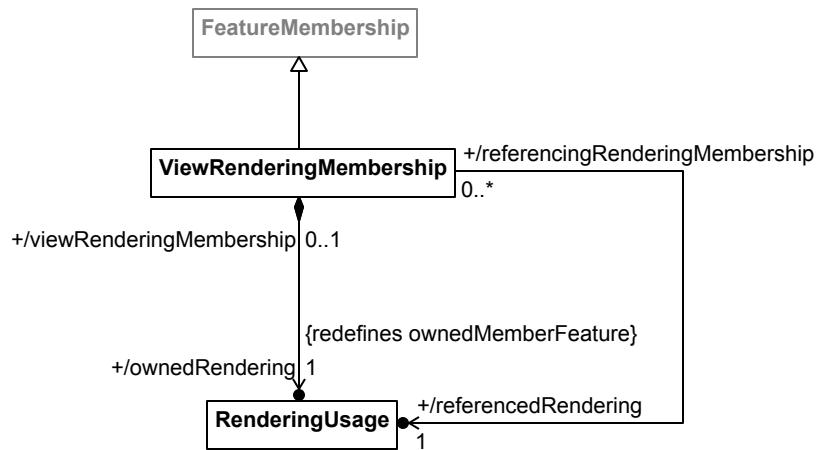


Figure 54. View Rendering Membership

7.25.3 Notation

Textual Notation

For Views and Viewpoints Textual Notation BNF, see [8.2.2.25](#).

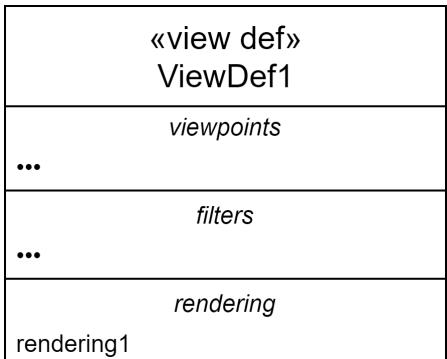
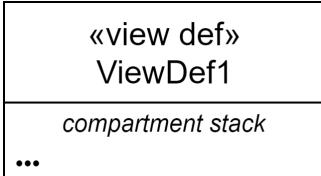
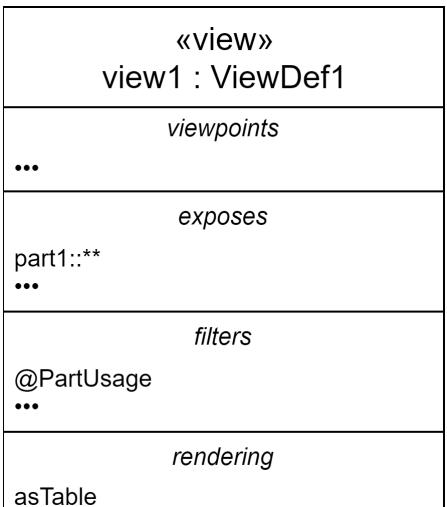
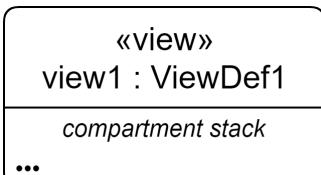
Release Note. Details to be provided.

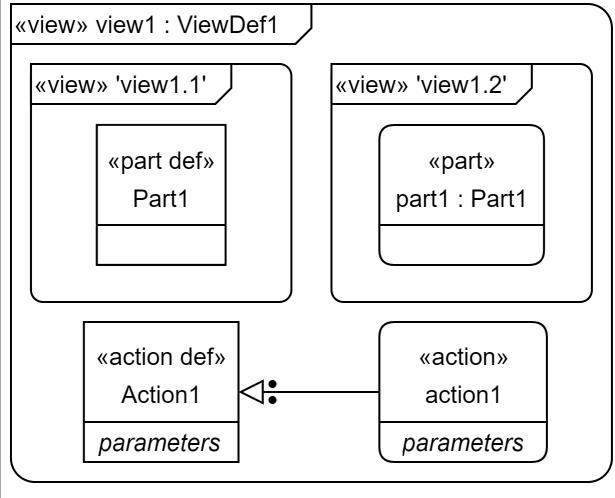
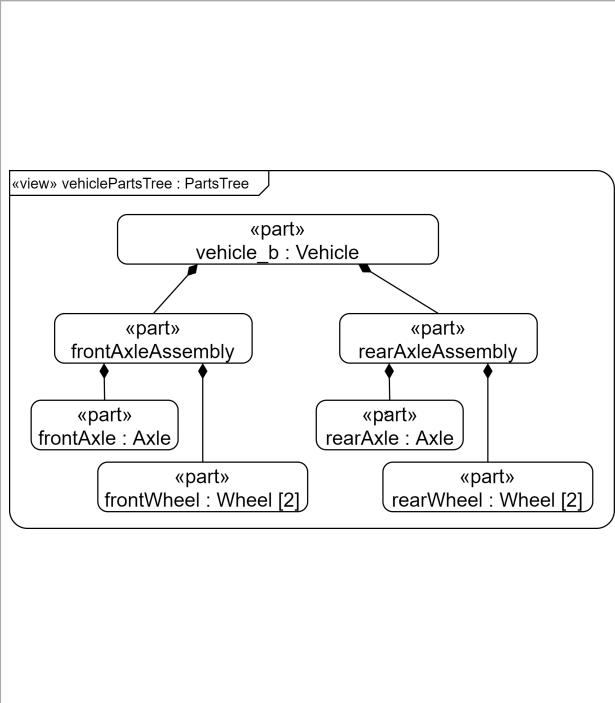
Graphical Notation

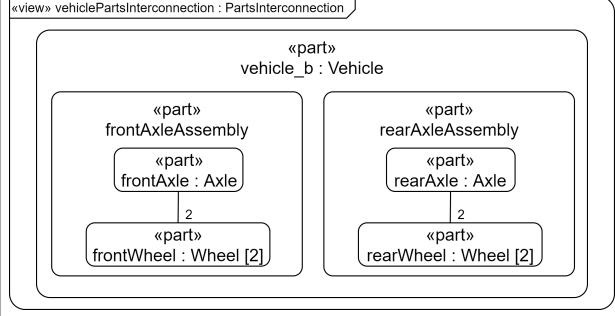
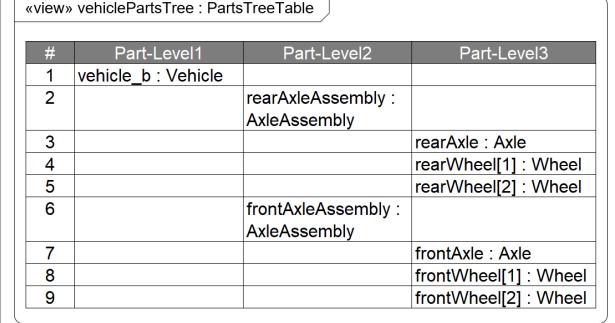
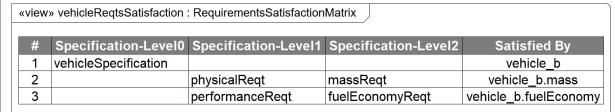
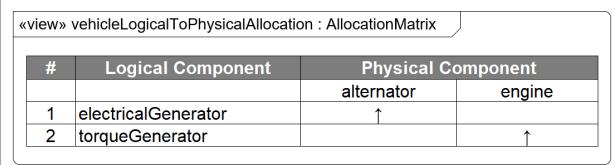
For Views and Viewpoints Graphical Notation BNF, see [8.2.3.25](#).

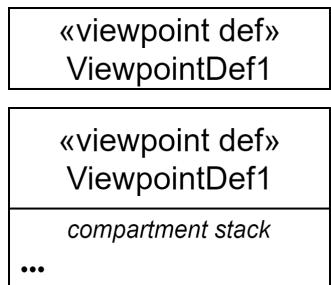
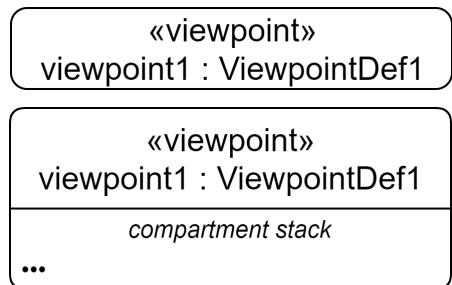
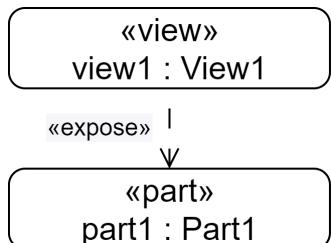
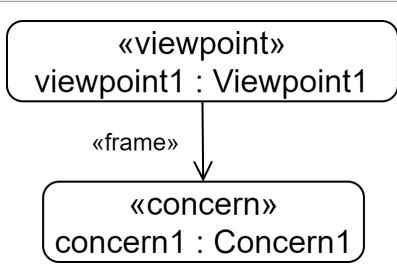
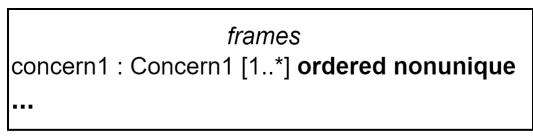
For the kinds of compartments that may appear in the *compartment stack* of a graphical symbol, see the [Representative Compartment Matrix](#) in [9.2.18.1](#).

Table 30. Views and Viewpoints - Representative Notation

Element	Graphical Notation	Textual Notation
View Definition	 	<pre> view def ViewDef1 { satisfy viewpoint1; /* ... */ filter filterExpression1; /* ... */ render rendering1; } view def ViewDef1 { /* members */ } </pre>
View	 	<pre> view view1 : ViewDef1 { satisfy viewpoint1; expose part1::**; filter @PartUsage; render asTable; } view view1 : ViewDef1 { /* members */ } </pre>

Element	Graphical Notation	Textual Notation
General View / Diagram	 <p>«view» view1 : ViewDef1</p> <p>«view» 'view1.1'</p> <p>«part def» Part1</p> <p>«view» 'view1.2'</p> <p>«part» part1 : Part1</p> <p>«action def» Action1</p> <p>«action» action1</p> <p>parameters</p>	
Tree View	 <p>«view» vehiclePartsTree : PartsTree</p> <p>«part» vehicle_b : Vehicle</p> <p>«part» frontAxleAssembly</p> <p>«part» rearAxleAssembly</p> <p>«part» frontAxe : Axle</p> <p>«part» rearAxe : Axle</p> <p>«part» frontWheel : Wheel [2]</p> <p>«part» rearWheel : Wheel [2]</p>	<pre> part vehicle_b : Vehicle { part frontAxleAssembly { part frontAxe : Axe; part frontWheel : Wheel[2]; } part rearAxleAssembly { part rearAxe : Axe; part rearWheel : Wheel[2]; } } view vehiclePartsTree : PartsTree { /*...*/ } </pre>

Element	Graphical Notation	Textual Notation																																								
Nested View	 <pre> <<view>> vehiclePartsInterconnection : PartsInterconnection <<part>> vehicle_b : Vehicle <<part>> frontAxleAssembly <<part>> frontAxle : Axle 2 <<part>> frontWheel : Wheel [2] <<part>> rearAxleAssembly <<part>> rearAxle : Axle 2 <<part>> rearWheel : Wheel [2] </pre>	<pre> part vehicle_b : Vehicle { part frontAxleAssembly { part frontAxle : Axe; part frontWheel : Wheel [2]; } part rearAxleAssembly { part rearAxle : Axe; part rearWheel : Wheel [2]; } } view vehiclePartsInterconnection : PartsInterconnection { /*...*/ } </pre>																																								
Table View	 <table border="1"> <thead> <tr> <th>#</th> <th>Part-Level1</th> <th>Part-Level2</th> <th>Part-Level3</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>vehicle_b : Vehicle</td> <td></td> <td></td> </tr> <tr> <td>2</td> <td></td> <td>rearAxleAssembly : AxeAssembly</td> <td></td> </tr> <tr> <td>3</td> <td></td> <td></td> <td>rearAxle : Axe</td> </tr> <tr> <td>4</td> <td></td> <td></td> <td>rearWheel[1] : Wheel</td> </tr> <tr> <td>5</td> <td></td> <td></td> <td>rearWheel[2] : Wheel</td> </tr> <tr> <td>6</td> <td></td> <td>frontAxleAssembly : AxeAssembly</td> <td></td> </tr> <tr> <td>7</td> <td></td> <td></td> <td>frontAxle : Axe</td> </tr> <tr> <td>8</td> <td></td> <td></td> <td>frontWheel[1] : Wheel</td> </tr> <tr> <td>9</td> <td></td> <td></td> <td>frontWheel[2] : Wheel</td> </tr> </tbody> </table>	#	Part-Level1	Part-Level2	Part-Level3	1	vehicle_b : Vehicle			2		rearAxleAssembly : AxeAssembly		3			rearAxle : Axe	4			rearWheel[1] : Wheel	5			rearWheel[2] : Wheel	6		frontAxleAssembly : AxeAssembly		7			frontAxle : Axe	8			frontWheel[1] : Wheel	9			frontWheel[2] : Wheel	
#	Part-Level1	Part-Level2	Part-Level3																																							
1	vehicle_b : Vehicle																																									
2		rearAxleAssembly : AxeAssembly																																								
3			rearAxle : Axe																																							
4			rearWheel[1] : Wheel																																							
5			rearWheel[2] : Wheel																																							
6		frontAxleAssembly : AxeAssembly																																								
7			frontAxle : Axe																																							
8			frontWheel[1] : Wheel																																							
9			frontWheel[2] : Wheel																																							
Matrix View	 <table border="1"> <thead> <tr> <th>#</th> <th>Specification-Level0</th> <th>Specification-Level1</th> <th>Specification-Level2</th> <th>Satisfied By</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>vehicleSpecification</td> <td></td> <td></td> <td>vehicle_b</td> </tr> <tr> <td>2</td> <td></td> <td>physicalReqt</td> <td>massReqt</td> <td>vehicle_b.mass</td> </tr> <tr> <td>3</td> <td></td> <td>performanceReqt</td> <td>fuelEconomyReqt</td> <td>vehicle_b.fuelEconomy</td> </tr> </tbody> </table>	#	Specification-Level0	Specification-Level1	Specification-Level2	Satisfied By	1	vehicleSpecification			vehicle_b	2		physicalReqt	massReqt	vehicle_b.mass	3		performanceReqt	fuelEconomyReqt	vehicle_b.fuelEconomy																					
#	Specification-Level0	Specification-Level1	Specification-Level2	Satisfied By																																						
1	vehicleSpecification			vehicle_b																																						
2		physicalReqt	massReqt	vehicle_b.mass																																						
3		performanceReqt	fuelEconomyReqt	vehicle_b.fuelEconomy																																						
Matrix View	 <table border="1"> <thead> <tr> <th>#</th> <th>Logical Component</th> <th>Physical Component</th> <th></th> </tr> </thead> <tbody> <tr> <td></td> <td></td> <td>alternator</td> <td>engine</td> </tr> <tr> <td>1</td> <td>electricalGenerator</td> <td>↑</td> <td></td> </tr> <tr> <td>2</td> <td>torqueGenerator</td> <td></td> <td>↑</td> </tr> </tbody> </table>	#	Logical Component	Physical Component				alternator	engine	1	electricalGenerator	↑		2	torqueGenerator		↑																									
#	Logical Component	Physical Component																																								
		alternator	engine																																							
1	electricalGenerator	↑																																								
2	torqueGenerator		↑																																							

Element	Graphical Notation	Textual Notation
Viewpoint Definition	 <pre data-bbox="1065 304 1274 361"> viewpoint def ViewPointDef1; </pre> <pre data-bbox="1065 388 1290 494"> viewpoint def ViewPointDef1 { /* members */ } </pre>	
Viewpoint	 <pre data-bbox="1065 620 1388 677"> viewpoint viewpoint1 : ViewPointDef1; </pre> <pre data-bbox="1065 705 1388 811"> viewpoint viewpoint1 : ViewPointDef1 { /* members */ } </pre>	
Expose	 <pre data-bbox="1065 952 1361 1058"> part part1 : Part1; view view1 : View1 { expose part1; } </pre>	
Frame	 <pre data-bbox="1065 1205 1388 1374"> concern concern1 : Concern1; viewpoint viewpoint1 : Viewpoint1 { frame concern1; } </pre>	
Frames Compartment	 <pre data-bbox="1065 1480 1416 1712"> { frame concern concern1 : Concern1 [1..*] ordered nonunique; /* ... */ } </pre>	

7.26 Metadata

7.26.1 Overview

A *metadata usage* 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. A metadata usage is defined by a single *metadata definition*. If the definition has no nested features itself, then the metadata usage simply acts as a user-defined syntactic tag on the annotated element. If the definition does have features, then the metadata usage must provide value bindings for all of them, specifying metadata for the annotated element.

7.26.2 Abstract Syntax

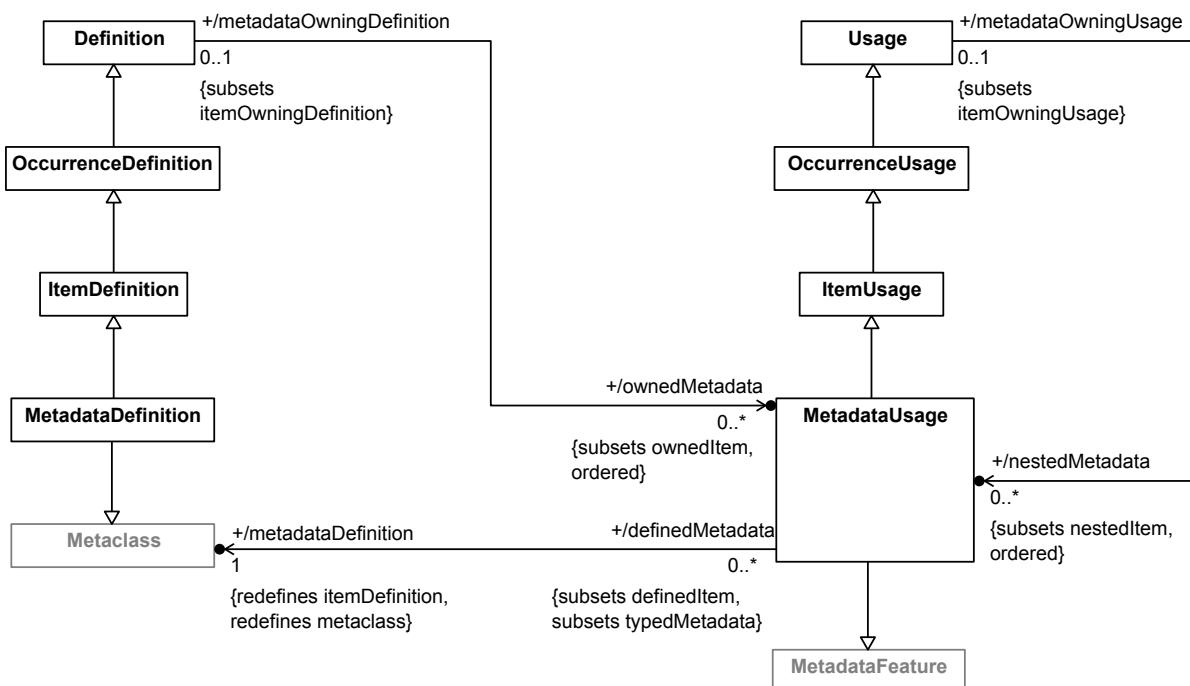


Figure 55. Metadata Definition and Usage

7.26.3 Notation

For Metadata Textual Notation BNF, see [8.2.2.26](#).

Metadata Definition and Usage

A MetadataDefinition is declared like an ItemDefinition (see [7.10](#)), but using the keyword `metadata def`. If no ownedSuperclassing is explicitly given for the MetadataDefinition, then it is implicitly given a default Superclassing to the MetadataDefinition `MetadataItem` from the `Metadata` library model (see [9.2.20](#)).

```

metadata def SecurityRelated;

metadata def ApprovalAnnotation {
    attribute approved : Boolean;
    attribute approver : String;
}
  
```

A MetadataUsage 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 Metaclass (see [KerML]) or SysML MetadataDefinition. 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 MetadataUsage after the keyword **about**, indicating that the MetadataUsage has Annotation Relationships to each of the identified Elements.

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

If the specified Metaclass or MetadataDefinition has `features`, then a body must be given for the MetadataUsage that declares ReferenceUsages (see [7.6](#)) that redefine each of the `features` of the Metaclass or MetadataDefinition and binds them to the result of model-level evaluable Expressions (see [KerML]). These nested ReferenceUsages of a MetadataUsage must always have the same names as the names of the `features` of its `metadataDefinition`, so the shorthand prefix redefines notation (see [7.6](#)) is always used

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

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

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

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

Annotated Elements

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

```
metadata def CommandMetadata {
    // A MetadataUsage of this MetadataDefinition may annotate
    // an ActionDefinition or an ActionUsage.
    :> annotatedElement : SysML::ActionDefinition;
    :> annotatedElement : SysML::ActionUsage;
}

action def Save specializes UserAction {
    @CommandMetadata; // This is valid.
```

```

    redefine action doAction {
        @CommandMetadata; // This is valid.
    }
}
item def Options {
    @CommandMetadata; // This is INVALID.
}

```

Semantic Metadata

If the `metadataDefinition` of a `MetadataUsage` is a direct or indirect specialization of `Metaobjects::SemanticMetadata` (see [KerML]), then the `annotatedElements` must all be Types (e.g., Definitions or Usages) and the Feature `SemanticMetadata::baseType` must be bound to a value of type `KerML::Type`. Then each annotated Type shall implicitly specialize a Type determined from the `baseType` value as follows:

- If the annotated Type is a Definition and the `baseType` is a Definition (or KerML Classifier), then annotated the Definition shall implicitly subclassify the `baseType`.
- If the annotated Type is a Definition and the `baseType` is a Usage (or KerML Feature), then the annotated Definition shall implicitly subclassify each definition (or type) of the `baseType`.
- If the annotated type is a Usage and the `baseType` is a Usage (or KerML Feature), then the annotated Usage shall implicitly subset the `baseType`.
- Otherwise no implicit specialization is added.

When evaluated in a model-level evaluable expression, the cast operator `as` (see [KerML]) may be used to cast a Usage (or KerML Feature) referenced as its first operand to the actual reflective `MetadataDefinition` (or KerML Metaclass) value for this Usage (or Feature), which may then be bound to the `baseType` Feature of `SemanticMetadata`.

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

```

action def UserAction;
action userActions : UserAction[*] nonunique,

metadata def CommandMetadata :> SemanticMetadata {
    // The cast operation "userAction as SysML::Usage" has
    // type Usage, which conforms to the type Type of baseType.
    // Since userActions is an ActionUsage, the expression evaluates
    // at model level to a value of type SysML::ActionUsage.
    baseType = userActions as SysML::Usage;
}

// Save implicitly subclassifies UserAction
// (which is the definition of userActions).
action def Save {
    @CommandMetadata;
}

// previousAction implicitly subsets userActions.
action previousAction[1] {
    @CommandMetadata;
}

```

User-Defined Keywords

A *user-defined keyword* is a (possibly qualified) `MetadataDefinition` (or KerML Metaclass) name or `shortName` preceded by the symbol `#`. Such a keyword can be used in Package, Dependency, Definition and Usage declarations.

The user-defined keyword is placed immediately before the language-defined (reserved) keyword for the declaration and specifies a metadata annotation of the declared element.

```
// It is often convenient to use a lower-case initial name or
// short name for semantic metadata intended to be used as a keyword.
metadata def <command> CommandMetadata :> SemanticMetadata {
    baseType = userActions as SysML::Usage;
}

// Save is an ActionDefinition that implicitly subclasses UserAction.
#command action def Save;

// previousAction is an ActionUsage that implicitly subsets userActions.
#command action previousAction[1];
```

If the named MetadataDefinition is a kind of *SemanticMetadata*, then the implicit specialization rules given above for semantic metadata apply. In addition, a user-defined keyword for semantic metadata may also be used to declare a Definition or Usage without using any language-defined keyword.

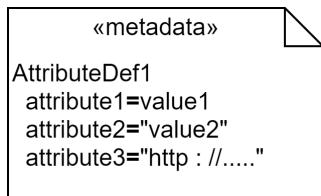
```
// Save is a Definition that implicitly subclasses UserAction.
#command def Save;

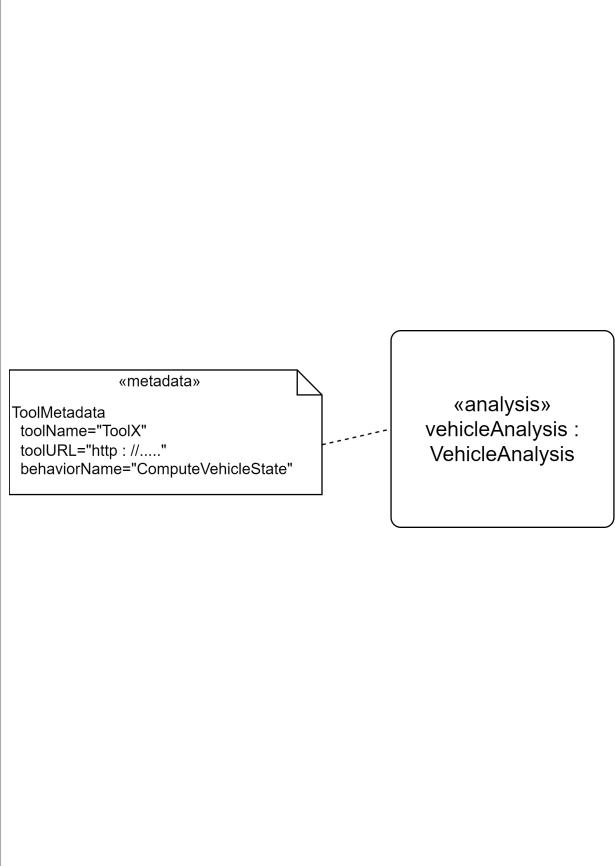
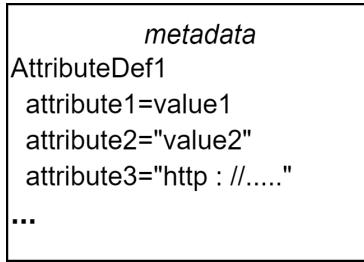
// previousAction is a Usage implicitly subsets userActions.
#command previousAction[1];
```

Graphical Notation

For Metadata Graphical Notation BNF, see [8.2.3.26](#).

Table 31. Metadata - Representative Notation

Element	Graphical Notation	Textual Notation
Metadata	 <pre><<metadata>> AttributeDef1 attribute1=value1 attribute2="value2" attribute3="http://....."</pre>	<pre>metadata MetadataDef1 { attribute1=value1; attribute2="value2"; attribute3="http://....." }</pre> <p>or</p> <pre>@MetadataDef1 { attribute1=value1; attribute2="value2"; attribute3="http://....." }</pre>

Element	Graphical Notation	Textual Notation
Annotation-Metadata		<pre> analysis vehicleAnalysis : VehicleAnalysis; metadata ToolMetadata about vehicleAnalysis { toolName="ToolX"; toolURL="http://....."; behaviorName= "ComputeVehicleState"; } or analysis vehicleAnalysis : VehicleAnalysis { metadata ToolMetadata { toolName="ToolX"; toolURL="http://....."; behaviorName= "ComputeVehicleState"; } } </pre>
Metadata Compartment		<pre> metadata MetadataDef1 { attribute1=value1; attribute2="value2"; attribute3= "http://....."; } </pre>

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 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 32](#)) that includes further notations to describe how the concrete syntax maps to the abstract syntax (see [Table 33](#)).

Productions in the grammar formally result in the synthesis of classes in the abstract syntax and the population of their properties (see [Table 34](#)). 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 32. 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 33. Abstract Syntax Synthesis Notation

Property assignment	p = Element	Assign the result of parsing the concrete syntax Element to abstract syntax property p.
List property construction	p += Element	Add the result of parsing the concrete syntax Element to the abstract syntax list property p.
Boolean property assignment	p ?= Element	If the concrete syntax Element is parsed, then set the abstract Boolean property p to true.
Non-parsing assignment	{ p = value } { p += value }	Assign (or add) the given value to the abstract syntax property p, without parsing any input. The value may be a literal or a reference to another abstract syntax property. The symbol "this" refers to the element being synthesized.
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 34. Grammar Production Definitions

Production definition	NonterminalElement : AbstractSyntaxElement = ...	Define a production for the NonterminalElement that synthesizes the AbstractSyntaxElement. If the NonterminalElement has the same name as the AbstractSyntaxElement, then ": AbstractSyntaxElement" may be omitted.
Parameterized production definition	NonterminalElement (p : Type) : AbstractSyntaxElement = ...	Define a production for the NonterminalElement that synthesizes the AbstractSyntaxElement, with a parameter named p, whose type is an abstract syntax class.

8.2.2.1.2 Lexical Structure

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

1. The reserved keywords of SysML are the following.

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

3. SysML uses the following additional symbol: #

8.2.2.2 Elements and Relationships Textual Notation

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

8.2.2.3 Annotations Textual Notation

8.2.2.3.1 Annotations

```
Annotation =
annotatedElement = [QualifiedName]

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

AnnotatingMember : OwningMembership =
ownedRelatedElement += AnnotatingElement

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

```
RelationshipBody : Relationship =
  ';' | '{' ( ownedRelationship += OwnedAnnotation )* '}'
```

8.2.2.3.2 Comments and Documentation

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

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

8.2.2.3.3 Textual Representation

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

8.2.2.4 Namespaces and Packages Textual Notation

8.2.2.4.1 Packages

```

RootNamespace : Namespace =
    PackageBodyElement*

Package =
( ownedRelationship += PrefixMetadataMember )?
    PackageDeclaration PackageBody

LibraryPackage =
( isStandard ?= 'standard' ) 'library'
( ownedRelationship += PrefixMetadataMember )?
    PackageDeclaration PackageBody

PackageDeclaration : Package =
    'package' Identification

PackageBody : Package =
    ';' | '{' PackageBodyElement* '}''

PackageBodyElement : Package =
    ownedRelationship += PackageMember
    | ownedRelationship += ElementFilterMember
    | ownedRelationship += AliasMember
    | ownedRelationship += Import

MemberPrefix : Membership =
( visibility = VisibilityIndicator )?

PackageMember : OwningMembership
    MemberPrefix
    ( ownedRelatedElement += DefinitionElement
    | ownedRelatedElement = UsageElement )

ElementFilterMember : ElementFilterMembership =
    MemberPrefix
    'filter' ownedRelatedElement += OwnedExpression ';'

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

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

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

ImportedFilterPackage : Import :
    importedNamespace = FilterPackage

```

```
{ ownedRelatedElement += FilterPackage }

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

FilterPackageImport : Import =
    ImportedNamespace

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

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

8.2.2.4.2 Package Elements

```
AnnotatingElement =  
    Comment  
    | PrefixComment  
    | Documentation  
    | TextualRepresentation  
    | MetadataUsage  
  
DefinitionElement : Element =  
    Package  
    | LibraryPackage  
    | AnnotatingElement  
    | Dependency  
    | AttributeDefinition  
    | EnumerationDefinition  
    | OccurrenceDefinition  
    | IndividualDefinition  
    | ItemDefinition  
    | PartDefinition  
    | ConnectionDefinition  
    | FlowConnectionDefinition  
    | InterfaceDefinition  
    | PortDefinition  
    | ActionDefinition  
    | CalculationDefinition  
    | StateDefinition  
    | ConstraintDefinition  
    | RequirementDefinition  
    | ConcernDefinition  
    | StakeholderDefinition  
    | CaseDefinition  
    | AnalysisCaseDefinition  
    | VerificationCaseDefinition  
    | UseCaseDefinition  
    | ViewDefinition  
    | ViewpointDefinition  
    | RenderingDefinition  
    | MetadataDefinition  
    | ExtendedDefinition  
  
UsageElement : Usage =  
    NonOccurrenceUsageElement  
    | OccurrenceUsageElement
```

8.2.2.5 Dependencies Textual Notation

```
Dependency =
( ownedRelationship += PrefixMetadataAnnotation )?
'dependency' DependencyDeclaration
RelationshipBody

DependencyDeclaration =
( Identification '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

```
BasicDefinitionPrefix =
    isAbstract ?= 'abstract' | isVariation ?= 'variation'

DefinitionExtensionKeyword : Definition =
    ownedRelationship += PrefixMetadataMember

DefinitionPrefix : Definition =
    BasicDefinitionPrefix? DefinitionExtensionKeyword?

Definition =
    DefinitionDeclaration DefinitionBody

DefinitionDeclaration : Definition
    Identification SubclassificationPart?

DefinitionBody : Type =
    ';' | '{' DefinitionBodyItem* '}''

DefinitionBodyItem : Type =
    ownedRelationship += DefinitionMember
    | ownedRelationship += VariantUsageMember
    | ownedRelationship += NonOccurrenceUsageMember
    | ( ownedRelationship += SourceSuccessionMember )?
        ownedRelationship += OccurrenceUsageMember
    | ownedRelationship += AliasMember
    | ownedRelationship += Import

DefinitionMember : OwningMembership =
    MemberPrefix
    ownedRelatedElement += DefinitionElement

VariantUsageMember : VariantMembership =
    MemberPrefix 'variant'
    ownedVariantUsage = VariantUsageElement

NonOccurrenceUsageMember : FeatureMembership =
    MemberPrefix
    ownedRelatedElement += NonOccurrenceUsageElement

OccurrenceUsageMember : FeatureMembership =
    MemberPrefix
    ownedRelatedElement += OccurrenceUsageElement

StructureUsageMember : FeatureMembership =
    MemberPrefix
    ownedRelatedElement += StructureUsageElement

BehaviorUsageMember : FeatureMembership =
    MemberPrefix
    ownedRelatedElement += BehaviorUsageElement
```

8.2.2.6.2 Usages

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

RefPrefix : Usage =
    ( direction = FeatureDirection )?
    ( isAbstract ?= 'abstract' | isVariation ?= 'variation')?
    ( isReadOnly ?= 'readonly' )?
    ( isDerived ?= 'derived' )?
    ( isEnd ?= 'end' )?

BasicUsagePrefix : Usage =
    RefPrefix
    ( isReference ?= 'ref' )?

UsageExtensionKeyword : Usage =
    ownedRelationship += PrefixMetadataMember

UsagePrefix : Usage =
    BasicUsagePrefix UsageExtensionKeyword?

Usage =
    UsageDeclaration UsageCompletion

UsageDeclaration : Usage =
    Identification FeatureSpecializationPart?

UsageCompletion : Usage =
    ValuePart? UsageBody

UsageBody : Usage =
    DefinitionBody

ValuePart : Feature =
    ownedRelationship += FeatureValue

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

8.2.2.6.3 Reference Usages

```
DefaultReferenceUsage : ReferenceUsage =
    RefPrefix Usage

ReferenceUsage =
    RefPrefix 'ref' Usage

VariantReference : ReferenceUsage =
    ownedRelationship += OwnedReferenceSubsetting
    FeatureSpecialization* UsageBody
```

8.2.2.6.4 Body Elements

```

NonOccurrenceUsageElement : Usage =
    DefaultReferenceUsage
| ReferenceUsage
| AttributeUsage
| EnumerationUsage
| BindingConnectorAsUsage
| SuccessionAsUsage
| ExtendedUsage

OccurrenceUsageElement : Usage =
    StructureUsageElement | BehaviorUsageElement

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

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

VariantUsageElement : Usage =
    VariantReference
| ReferenceUsage
| AttributeUsage
| BindingConnector
| Succession
| OccurrenceUsage
| IndividualUsage
| PortionUsage
| EventOccurrenceUsage

```

```
| ItemUsage  
| PartUsage  
| ViewUsage  
| RenderingUsage  
| PortUsage  
| ConnectionUsage  
| InterfaceUsage  
| AllocationUsage  
| Message  
| FlowConnectionUsage  
| SuccessionFlowConnectionUsage  
| BehaviorUsageElement
```

8.2.2.6.5 Specialization

```

SubclassificationPart : Classifier =
    SPECIALIZES ownedRelationship += OwnedSubclassification
    ( ',' ownedRelationship += OwnedSubclassification )*

OwnedSubclassification : Subclassification =
    superClassifier = [QualifiedName]

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

FeatureSpecialization : Feature =
    Typings | Subsettings | References | Redefinitions

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

TypedBy : Feature =
    DEFINED_BY ownedRelationship += FeatureTyping

FeatureTyping =
    OwnedFeatureTyping | ConjugatePortTyping

OwnedFeatureTyping : FeatureTyping =
    type = [QualifiedName]
    | type = OwnedFeatureChain
    { ownedRelatedElement += type }

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

Subsets : Feature =
    SUBSETS ownedRelationship += OwnedSubsetting

OwnedSubsetting : Subsetting =
    subsettetedFeature = [QualifiedName]
    | subsettetedFeature = OwnedFeatureChain
    { ownedRelatedElement += subsettetedFeature }

References : Feature =
    REFERENCES ownedRelationship += OwnedReferenceSubsetting

OwnedReferenceSubsetting : ReferenceSubsetting =
    referencedFeature = [QualifiedName]
    | referencedFeature = OwnedFeatureChain
    { ownedRelatedElement += referenceFeature }

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

Redefines : Feature =
    REDEFINES ownedRelationship += OwnedRedefinition

OwnedRedefinition : Redefinition =
    redefinedFeature = [QualifiedName]
    | redefinedFeature = OwnedFeatureChain
    { ownedRelatedElement += redefinedFeature }

```

```

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

```

8.2.2.6.6 Multiplicity

```

MultiplicityPart : Feature =
    ownedRelationship += OwnedMultiplicity
    | ( ownedRelationship += OwnedMultiplicity )?
        ( isOrdered ?= 'ordered' ( { isUnique = false } 'nonunique' )?
            | { isUnique = false } 'nonunique' ( isOrdered ?= 'ordered' )? )
    
OwnedMultiplicity : OwningMembership =
    ownedRelatedElement += MultiplicityRange
    
MultiplicityRange =
    '[' ( ownedRelationship += MultiplicityExpressionMember '..')?
        ownedRelationship += MultiplicityExpressionMember ']'
    
MultiplicityExpressionMember : OwningMembership =
    ownedRelatedElement += ( LiteralExpression | FeatureReferenceExpression )

```

8.2.2.7 Attributes Textual Notation

```

AttributeDefinition : AttributeDefinition =
    DefinitionPrefix 'attribute' 'def' Definition
    
AttributeUsage : AttributeUsage =
    UsagePrefix 'attribute' Usage

```

8.2.2.8 Enumerations Textual Notation

```
EnumerationDefinition =
    'enum' 'def' DefinitionDeclaration EnumerationBody

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

AnnotatingMember : OwningMembership =
    ownedMemberElement = AnnotatingElement

EnumerationUsageMember : VariantMembership =
    MemberPrefix ownedRelatedElement += EnumeratedValue

EnumeratedValue : EnumerationUsage =
    'enum'? Usage

EnumerationUsage : EnumerationUsage =
    UsagePrefix 'enum' Usage
```

8.2.2.9 Occurrences Textual Notation

8.2.2.9.1 Occurrence Definitions

```
OccurrenceDefinitionPrefix : OccurrenceDefinition =
    BasicDefinitionPrefix?
    ( isIndividual ?= 'individual' )?
    DefinitionExtensionKeyword?

OccurrenceDefinition =
    OccurrenceDefinitionPrefix 'occurrence' 'def' Definition

IndividualDefinition : OccurrenceDefinition =
    BasicDefinitionPrefix? isIndividual ?= 'individual'
    DefinitionExtensionKeyword? 'def' Definition
```

8.2.2.9.2 Occurrence Usages

```
OccurrenceUsagePrefix : OccurrenceUsage =
    BasicUsagePrefix
    ( isIndividual ?= 'individual' )?
    ( portionKind = PortionKind )?
    UsageExtensionKeyword?

OccurrenceUsage =
    OccurrenceUsagePrefix 'occurrence' Usage

IndividualUsage : OccurrenceUsage =
    BasicUsagePrefix isIndividual ?= 'individual'
    UsageExtensionKeyword Usage

PortionUsage : OccurrenceUsage =
    BasicUsagePrefix ( isIndividual ?= 'individual' )?
    portionKind = PortionKind
    UsageExtensionKeyword Usage

PortionKind =
    'snapshot' | 'timeslice'

EventOccurrenceUsage =
    OccurrenceUsagePrefix 'event'
    ( ownedRelationship += OwnedReferenceSubsetting
        FeatureSpecializationPart?
    | 'occurrence' UsageDeclaration? )
    UsageCompletion
```

8.2.2.9.3 Occurrence Successions

```
SourceSuccessionMember : FeatureMembership =
    'then' ownedRelatedElement + SourceSuccession

SourceSuccession : SuccessionAsUsage =
    ownedRelationship += SourceEndMember

SourceEndMember : EndFeatureMembership =
    ownedRelatedElement + SourceEnd

SourceEnd : Feature =
    ( ownedRelationship += OwnedMultiplicity )?
```

8.2.2.10 Items Textual Notation

```
ItemDefinition =
    OccurrenceDefinitionPrefix
    'item' 'def' Definition

ItemUsage =
    OccurrenceUsagePrefix 'item' Usage
```

8.2.2.11 Parts Textual Notation

```
PartDefinition =
    OccurrenceDefinitionPrefix 'part' 'def' Definition

PartUsage =
    OccurrenceUsagePrefix 'part' Usage
```

8.2.2.12 Ports Textual Notation

```
PortDefinition =
    DefinitionPrefix 'port' 'def' Definition

PortUsage =
    OccurrenceUsagePrefix 'port' Usage

ConjugatedPortTyping : ConjugatedPortTyping =
    '~' originalPortDefinition = ~[QualifiedName]
(see Note 1)
```

Notes.

1. The notation `~[QualifiedName]` indicates that a `QualifiedName` shall be parsed from the input text, but that it shall be resolved as if it was the qualified name constructed as follows
 - Extract the last segment name of the given `QualifiedName` and prepend the symbol `~` to it.
 - Append the name so constructed to the end of the entire original `QualifiedName`.

For example, if the `ConjugatedPortTyping` is `~A::B::C`, then the given `QualifiedName` is `A::B::C`, and `~[QualifiedName]` is resolved as `A::B::C::'~C'`.

Alternatively, a conforming tool may first resolve the given `QualifiedName` as usual to a `PortDefinition` and then use the `conjugatedPortDefinition` of this `PortDefinition` as the resolution of `~[QualifiedName]`.

8.2.2.13 Connections Textual Notation

8.2.2.13.1 Connection Definition and Usage

```
ConnectionDefinition =
    OccurrenceDefinitionPrefix 'connection' 'def' Definition

ConnectionUsage =
    OccurrenceUsagePrefix
    ( 'connection' UsageDeclaration
        ( 'connect' ConnectorPart )?
        | 'connect' ConnectorPart )
    UsageBody

ConnectorPart : ConnectionUsage =
    BinaryConnectorPart | NaryConnectorPart

BinaryConnectorPart : ConnectionUsage =
    ownedRelationship += ConnectorEndMember 'to'
    ownedRelationship += ConnectorEndMember

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

ConnectorEndMember : EndFeatureMembership :
    ownedRelatedElement += ConnectorEnd

ConnectorEnd : Feature =
    ( name = NAME REFERENCES )?
    ownedRelationship += OwnedReferenceSubsetting
    ( ownedRelationship += OwnedMultiplicity )?
```

8.2.2.13.2 Binding Connectors

```
BindingConnectorAsUsage =
    UsagePrefix ( 'binding' UsageDeclaration )?
    'bind' ownedRelationship += ConnectorEndMember
    '=' ownedRelationship += ConnectorEndMember
    UsageBody
```

8.2.2.13.3 Successions

```
SuccessionAsUsage =
    UsagePrefix ( 'succession' UsageDeclaration )?
    'first' s.ownedRelationship += ConnectorEndMember
    'then' s.ownedRelationship += ConnectorEndMember
    UsageBody
```

8.2.2.13.4 Messages and Flow Connections

```

FlowConnectionDefinition :
    OccurrenceDefinitionPrefix 'flow' 'def' Definition

Message : FlowConnectionUsage =
    OccurrenceUsagePrefix 'message'
    MessageDeclaration DefinitionBody
    { isAbstract = true }

MessageDeclaration : FlowConnectionUsage =
    UsageDeclaration
    ( 'of' ownedRelationship += PayloadFeatureMember )?
    ( ValuePart
    | 'from' ownedRelationship += MessageEventMember
    'to' ownedRelationship += MessageEventMember
    )?

MessageEventMember : ParameterMembership =
    ownedRelatedElement += MessageEvent

MessageEnd : EventOccurrenceUsage =
    ownedRelationship += OwnedReferenceSubsetting

FlowConnectionUsage =
    OccurrenceUsagePrefix 'flow'
    FlowConnectionDeclaration DefinitionBody

SuccessionFlowConnectionUsage =
    OccurrenceUsagePrefix 'succession' 'flow'
    FlowConnectionDeclaration DefinitionBody

FlowConnectionDeclaration : FlowConnectionUsage =
    ( UsageDeclaration
    ( 'of' ownedRelationship += PayloadFeatureMember )?
    'from'
    )?
    ownedRelationship += FlowEndMember 'to'
    ownedRelationship += FlowEndMember

PayloadFeatureMember : FeatureMembership =
    ownedRelatedElement += PayloadFeature

PayloadFeature : Feature =
    ( name = NAME DEFINED_BY )?
    ( ownedRelationship += OwnedFeatureTyping
    ( ownedRelationship += OwnedMultiplicity )?
    | ownedRelationship += OwnedMultiplicity
    ( ownedRelationship += OwnedFeatureTyping )?
    )

FlowEndMember : EndFeatureMembership =
    ownedRelatedElement += FlowEnd

FlowEnd : Feature =
    ( ownedRelationship += FlowEndSubsetting )?
    ownedRelationship += FlowFeatureMember

FlowEndSubsetting : ReferenceSubsetting =

```

```

referencedFeature = [QualifiedName]
| referencedFeature = FeatureChainPrefix
{ ownedRelatedElement += referencedFeature }

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

FlowFeatureMember : FeatureMembership =
ownedRelatedElement += FlowFeature

FlowFeature : EVENT =
ownedRelationship += FlowFeatureRedefinition

FlowFeatureRefefinition : Redefinition =
redefinedFeature = [QualifiedName]

```

8.2.2.14 Interfaces Textual Notation

8.2.2.14.1 Interface Definitions

```
InterfaceDefinition =
    OccurrenceDefinitionPrefix 'interface' 'def'
    DefinitionDeclaration InterfaceBody

InterfaceBody : Type =
    ';' | '{' InterfaceBodyItem* '}''

InterfaceBodyItem : Type =
    ownedRelationship += DefinitionMember
    | ownedRelationship += VariantUsageMember
    | ownedRelationship += InterfaceNonOccurrenceUsageMember
    | ( ownedRelationship += SourceSuccessionMember )?
        ownedRelationship += InterfaceOccurrenceUsageMember
    | ownedRelationship += AliasMember
    | ownedRelationship += Import

InterfaceNonOccurrenceMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += InterfaceNonOccurrenceUsageElement

InterfaceNonOccurrenceUsageElement : Usage =
    ReferenceUsage
    | AttributeUsage
    | EnumerationUsage
    | BindingConnector
    | Succession

InterfaceOccurrenceUsageMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += InterfaceOccurrenceUsageElement

InterfaceOccurrenceUsageElement : Feature =
    DefaultInterfaceEnd | StructureUsageElement | BehaviorUsageElement

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

8.2.2.14.2 Interface Usages

```
InterfaceUsage =
    OccurrenceUsagePrefix 'interface'
    InterfaceUsageDeclaration InterfaceBody

InterfaceUsageDeclaration : InterfaceUsage =
    UsageDeclaration ( 'connect' InterfacePart )?
    | InterfacePart

InterfacePart : InterfaceUsage =
    BinaryInterfacePart | NaryInterfacePart

BinaryInterfacePart : InterfaceUsage =
    ownedRelationship += InterfaceEndMember 'to'
    ownedRelationship += InterfaceEndMember

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

InterfaceEndMember : EndFeatureMembership =
    ownedRelatedElement += InterfaceEnd

InterfaceEnd : PortUsage :
    ( name = Name REFERENCES )?
    ownedRelationship += OwnedReferenceSubsetting
    ( ownedRelationship += OwnedMultiplicity )?
```

8.2.2.15 Allocations Textual Notation

```
AllocationDefinition =
    OccurrenceDefinitionPrefix 'allocation' 'def' Definition

AllocationUsage =
    OccurrenceUsagePrefix
    AllocationUsageDeclaration UsageBody

AllocationUsageDeclaration : AllocationUsage =
    'allocation' UsageDeclaration
    ( 'allocate' ConnectorPart )?
    | 'allocate' ConnectorPart
```

8.2.2.16 Actions Textual Notation

8.2.2.16.1 Action Definitions

```
ActionDefinition =
    OccurrenceDefinitionPrefix 'action' 'def'
    DefinitionDeclaration ActionBody

ActionBody : Type =
    ';' | '{' ActionBodyItem* '}''

ActionBodyItem : Type =
    NonBehaviorBodyItem
    | ownedRelationship += InitialNodeMember
        ( ownedRelationship += ActionTargetSuccessionMember )*
    | ( ownedRelationship += SourceSuccessionMember )?
        ownedRelationship += ActionBehaviorMember
        ( ownedRelationship += ActionTargetSuccessionMember )*
    | ownedRelationship += GuardedSuccessionMember

NonBehaviorBodyItem =
    ownedRelationship += Import
    | ownedRelationship += AliasMember
    | ownedRelationship += DefinitionMember
    | ownedRelationship += VariantUsageMember
    | ownedRelationship += NonOccurrenceUsageMember
    | ( ownedRelationship += SourceSuccessionMember )?
        ownedRelationship += StructureUsageMember

ActionBehaviorMember : FeatureMembership =
    BehaviorUsageMember | ActionNodeMember

InitialNodeMember : FeatureMembership =
    MemberPrefix 'first' memberFeature = [QualifiedName]
    RelationshipBody

ActionNodeMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += ActionNode

ActionTargetSuccessionMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += ActionTargetSuccession

GuardedSuccessionMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += GuardedSuccession
```

8.2.2.16.2 Action Usages

```
ActionUsage =
  OccurrenceUsagePrefix 'action'
  ActionUsageDeclaration ActionBody

ActionUsageDeclaration : ActionUsage =
  UsageDeclaration ValuePart?

PerformActionUsage =
  OccurrenceUsagePrefix 'perform'
  PerformActionUsageDeclaration ActionBody

PerformActionUsageDeclaration : PerformActionUsage =
  ( ownedRelationship += OwnedReferenceSubsetting
    FeatureSpecializationPart?
  | 'action' UsageDeclaration )
  ValuePart?
```

8.2.2.16.3 Action Nodes

```

ActionNode : ActionUsage =
    SendNode | AcceptNode | AssignmentNode
    | IfNode | WhileLoopNode | ForLoopNode
    | ControlNode

ActionNodeUsageDeclaration : ActionUsage =
    'action' UsageDeclaration?

AcceptNode : AcceptActionUsage =
    OccurrenceUsagePrefix
    AcceptNodeDeclaration ActionBody

AcceptNodeDeclaration : AcceptActionUsage =
    ActionNodeUsageDeclaration?
    'accept' AcceptParameterPart

AcceptParameterPart : AcceptActionUsage =
    ownedRelationship += PayloadParameterMember
    ( 'via' ownedRelationship += NodeParameterMember
    | ownedRelationship += EmptyParameterMember )

PayloadParameterMember : ParameterMembership =
    ownedRelatedElement += PayloadParameter

PayloadParameter : ReferenceUsage =
    Identification
    ( PayloadParameterSpecializationPart ValuePart?
    | PayloadParameterSpecializationPart? TriggerValuePart )
    | ownedRelationship += OwnedFeatureTyping
    ( ownedRelationship += OwnedMultiplicity )?
    | ownedRelationship += OwnedMultiplicity
    ownedRelationship += OwnedFeatureTyping

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

TriggerValuePart : Feature =
    ownedRelationship += TriggerFeatureValue

TriggerFeatureValue : FeatureValue =
    ownedRelatedElement += TriggerExpression

TriggerExpression : TriggerInvocationExpression =
    kind = ( 'at' | 'after' )
    ownedRelationship += OwnedExpressionMember
    | kind = "when"
    ownedRelationship += ChangeExpressionMember

ChangeExpressionMember : FeatureMembership =
    ownedRelatedElement += ChangeExpression

ChangeExpression : Expression =
    ownedRelationship += ChangeExpressionMember

ChangeResultExpressionMember : ResultExpressionMember =
    ownedRelatedElement += OwnedExpression

```

```

SendNode : SendActionUsage =
    OccurrenceUsagePrefix
    SendNodeDeclaration ActionBody

SendNodeDeclaration : SendActionUsage =
    ActionNodeUsageDeclaration?
    'send' ownedRelationship += NodeParameterMember
    ( 'via' ownedRelationship += NodeParameterMember
    | ownedRelationship += EmptyParameterMember )
    ( 'to' ownedRelationship += NodeParameterMember
    | ownedRelationship += EmptyParameterMember )

NodeParameterMember : ParameterMembership =
    ownedRelatedElement += NodeParameter

NodeParameter : ReferenceUsage =
    ownedRelationship += FeatureBinding

FeatureBinding : FeatureValue =
    ownedRelatedElement += ownedExpression

AssignmentNode : AssignmentActionUsage =
    OccurrenceUsagePrefix
    AssignmentNodeDeclaration ActionBody

AssignmentNodeDeclaration: ActionUsage =
    ( ActionNodeUsageDeclaration )? 'assign'
    ownedRelationship += AssignmentTargetMember
    ownedRelationship += FeatureChainMember ':='
    ownedRelationship += NodeParameterMember

AssignmentTargetMember : ParameterMembership =
    ownedRelatedElement += AssignmentTargetParameter

AssignmentTargetParameter : ReferenceUsage =
    ( ownedRelationship += AssignmentTargetBinding '.' )?

AssignmentTargetBinding : FeatureValue =
    ownedRelatedElement += NonFeatureChainPrimaryExpression

FeatureChainMember : Membership =
    memberElement = [QualifiedName]
    | OwnedFeatureChainMember

OwnedFeatureChainMember : OwnedMembership =
    ownedRelatedElement += FeatureChain

ActionNodePrefix : ActionUsage =
    OccurrenceUsagePrefix ActionNodeUsageDeclaration?

ExpressionParameterMember : ParameterMembership =
    ownedRelatedElement += OwnedExpression

IfNode : IfActionUsage =
    ActionNodePrefix
    'if' ownedRelationship += ExpressionParameterMember

```

```

ownedRelationship += ActionBodyParameterMember
( 'else' ownedRelationship +=
( ActionBodyParameterMember | IfNodeParameterMember ) )?

ActionBodyParameterMember : ParameterMembership =
ownedRelatedElement += ActionBodyParameter

ActionBodyParameter : ActionUsage =
( 'action' UsageDeclaration? )?
'{` ActionBodyItem* '}`

IfNodeParameterMember : ParameterMembership =
ownedRelatedElement += IfNode

WhileLoopNode : WhileLoopActionUsage =
ActionNodePrefix
( 'while' ownedRelationship += ExpressionParameterMember
| 'loop' ownedRelationship += EmptyParameterMember
)
ownedRelationship += ActionBodyParameterMember
( 'until' ownedRelationship += ExpressionParameterMember ';' )?

ForLoopNode : ForLoopActionUsage =
ActionNodePrefix
'for' ownedRelationship += ForVariableDeclarationMember
'in' ownedRelationship += NodeParameterMember
ownedRelationship += ActionBodyParameterMember

ForVariableDeclarationMember : FeatureMembership =
ownedRelatedElement += UsageDeclaration

ForVariableDeclaration : ReferenceUsage =
UsageDeclaration

ControlNode : ControlNode =
MergeNode | DecisionNode | JoinNode| ForkNode

ControlNodePrefix : OccurrenceUsage =
RefPrefix
( isIndividual ?= 'individual' )?
( portionKind = PortionKind )?

MergeNode =
ControlNodePrefix
isComposite ?= 'merge' UsageDeclaration
ActionNodeBody

DecisionNode =
ControlNodePrefix
isComposite ?= 'decide' UsageDeclaration
ActionNodeBody

JoinNode =
ControlNodePrefix
isComposite ?= 'join' UsageDeclaration
ActionNodeBody

```

```

ForkNode : ForkNode =
  ControlNodePrefix
  isComposite ?= 'fork' UsageDeclaration
  ActionNodeBody

ActionNodeBody : ControlNode =
  ';' | '{' ( ownedRelationship += AnnotatingMember )* '}'

EmptyParameterMember : ParameterMembership =
  ownedRelatedElement += EmptyUsage

EmptyUsage : ReferenceUsage =
  {}

```

8.2.2.16.4 Action Successions

```

ActionTargetSuccession : Feature =
  ( TargetSuccession | GuardedTargetSuccession | DefaultTargetSuccession )
  UsageBody

TargetSuccession : SuccessionAsUsage =
  ownedRelationship += SourceEndMember
  ownedRelationship += ConnectorEndMember

GuardedTargetSuccession : TransitionUsage =
  ownedRelationship += GuardExpressionMember
  'then' ownedRelationship += TransitionSuccessionMember

DefaultTargetSuccession : TransitionUsage =
  'else' ownedRelationship += TransitionSuccessionMember

GuardedSuccession : TransitionUsage =
  ( 'succession' UsageDeclaration )?
  'first' ownedRelationship += FeatureChainMember
  ownedRelationship += GuardExpressionMember
  'then' ownedRelationship += TransitionSuccessionMember
  UsageBody

```

8.2.2.17 States Textual Notation

8.2.2.17.1 State Definitions

```

StateDefinition =
    OccurrenceDefinitionPrefix 'state' 'def'
    DefinitionDeclaration StateDefBody

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

StateBodyItem : Type =
    NonBehaviorBodyItem
    | ( ownedRelationships += SourceSuccessionMember )?
    ownedRelationship += BehaviorUsageMember
    ( ownedRelationship += TargetTransitionUsageMember )*
    | ownedRelationship += TransitionUsageMember
    | ownedRelationship += EntryActionMember
    ( ownedRelationship += EntryTransitionMember )*
    | ownedRelationship += DoActionMember
    | ownedRelationship += ExitActionMember

EntryActionMember : StateSubactionMembership =
    MemberPrefix kind = 'entry'
    ownedRelatedElement += StateActionUsage

DoActionMember : StateSubactionMembership =
    MemberPrefix kind = 'do'
    ownedRelatedElement += StateActionUsage

ExitActionMember : StateSubactionMembership =
    MemberPrefix kind = 'exit'
    ownedRelatedElement += StateActionUsage

EntryTransitionMember : FeatureMembership :
    MemberPrefix
    ( ownedRelatedElement += GuardedTargetSuccession
    | 'then' ownedRelatedElement += TargetSuccession
    ) ';''

StateActionUsage : ActionUsage =
    EmptyActionUsage ';'
    | StatePerformActionUsage
    | StateAcceptActionUsage
    | StateSendActionUsage
    | StateAssignmentActionUsage

EmptyActionUsage : ActionUsage =
    {}

StatePerformActionUsage : PerformActionUsage =
    PerformActionUsageDeclaration ActionBody

StateAcceptActionUsage : AcceptActionUsage =
    AcceptNodeDeclaration ActionBody

StateSendActionUsage : SendActionUsage
    SendNodeDeclaration ActionBody

```

```

StateAssignmentActionUsage : AssignmentActionUsage =
    AssignmentNodeDeclaration ActionBody

TransitionUsageMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += TransitionUsage

TargetTransitionUsageMember : FeatureMembership =
    MemberPrefix ownedRelatedElement += TargetTransitionUsage

```

8.2.2.17.2 State Usages

```

StateUsage =
    OccurrenceUsagePrefix 'state'
    ActionUsageDeclaration StateUsageBody

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

ExhibitStateUsage =
    OccurrenceUsagePrefix 'exhibit'
    ( ownedRelationship += OwnedReferenceSubsetting
        FeatureSpecializationPart?
    | 'state' UsageDeclaration )
    ValuePart? StateUsageBody

```

8.2.2.17.3 Transition Usages

```

TransitionUsage =
  'transition' ( UsageDeclaration 'first' )?
  ownedRelationship += FeatureChainMember
  ownedRelationship += EmptyParameterMember
  ( ownedRelationship += EmptyParameterMember
    ownedRelationship += TriggerActionMember )?
  ( ownedRelationship += GuardExpressionMember )?
  ( ownedRelationship += EffectBehaviorMember )?
  'then' ownedRelationship += TransitionSuccessionMember
  ActionBody

TargetTransitionUsage : 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 += TransitionSuccessionMember
  ActionBody

TriggerActionMember : TransitionFeatureMembership =
  'accept' { kind = 'trigger' } ownedRelatedElement += TriggerAction

TriggerAction : AcceptActionUsage =
  AcceptParameterPart

GuardExpressionMember : TransitionFeatureMembership =
  'if' { kind = 'guard' } ownedRelatedElement += OwnedExpression

EffectBehaviorMember : TransitionFeatureMembership =
  'do' { kind = 'effect' } ownedRelatedElement += EffectBehaviorUsage

EffectBehaviorUsage : ActionUsage =
  EmptyActionUsage
  | TransitionPerformActionUsage
  | TransitionAcceptActionUsage
  | TransitionSendActionUsage
  | TransitionAssignmentActionUsage

TransitionPerformActionUsage : PerformActionUsage =
  PerformActionDeclaration ( '{' ActionBodyItem* '}' )?

TransitionAcceptActionUsage : AcceptActionUsage =
  AcceptNodeDeclaration ( '{' ActionBodyItem* '}' )?

TransitionSendActionUsage : SendActionUsage =
  SendNodeDeclaration ( '{' ActionBodyItem* '}' )?

TransitionAssignmentActionUsage : AssignmentActionUsage =

```

```

AssignmentNodeDeclaration ( '{' ActionBodyItem* '}' )?

TransitionSuccessionMember : OwningMembership =
    ownedRelatedElement += TransitionSuccession

TransitionSuccession : Succession =
    ownedRelationship += EmptyEndMember
    ownedRelationship += ConnectorEndMember

EmptyEndMember : EndFeatureMembership =
    ownedRelatedElement += EmptyFeature

EmptyFeature : Feature =
    {}

```

8.2.2.18 Calculations Textual Notation

8.2.2.18.1 Calculation Definitions

```

CalculationDefinition =
    OccurrenceDefinitionPrefix 'calc' 'def'
    DefinitionDeclaration CalculationBody

CalculationBody : Type =
    ';' | '{' CalculationBodyPart '}''

CalculationBodyPart : Type =
    CalculationBodyItem*
    ( ownedRelationship += ResultExpressionMember )?

CalculationBodyItem : Type =
    ActionBodyItem
    | ownedRelationship += ReturnParameterMember

ReturnParameterMember : ReturnParameterMembership =
    MemberPrefix? 'return' ownedRelatedElement += UsageElement

ResultExpressionMember : ResultExpressionMembership =
    MemberPrefix? ownedRelatedElement += OwnedExpression

```

8.2.2.18.2 Calculation Usages

```

CalculationUsage : CalculationUsage =
    OccurrenceUsagePrefix 'calc'
    CalculationUsageDeclaration CalculationBody

CalculationUsageDeclaration : Step =
    UsageDeclaration ValuePart?

```

8.2.2.19 Constraints Textual Notation

```
ConstraintDefinition =
    OccurrenceDefinitionPrefix 'constraint' 'def'
        DefinitionDeclaration CalculationBody

ConstraintUsage =
    OccurrenceUsagePrefix 'constraint'
        CalculationUsageDeclaration CalculationBody

AssertConstraintUsage =
    OccurrenceUsagePrefix 'assert' ( isNegated ?= 'not' )?
        ( ownedRelationship += OwnedReferenceSubsetting
            FeatureSpecializationPart?
            | 'constraint' UsageDeclaration )
        CalculationBody
```

8.2.2.20 Requirements Textual Notation

8.2.2.20.1 Requirement Definitions

```
RequirementDefinition =
    OccurrenceDefinitionPrefix 'requirement' 'def'
    DefinitionDeclaration RequirementBody?

RequirementBody : Type =
    ';' | '{' RequirementBodyItem* '}''

RequirementBodyItem : Type =
    DefinitionBodyItem
    | ownedRelationship += SubjectMember
    | ownedRelationship += RequirementConstraintMember
    | ownedRelationship += FramedConcernMember
    | ownedRelationship += RequirementVerificationMember
    | ownedRelationship += ActorMember
    | ownedRelationship += StakeholderMember

SubjectMember : SubjectMembership =
    MemberPrefix ownedRelatedElement += SubjectUsage

SubjectUsage : ReferenceUsage =
    'subject' Usage

RequirementConstraintMember : RequirementConstraintMembership =
    MemberPrefix? RequirementKind
    ownedRelatedElement += RequirementConstraintUsage

RequirementKind : RequirementConstraintMembership =
    'assume' { kind = 'assumption' }
    | 'require' { kind = 'requirement' }

RequirementConstraintUsage : ConstraintUsage =
    ownedRelationship += OwnedReferenceSubsetting
    FeatureSpecializationPart? RequirementBody
    | 'constraint' CalculationUsageDeclaration CalculationBody

FramedConcernMember : FramedConcernMembership =
    MemberPrefix? 'frame'
    ownedRelatedElement += FramedConcernUsage

FramedConcernUsage : ConcernUsage =
    ownedRelationship += OwnedReferenceSubsetting
    FeatureSpecializationPart? RequirementBody
    | 'concern' CalculationUsageDeclaration CalculationBody

ActorMember : ActorMembership =
    MemberPrefix ownedRelatedElement += ActorUsage

ActorUsage : PartUsage =
    'actor' Usage

StakeholderMember : StakeholderMembership =
    MemberPrefix ownedRelatedElement += StakeholderUsage

StakeholderUsage : PartUsage =
    'stakeholder' Usage
```

8.2.2.20.2 Requirement Usages

```
RequirementUsage =
    OccurrenceUsagePrefix 'requirement'
    CalculationUsageDeclaration RequirementBody

SatisfyRequirementUsage =
    OccurrenceUsagePrefix 'assert' ( isNegated ?= 'not' ) 'satisfy'
    ( ownedRelationship += OwnedReferenceSubsetting
        FeatureSpecializationPart?
    | 'requirement' UsageDeclaration )
    ValuePart?
    ( 'by' ownedRelationship += SatisfactionSubjectMember )?
    RequirementBody

SatisfactionSubjectMember : SubjectMembership =
    ownedRelatedElement += SatisfactionParameter

SatisfactionParameter : ReferenceUsage =
    ownedRelationship += SatisfactionFeatureValue

SatisfactionFeatureValue : FeatureValue =
    ownedRelatedElement += SatisfactionReferenceExpression

SatisfactionReferenceExpression : FeatureReferenceExpression =
    ownedRelationship += FeatureChainMember
```

8.2.2.20.3 Concerns

```
ConcernDefinition =
    OccurrenceDefinitionPrefix 'concern' 'def'
    DefinitionDeclaration RequirementBody?

ConcernUsage =
    OccurrenceUsagePrefix 'concern'
    CalculationUsageDeclaration RequirementBody
```

8.2.2.21 Cases Textual Notation

```
CaseDefinition =
    OccurrenceDefinitionPrefix 'case' 'def'
    DefinitionDeclaration CaseBody

CaseUsage =
    OccurrenceUsagePrefix 'case'
    CalculationUsageDeclaration CaseBody

CaseBody : Type =
    ';'
    | '{' CaseBodyItem*
        ( ownedRelationship += ResultExpressionMember ) ?
    '}'

CaseBodyItem : Type =
    ActionBodyItem
    | ownedRelationship += SubjectMember
    | ownedRelationship += ActorMember
    | ownedRelationship += ObjectiveMember

ObjectiveMember : ObjectiveMembership =
    MemberPrefix 'objective'
    ownedRelatedElement += ObjectiveRequirementUsage

ObjectiveRequirementUsage : RequirementUsage =
    CalculationUsageDeclaration RequirementBody
```

8.2.2.22 Analysis Cases Textual Notation

```
AnalysisCaseDefinition =
    OccurrenceDefinitionPrefix 'analysis' 'def'
    DefinitionDeclaration CaseBody

AnalysisCaseUsage =
    OccurrenceUsagePrefix 'analysis'
    CalculationUsageDeclaration CaseBody
```

8.2.2.23 Verification Cases Textual Notation

```
VerificationCaseDefinition =
    OccurrenceDefinitionPrefix 'verification' 'def'
    DefinitionDeclaration CaseBody

VerificationCaseUsage =
    OccurrenceUsagePrefix 'verification'
    CalculationUsageDeclaration CaseBody

RequirementVerificationMember : RequirementVerificationMembership =
    MemberPrefix 'verify' { kind = 'requirement' }
    ownedRelatedElement += RequirementVerificationUsage

RequirementVerificationUsage : RequirementUsage =
    ownedRelationship += OwnedReferenceSubsetting
    FeatureSpecialization* RequirementBody
    | 'requirement' CalculationUsageDeclaration RequirementBody
```

8.2.2.24 Use Cases Textual Notation

```
UseCaseDefinition =
    OccurrenceDefinitionPrefix 'use' 'case' 'def'
    DefinitionDeclaration CaseBody

UseCaseUsage =
    OccurrenceUsagePrefix 'use' 'case'
    CalculationUsageDeclaration CaseBody

IncludeUseCaseUsage :
    OccurrenceUsagePrefix 'include'
    ( ownedRelationship += OwnedReferenceSubsetting
        FeatureSpecializationPart?
        | 'use' 'case' UsageDeclaration )
    ValuePart?
    CaseBody
```

8.2.2.25 Views and Viewpoints Textual Notation

8.2.2.25.1 View Definitions

```
ViewDefinition =
    OccurrenceDefinitionPrefix 'view' 'def'
    DefinitionDeclaration ViewDefinitionBody

ViewDefinitionBody : ViewDefinition =
    ';' | '{' ViewDefinitionBodyItem* '}''

ViewDefinitionBodyItem : ViewDefinition =
    DefinitionBodyItem
    | ownedRelationship += ElementFilterMember
    | ownedRelationship += ViewRenderingMember

ViewRenderingMember : ViewRenderingMembership =
    MemberPrefix 'render'
    ownedRelatedElement += ViewRenderingUsage

ViewRenderingUsage : RenderingUsage =
    ownedRelationship += OwnedReferenceSubsetting
    FeatureSpecializationPart?
    UsageBody
```

8.2.2.25.2 View Usages

```
ViewUsage =
    OccurrenceUsagePrefix 'view'
    UsageDeclaration? ValuePart?
    ViewBody

ViewBody : ViewUsage =
    ';' | '{' ViewBodyItem* '}''

ViewBodyItem : ViewUsage =
    DefinitionBodyItem
    | ownedRelationship += ElementFilterMember
    | ownedRelationship += ViewRenderingMember
    | ownedRelationship += Expose

Expose =
    ( visibility = VisibilityIndicator )?
    'expose' ( ImportedNamespace | ImportedFilterPackage ) ';'
```

8.2.2.25.3 Viewpoints

```
ViewpointDefinition =
    OccurrenceDefinitionPrefix 'viewpoint' 'def'
    DefinitionDeclaration RequirementBody

ViewpointUsage =
    OccurrenceUsagePrefix 'viewpoint'
    CalculationUsageDeclaration RequirementBody
```

8.2.2.25.4 Renderings

```
RenderingDefinition =
    OccurrenceDefinitionPrefix 'rendering' 'def'
    Definition

RenderingUsage =
    OccurrenceUsagePrefix 'rendering'
    Usage
```

8.2.2.26 Metadata Textual Notation

```
MetadataDefinition =
  ( isAbstract ?= 'abstract')? 'metadata' 'def'
  Definition

PrefixMetadataAnnotation : Annotation =
  '#' annotatingElement = PrefixMetadataUsage
  { ownedRelatedElement += annotatingElement }

PrefixMetadataMember : OwningMembership =
  '#' ownedRelatedEleemnt = PrefixMetadataUsage

PrefixMetadataUsage : MetadataUsage =
  ownedRelationship += OwnedFeatureTyping

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

MetadataFeatureDeclaration : MetadataUsage =
  ( Identification ( ':' | 'typed' 'by' ) )?
  ownedRelationship += OwnedFeatureTyping

MetadataBody : Type =
  ';' |
  '{' ( ownedRelationship += DefinitionMember
    | ownedRelationship += MetadataBodyUsageMember
    | ownedRelationship += AliasMember
    | ownedRelationship += Import
    )*
  '}'

MetadataBodyUsageMember : FeatureMembership =
  ownedMemberFeature = MetadataBodyUsage

MetadataBodyUsage : ReferenceUsage :
  'ref'? ( ':>' | 'redefines' )? ownedRelationship += OwnedRedefinition
  FeatureSpecializationPart? ValuePart?
  MetadataBody

ExtendedDefinition : Definition =
  BasicDefinitionPrefix? DefinitionExtensionKeyword
  'def' Definition

ExtendedUsage : Usage =
  BasicUsagePrefix UsageExtensionKeyword
  Usage
```

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. [Table 35](#) summarizes the conventions used.

Table 35. Graphical BNF Conventions

Non-terminal element	non-terminal-element
Non-terminal element production (complete)	non-terminal-element = elements
Non-terminal element production (partial)	non-terminal-element = elements
Grouping	(elements)
Alternative elements	elements elements
Repeated elements (zero or more)	element *
Repeated elements (one or more)	element +
Optional elements (zero or one)	element ?
Elements	2-D layout of graphical and textual elements
Graphical element	graphical shape or graphical line
Graphical shape	2-D shape with optional nested elements
Graphical line	1-D shape with optional nested elements
Graphical line that connects other elements	&element graphical-line &element
Sequential text elements	element1 element2
Terminal text element as literal string	'terminal'
Terminal text element as lexical symbol	LEXICAL
Graphical Notation to Textual Notation mapping	graphical production <=> textual production

These conventions make a distinction between a complete production, which must include all alternatives within the production itself, and partial productions, which allow alternatives to be distributed across multiple productions located anywhere within a specification. This distinction allows greater reuse of production symbols across sections of a specification that build on partial productions given by earlier sections, while still making clear productions that are already complete within a given section.

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. All production symbols within the Graphical BNF follow a convention of all-lowercase names with optional internal hyphens. Elements of the Textual

Notation defined in Clause 8.2.2 of this specification may also be referenced by textual productions within the Graphical BNF. These imported Textual Notation elements can be distinguished from those of the Graphical BNF by their use of one or more uppercase letters within the name.

Release Note. Only a limited part of the graphical notation has been formally specified so far. Specification of the rest will be completed before the final submission.

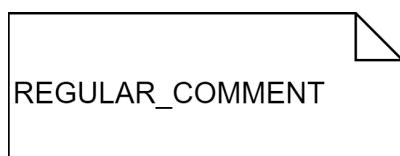
8.2.3.2 Elements and Relationships Graphical Notation

```
general-view =  
    (graphical-element)*  
  
graphical-element =  
    graphical-node  
    | graphical-relationship-with-name  
  
graphical-relationship-with-name =  
    (relationship-name)?  
    graphical-relationship  
  
relationship-name = name-with-optional-short-name  
  
name-with-optional-short-name = ( '<' NAME '>' )? NAME  
  
qualified-name = ( NAME '::' )* NAME
```

8.2.3.3 Annotations Graphical Notation

8.2.3.3.1 Nodes

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



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



```

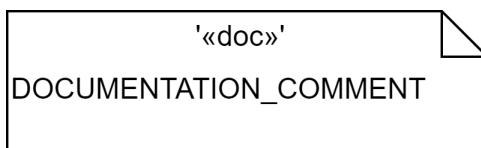
comment-with-name =
'«comment»'
comment-name
REGULAR_COMMENT

comment-name = name-with-optional-id

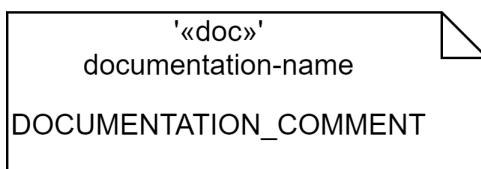
documentation-annotation-node =
  documentation-annotation-without-name
  | documentation-annotation-with-name

```

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



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



```
documentation-name = name-with-optional-id
```

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



```
language-string = 'language' '=' '''language-name'''
language-name = STRING_VALUE
```

8.2.3.3.2 Relationships

```
graphical-relationship =|
  annotation
```

```
annotation =
```

```

&annotated-element annotation-link &annotation-node

annotated-element =
    graphical-element
    | textual-element-in-compartment

annotation-link =
    ●-----
```

Note. 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

8.2.3.4.1 Nodes

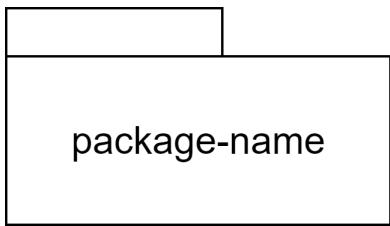
```

package-view = ( pv-graphical-element | annotation-element )*
pv-graphical-element =
    pv-graphical-node
    | pv-graphical-relationship

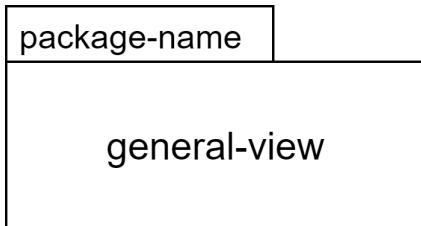
pv-graphical-node =|
    namespace-node
    | package-node

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 =
```

```

-----+
|       |
| package-name   |
|       |
-----+
imported-package-with-name-in-tab =
-----+
| package-name |
|       |
| graphical-view |
|       |
-----+

```

package-name = name-with-optional-id

8.2.3.4.2 Relationships

```

pv-graphical-relationship =|
  import
  | top-level-import
  | recursive-import
  | owned-membership
  | unowned-membership

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

top-level-import =
  &namespace-node — '«import» *' —> &namespace-node

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

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

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

```

8.2.3.5 Dependencies Graphical Notation

```
graphical-relationship =|
    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 =
    ----->
```

Note. 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

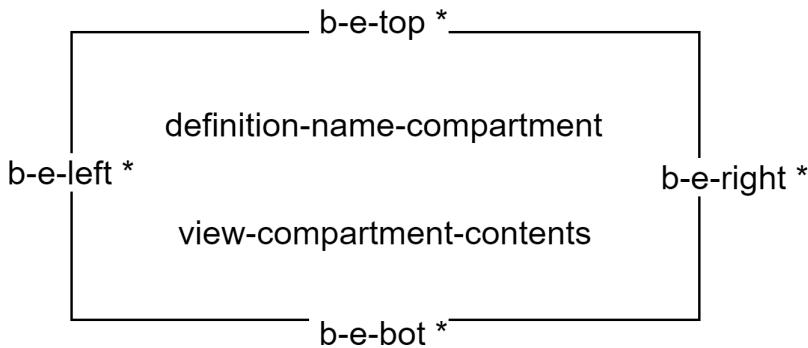
8.2.3.6.1 Nodes

```
graphical-node =|
    type-node

type-node =
    definition-node
    | usage-node

definition-node =
    b-e-left * _____ b-e-top *
    |                                |
    |      definition-name-compartment      |
    |                                |
    |      ( vertical-view-compartment ) * |
    |                                |
    |_____ b-e-right *
```

|

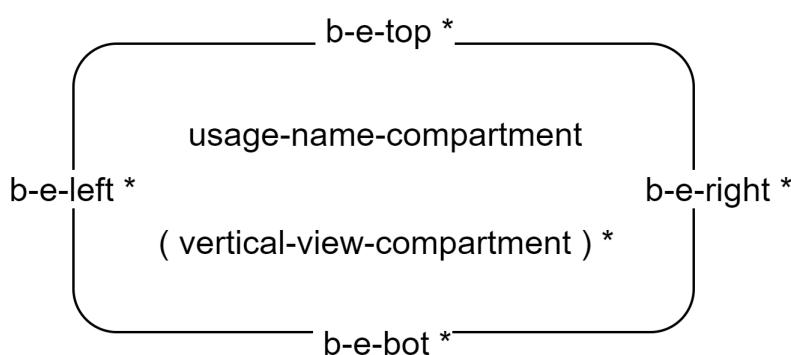


```

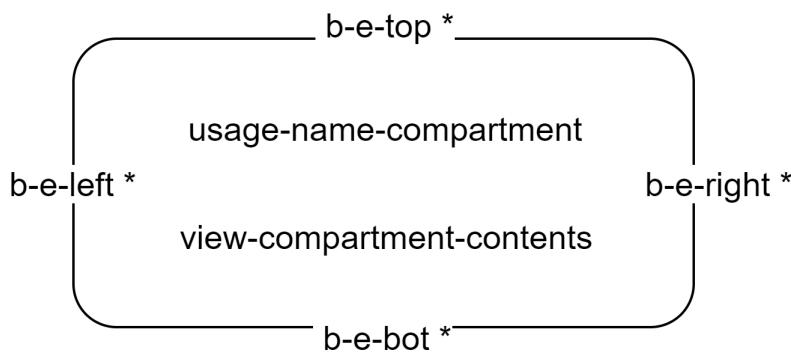
definition-name-compartment =
('«abstract»')?
('«variation»')?
definition-type-keyword
definition-name (':> qualified-name)?
('«alias»' ( qualified-name (',' qualified-name)* ) )?

```

usage-node =



|



```

usage-name-compartment =
('«abstract»')?
('«variation»' | '«variant»')?
usage-type-keyword
qualified-name (':> qualified-name)?
('«alias»' ( qualified-name (',' qualified-name)* ) )?

```

```

vertical-view-compartment =

```

view-compartment-contents

```

definition-name = name-with-optional-short-name

b-e-top =|  ()
b-e-bottom =|  ()
b-e-left =|  ()
b-e-right =|  ()

el-prefix = '^' | '/'

view-compartment-contents =|
    documentation-compartment
| metadata-compartment
| namespaces-compartment
| relationships-compartment
| variants-compartment
| variant-elementusages-compartment

```

8.2.3.6.2 Relationships

```

graphical-relationship =|
    type-relationship

type-relationship =
    subclassification
| subsetting
| definition
| redefinition
| composite-feature-membership
| noncomposite-feature-membership

subclassification =

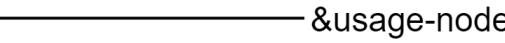
```

&definition-node  &definition-node

```

subsetting =

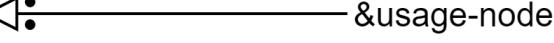
```

&usage-node  &usage-node

```

definition =

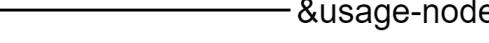
```

&definition-node  &usage-node

```

redefinition =

```

&usage-node  &usage-node

```

composite-feature-membership =
  &type-node ◆—————&usage-node

```

```

noncomposite-feature-membership =
  &type-node ◇—————&usage-node

```

8.2.3.7 Attributes Graphical Notation

```

definition-type-keyword =| '<<attribute def>>'
usage-type-keyword =| '<<attribute>>'

view-compartment-contents =| attributes-compartment

attributes-compartment = attributes-compartment-element* '...'?
attributes-compartment-element =
  el-prefix? UsagePrefix UsageDeclaration DefinitionBodyItem*

```

8.2.3.8 Enumerations Graphical Notation

```

definition-type-keyword =| '<<enumeration def>>'
usage-type-keyword =| '<<enumeration>>'

view-compartment-contents =| enums-compartment

enums-compartment = enums-compartment-element* '...'?
enums-compartment-element = el-prefix? UsagePrefix Usage

```

8.2.3.9 Occurrences Graphical Notation

```

definition-type-keyword =|
  '<<occurrence def>>'|
  '| '<<individual>>''

usage-type-keyword =|
  '<<occurrence>>'|
  '| '<<individual>>'|
  '| '<<timeslice>>'|
  '| '<<snapshot>>'

view-compartment-contents =|
  occurrences-compartment
  | individuals-compartment
  | timeslices-compartment
  | snapshots-compartment

occurrences-compartment = occurrences-compartment-element* '...'?
occurrences-compartment-element =
  el-prefix? OccurrenceUsagePrefix Usage DefinitionBodyItem*

individuals-compartment = individuals-compartment-element* '...'?
individuals-compartment-element =
  el-prefix? OccurrenceUsagePrefix Usage DefinitionBodyItem*

```

```

timeslices-compartment = timeslices-compartment-element* '...'?
timeslices-compartment-element =
    el-prefix? OccurrenceUsagePrefix Usage DefinitionBodyItem*

snapshots-compartment = snapshots-compartment-element* '...'?
snapshots-compartment-element =
    el-prefix? OccurrenceUsagePrefix Usage DefinitionBodyItem*

```

8.2.3.10 Items Graphical Notation

```

definition-type-keyword =| '«item def»'
usage-type-keyword =| '«item»'

view-compartment-contents =| items-compartment

items-compartment = items-compartment-element* '...'
items-compartment-element =
    el-prefix? OccurrenceUsagePrefix Usage DefinitionBodyItem*

```

8.2.3.11 Parts Graphical Notation

```

definition-type-keyword =| '«part def»'
usage-type-keyword =| '«part»'

view-compartment-contents =|
    parts-compartment
    | parts-compartment-graphical
    | directed-features-compartment
    | shapes-compartment

parts-compartment = parts-compartment-element* '...'?
parts-compartment-element =
    el-prefix? OccurrenceUsagePrefix UsageDeclaration
    ValueOrFlowPart? DefinitionBodyItem*

directed-features-compartment = directed-features-compartment-element* '...'?
directed-features-compartment-element =
    el-prefix FeatureDirection Definition-Body-Item*

```

8.2.3.12 Ports Graphical Notation

```

definition-type-keyword =| '«port def»'
usage-type-keyword =| '«port»'

view-compartment-contents =| ports-compartment

ports-compartment = port-compartment-element* '...'?
port-compartment-element =
    el-prefix? OccurrenceUsagePrefix UsageDeclaration
    ValueOrFlowPart? DefinitionBodyItem*

```

8.2.3.13 Connections Graphical Notation

```

definition-type-keyword =| '«connection def»'
usage-type-keyword =| '«connection»'

view-compartment-contents =| connections-compartment

connections-compartment = connections-compartment-element* '...'?

```

```
connections-compartment-element =
    el-prefix? OccurrenceUsagePrefix UsageDeclaration ConnectorPart+ DefinitionBodyItem*
```

8.2.3.14 Interfaces Graphical Notation

```
definition-type-keyword =| '«interface def»'
usage-type-keyword =| '«interface»'

view-compartment-contents =|
    interfaces-compartment
| ends-compartment

interfaces-compartment = interfaces-compartment-element* '...'?
interfaces-compartment-element =
    el-prefix? InterfaceUsageDeclaration InterfaceBodyDefinition*
```

8.2.3.15 Allocations Graphical Notation

```
definition-type-keyword =| '«allocation def»'
usage-type-keyword =| '«allocation»'

view-compartment-contents =|
    allocations-compartment
| allocated-compartment

allocations-compartment = allocations-compartment-element* '...'?
allocations-compartment-element =
    el-prefix? OccurrenceUsagePrefix AllocationUsageDeclaration UsageBody*

allocated-compartment = allocated-compartment-element* '...'?
allocated-compartment-element =
    el-prefix? AllocationUsageDeclaration ValuePartOrFlowPart DefBodyItem*
```

8.2.3.16 Actions Graphical Notation

```
definition-type-keyword =| '«action def»'
usage-type-keyword =| '«action»'

view-compartment-contents =|
    actions-compartment
| actions-compartment-graphical
| action-flow-compartment-graphical
| state-actions-compartment
| perform-actions-compartment
| perform-actions-compartment-graphical
| performed-by-compartment
| parameters-compartment

actions-compartment =actions-compartment-element* '...'?
actions-usage-compartment-element =
    el-prefix? OccurrenceUsagePrefix ActionUsageDeclaration ActionBodyItem*

state-actions-compartment =state-actions-compartment-element* '...'?
state-action-compartment-element =
    el-prefix? EntryActionMember | DoActionMember | ExitActionMember

perform-action-compartment = perform-action-compartment-element* '...'?
perform-action-compartment-element =
    el-prefix? PerformActionUsageDeclaration ActionBodyItem*
```

```

parameter-compartment = parameter-compartment-element* '...'?
parameter-compartment-element =
    el-prefix? FeatureDirection UsageDeclaration ValueOrFlowPart? DefinitionBodyItem*

```

8.2.3.17 States Graphical Notation

```

definition-type-keyword =| '«state def»'
usage-type-keyword =| '«state»'

view-compartment-contents =|
    states-compartment
| states-compartment-graphical
| exhibit-states-compartment
| successions-compartment

states-compartment = states-compartment-element* '...'?
states-compartment-element = el-prefix? UsageDeclaration

exhibit-states-compartment = exhibit-state-scompartment-element* '...'?
exhibit-states-compartment-element-compartment = UsageDeclaration

```

8.2.3.18 Calculations Graphical Notation

```

definition-type-keyword =| '«calc def»'
usage-type-keyword =| '«calc»'

view-compartment-contents =|
    calc-compartment
| result-compartment

```

8.2.3.19 Constraints Graphical Notation

```

definition-type-keyword =| '«constraint def»'
usage-type-keyword =| '«constraint»'

view-compartment-contents =|
    constraints-compartment
| assert-constraints-compartment

constraints-compartment = constraints-usage-compartment* '...'?
constraints-usage-compartment =
    el-prefix? OccurrenceUsagePrefix CalculationUsageDeclaration CalculationBody*

assert-constraints-compartment= assert-constraints-compartment-element* '...'?
assert-constraints-compartment-element =
    el-prefix? OccurrenceUsagePrefix ( 'not' )?
        ( OwnedSubsetting FeatureSpecializationPart? | UsageDeclaration )
    CalculationUsageParameterPart CalculationBody

```

8.2.3.20 Requirements Graphical Notation

```

definition-type-keyword =| '«requirement def»'
usage-type-keyword =| '«requirement»'

view-compartment-contents =|
    requirements-compartment
| satisfy-requirements-compartment

requirements-compartment = requirements-compartment-element* '...'?

```

```

requirements-compartment-element = OccurrenceUsagePrefix CalculationUsageDeclaration

satisfy-requirements-compartment = satisfy-requirements-compartment-element* '...'?
satisfy-requirements-compartment-element =
    OccurrenceUsagePrefix ('not' )?
    ( OwnedSubsetting FeatureSpecializationPart? | UsageDeclaration )
    ( ValuePart | ActionParameterList ) ( 'by' SatisfactionSubjectMember )? RequirementBody

```

8.2.3.21 Cases Graphical Notation

```

view-compartment-contents =|
    subject-compartment
| objective-compartment
| actors-compartment
| body-compartment

subject-compartment = subject-compartment-element* '...'?
subject-compartment-element = el-prefix? MemberPrefix Usage

objective-compartment = objective-compartment-element* '...'?
objective-compartment-element =
    comp-prefix? MemberPrefix CalculationUsageDeclaration RequirementBody

actors-compartment = actors-compartment-element* '...'?
actors-compartment-element = el-prefix? MemberPrefix Usage

body-compartment = body-compartment-element* '...'?
body-compartment-element = el-prefix? DefinitionBodyItem*

```

8.2.3.22 Analysis Cases Graphical Notation

```

definition-type-keyword =| '«analysis def»'
usage-type-keyword =| '«analysis»'

view-compartment-contents =| analyses-compartment

analyses-compartment = analyses-compartment-element* '...'?
analyses-compartment-element =
    el-prefix? OccurrenceUsagePrefix CalculationUsageDeclaration CaseBody

```

8.2.3.23 Verification Cases Graphical Notation

```

definition-type-keyword =| '«verification def»'
usage-type-keyword =| '«verification»'

view-compartment-contents =|
    verifications-compartment
| verifies-compartment
| verification-methods-compartment

verifications-compartment = verifications-compartment-element* '...'?
verifications-compartment-element =
    el-prefix? OccurrenceUsagePrefix CalculationUsageDeclaration CaseBody

verifies-compartment = verifies-compartment-element* '...'?
verifies-compartment-element = el-prefix? MemberPrefix RequirementVerificationUsage

verification-methods-compartment = verification-methods-compartment-element* '...'?
verification-methods-compartment-element = MetadataBody

```

8.2.3.24 Use Cases Graphical Notation

```
definition-type-keyword =| '«use case def»'
usage-type-keyword =| '«user case»'

view-compartment-contents =|
    use-case-compartment-graphical
| includes-compartment
| include-actions-compartment

includes-compartment = includes-compartment-element* '...'?
includes-compartment-element =
    el-prefix? OccurrenceUsagePrefix
    ( OwnedSubsetting FeatureSpecializationPart? | UsageDeclaration )

include-actions-compartment = use-cases-includes-action-compartment-element* '...'?
include-actions-compartment-element =
    el-prefix? OccurrenceUsagePrefix
    ( OwnedSubsetting FeatureSpecializationPart? | UsageDeclaration )
    ( ValuePart | ActionUsageParameterList )? CaseBody
```

8.2.3.25 Views and Viewpoints Graphical Notation

```
definition-type-keyword =| '«viewpoint def»'
usage-type-keyword =| '«viewpoint»'

definition-type-keyword =| '«view def»'
usage-type-keyword =| '«view»'

view-compartment-contents =|
    stakeholders-compartment
| concerns-compartment
| frames-compartment
| views-compartment
| views-compartment-graphical
| viewpoint-compartment
| filters-compartment
| rendering-compartment

stakeholders-compartment = stakeholders-compartment-element* '...'?
stakeholders-compartment-element = el-prefix? MemberPrefix Usage

frames-compartment = frames-compartment-element* '...'?
frames-compartment-element = el-prefix* MemberPrefix? FramedConcernUsage

views-compartment = views-compartment-element* '...'
views-compartment-element =
    el-prefix? OccurrenceUsagePrefix UsageDeclaration? ValueOrFlowPart? ViewBody

viewpoint-compartment = viewpoint-compartment-element* '...'?
viewpoint-compartment-element =
    el-prefix? OccurrenceUsagePrefix CalculationUsageDeclaration RequirementBody

filters-compartment = filters-compartment-element* '...'?
filters-compartment-element = el-prefix? MemberPrefix OwnedExpression
```

A model library in Section 9.2.18 defines standard graphical view definitions for SysML. These may be supplemented by further, customized View Definitions specific to a model.

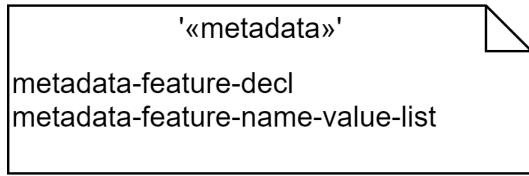
8.2.3.26 Metadata Graphical Notation

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

metadata-feature-annotation-node =
  '«metadata»'
  metadata-feature-decl
  metadata-feature-name-value-list

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

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



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 Elements and Relationships Abstract Syntax

This is a Kernel abstract syntax model. For Elements and Relationships Abstract Syntax class descriptions, see [KerML, 8.3.2.1].

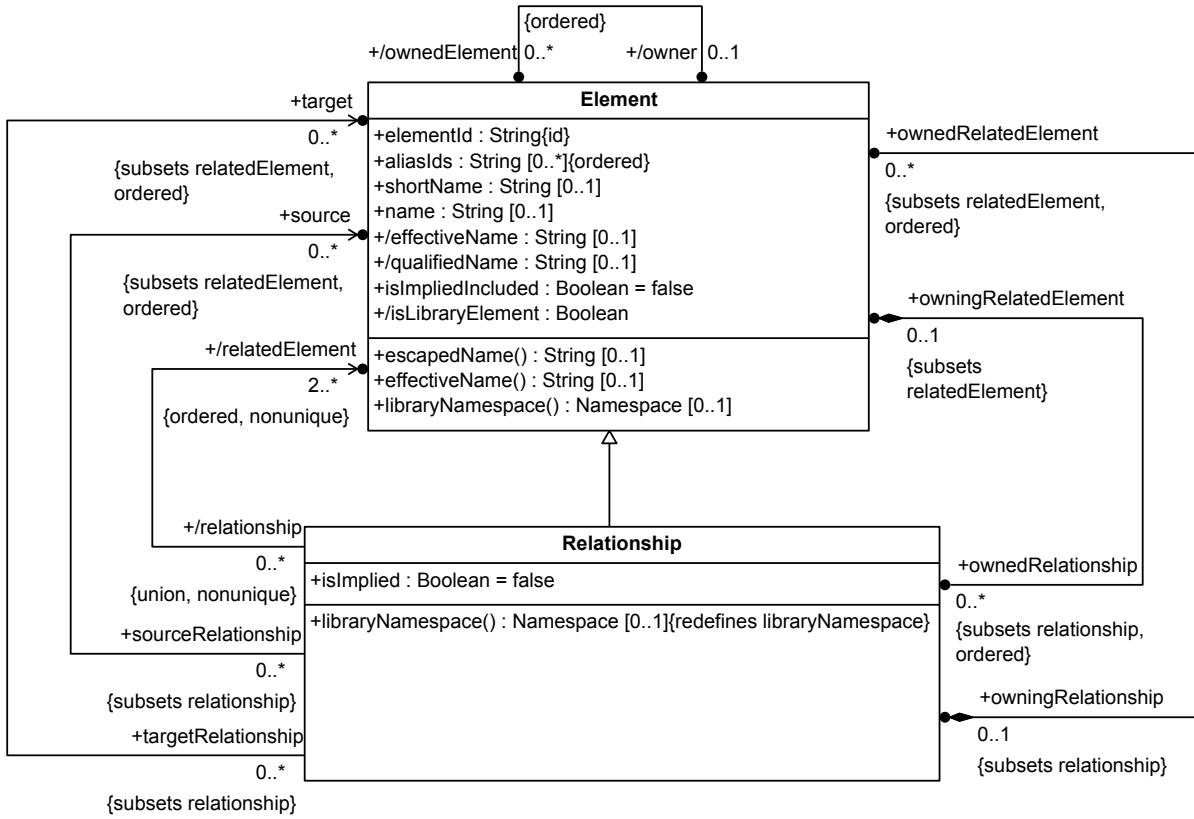


Figure 56. Elements

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

8.3.3 Annotations Abstract Syntax

This is a Kernel abstract syntax model. For Annotations Abstract Syntax class descriptions, see [KerML, 8.3.2.2].

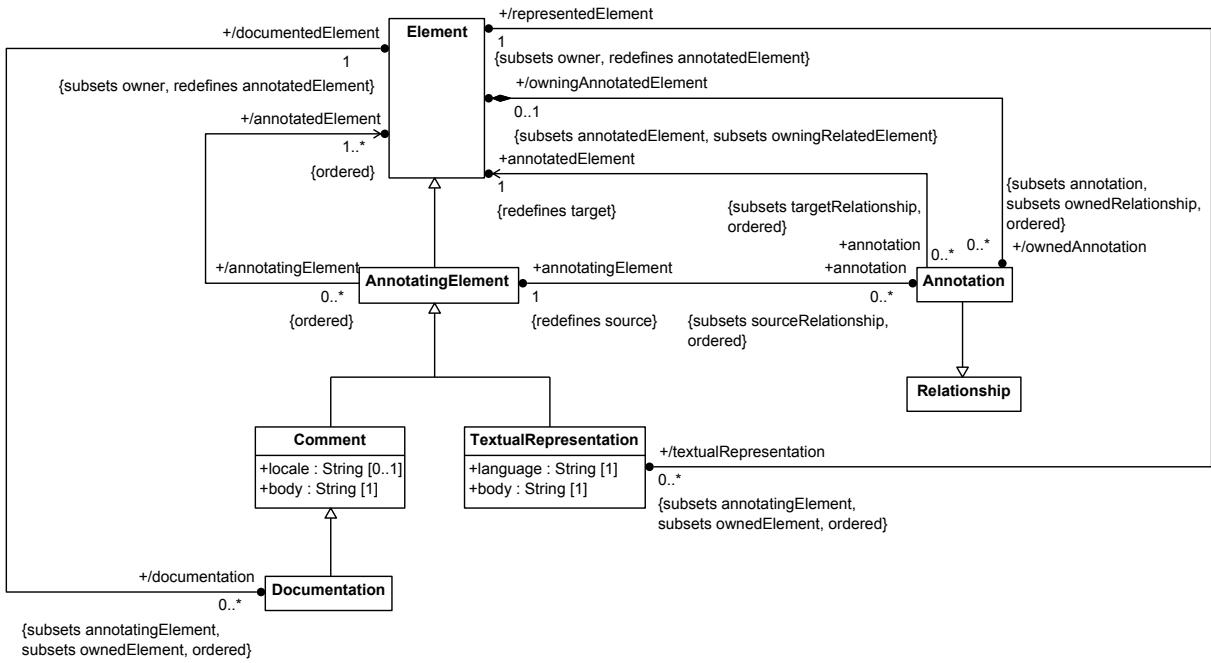


Figure 57. Annotation

8.3.4 Namespaces and Packages Abstract Syntax

This is a Kernel abstract syntax model. For Namespaces Abstract Syntax class descriptions, see [KerML, 8.3.2.3]. For Packages Abstract Syntax class descriptions, see [KerML, 8.3.4.13].

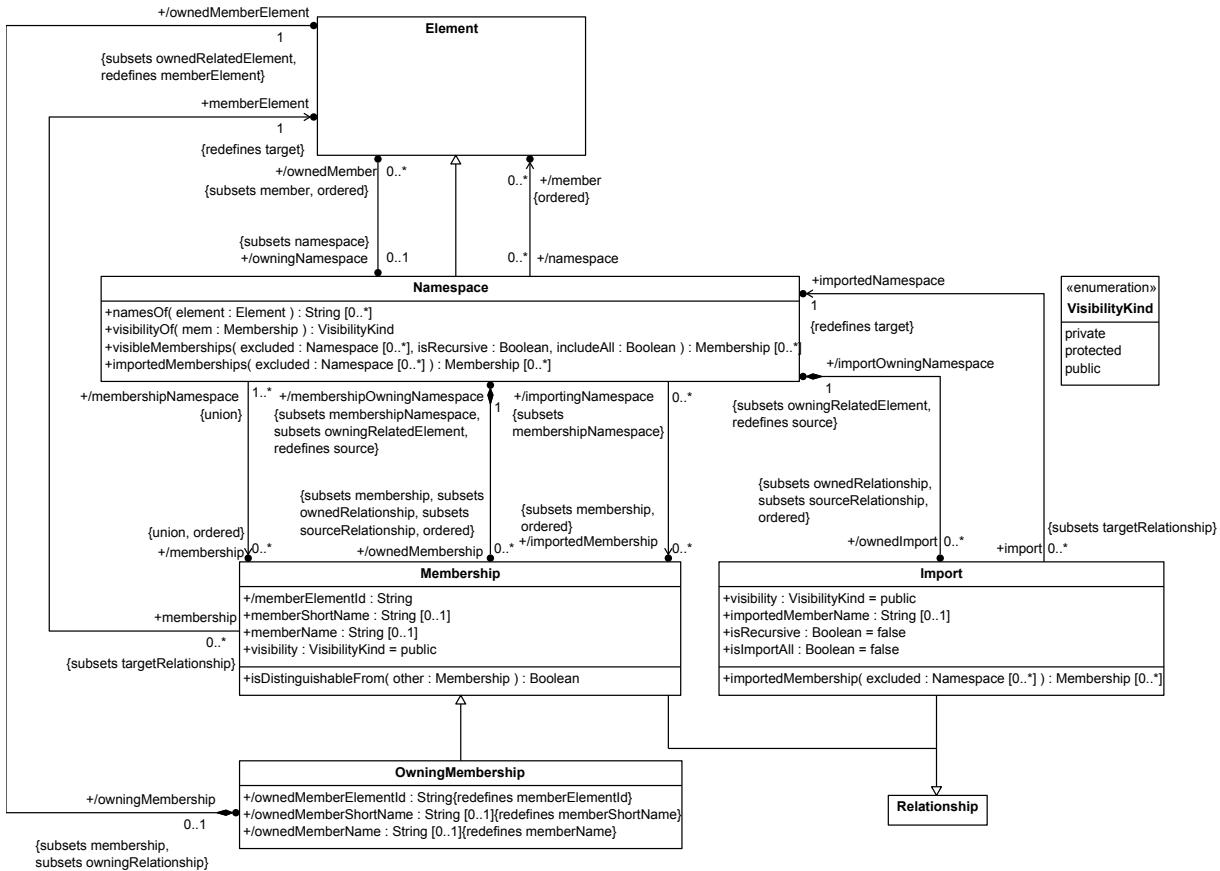


Figure 58. Namespaces

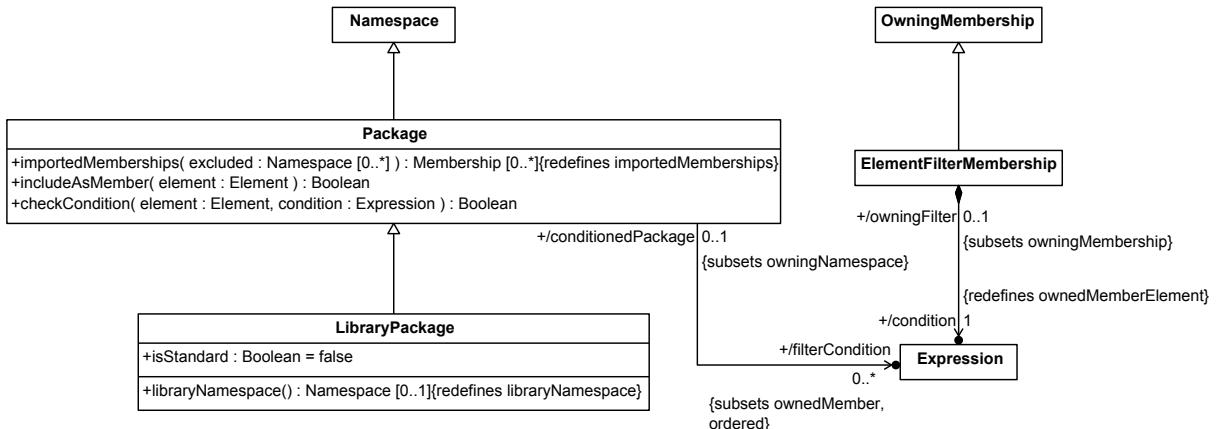


Figure 59. Packages

8.3.5 Dependencies Abstract Syntax

8.3.5.1 Overview

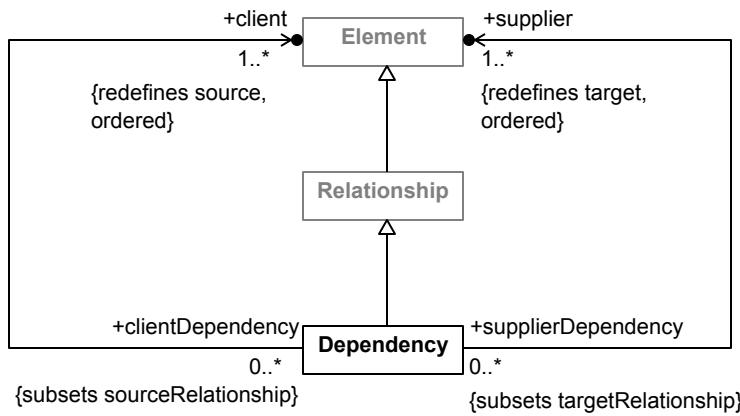


Figure 60. Dependencies

8.3.5.2 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

None.

8.3.6 Definition and Usage Abstract Syntax

8.3.6.1 Overview

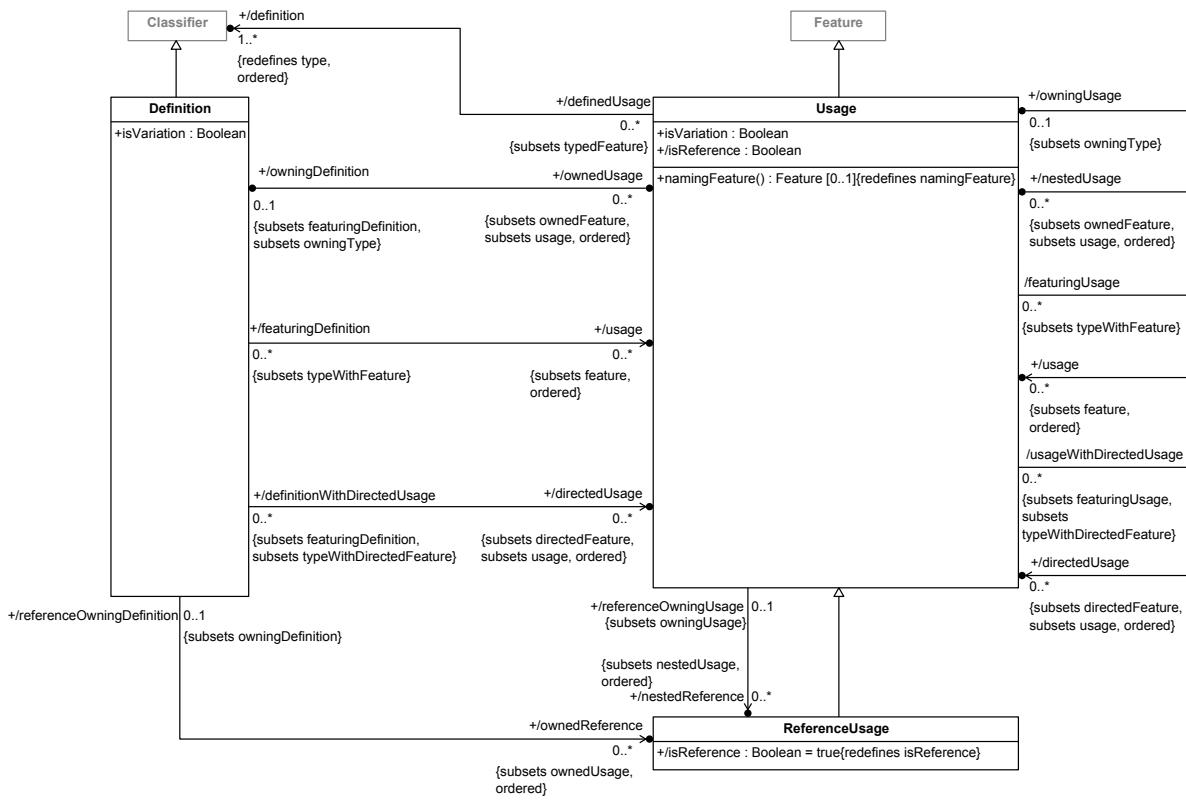


Figure 61. Definition and Usage

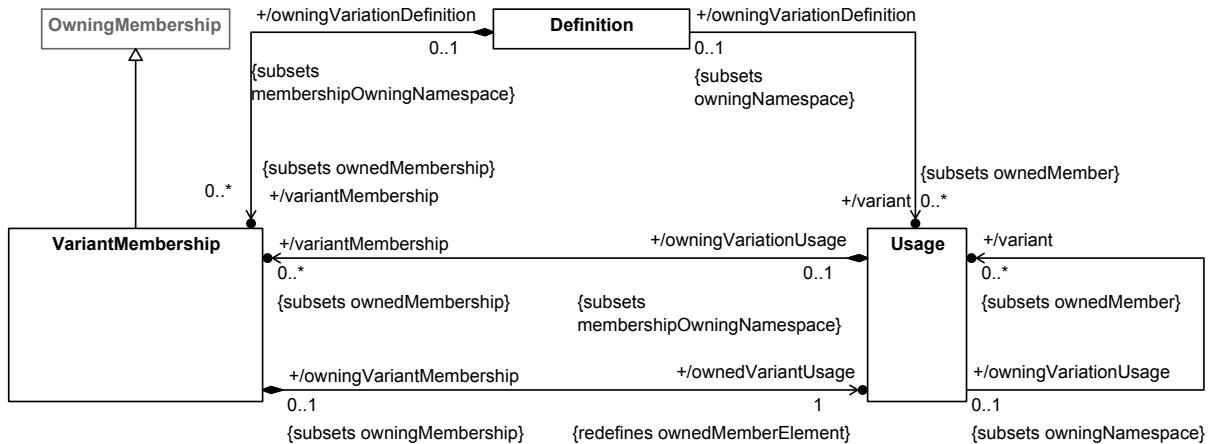


Figure 62. Variant Membership

8.3.6.2 Definition

Description

A Definition is a Classifier of Usages. The actual kinds of Definitions that may appear in a model are given by the subclasses of Definition (possibly as extended with user-defined *SemanticMetadata*).

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.

`/ownedMetadata : MetadataUsage [0..*] {subsets ownedItem, ordered}`

The MetadataUsages 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)
```

8.3.6.3 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

None.

8.3.6.4 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.

`/nestedMetadata : MetadataUsage [0..*] {subsets nestedItem, ordered}`

The MetadataUsages 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`, the there must be no `variants`.

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

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

Operations

`namingFeature() : Feature [0..1]`

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

```
body: if not owningMembership.oclIsKindOf(VariantMembership) then  
    self.oclaSType(Feature).namingFeature()
```

```
else
    let namingFeature : Feature = firstSubsettedFeature() 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
```

8.3.6.5 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

OwningMembership

Attributes

/ownedVariantUsage : Usage {redefines ownedMemberElement}

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

Operations

No operations.

Constraints

None.

8.3.7 Attributes Abstract Syntax

8.3.7.1 Overview

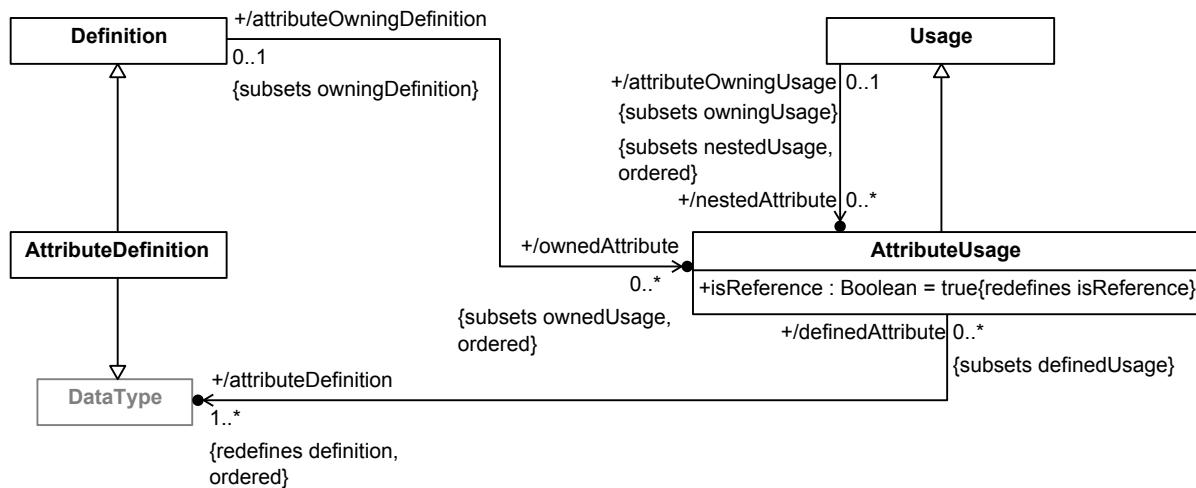


Figure 63. Attribute Definition and Usage

8.3.7.2 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 definition, 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

None.

8.3.7.3 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 Types

Definition
DataType

Features

None.

Constraints

None.

8.3.8 Enumerations Abstract Syntax

8.3.8.1 Overview

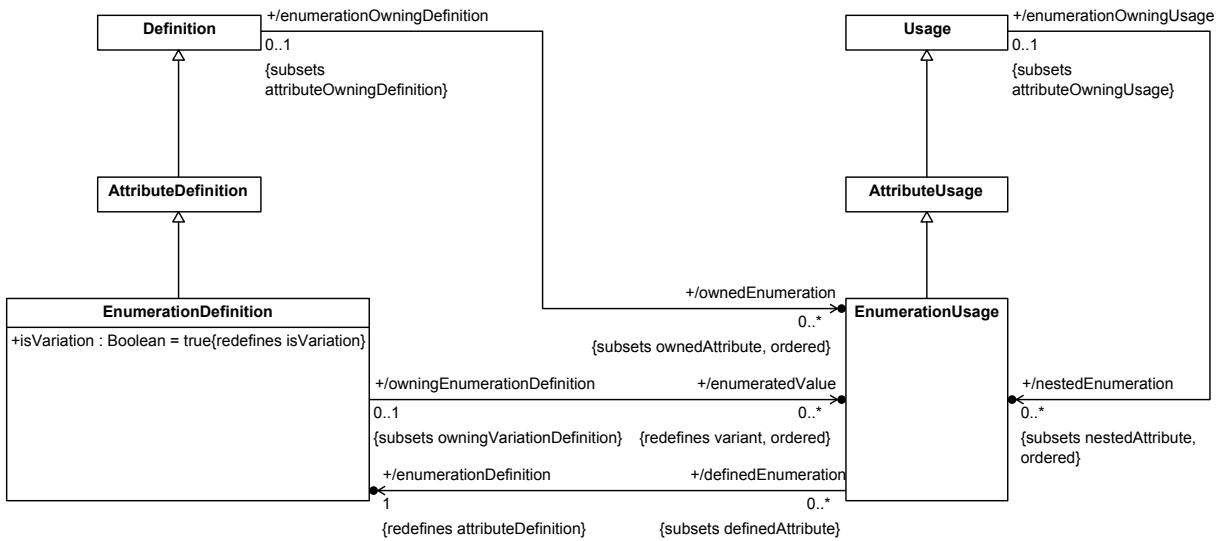


Figure 64. Enumeration Definition and Usage

8.3.8.2 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

None.

8.3.8.3 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

None.

8.3.9 Occurrences Abstract Syntax

8.3.9.1 Overview

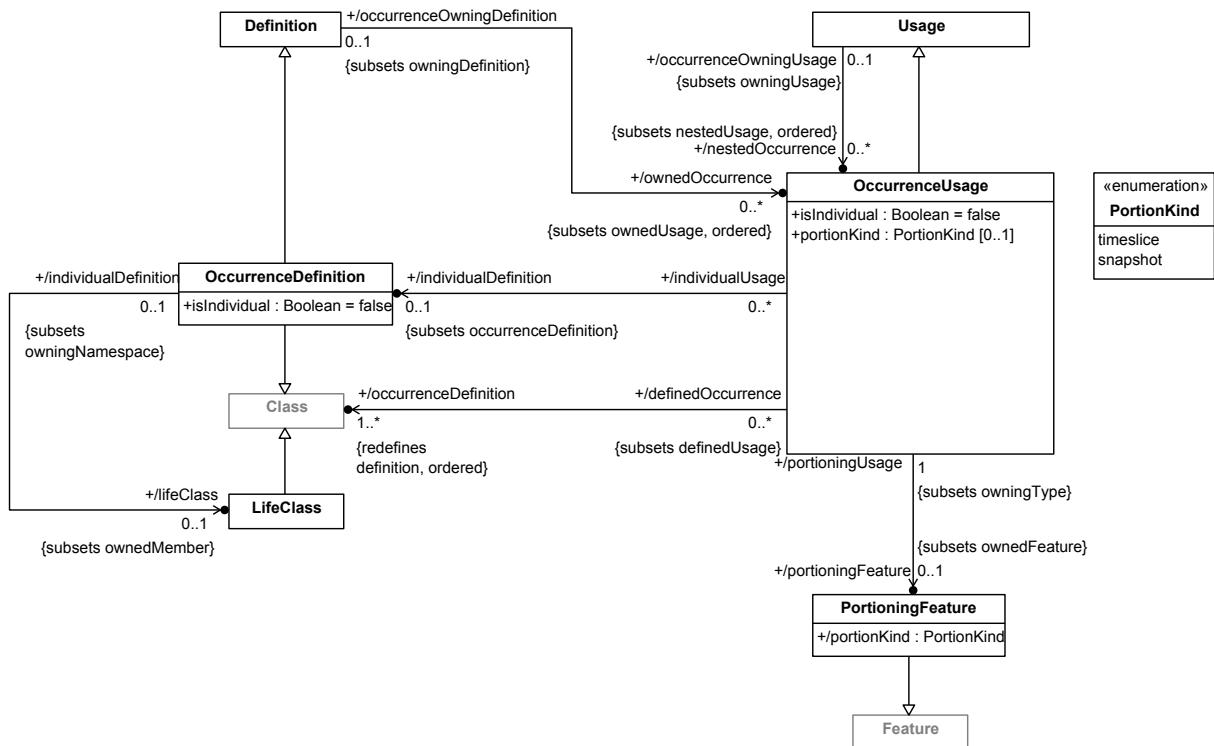


Figure 65. Occurrence Definition and Usage

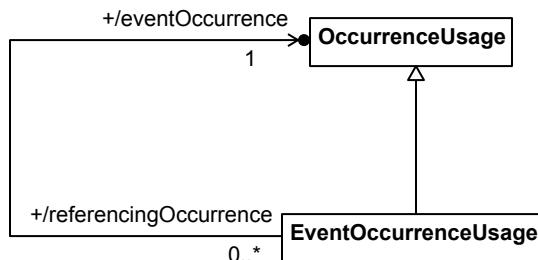


Figure 66. Event Occurrences

8.3.9.2 EventOccurrenceUsage

Description

A **EventOccurrenceUsage** is an **OccurrenceUsage** that represents another **OccurrenceUsage** occurring as a suboccurrence of the containing occurrence of the **EventOccurrenceUsage**. Unless it is the **EventOccurrenceUsage** itself, the referenced **OccurrenceUsage** performed is related to the **EventOccurrenceUsage** by a **ReferenceSubsetting** relationship.

If the **EventOccurrenceUsage** is owned by an **OccurrenceDefinition** or **OccurrenceUsage**, then it also subsets the `timeEnclosedOccurrences` property of the Class `Occurrence` from the Kernel Semantic Library model `Occurrences`.

General Classes

OccurrenceUsage

Attributes

/eventOccurrence : OccurrenceUsage

The OccurrenceUsage referenced as an event by this EventOccurrenceUsage. It is the `referenceFeature` of the `ownedReferenceSubsetting` for the EventOccurrenceUsage, if there is one, and, otherwise, the EventOccurrenceUsage itself.

Operations

No operations.

Constraints

eventOccurrenceUsageEventOccurrence

[no documentation]

```
eventOccurrence =  
    if ownedReferenceSubsetting = null then self  
    else ownedReferenceSubsetting.referencedFeature.oclAsType(OccurrenceUsage)  
    endif
```

8.3.9.3 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

None.

Operations

No operations.

Constraints

None.

8.3.9.4 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
```

8.3.9.5 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
```

8.3.9.6 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.

8.3.9.7 PortionKind

Description

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

General Classes

None.

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).

8.3.10 Items Abstract Syntax

8.3.10.1 Overview

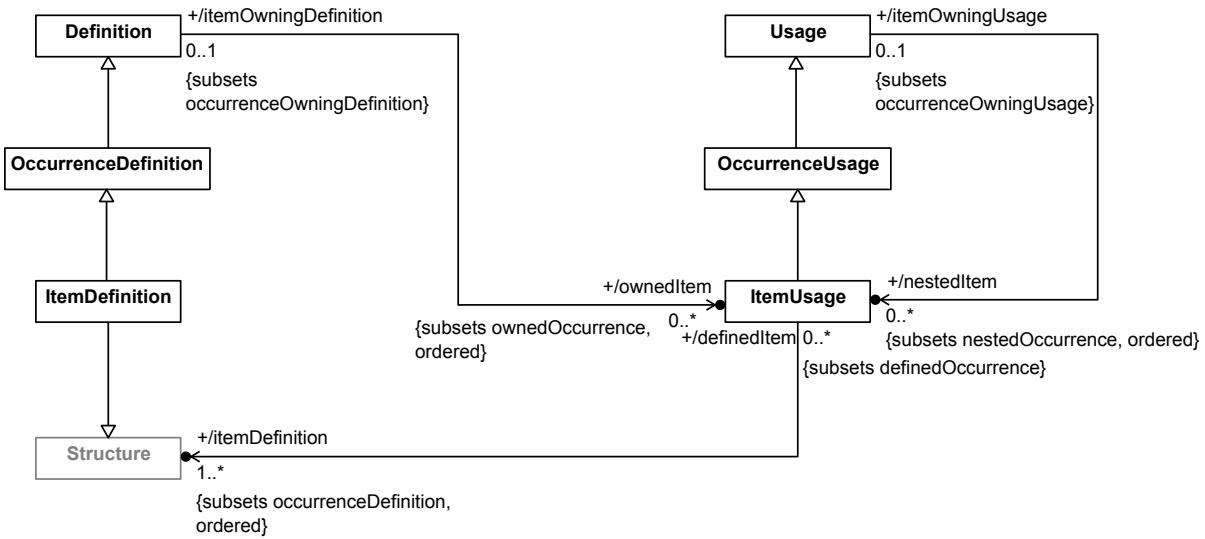


Figure 67. Item Definition and Usage

8.3.10.2 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

None.

Operations

No operations.

Constraints

None.

8.3.10.3 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..*] {subsets occurrenceDefinition, 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

None.

8.3.11 Parts Abstract Syntax

8.3.11.1 Overview

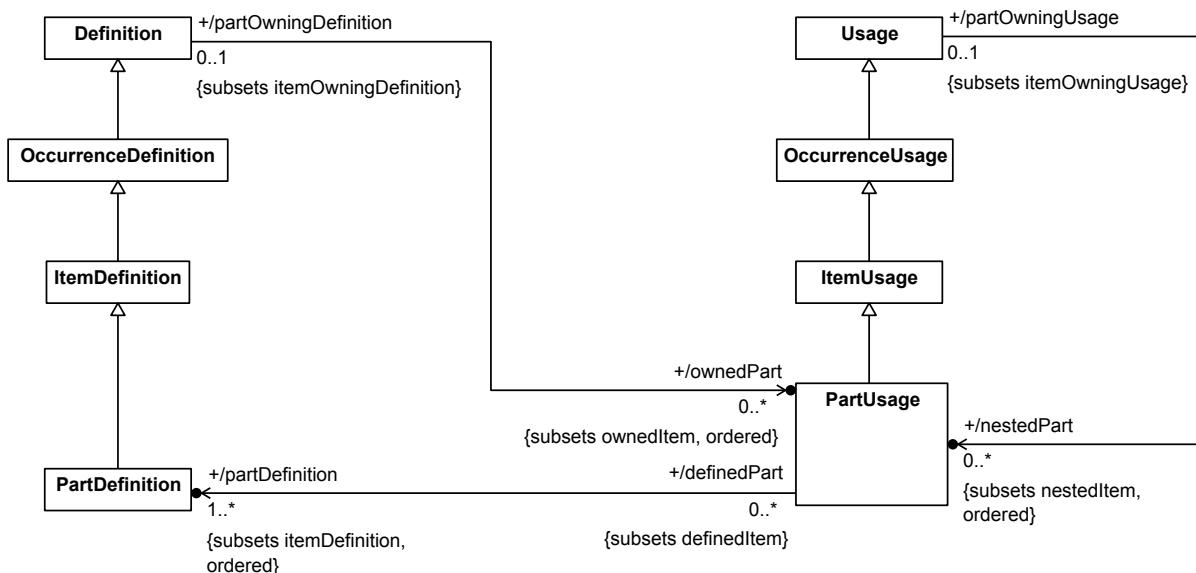


Figure 68. Part Definition and Usage

8.3.11.2 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

None.

Operations

No operations.

Constraints

None.

8.3.11.3 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

None.

8.3.12 Ports Abstract Syntax

8.3.12.1 Overview

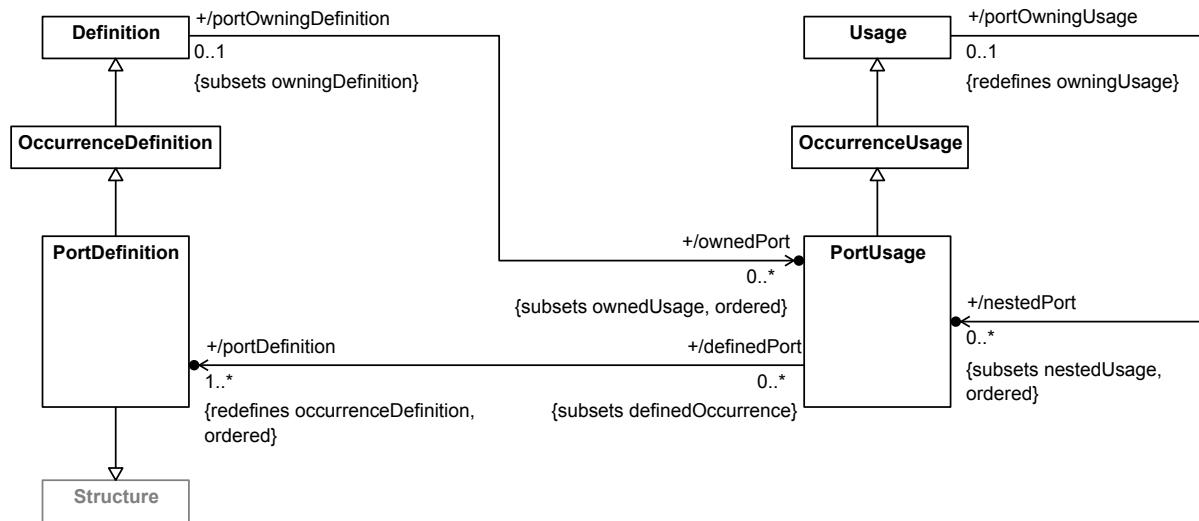


Figure 69. Port Definition and Usage

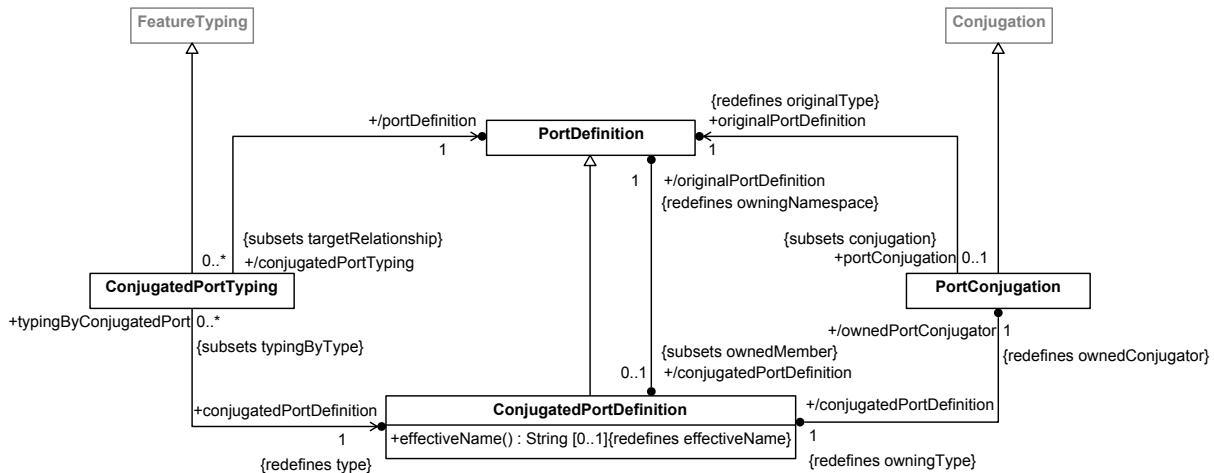


Figure 70. Port Conjugation

8.3.12.2 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 (that is not itself a ConjugatedPortDefinition) has exactly one corresponding ConjugatedPortDefinition, whose effective name is the name 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

effectiveName() : String [0..1]

If the name of the `originalPortDefinition` is non-empty, then return that with the character `~` prepended.

```
body: let originalName : String = originalPortDefinition.name in
if originalName.isEmpty() then null
else '~' + originalName
endif
```

Constraints

conjugatedPortDefinitionConjugatedPortDefinitionIsEmpty

[no documentation]

conjugatedPortDefinition = null

conjugatedPortDefinitionOriginalPortDefinition

[no documentation]

originalPortDefinition = ownedPortConjugator.originalPortDefinition

8.3.12.3 ConjugatedPortTyping

Description

A ConjugatedPortTyping is a FeatureTyping whose `type` is a ConjugatedPortDefinition. (This relationship is intended to be an abstract syntax marker for a special surface notation for conjugated typing of ports.)

General Classes

FeatureTyping

Attributes

conjugatedPortDefinition : ConjugatedPortDefinition {redefines type}

The `type` of this ConjugatedPortTyping considered as a FeatureTyping, which must be a ConjugatedPortDefinition.

/portDefinition : PortDefinition

The `originalPortDefinition` of the `conjugatedPortDefinition` of this ConjugatedPortTyping.

Operations

No operations.

Constraints

conjugatedPortTypingConjugatedPortDefinition

[no documentation]

```
conjugatedPortDefinition = portDefinition.conjugatedPortDefinition
```

8.3.12.4 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

None.

8.3.12.5 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, other than PortUsages, must not be composite.

A PortDefinition must subclass, directly or indirectly, the base Class *Port* from the Systems model library.

General Classes

OccurrenceDefinition
Structure

Attributes

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

The ConjugatedPortDefinition that is conjugate to this PortDefinition.

Operations

No operations.

Constraints

portDefinitionOwnedUsagesNotComposite

The ownedUsages of a PortUsage that are not PortUsages must not be composite.

```
ownedUsage->
    select(not oclIsKindOf(PortUsage)) ->
        forAll(not isComposite)
```

portDefinitionConjugatedPortDefinition

[no documentation]

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

8.3.12.6 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 nestedUsages of a PortUsage, other than nested PortUsages, 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 occurrenceDefinition, ordered}

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

Operations

No operations.

Constraints

portUsageNestedUsagesNotComposite

The `nestedUsages` of a PortUsage that are not themselves PortUsages must not be composite.

```
nestedUsage->
  select(not oclIsKindOf(PortUsage)) ->
  forAll(not isComposite)
```

8.3.13 Connections Abstract Syntax

8.3.13.1 Overview

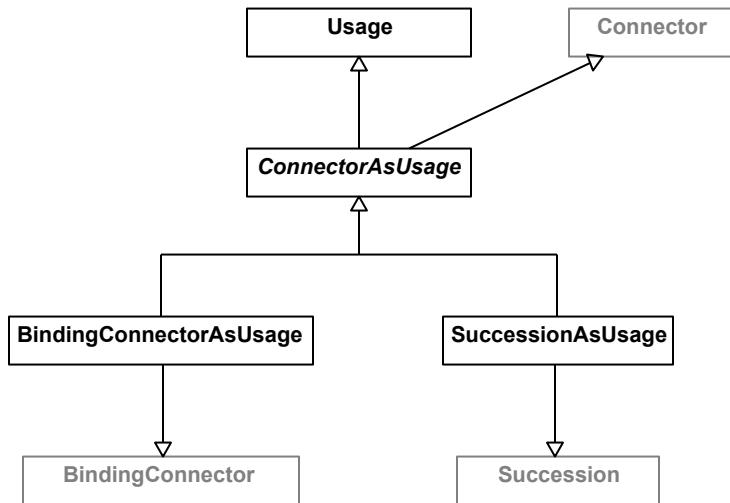


Figure 71. Connectors as Usages

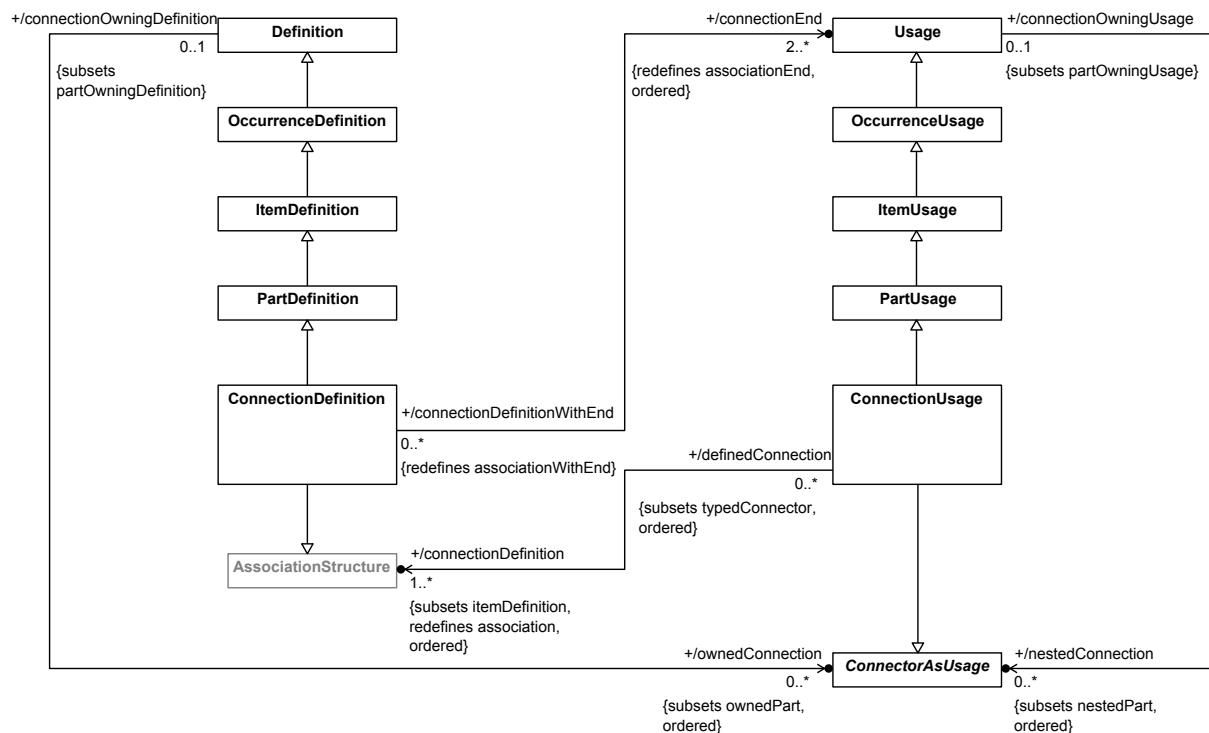


Figure 72. Connection Definition and Usage

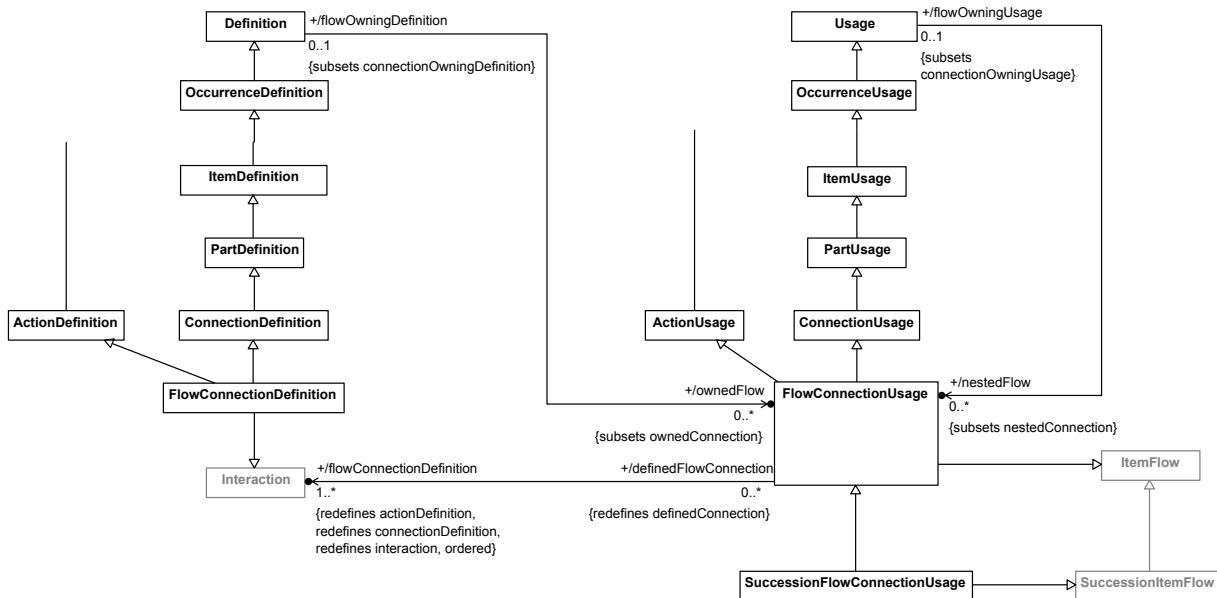


Figure 73. Flow Connections

8.3.13.2 BindingConnectorAsUsage

Description

A BindingConnectorAsUsage is both a BindingConnector and a ConnectorAsUsage.

General Classes

BindingConnector
ConnectorAsUsage

Attributes

None.

Operations

No operations.

Constraints

None.

8.3.13.3 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

None.

8.3.13.4 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

None.

8.3.13.5 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

None.

Operations

No operations.

Constraints

None.

8.3.13.6 FlowConnectionDefinition

Description

A FlowConnectionDefinition is a ConnectionDefinition and ActionDefinition that is also an Interaction representing flows between Usages.

A FlowConnectionDefinition must subclassify, directly or indirectly, the base FlowConnectionDefinition *FlowConnection* from the Systems model library.

General Classes

ActionDefinition
ConnectionDefinition
Interaction

Attributes

None.

Operations

No operations.

Constraints

None.

8.3.13.7 FlowConnectionUsage

Description

A FlowConnectionUsage is a ConnectionUsage that is also an ItemFlow.

A FlowConnectionUsage must subset the base FlowConnectionUsage *flowConnections* from the Systems model library.

General Classes

ConnectionUsage
ItemFlow
ActionUsage

Attributes

/flowConnectionDefinition : Interaction [1..*] {redefines actionDefinition, connectionDefinition, interaction, ordered}

The Interactions that are the types of this FlowConnectionUsage. Nominally, these are FlowConnectionDefinitions, but other kinds of Kernel Interactions are also allowed, to permit use of Interactions from the Kernel Library.

Operations

No operations.

Constraints

None.

8.3.13.8 SuccessionAsUsage

Description

A SuccessionAsUsage is both a ConnectorAsUsage and a Succession.

General Classes

Succession
ConnectorAsUsage

Attributes

None.

Operations

No operations.

Constraints

None.

8.3.13.9 SuccessionFlowConnectionUsage

Description

A SuccessionFlowConnectionUsage is a FlowConnectionUsage that is also a SuccessionItemFlow.

A FlowConnectionUsage must subset the base SuccessionFlowConnectionUsage *successionFlowConnections* from the Systems model library.

General Classes

SuccessionItemFlow
FlowConnectionUsage

Attributes

None.

Operations

No operations.

Constraints

None.

8.3.14 Interfaces Abstract Syntax

8.3.14.1 Overview

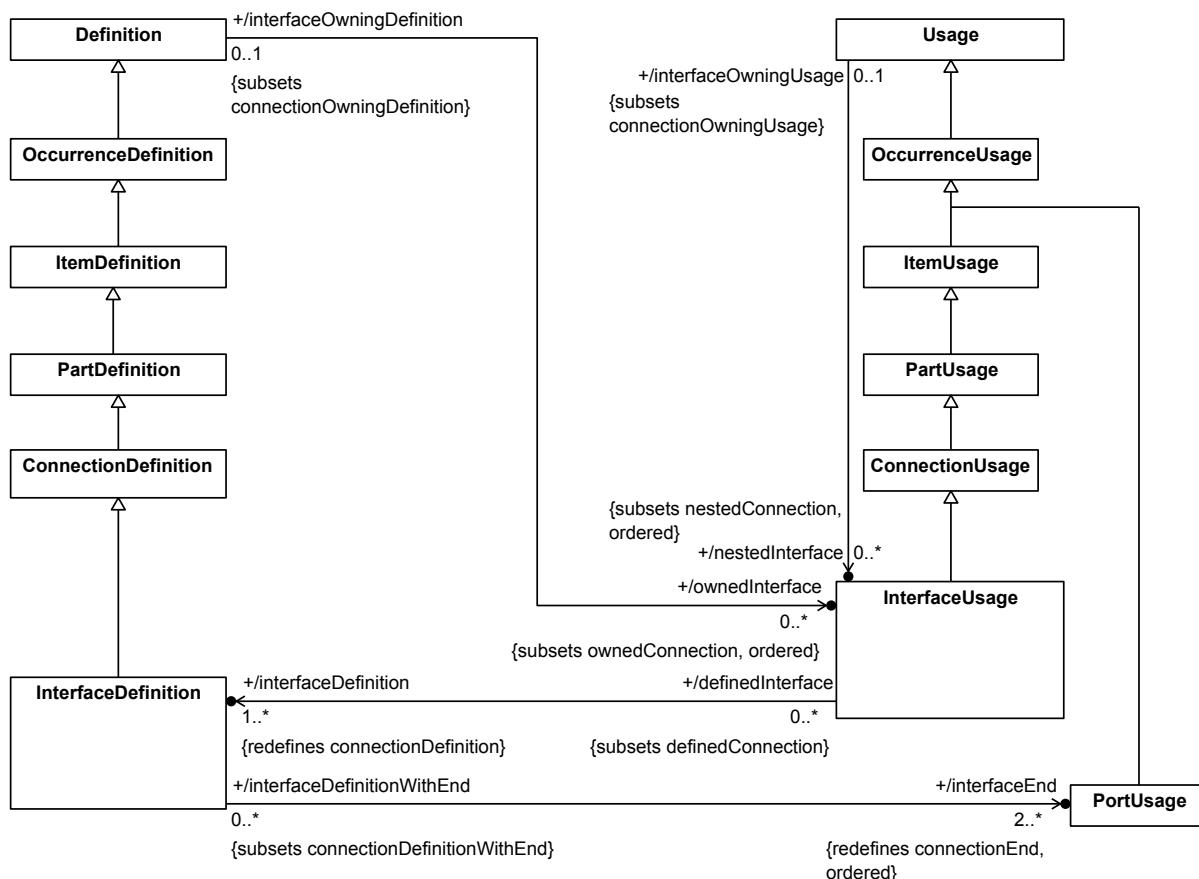


Figure 74. Interface Definition and Usage

8.3.14.2 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

None.

8.3.14.3 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

None.

8.3.15 Allocations Abstract Syntax

8.3.15.1 Overview

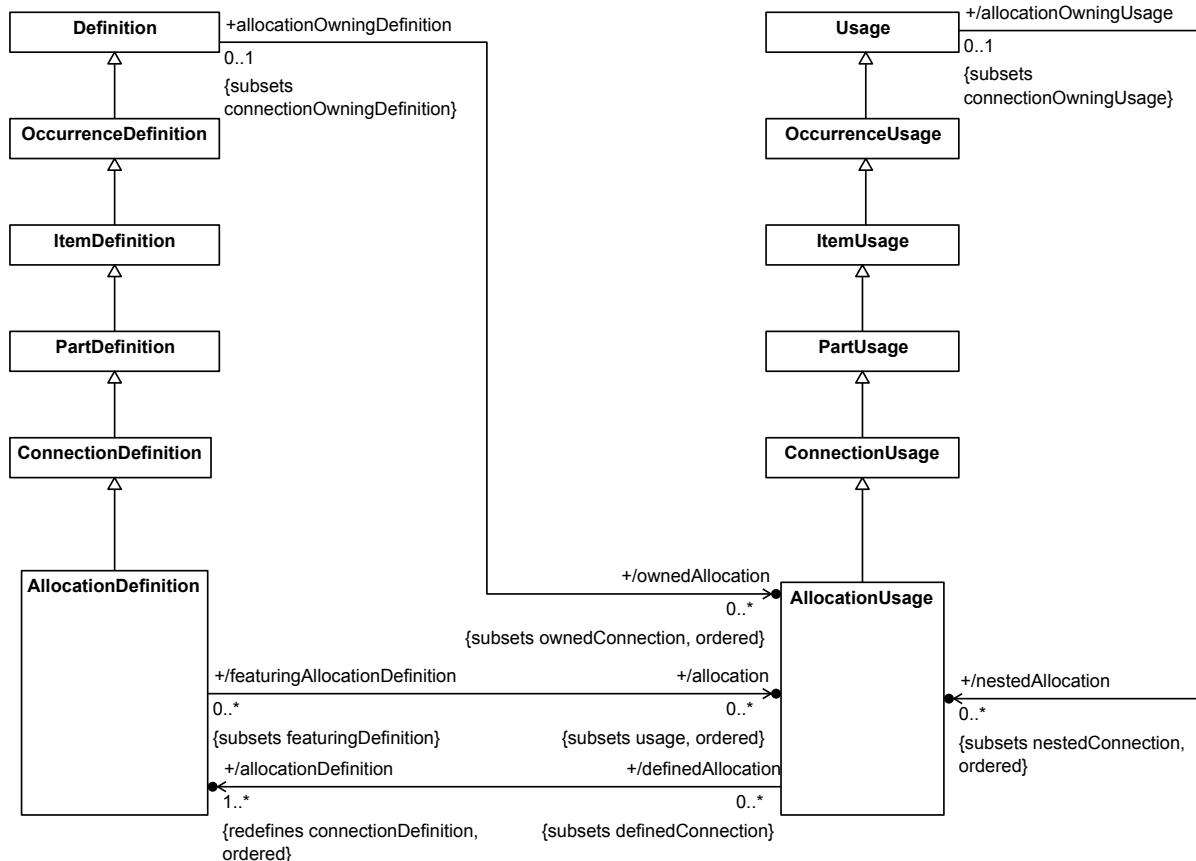


Figure 75. Allocation Definition and Usage

8.3.15.2 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

None.

8.3.15.3 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 AllocatopnUsage 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

None.

8.3.16 Actions Abstract Syntax

8.3.16.1 Overview

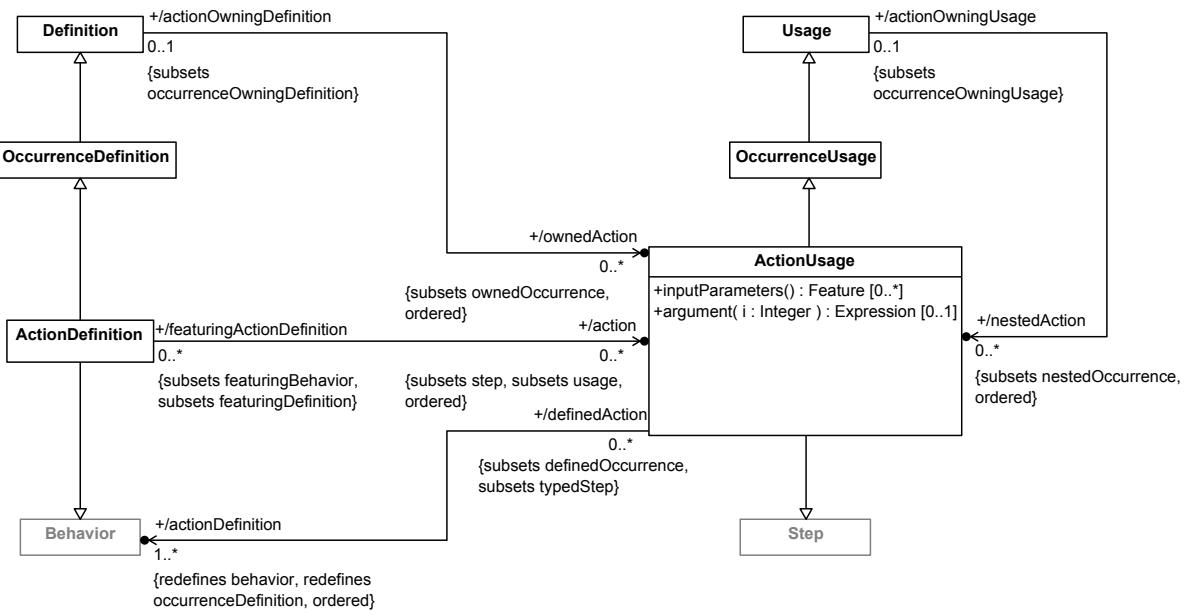


Figure 76. Action Definition and Usage

Return the owned `input` Features of an `ActionUsage`.

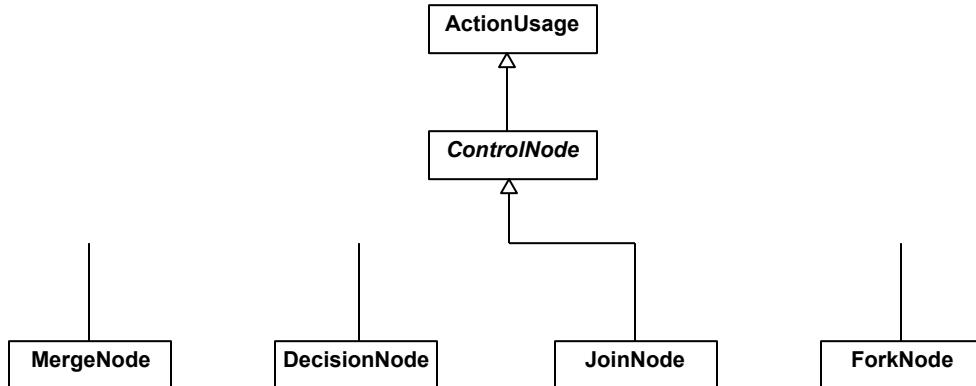


Figure 77. Control Nodes

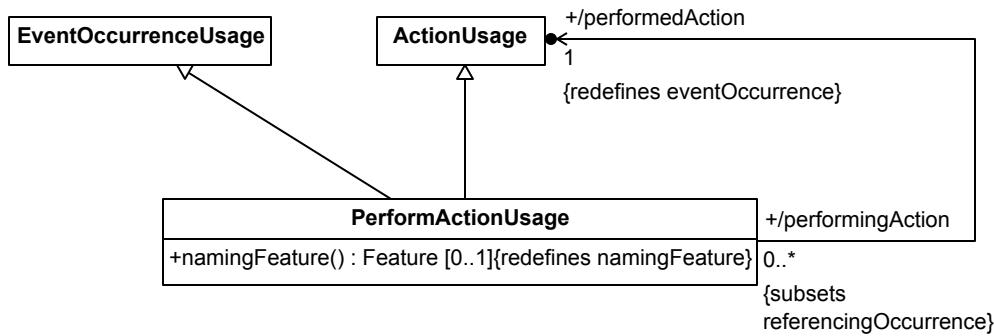


Figure 78. Action Performance

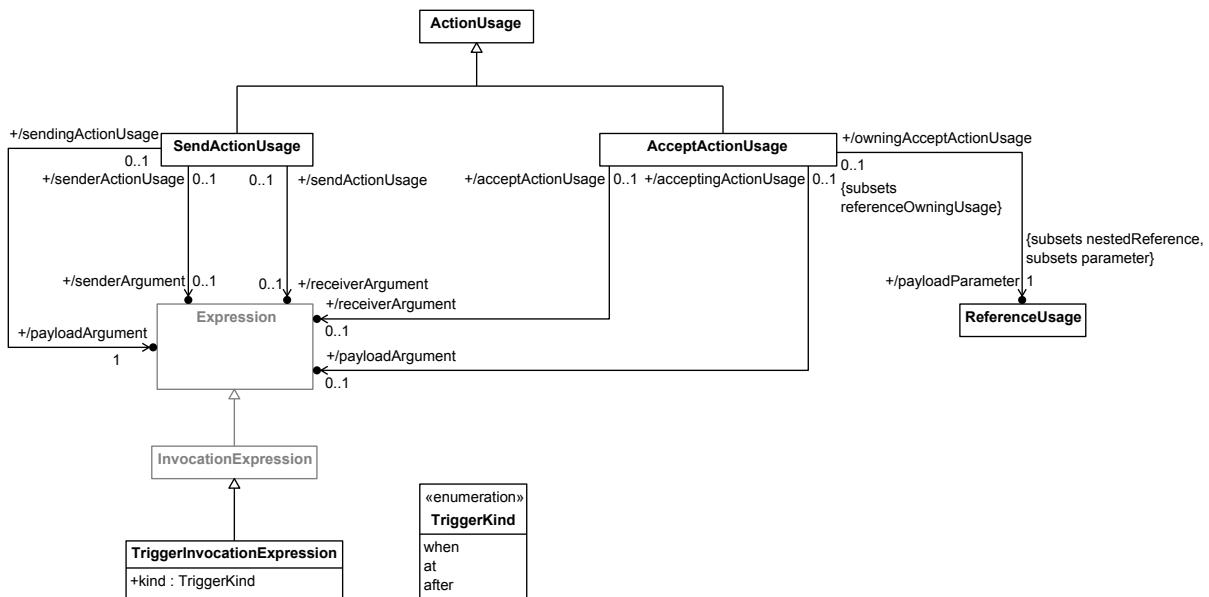


Figure 79. Send and Accept Actions

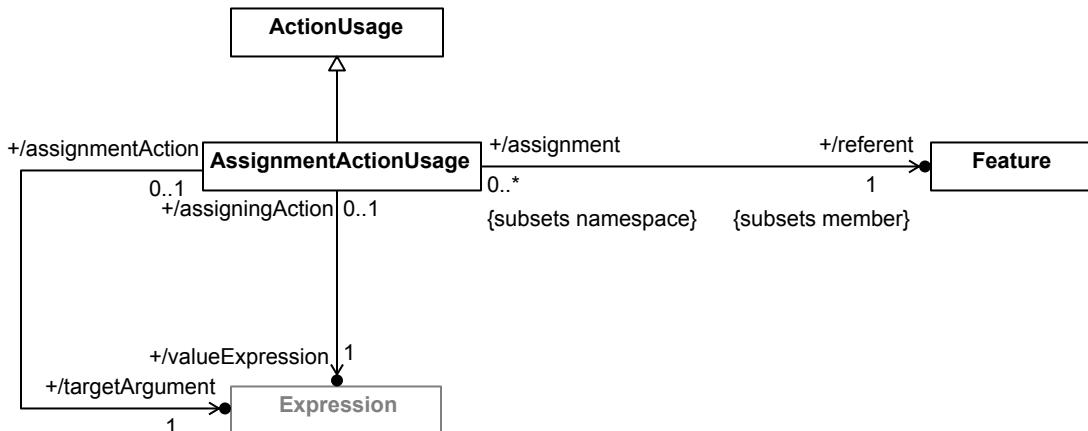


Figure 80. Assignment Actions

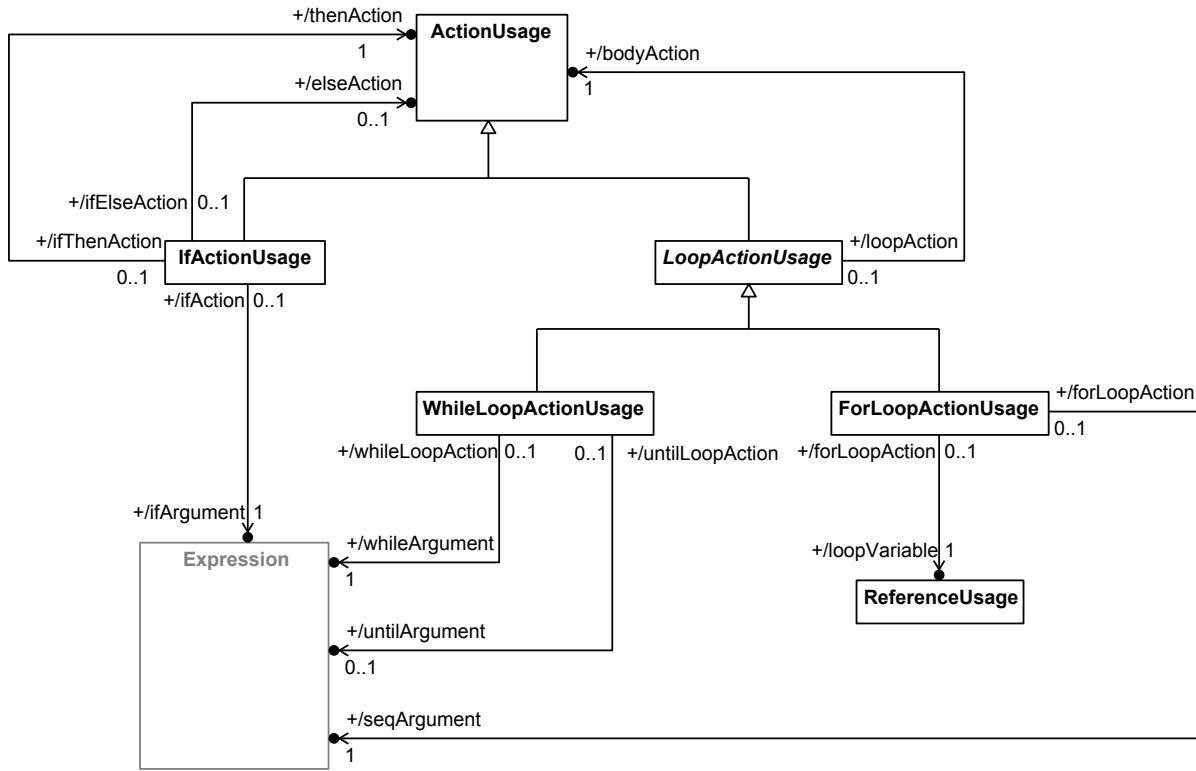


Figure 81. Structured Control Actions

8.3.16.2 AcceptActionUsage

Description

An AcceptActionUsage is an ActionUsage that is typed, directly or indirectly, by the ActionDefinition *AcceptAction* from the Systems model library (unless it is owned by a TransitionAction, in which case it is typed by *AcceptMessageAction*). It specifies the acceptance of an *incomingTransfer* from the *Occurrence* given by the result of its *receiverArgument* Expression. (If no *receiverArgument* is provided, the default is the *this* context of the AcceptActionUsage.) The payload of the accepted *Transfer* is output on its *payloadParameter*.

Which *Transfers* may be accepted is determined by the typing and binding of the *payloadParameter*. If the *triggerKind* has any value other than *accept*, then the *payloadParameter* must be bound to a *payloadArgument* that is an *InvocationExpression* whose *function* is determined by the *triggerKind*.

General Types

ActionUsage

Features

/payloadArgument : Expression [0..1]

An Expression whose result is bound to the *payload* parameter of this AcceptActionUsage. If provided, the AcceptActionUsage will only accept a *Transfer* with exactly this *payload*.

/payloadParameter : ReferenceUsage {subsets nestedReference, parameter}

The `nestedReference` of this `AcceptActionUsage` that redefines the `payload` output parameter of the base `AcceptActionUsage` `AcceptAction` from the Systems model library.

/receiverArgument : Expression [0..1]

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

Constraints

acceptActionUsageDeriveReceiverArgument

The `receiverArgument` of an `AcceptUsageAction` is its second argument Expression.

`receiverArgument = argument(2)`

acceptActionUsageDerivePayloadArgument

The `payloadArgument` of an `AcceptUsageAction` is its first argument Expression.

`payloadArgument = argument(1)`

acceptActionUsageValidateParameters

An `AcceptUsageAction` must have at least two input parameters, corresponding to its `payload` and `receiver`, respectively (even if they have no `FeatureValue`). (Note that the `payloadParameter` is an input as well as an output.)

`inputParameters->size() >= 2`

acceptActionUsageDerivePayloadParameter

The `payloadParameter` of an `AcceptActionUsage` is its first parameter.

```
payloadParameter =
  if parameter->isEmpty() then null
  else parameter->at(1) endif
```

8.3.16.3 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 ActionDefinition, which define the actions that specify the behavior of the ActionDefinition.

Operations

No operations.

Constraints

None.

8.3.16.4 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 must subset, directly or indirectly, the base ActionUsage *actions* from the Systems model library. If it is a feature of an ActionDefinition or ActionUsage, then it must subset, directly or indirectly, the ActionUsage *Action::subactions*.

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

argument(i : Integer) : Expression [0..1]

Return the i-th argument Expression of an ActionUsage, defined as the value Expression of the FeatureValue of the i-th owned parameter of the ActionUsage. Return null if the ActionUsage has less than i owned parameters or the i-th owned parameter has no FeatureValue.

```
body: let ownedInputParameters : Sequence(Feature) = input->select(owner = self) in
if ownedInputParameters->size() < i then null
else
    let featureValue : Sequence(FeatureValue) = ownedInputParameters->at(i).
        ownedMembership->select(oclIsKindOf(FeatureValue)) in
        if featureValue->isEmpty() then null
        else featureValue->at(1).value
        endif
endif
```

inputParameters() : Feature [0..*]

Return the owned input parameters of this ActionUsage.

```
body: input->select(f | f.owner = self)
```

Constraints

None.

8.3.16.5 AssignmentActionUsage

Description

An AssignmentActionUsage is an ActionUsage that is typed, directly or indirectly, by the ActionDefinition *AssignmentAction* from the Systems model library. It specifies that the value of the `referent` Feature, relative to the target given by the result of the `targetArgument` Expression, should be set to the result of the `valueExpression`.

General Classes

ActionUsage

Attributes

/referent : Feature {subsets member}

The Feature whose value is to be set, derived as an unowned `member` of the AssignmentActionUsage. It shall not be a FeatureChain. It is redefined by the `target::accessedFeature` of the AssignmentActionUsage.

/targetArgument : Expression

The Expression whose value is an occurrence in the domain of the `referent` Feature, for which the value of the `referent` will be set to the result of the `valueExpression` by this AssignmentActionUsage. Derived as the `value` of the FeatureValue of the redefined `target` parameter of the AssignmentActionUsage.

/valueExpression : Expression

The Expression whose result is to be assigned to the `referent` Feature. Derived as the `value` of the FeatureValue of the redefined `replacementValues` parameter of the AssignmentActionUsage.

Operations

No operations.

Constraints

None.

8.3.16.6 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 is 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

None.

Operations

No operations.

Constraints

None.

8.3.16.7 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

None.

Operations

No operations.

Constraints

decisionNodeIncomingSuccession

A DecisionNode may have at most one incoming Succession Connector.

8.3.16.8 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

None.

Operations

No operations.

Constraints

forkNodeIncomingSuccession

A ForkNode may have at most one incoming Succession Connector.

8.3.16.9 ForLoopActionUsage

Description

A ForLoopActionUsage is a LoopActionUsage that is typed, directly or indirectly, by the ActionDefinition *ForLoopAction* from the Systems model library. It specifies that the *bodyClause* ActionUsage should be performed once for each value, in order, from the sequence of values obtained as the result of the *seqArgument* Expression. The *bodyAction* must have one input parameter, which receives a value from the sequence on each iteration.

General Classes

LoopActionUsage

Attributes

/loopVariable : ReferenceUsage

The *feature* of this ForLoopActionUsage that acts as the loop variable, which is assigned the successive elements of the input sequence on each iteration. Derived as the *feature* that subsets the library ReferenceUsage *ForLoopAction::var*.

/seqArgument : Expression

The Expression whose result provides the sequence of values to be passed to each *bodyAction* performance. Derived as the *value* Expression of the FeatureValue for the redefined *seq* parameter of the ForLoopActionUsage.

Operations

No operations.

Constraints

None.

8.3.16.10 IfActionUsage

Description

An IfActionUsage is an ActionUsage that is typed, directly or indirectly, by the ActionDefinition *IfThenAction* from the Systems model library, or, more specifically, by *IfThenElseAction*, if it has an *elseAction*. It specifies that the *thenAction* ActionUsage should be performed if the result of the *ifArgument* Expression is true. It may also optionally specify a *elseAction* ActionUsage that is performed if the result of the *ifArgument* is false.

General Classes

ActionUsage

Attributes

/elseAction : ActionUsage [0..1]

The ActionUsage that is to be performed if the result of the *ifArgument* is false. Derived as the owned ActionUsage that redefines *elseClause* parameter of the IfActionUsage.

/ifArgument : Expression

The Expression whose result determines whether the *thenAction* or (optionally) the *elseAction* is performed. Derived as the *value* Expression of the FeatureValue for the redefined *ifTest* parameter of the IfActionUsage.

/thenAction : ActionUsage

The ActionUsage that is to be performed if the result of the *ifArgument* is true. Derived as the owned ActionUsage that redefines the *thenClause* parameter of the IfActionUsage.

Operations

No operations.

Constraints

None.

8.3.16.11 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

None.

Operations

No operations.

Constraints

joinNodeOutgoingSuccession

A JoinNode may have at most one outgoing Succession Connector.

8.3.16.12 LoopActionUsage

Description

A LoopActionUsage is an ActionUsage that is typed, directly or indirectly, by the ActionDefinition *LoopAction* from the Systems model library. It specifies that its *bodyAction* should be performed repeatedly. Its subclasses WhileLoopActionUsage and ForLoopActionUsage provide different ways to determine how many times the *bodyAction* should be performed.

General Classes

ActionUsage

Attributes

/bodyAction : ActionUsage

The ActionUsage to be performed repeatedly by the LoopActionUsage. Derived as the owned ActionUsage that redefines the *body* parameter of the LoopActionUsage.

Operations

No operations.

Constraints

None.

8.3.16.13 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

None.

Operations

No operations.

Constraints

mergeNodeOutgoingSuccession

A MergeNode may have at most one outgoing Succession Connector.

8.3.16.14 PerformActionUsage

Description

A PerformActionUsage is an ActionUsage that represents the performance of an ActionUsage. Unless it is the PerformActionUsage itself, the ActionUsage to be performed is related to the PerformActionUsage by a ReferenceSubsetting relationship. A PerformActionUsage is also an EventOccurrenceUsage, with its *performedAction* as the *eventOccurrence*.

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 *eventOccurrence* of the PerformActionUsage considered as an EventOccurrenceUsage, which must be an ActionUsage.

Operations

namingFeature() : Feature [0..1]

The naming Feature of an PerformActionUsage is its *performedAction*.

body: exhibitedState

Constraints

None.

8.3.16.15 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 payloadArgument Expression via a Transfer that from whose `source` is given by the result of the `senderArgument` Expression and whose `target` is given by the result of the `receiverArgument`. At least one of `senderArgument` and `receiverArgument` must be provided. If no `senderArgument` is provided, the default is the `this` context for the action. If no `receiverArgument` is given, then the receiver is to be determined from outgoing connections from the sender.

General Classes

ActionUsage

Attributes

/payloadArgument : Expression

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

/receiverArgument : Expression [0..1]

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

/senderArgument : Expression [0..1]

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

Operations

No operations.

Constraints

sendActionUsageValidateArguments

At least one of the `senderArgument` and `receiverArgument` of a SendActionUsage must be non-null.

`senderArgument <> null` or `receiverArgument <> null`

sendActionUsageDeriveReceiverArgument

The `receiverArgument` of a SendActionUsage is its third argument Expression.

`receiverArgument = argument(3)`

sendActionUsageDerivePayloadArgument

The `payloadArgument` of a SendActionUsage is its first argument Expression.

`payloadArgument = argument(1)`

sendActionValidateParameter

A SendActionUsage must have at least three owned input parameters, corresponding to its *payload*, *sender* and *receiver*, respectively (whether or not they have FeatureValues).

```
inputParameters->size() >= 3  
sendActionUsageDeriveSenderArgument
```

The *senderArgument* of a SendActionUsage is its second argument Expression.

```
senderArgument = argument(2)
```

8.3.16.16 TriggerInvocationExpression

Description

A TriggerInvocationExpression is an InvocationExpression that invokes one of the trigger Functions from the Kernel *Triggers package*, as indicated by its *kind*.

General Classes

InvocationExpression

Attributes

kind : TriggerKind

Indicates which of the Functions from the Kernel *Triggers package* is to be invoked by this TriggerInvocationExpression.

Operations

No operations.

Constraints

None.

8.3.16.17 TriggerKind

Description

TriggerKind enumerates the kinds of triggers that can be represented by TriggerInvocationExpression.

General Classes

None.

Literal Values

after

Indicates a relative time trigger, corresponding to the *TriggerAfter* Function from the *Triggers library model*.

at

Indicates an absolute time trigger, corresponding to the *TriggerAt* Function from the *Triggers* library model.
when

Indicates a change trigger, corresponding to the *TriggerWhen* Function from the *Triggers* library model.

8.3.16.18 WhileLoopsActionusage

Description

A WhileLoopActionUsage is a LoopActionUsage that is typed, directly or indirectly, by the ActionDefinition *WhileLoopAction* from the Systems model library. It specifies that the *bodyClause* ActionUsage should be performed repeatedly while the result of the *whileArgument* Expression is true or until the result of the *untilArgument* Expression (if provided) is true. The *whileArgument* Expression is evaluated before each (possible) performance of the *bodyClause*, and the *untilArgument* Expression is evaluated after each performance of the *bodyClause*.

General Classes

LoopActionUsage

Attributes

/untilArgument : Expression [0..1]

The Expression whose result, if false, determines that the *bodyAction* should continue to be performed. Derived as the owned Expression that redefines the *untilTest* parameter of the WhileLoopActionUsage.

/whileArgument : Expression

The Expression whose result, if true, determines that the *bodyAction* should continue to be performed. Derived as the owned Expression that redefines the *whileTest* parameter of the WhileLoopActionUsage.

Operations

No operations.

Constraints

None.

8.3.17 States Abstract Syntax

8.3.17.1 Overview

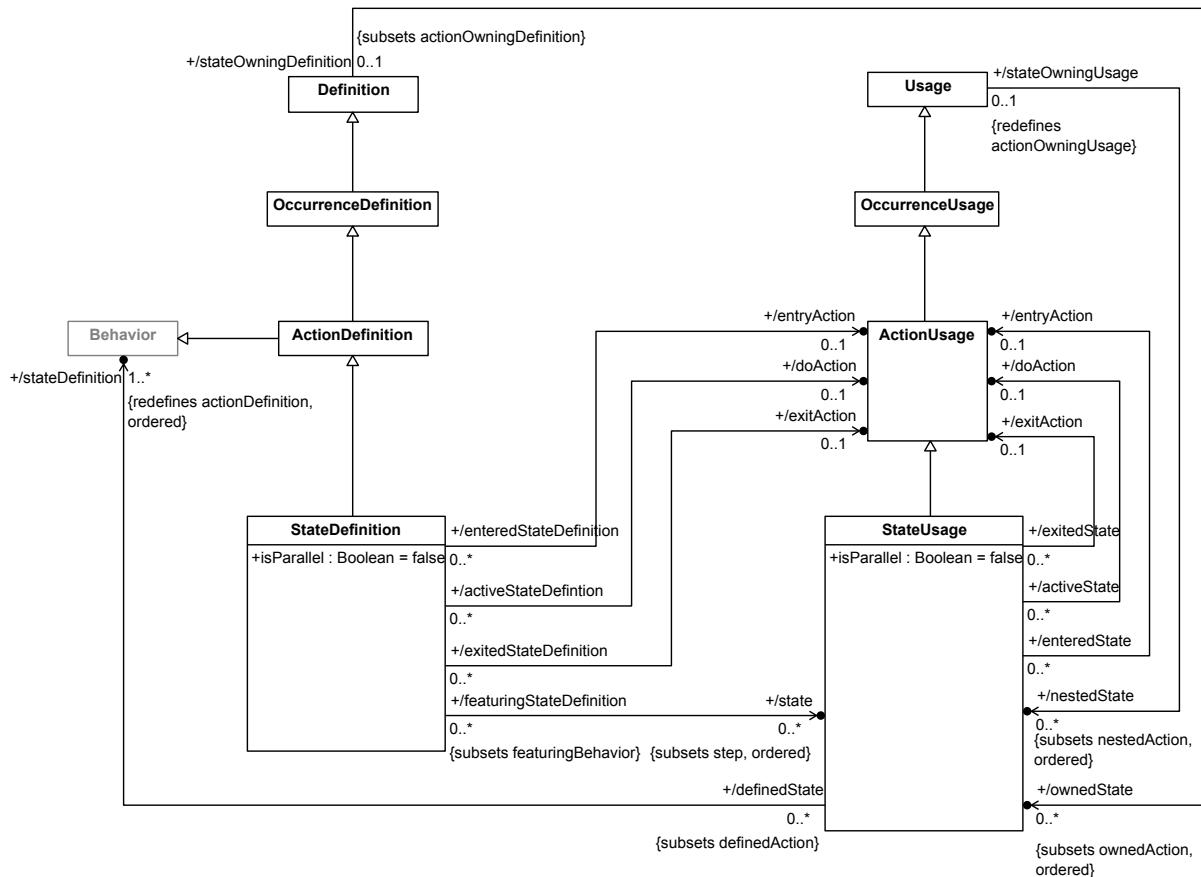


Figure 82. State Definition and Usage

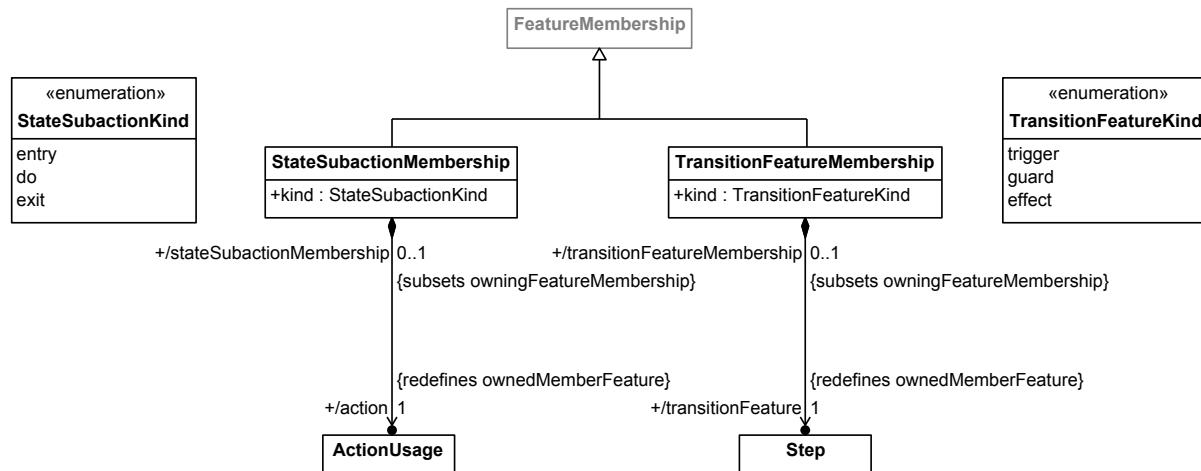


Figure 83. State Membership

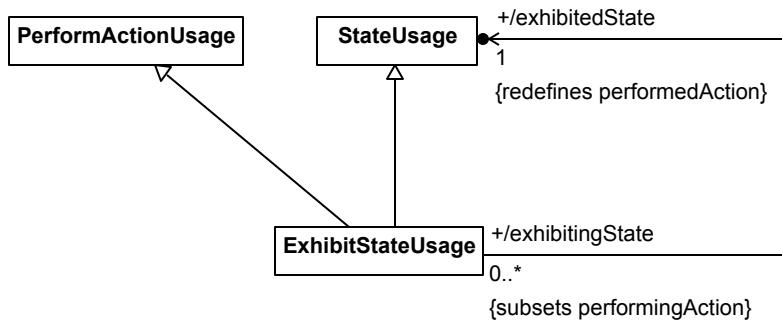


Figure 84. State Exhibition

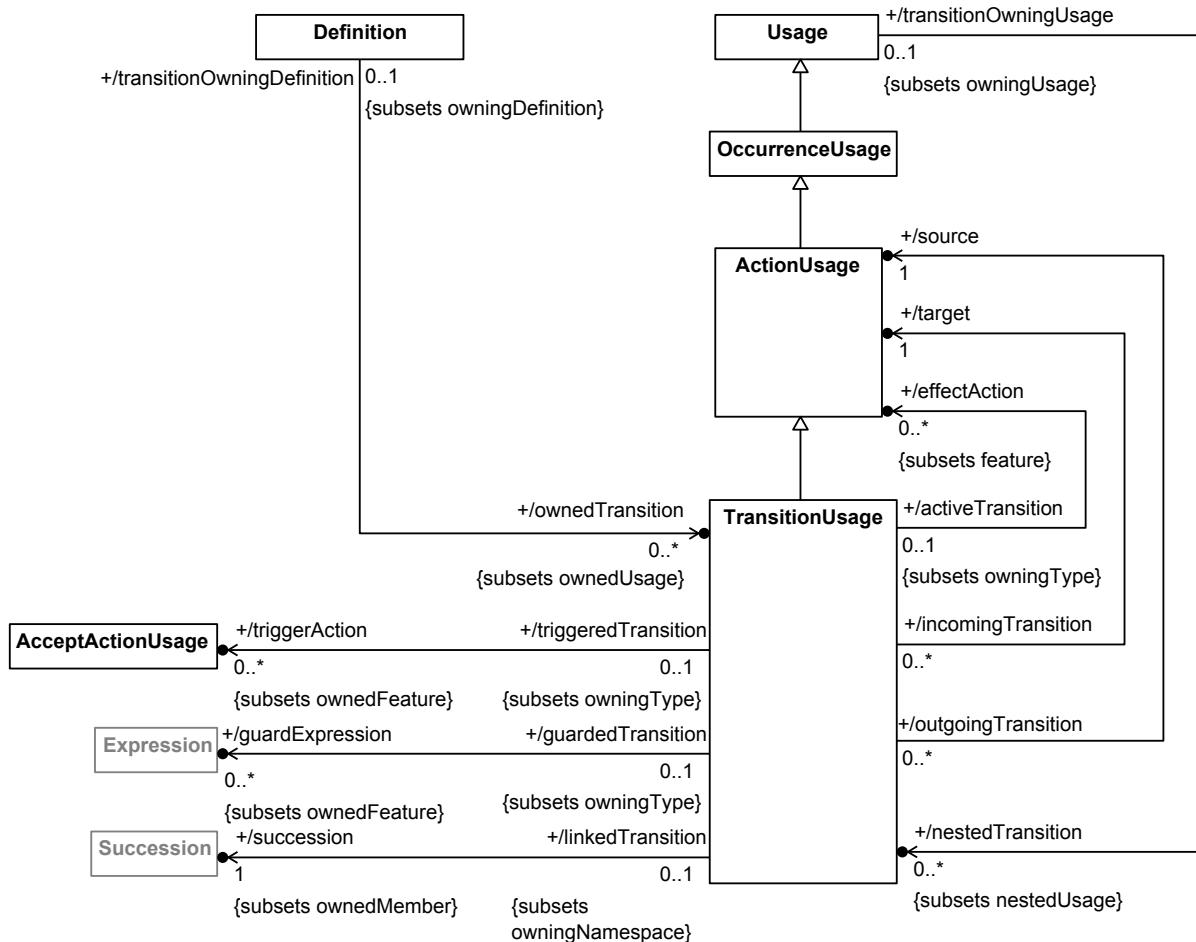


Figure 85. Transition Usage

8.3.17.2 ExhibitStateUsage

Description

An **ExhibitStateUsage** is a **StateUsage** that represents the exhibiting of a **StateUsage**. Unless it is the **StateUsage** itself, the **StateUsage** to be exhibited is related to the **ExhibitStateUsage** by a **ReferenceSubsetting** 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 *performedAction* of the ExhibitStateUsage considered as an PerformActionUsage, which must be an StateUsage.

Operations

No operations.

Constraints

None.

8.3.17.3 StateSubactionKind

Description

A StateSubactionKind indicates whether the *action* of a StateSubactionMembership is an entry, do or exit action.

General Classes

None.

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.

8.3.17.4 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

None.

8.3.17.5 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)
```

8.3.17.6 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 actionDefinition, 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)
```

8.3.17.7 TransitionFeatureKind

Description

A TransitionActionKind indicates whether the transitionFeature of a TransitionFeatureMembership is a trigger, guard or effect.

General Classes

None.

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.

8.3.17.8 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

None.

8.3.17.9 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`. All `triggerActions` must be directly typed by `AcceptMessageAction` from the Systems model library.

Operations

No operations.

Constraints

None.

8.3.18 Calculations Abstract Syntax

8.3.18.1 Overview

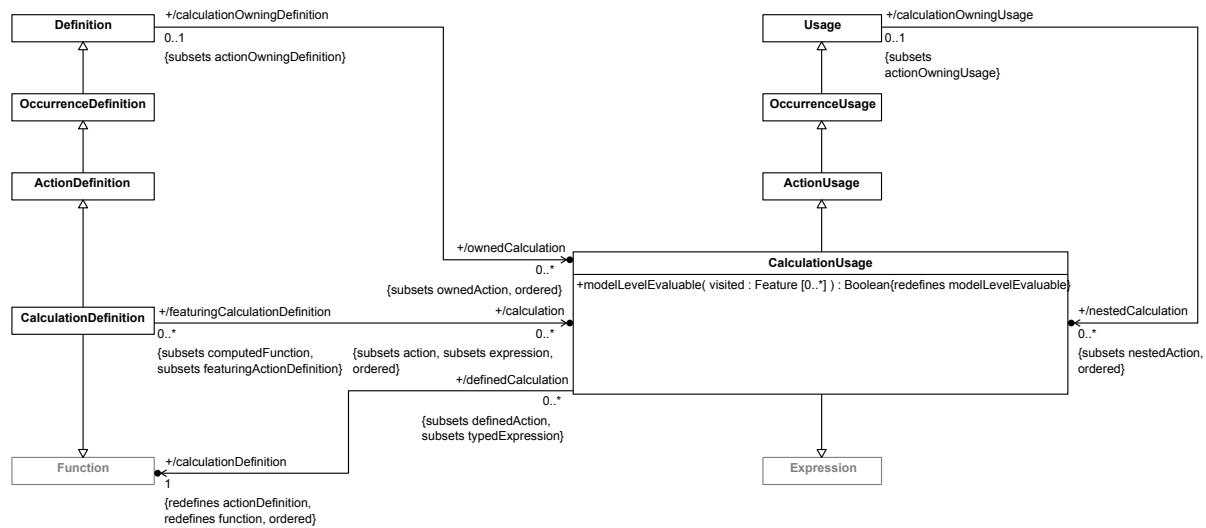


Figure 86. Calculation Definition and Usage

8.3.18.2 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

None.

8.3.18.3 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

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

A CalculationUsage is not model-level evaluable.

body: false

Constraints

None.

8.3.19 Constraints Abstract Syntax

8.3.19.1 Overview

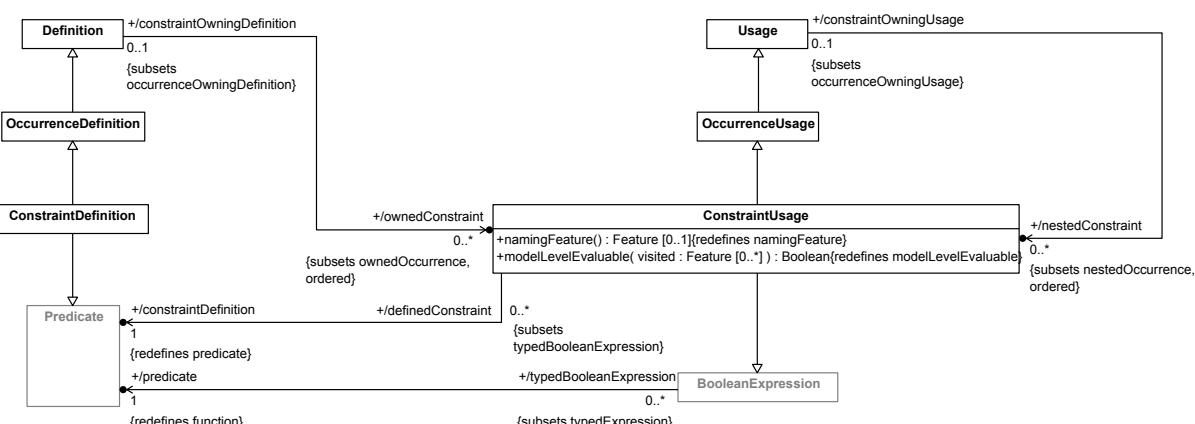


Figure 87. Constraint Definition and Usage

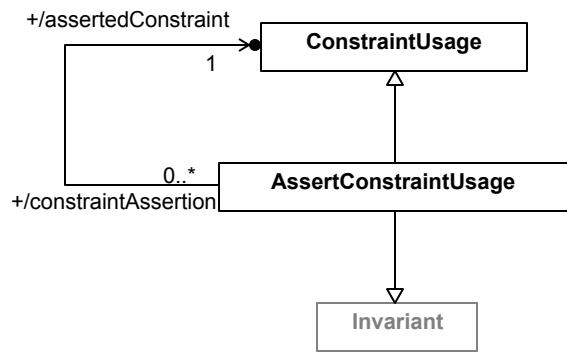


Figure 88. Constraint Assertion

8.3.19.2 AssertConstraintUsage

Description

An **AssertConstraintUsage** is a **ConstraintUsage** that is also an **Invariant** and, so, is asserted to be true (by default). Unless it is the **AssertConstraintUsage** itself, the asserted **ConstraintUsage** is related to the **AssertConstraintUsage** by a **ReferenceSubsetting** relationship.

If the **AssertConstraintUsage** is owned by a **PartDefinition** or **PartUsage**, then it also subsets the `assertedConstraints` feature of the **PartDefinition** *Part* from the System Library model *Parts*.

General Classes

Invariant
ConstraintUsage

Attributes

`/assertedConstraint : ConstraintUsage`

The **ConstraintUsage** to be performed by the **AssertConstraintUsage**. It is the `referenceFeature` of the `ownedReferenceSubsetting` for the **AssertConstraintUsage**, if there is one, and, otherwise, the **AssertConstraintUsage** itself.

Operations

No operations.

Constraints

`assertConstraintUsageAssertedConstraint`

If an **AssertConstraintUsage** has no `ownedReferenceSubsetting`, then its `assertedConstraint` is the **AssertConstraintUsage** itself. Otherwise, the `assertedConstraint` is the `referenceFeature` of the `ownedReferenceSubsetting`, which must be a **ConstraintUsage**.

```

assertedConstraint =
    if ownedReferenceSubsetting = null then self
    else ownedReferenceSubsetting.referencedFeature.oclAsType(ConstraintUsage)
    endif

```

8.3.19.3 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

None.

Operations

No operations.

Constraints

None.

8.3.19.4 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

modelLevelEvaluable(visited : Feature [0..*]) : Boolean

A ConstraintUsage is not model-level evaluable.

body: false

namingFeature() : Feature [0..1]

If this ConstraintUsage is an assumedConstraint or requiredConstraint of a RequirementUsage, then its naming Feature is the target of its ownedReferenceSubsetting, if it has one.

Constraints

None.

8.3.20 Requirements Abstract Syntax

8.3.20.1 Overview

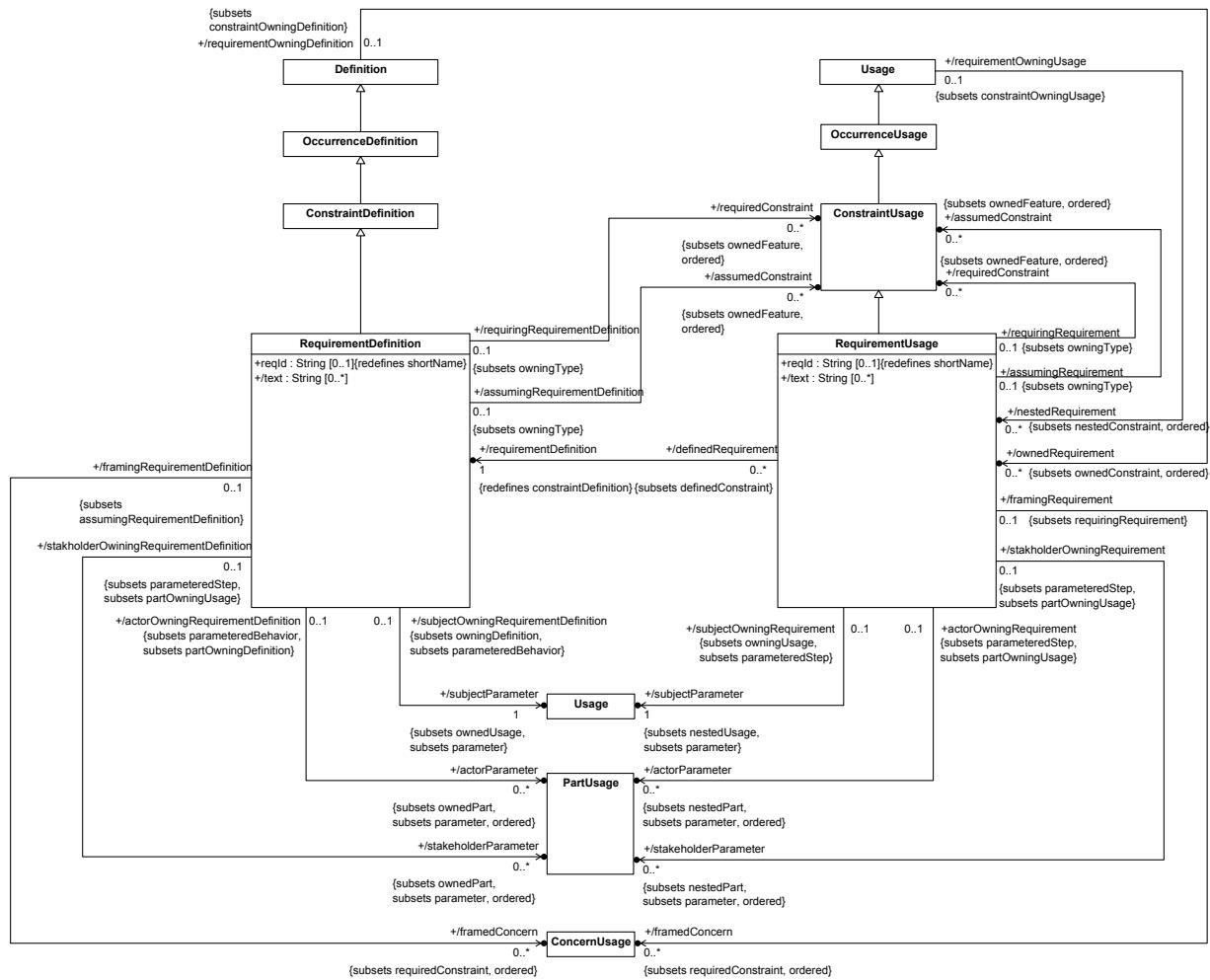


Figure 89. Requirement Definition and Usage

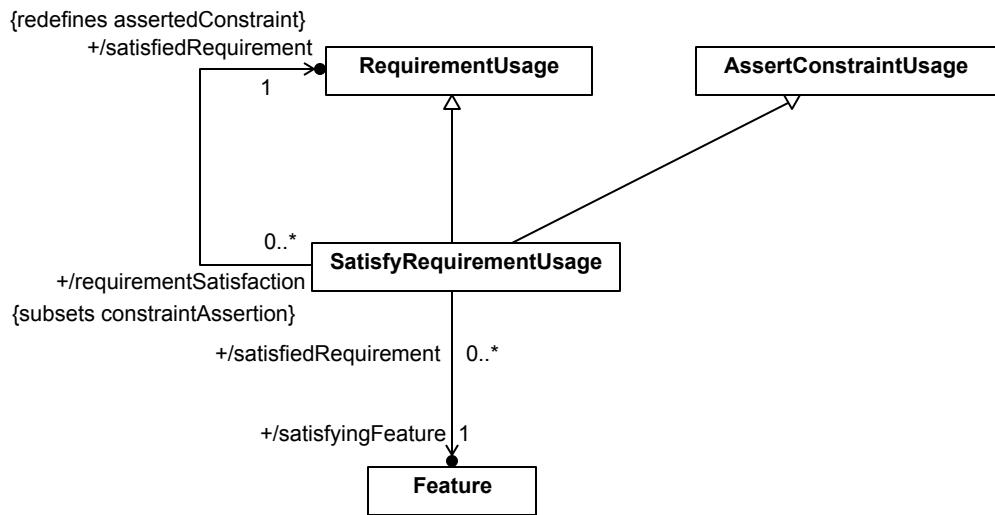


Figure 90. Requirement Satisfaction

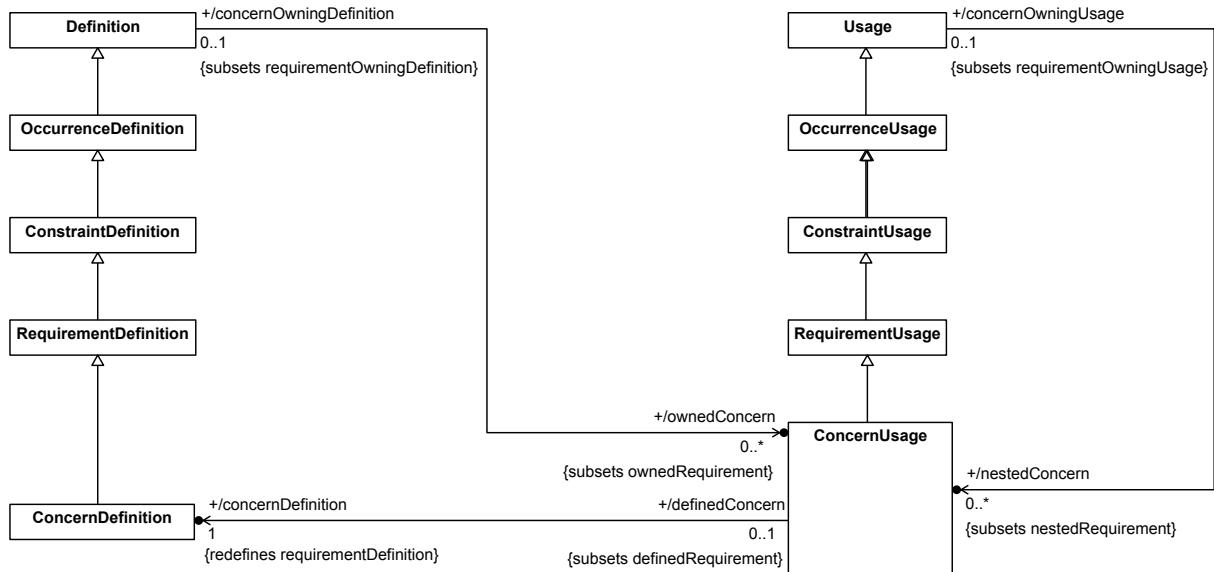


Figure 91. Concern Definition and Usage

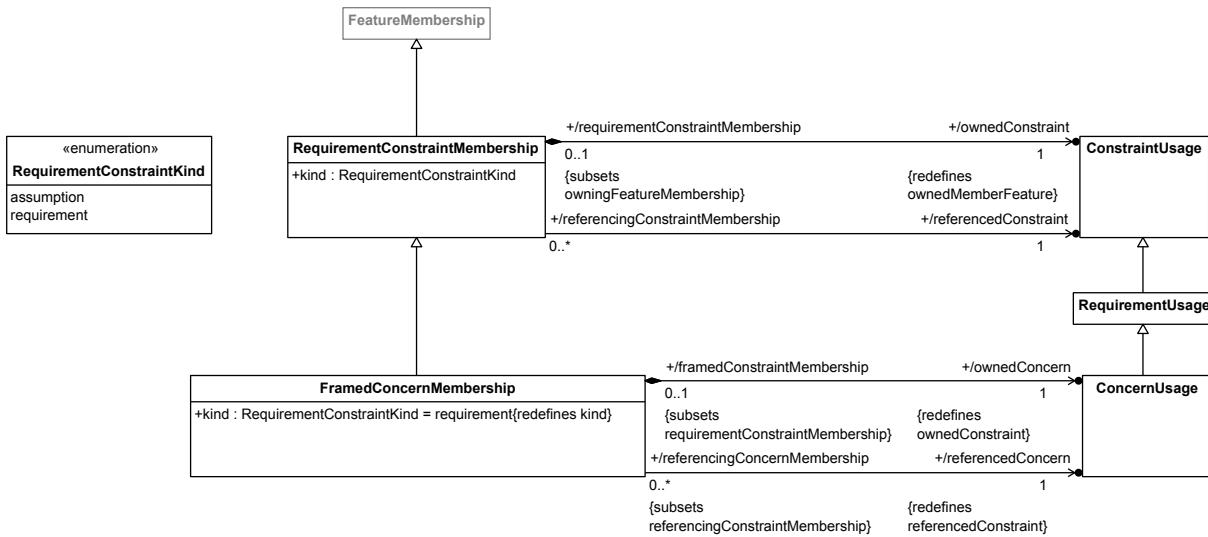


Figure 92. Requirement Constraint Membership

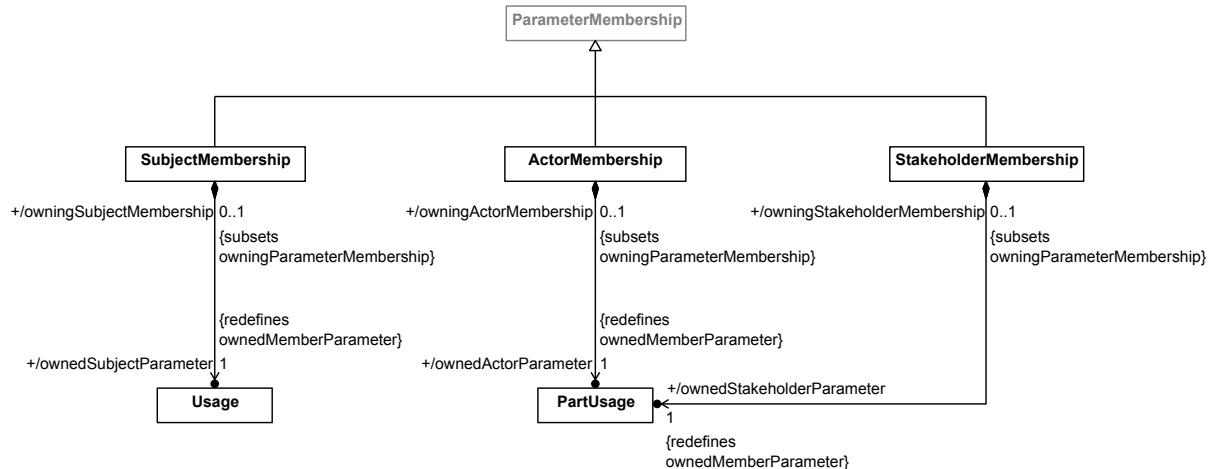


Figure 93. Requirement Parameter Memberships

8.3.20.2 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 parameterized element.

General Classes

ParameterMembership

Attributes

/ownedActorParameter : **PartUsage** {redefines ownedMemberParameter}

The **PartUsage** specifying the actor.

Operations

No operations.

Constraints

None.

8.3.20.3 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

None.

Operations

No operations.

Constraints

None.

8.3.20.4 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

None.

8.3.20.5 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 {redefines kind}`

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. It is the `referencedConstraint` of the FramedConcernMembership considered as a RequirementConstraintMembership, which must be a ConcernUsage.

Operations

No operations.

Constraints

None.

8.3.20.6 RequirementConstraintKind

Description

A RequirementConstraintKind indicates whether a ConstraintUsage is an assumption or a requirement in a RequirementDefinition or RequirementUsage.

General Classes

None.

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.

8.3.20.7 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 `referencedFeature` of the `ownedReferenceSubsetting` of the `ownedConstraint`, if there is one, and, otherwise, the `ownedConstraint` itself.

Operations

No operations.

Constraints

None.

8.3.20.8 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 shortName}

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 *modelerId* 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

None.

8.3.20.9 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 shortName}

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

None.

8.3.20.10 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 assertedConstraint of the SatisfyRequirementUsage considered as an AssertConstraintUsage, which must be a RequirementUsage.

/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

None.

8.3.20.11 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

None.

8.3.20.12 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 {redefines ownedMemberParameter}`

The PartUsage specifying the stakeholder.

Operations

No operations.

Constraints

None.

8.3.21 Cases Abstract Syntax

8.3.21.1 Overview

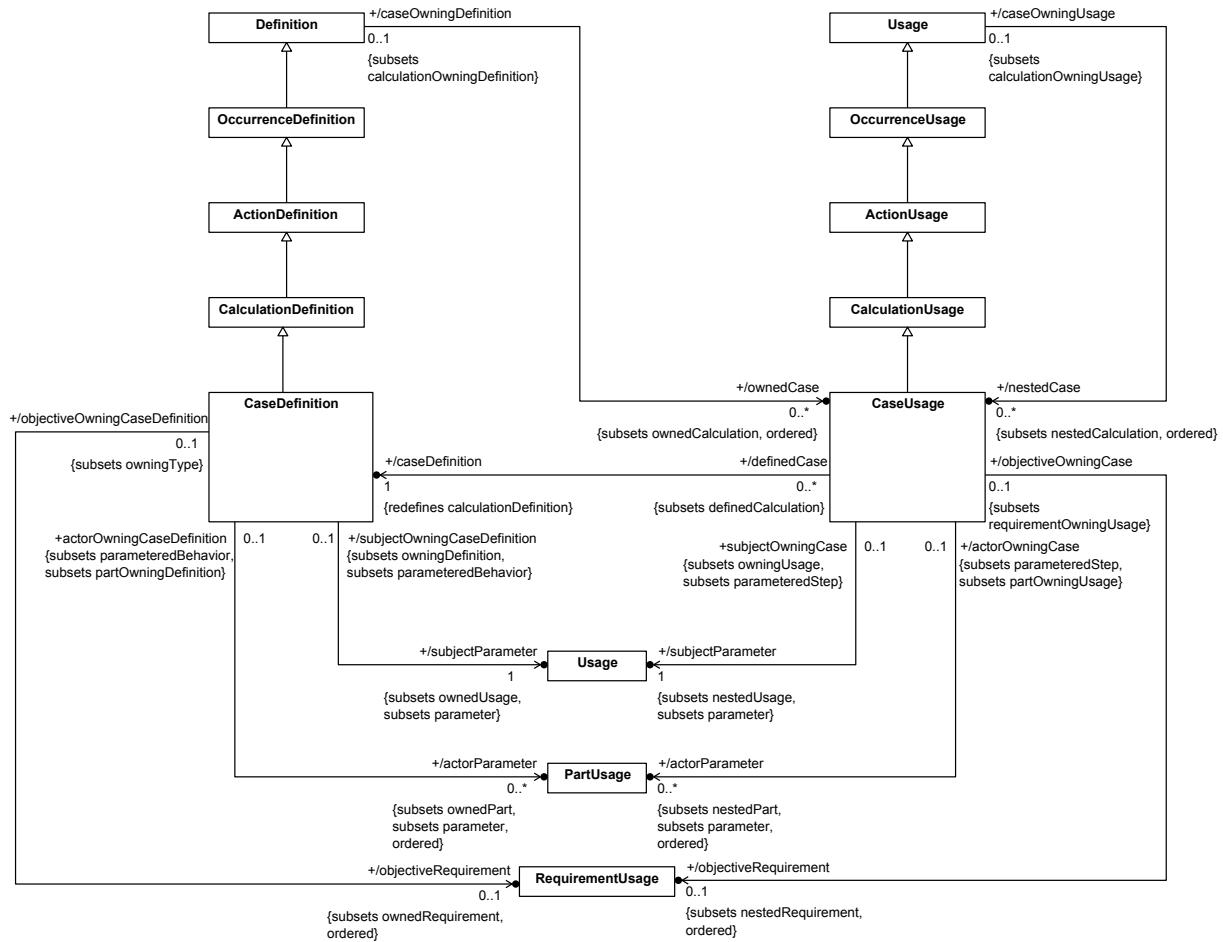


Figure 94. Case Definition and Usage

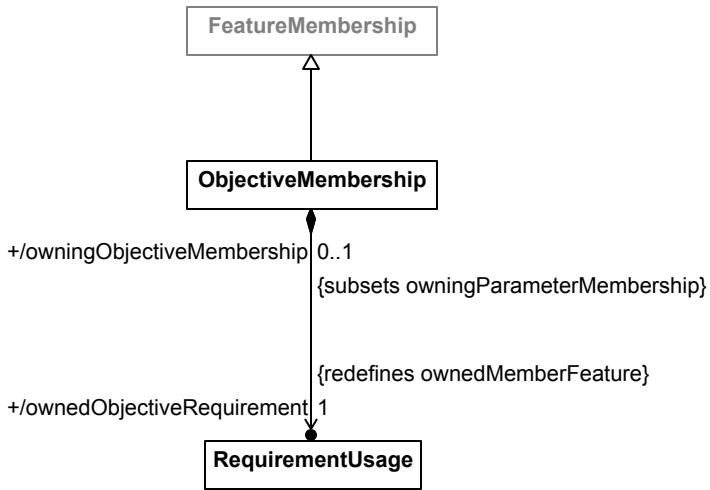


Figure 95. Case Membership

8.3.21.2 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

None.

8.3.21.3 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

None.

8.3.21.4 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

None.

8.3.22 Analysis Cases Abstract Syntax

8.3.22.1 Overview

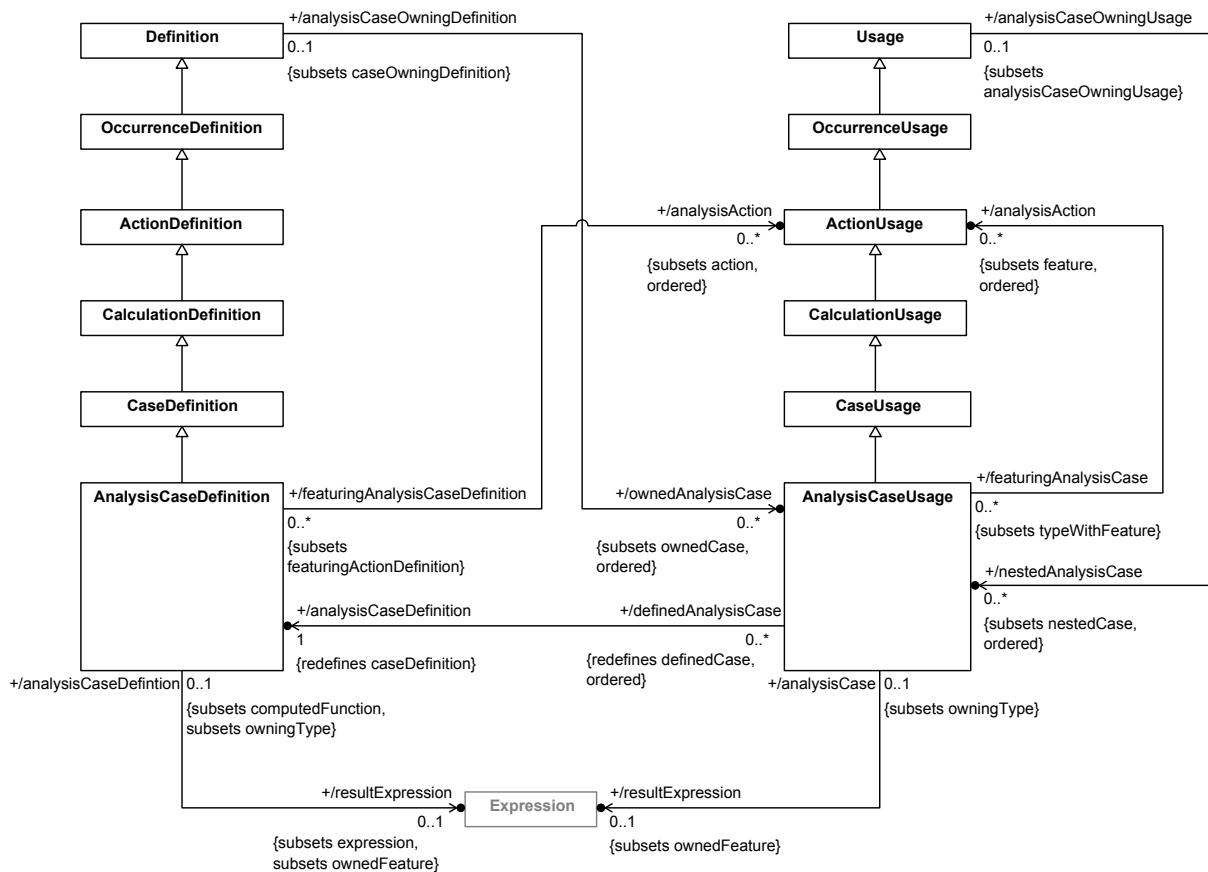


Figure 96. Analysis Case Definition and Usage

8.3.22.2 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

None.

8.3.22.3 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

None.

8.3.23 Verification Cases Abstract Syntax

8.3.23.1 Overview

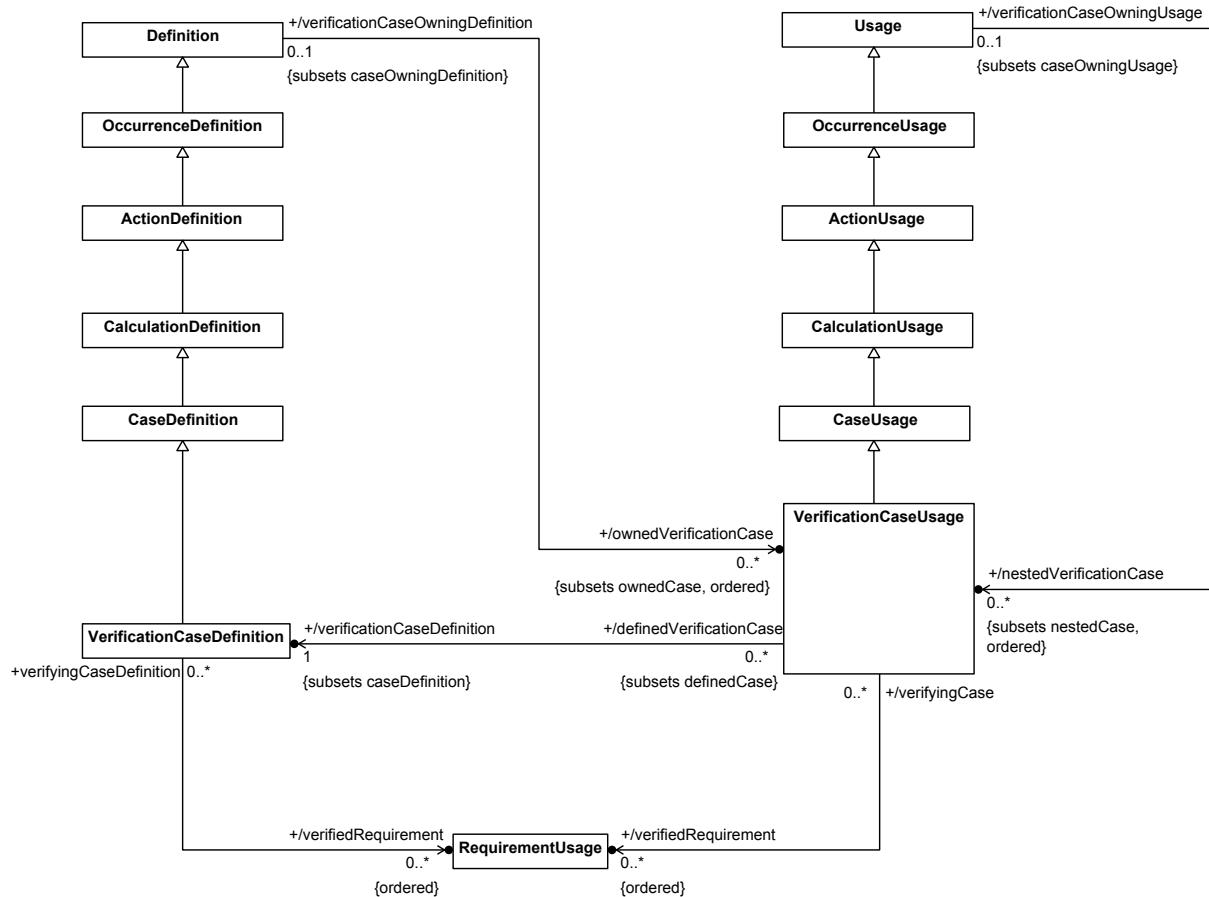


Figure 97. Verification Case Definition and Usage

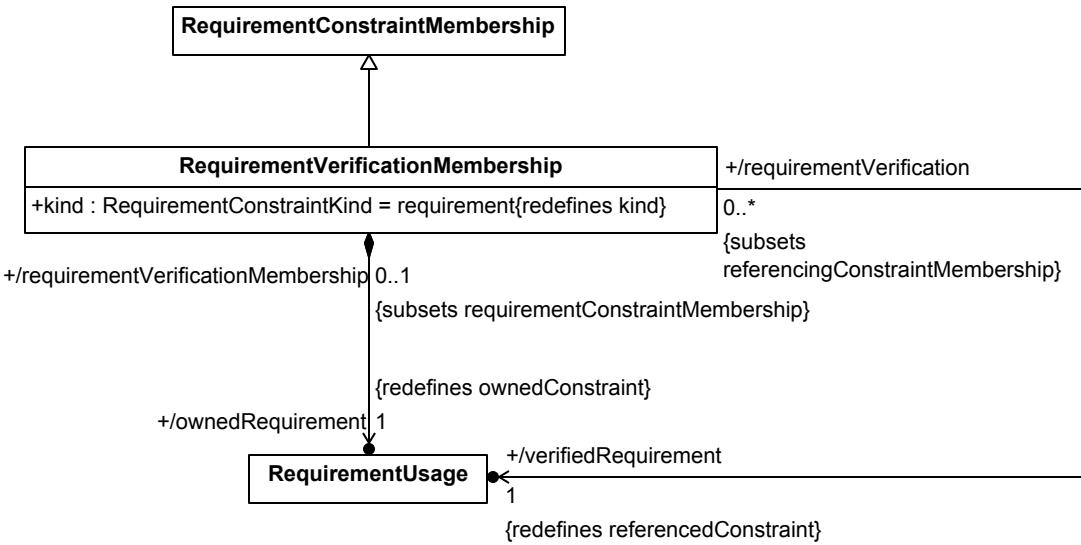


Figure 98. Verification Membership

8.3.23.2 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 {redefines kind}`

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. It is the `referencedConstraint` of the RequirementVerificationMembership considered as a RequirementConstraintMembership, which must be a RequirementUsage.

Operations

No operations.

Constraints

None.

8.3.23.3 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

None.

8.3.23.4 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

None.

8.3.24 Use Cases Abstract Syntax

8.3.24.1 Overview

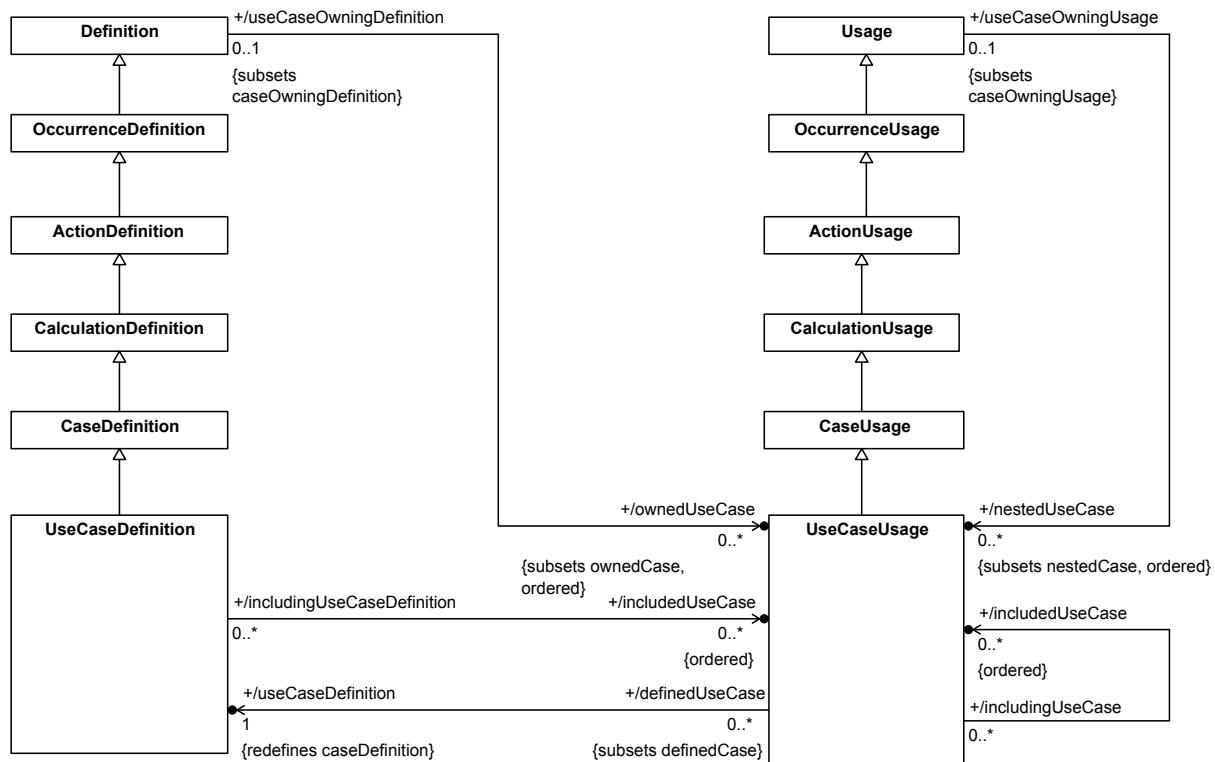


Figure 99. Use Case Definition and Usage

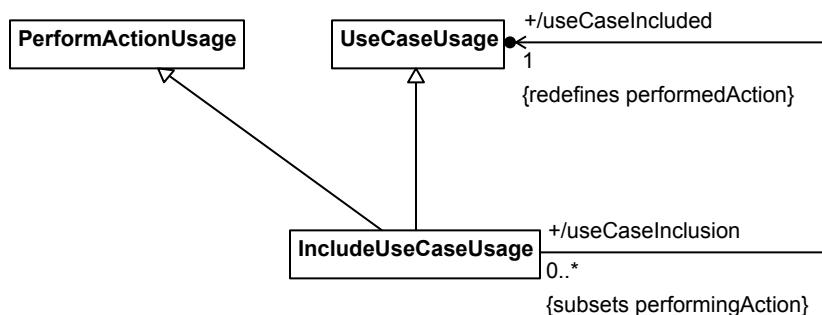


Figure 100. Use Case Inclusion

8.3.24.2 IncludeUseCaseUsage

Description

An `IncludeUseCaseUsage` is a `UseCaseUsage` that represents the inclusion of a `UseCaseUsage` by a `UseCaseDefinition` or `UseCaseUsage`. Unless it is the `IncludeUseCaseUsage` itself, the `UseCaseUsage` to be included is related to the `includedUseCase` by a `ReferenceSubsetting` 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 `subsettetedFeature` of the first owned `Subsetting` Relationship of the `IncludeUseCaseUsage`.

Operations

No operations.

Constraints

None.

8.3.24.3 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

None.

8.3.24.4 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

None.

8.3.25 Views Abstract Syntax

8.3.25.1 Overview

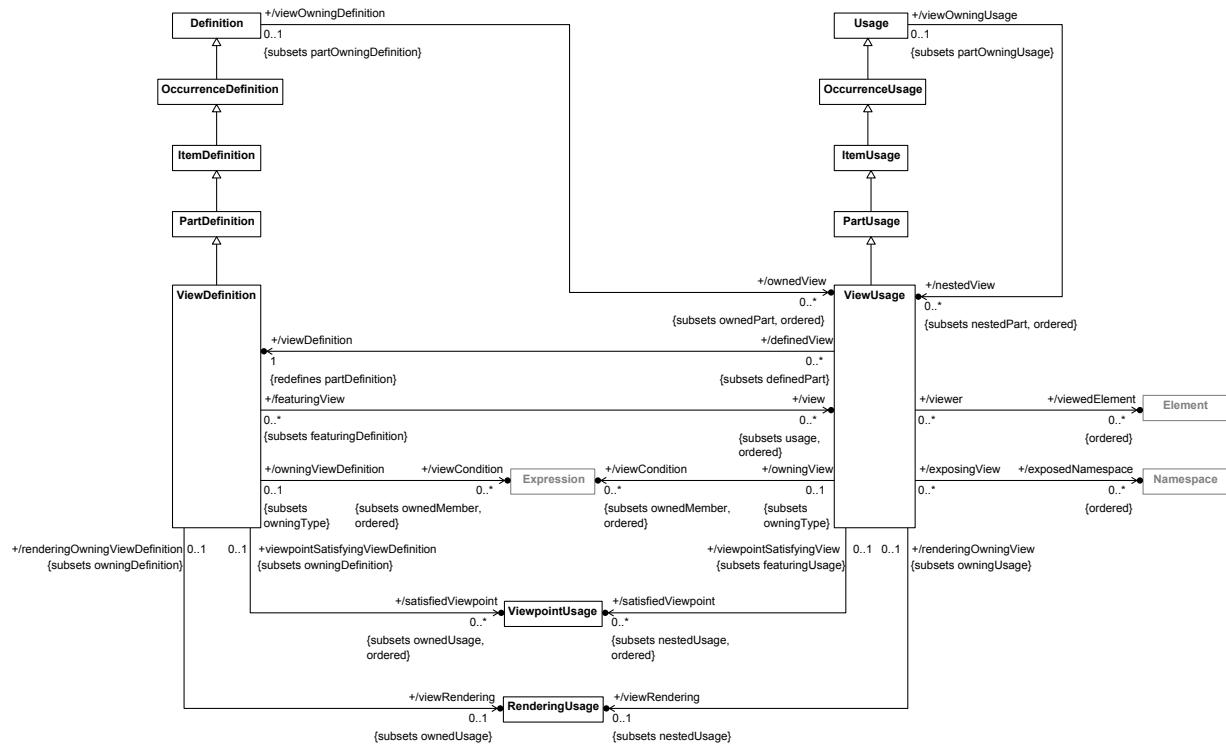


Figure 101. View Definition and Usage

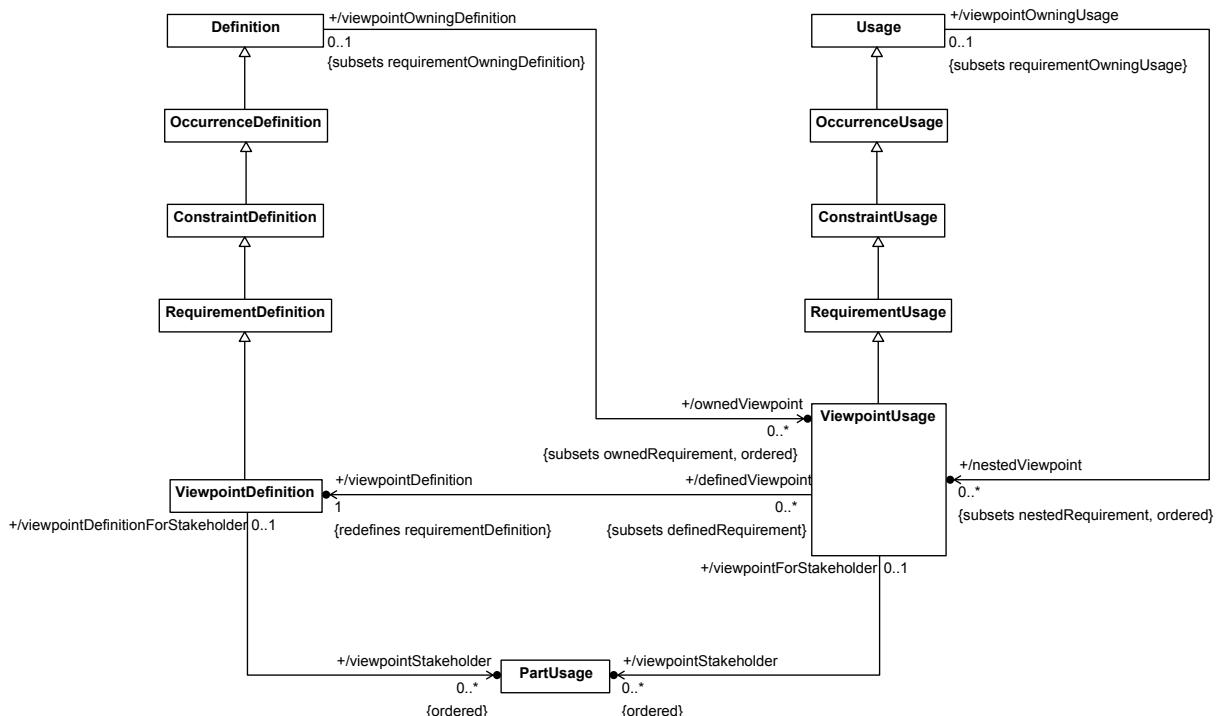


Figure 102. Viewpoint Definition and Usage

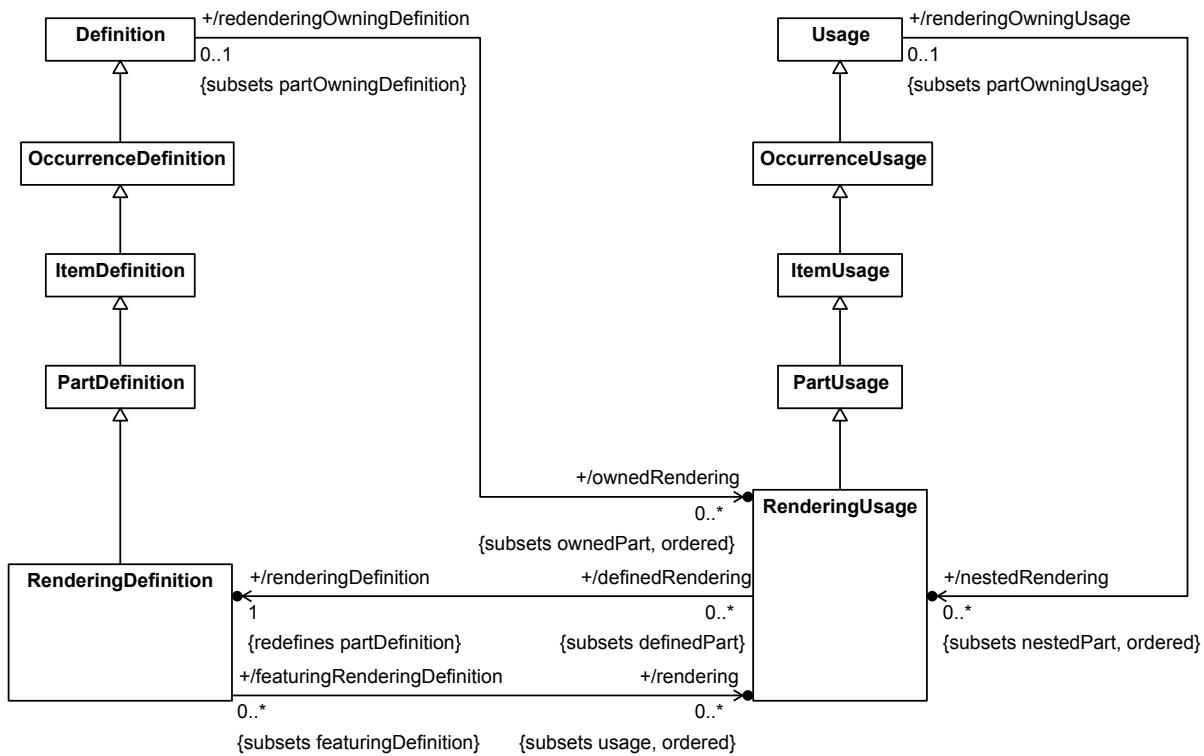


Figure 103. Rendering Definition and Usage

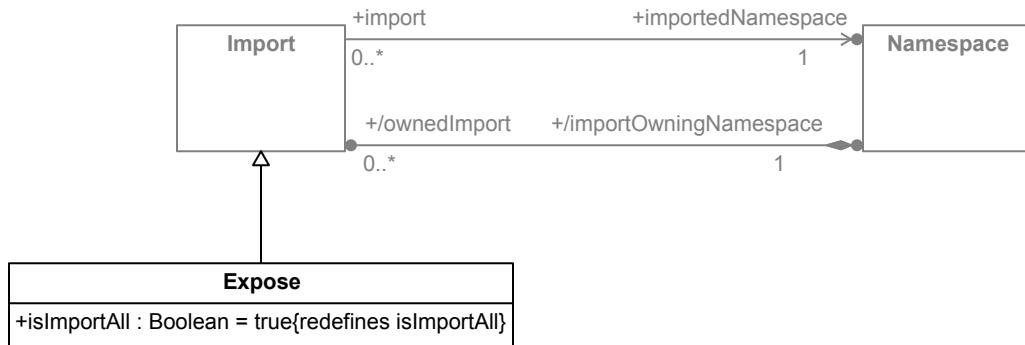


Figure 104. Expose Relationship

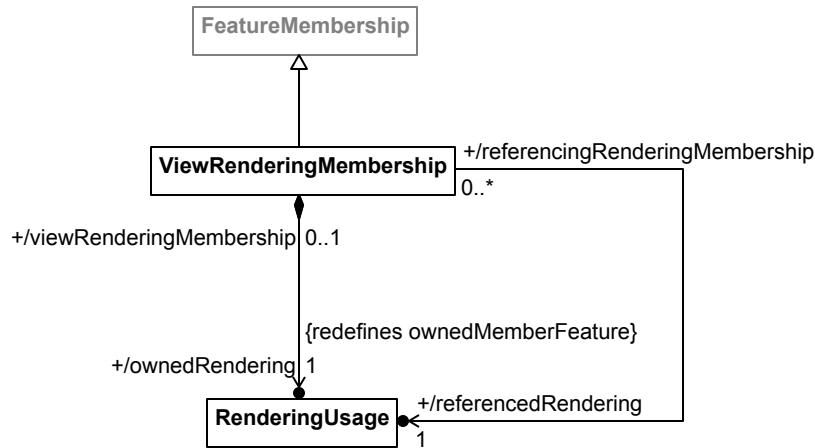


Figure 105. View Rendering Membership

8.3.25.2 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`

8.3.25.3 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

None.

8.3.25.4 `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

None.

8.3.25.5 `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

None.

8.3.25.6 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

None.

8.3.25.7 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

None.

8.3.25.8 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. It is the `referenceFeature` of the `ownedReferenceSubsetting` for the `ownedRendering`, if there is one, and, otherwise, the `ownedRendering` itself.

Operations

No operations.

Constraints

None.

8.3.25.9 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

None.

8.3.26 Metadata Abstract Syntax

8.3.26.1 Overview

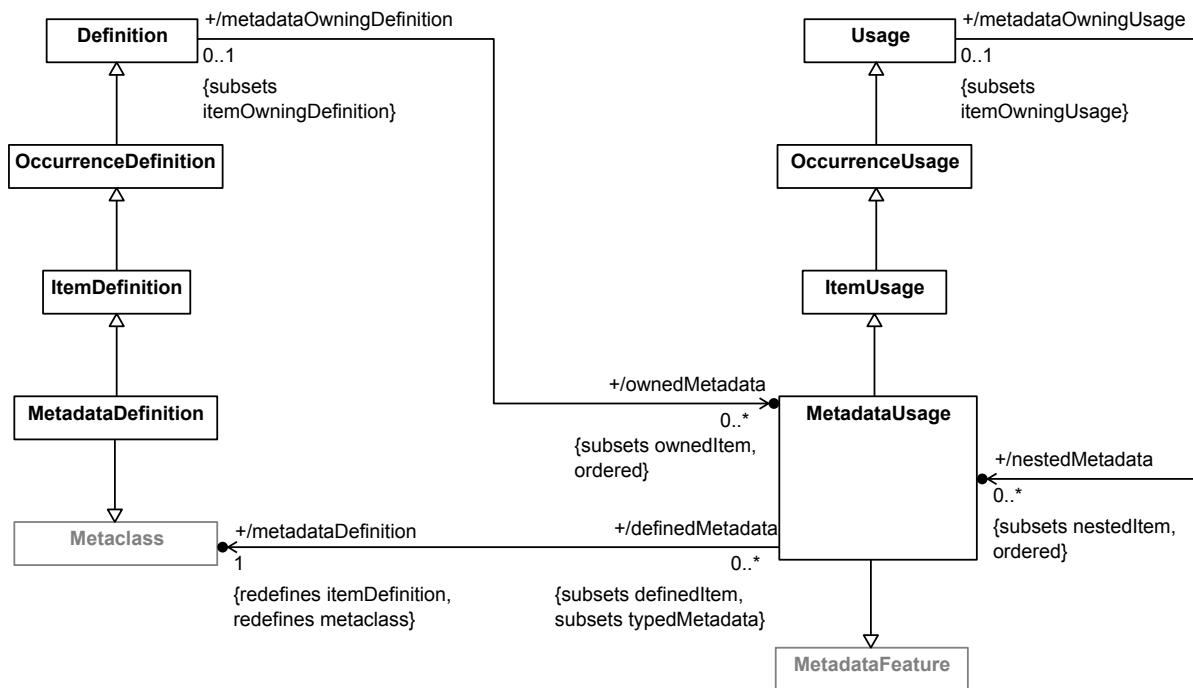


Figure 106. Metadata Definition and Usage

8.3.26.2 MetadataDefinition

Description

A MetadataDefinition is an ItemDefinition that is also a Metaclass.

A MetadataDefinition must subclassify, directly or indirectly, the base MetadataDefinition MetadataItem from the Systems model library.

General Classes

MetaClass
ItemDefinition

Attributes

None.

Operations

No operations.

Constraints

None.

8.3.26.3 MetadataUsage

Description

An MetadataUsage is a Usage and a MetadataFeature, used to annotate other Elements in a system model with metadata. As a MetadataFeature, its type must be a MetaClass, which will nominally be a MetadataDefinition. However, any Kernel MetaClass is also allowed, to permit use of Metaclasses from the Kernel Library.

A MetadataUsage must subset, directly or indirectly, the base MetadataUsage `metadataItems` from the Systems model library.

General Classes

ItemUsage
MetadataFeature

Attributes

`/metadataDefinition : MetaClass {redefines itemDefinition, metaClass}`

Operations

No operations.

Constraints

None.

8.4 Semantics

8.4.1 Semantics Overview

Release 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**
- 8.4.22 Metadata 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 base 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 *Cause and Effect Domain Library* contains a language extension for modeling cause and effect relationships (see [9.5](#)).
- The *Requirement Derivation Domain Library* contains a language extension for modeling requirement derivation relationships (see [9.6](#)).
- The *Geometry Domain Library* contains a model for physical items with spacial extent, including an extensive set of basic geometric shapes that can be used to construct such items (see [9.7](#)).
- 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/IEC 80000 and ISO 8601-1 standards (see [9.8](#)).

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.

Release 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 Systems Model Library 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 [8.3.7](#)). These library model packages are all contained within a top-level package called *SystemsLibrary*.

Implementation Note. The pilot implementation does not currently include the containing *SystemsLibrary* package. Instead, all Systems Model Library packages are visible in the global namespace.

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 AttributeValue

Element

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 Types

DataValue

Features

None.

Constraints

None.

9.2.2.2.2 attributeValues

Element

AttributeUsage

Description

attributeValues is the base feature for all AttributeUsages.

General Types

AttributeValue
dataValues

Features

None.

Constraints

None.

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

Element

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 Types

Object

Features

boundingShapes : Item [0..*] {subsets envelopingShapes}

Enveloping shapes that are StructuredSpaceObjects with every `face` or every `edge` intersecting this Item.

checkedConstraints : ConstraintCheck [0..*] {subsets ownedPerformances}

Constraints that have been checked by this Item.

envelopingShapes : Item [0..*]

Shapes that are the `shape` of an Item that includes this Item in space and time.

isSolid : Boolean

An Item is solid if it has no `voids`.

shape : Item {redefines spaceBoundary}

Spatial boundary of this Item.

subitems : Item [0..*] {subsets suboccurrences}

The Items that are composite subitems of this Item.

subparts : Part [0..*] {subsets subitems}

The `subitems` of this item that are `parts`.

voids : Item [0..*] {redefines innerSpaceOccurrences}

Voids are the `innerSpaceOccurrences` of this Item.

Constraints

None.

9.2.3.2.2 items

Element

ItemUsage

Description

items is the base feature of all ItemUsages.

General Types

objects
Item

Features

None.

Constraints

None.

9.2.3.2.3 Touches

Element

ConnectionDefinition

Description

Touching Occurrences are JustOutsideOf each other and happen at the same time (HappensWhile).

General Types

HappensWhile
JustOutsideOf

Features

touchedItem : Item {redefines separateSpace, thatOccurrence}

touchedItemToo : Item {redefines thisOccurrence, separateSpaceToo}

touches : Item [0..*] {subsets justOutsideOf, happensWhile}

touchesToo : Item [0..*] {subsets happensWhile¹, justOutsideOfToo}

Constraints

None.

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

Element

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 Types

Item

Features

exhibitedStates : StateAction [0..*] {subsets performedActions}

StateActions that are exhibited by this Part.

ownedActions : Action [0..*] {subsets ownedPerformances}

Actions that are owned by this Part. The `this` reference of a `ownedAction` is always its owning Part.

ownedPorts : Port [0..*] {subsets timeEnclosedOccurrences}

Ports that are owned by this Part.

ownedStates : StateAction [0..*] {subsets ownedActions}

StateActions that are owned by this Part.

performedActions : Action [0..*] {subsets enactedPerformances}

Actions that are performed by this Part.

Constraints

None.

9.2.4.2.2 parts

Element

PartUsage

Description

`parts` is the base feature of all PartUsages.

General Types

Part
items

Features

None.

Constraints

None.

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

Element

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 Types

Object

Features

subports : Port [0..*] {subsets timeEnclosedOccurrences}

Constraints

None.

9.2.5.2.2 ports

Element

PortUsage

Description

ports is the base feature of all PortUsages.

General Types

Port
objects

Features

None.

Constraints

None.

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 BinaryConnection

Element

ConnectionDefinition

Description

BinaryConnection is the most general class of binary links between two things within some containing structure.
BinaryConnection is the base type of all ConnectionDefinitions with exactly two ends.

General Types

Connection
BinaryLinkObject

Features

source : Anything [0..*]

target : Anything [0..*]

Constraints

None.

9.2.6.2.2 binaryConnections

Element

ConnectionUsage

Description

binaryConnections is the base feature of all binary ConnectionUsages.

General Types

binaryLinkObjects
connections
BinaryConnection

Features

[no name] : Anything

[no name] : Anything

Constraints

None.

9.2.6.2.3 Connection

Element

ConnectionDefinition

Description

Connection is the most general class of links between things within some containing structure. Connection is the base type of all ConnectionDefinitions.

General Types

LinkObject
Part

Features

None.

Constraints

None.

9.2.6.2.4 connections

Element

ConcernUsage

Description

connections is the base feature of all ConnectionUsages.

General Types

parts
linkObjects

Features

None.

Constraints

None.

9.2.6.2.5 FlowConnection

Element

ConnectionDefinition

FlowConnectionDefinition

Description

FlowConnection is the class of binary connections that represent a transfer of objects or values between two occurrences. It is the base type of all FlowConnectionUsages.

General Types

Transfer

BinaryConnection

Action

Features

source : Occurrence [0..*] {redefines source, toTransferTargets}

target : Occurrence [0..*] {redefines target, toTransferTargets}

Constraints

None.

9.2.6.2.6 flowConnections

Element

FlowConnectionUsage

ConcernUsage

Description

flowConnections is the base feature of all FlowConnectionUsages.

General Types

transfers

binaryConnections

actions

FlowConnection

Features

source : Occurrence [0..*] {redefines source}

target : Occurrence [0..*] {redefines target}

Constraints

None.

9.2.6.2.7 SuccessionFlowConnection

Element

FlowConnectionDefinition

ConnectionDefinition

Description

SuccessionFlowConnection is the subclass of flow connections that represent temporally ordered transfers. It is the base type of all SuccessionFlowConnectionUsages.

General Types

TransferBefore

FlowConnection

Features

source : Occurrence [0..*] {redefines source, toTransferSources}

target : Occurrence [0..*] {redefines target, toTransferTargets}

Constraints

None.

9.2.6.2.8 successionFlowConnections

Element

ConcernUsage

FlowConnectionUsage

Description

successionFlowConnections is the base feature of all SuccessionFlowConnectionUsages.

General Types

transfersBefore

flowConnections

SuccessionFlowConnection

Features

source : Occurrence [0..*] {redefines source}

target : Occurrence [0..*] {redefines target}

Constraints

None.

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

Element

InterfaceDefinition

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 Types

BinaryConnection

Features

source : Port [0..*] {redefines source}

target : Port [0..*] {redefines target}

Constraints

None.

9.2.7.2.2 interfaces

Element

InterfaceUsage

Description

interfaces is the base feature of all InterfaceUsages.

General Types

Interface

binaryConnections

Features

[no name] : Port

[no name] : Port

Constraints

None.

9.2.8 Allocations

9.2.8.1 Allocations Overview

This package defines the base types for allocations and related structural elements in the SysML language.

9.2.8.2 Elements

9.2.8.2.1 Allocation

Element

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 Types

BinaryConnection

Features

source : Anything [0..*] {redefines source}

suballocations : Allocation [0..*]

target : Anything [0..*] {redefines target}

Constraints

None.

9.2.8.2.2 allocations

Element

AllocationUsage

Description

allocations is the base feature of all ConnectionUsages.

General Types

Allocation
binaryConnections

Features

[no name] : Anything

[no name] : Anything

Constraints

None.

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

Element

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 Types

Action

Features

incomingTransfer : TransferBefore {redefines incomingTransferToSelf}

The Transfer accepted by this *AcceptAction*.

payload : Anything [1..*]

The payload received from the incoming *Transfer*. If an input value is provided for this parameter, then the *Transfer* payload must match that value.

receiver : Occurrence

The Occurrence from whose *incomingTransfers* the *incomingTransfer* of the *AcceptAction* is accepted. By default, this is the *this* context reference for the *AcceptAction*.

Constraints

None.

9.2.9.2.2 acceptActions

Element

ActionUsage

Description

acceptActions is the base feature for all SendActionUsages.

General Types

AcceptAction
actions

Features

None.

Constraints

None.

9.2.9.2.3 Action

Element

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 Types

Performance

Features

acceptSubactions : AcceptAction [0..*] {subsets subactions}

The *subactions* of this Action that are *AcceptActions*.

assignments : AssignmentAction [0..*] {subsets subactions}

The *subactions* of this Action that are *AssignmentActions*.

controls : ControlAction [0..*] {subsets subactions}

The *subactions* of this Action that are *ControlActions*.

decisions : DecisionAction [0..*] {subsets controls}

The *controls* of this Action that are *DecisionActions*.

done : Action {redefines endShot}

The ending *snapshot* of this *Action*.

forks : ForkAction [0..*] {subsets controls}

The *controls* of this *Action* that are *ForkActions*.

forLoops : ForLoopAction [0..*] {subsets loops}

The *loops* of this *Action* that are *ForLoopActions*.

ifSubactions : IfThenAction [0..*] {subsets subactions}

The *subactions* of this *Action* that are *IfThenActions* (including *IfThenElseActions*).

joins : JoinAction [0..*] {subsets controls}

The *controls* of this activity that are *JoinActions*.

loops : LoopAction [0..*] {subsets subactions}

The *subactions* of this *Action* that are *LoopActions*.

merges : MergeAction [0..*] {subsets controls}

The *controls* of this *Action* that are *MergeActions*.

sendSubactions : SendAction [0..*] {subsets subactions}

The *subactions* of this *Action* that are *SendActions*.

start : Action {redefines startShot}

The starting *snapshot* of this *Action*.

subactions : Action [0..*] {subsets enclosedPerformances}

The *subperformances* of this *Action* that are *Actions*. The *this* reference of a *subaction* is always the same as that of its owning *Action*.

transitions : TransitionAction [0..*] {subsets subactions}

The *subactions* of this *Action* that are *TransitionActions*.

whileLoops : WhileLoopAction [0..*] {subsets loops}

The *loops* of this *Action* that are *WhileLoopActions*.

Constraints

None.

9.2.9.2.4 actions

Element

ActionUsage

Description

actions is the base feature for all ActionUsages.

General Types

Action
performances

Features

None.

Constraints

None.

9.2.9.2.5 AssignmentAction

Element

ActionDefinition

Description

An *AssignmentAction* is an *Action* used to type an *AssignmentActionUsage*. It is also a *FeatureWritePerformance* that updates the *accessedFeature* of its target *Occurrence* with the given *replacementValues*.

General Types

Action
FeatureWritePerformance

Features

replacementValues : Anything [0..*] {redefines replacementValues}

The values to be assigned to the *accessedFeature* of the *target*.

target : Occurrence {redefines onOccurrence}

The target *Occurrence* whose *accessedFeature* is being assigned.

Constraints

None.

9.2.9.2.6 assignmentActions

Element

ActionUsage

Description

assignmentActions is the base feature for all AssignmentActionUsages.

General Types

AssignmentAction
actions

Features

None.

Constraints

None.

9.2.9.2.7 ControlAction

Element

ActionDefinition

Description

A *ControlAction* is the *Action* of a ControlNode, which has no inherent behavior.

General Types

Action

Features

None.

Constraints

None.

9.2.9.2.8 DecisionAction

Element

ActionDefinition

Description

A *DecisionAction* is the *ControlAction* for a DecisionNode. It is a *DecisionPerformance* that selects one outgoing *HappensBeforeLink*.

General Types

DecisionPerformance
ControlAction

Features

None.

Constraints

None.

9.2.9.2.9 DecisionTransitionAction

Element

ActionDefinition

Description

A *DecisionTransitionAction* is a *TransitionAction* and *NonStateTransitionPerformance* that has a single *guard*, but no *trigger* or *effects*. It is the base type of *TransitionUsages* used as conditional successions in action models.

General Types

NonStateTransitionPerformance
TransitionAction

Features

accepter : AcceptAction {redefines accepter}

effect : Action {redefines effect}

guard : Calculation {redefines guard}

Constraints

None.

9.2.9.2.10 ForkAction

Element

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 Types

ControlAction

Features

None.

Constraints

None.

9.2.9.2.11 ForLoopAction

Element

ActionDefinition

Description

A *ForLoopAction* is a *LoopAction* that iterates over an ordered sequence of values. It is the base type for all *ForLoopActionUsages*.

General Types

LoopAction

Features

body : Action [0..*] {redefines body}

The *Action* that is performed on each iteration of the loop.

index : Positive

The index of the element of *seq* assigned to *var* on the current iteration of the loop.

initialization : AssignmentAction

Initializes *index* to 1.

seq : Anything [0..*] {ordered, nonunique}

The sequence of values over which the loop iterates.

var : Anything

The loop variable that is assigned successive elements of *seq* on each iteration of the loop.

whileLoop : WhileLoopAction

While *index* is less than or equal to the size of *seq*, assigns *var* to the *index* element of *seq*, then performs *body* and increments *index*.

Constraints

None.

9.2.9.2.12 forLoopActions

Element

ActionUsage

Description

forLoopActions is the base feature for all ForLoopActionUsages.

General Types

loopActions
ForLoopAction

Features

None.

Constraints

None.

9.2.9.2.13 IfThenAction

Element

ActionDefinition

Description

An *IfThenAction* is a Kernel *IfThenPerformance* that is also an Action. It is the base type for all IfActionUsages.

General Types

Action
IfThenPerformance

Features

ifTest : BooleanEvaluation {redefines ifTest}

An evaluation of a *Boolean*-valued Expression whose result determines whether or not the *thenClause* is performed.

thenClause : Performance [0..1] {redefines thenClause}

An optional *Performance* that occurs if and only if the result of the *ifTest* is true.

Constraints

None.

9.2.9.2.14 ifThenActions

Element

ActionUsage

Description

ifThenActions is the base feature for all IfActionUsages.

General Types

IfThenAction
actions

Features

None.

Constraints

None.

9.2.9.2.15 IfThenElseAction

Element

ActionDefinition

Description

An *IfThenElseAction* is a Kernel *IfThenElsePerformance* that is also an *IfThenAction*. It is the base type for all IfActionUsages that have an *elseAction*.

General Types

IfThenAction
IfThenElsePerformance

Features

elseClause : Performance [0..1] {redefines *elseClause*}

An optional *Performance* that occurs if and only if the result of the *ifTest* is false.

Constraints

None.

9.2.9.2.16 ifThenElseActions

Element

ActionUsage

Description

`ifThenElseactions` is the base feature for all `IfActionUsages` that have an `elseAction`.

General Types

`IfThenElseAction`
`ifThenActions`

Features

None.

Constraints

None.

9.2.9.2.17 JoinAction

Element

`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 Types

`ControlAction`

Features

None.

Constraints

None.

9.2.9.2.18 LoopAction

Element

`ActionDefinition`

Description

A `LoopAction` is the base type for all `LoopActionUsages`.

General Types

`Action`

Features

body : Action [0..*]

The action that is performed repeatedly in the loop.

Constraints

None.

9.2.9.2.19 loopActions

Element

ActionUsage

Description

loopActions is the base feature for all LoopActionUsages.

General Types

actions
LoopAction

Features

None.

Constraints

None.

9.2.9.2.20 MergeAction

Element

ActionDefinition

Description

A *MergeAction* is the *ControlAction* for a *MergeNode*. It is a *MergePerformance* that selects exactly one incoming *HappensBefore* link.

General Types

ControlAction
MergePerformance

Features

None.

Constraints

None.

9.2.9.2.21 SendAction

Element

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 Types

Action

Features

outgoingTransfer : TransferBefore {redefines outgoingTransferFromSelf}

The *Transfer* initiated by this *SendAction*.

payload : Anything [1..*]

The payload to be sent in the outgoing *Transfer*.

receiver : Occurrence

The *Occurrence* that receives the *outgoingTransfer* as an *incomingTransfer*.

Constraints

None.

9.2.9.2.22 sendActions

Element

ActionUsage

Description

sendActions is the base feature for all *SendActionUsages*.

General Types

actions
SendAction

Features

None.

Constraints

None.

9.2.9.2.23 TransitionAction

Element

ActionDefinition

Description

A TransitionAction is a TransitionPerformance whose `transitionLinkSource` is an Action. It is the base type of all TransitionUsages.

General Types

TransitionPerformance

Action

Features

accepter : AcceptAction [0..1] {subsets subactions}

effect : Action [0..*] {subsets subactions, redefines effect}

transitionLinkSource : Action {redefines transitionLinkSource}

Constraints

None.

9.2.9.2.24 transitionActions

Element

TransitionUsage

Description

`transitionActions` is the base feature for all TransitionUsages.

General Types

actions

Action

TransitionAction

Features

None.

Constraints

None.

9.2.9.2.25 WhileLoopAction

Element

ActionDefinition

Description

A *WhileLoopAction* is a Kernel *LoopPerformance* that is also a *LoopAction*. It is the base type for all *WhileLoopActionUsages*.

General Types

LoopPerformance
LoopAction

Features

body : Action [0..*] {redefines body, body}

The *Action* that is performed while the *whileTest* is true and the *untilTest* is false.

untilTest : BooleanEvaluation [0..*] {redefines untilTest}

Successive evaluations of a *Boolean*-valued Expression that must be false for the loop to continue. The Expression is evaluated after the *body* is performed.

whileTest : BooleanEvaluation [1..*] {redefines whileTest}

Successive evaluations of a *Boolean*-valued Expression that must be true for the loop to continue. the Expression is evaluated before the *body* is performed and is always evaluated at least once.

Constraints

None.

9.2.9.2.26 whileLoopActions

Element

ActionUsage

Description

whileLoopActions is the base feature for all *WhileLoopActionUsages*.

General Types

WhileLoopAction
loopActions

Features

None.

Constraints

None.

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

Element

StateDefinition

Description

A StateAction is a kind of Action that is also a StatePerformance. It is the base type for all StateDefinitions.

General Types

StatePerformance

Action

Features

stateTransitions : StateTransitionAction [0..*] {subsets transitions}

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

None.

9.2.10.2.2 stateActions

Element

StateUsage

Description

stateActions is the base feature for all StateUsages.

General Types

actions
StateAction

Features

None.

Constraints

None.

9.2.10.2.3 StateTransitionAction

Element

ActionDefinition

Description

A StateTransitionAction is a TransitionAction and a StateTransitionPerformance whose transitionLinkSource is a State. It is the base type of TransitionUsages used transitions in state models.

General Types

StateTransitionPerformance
TransitionAction

Features

payload : Anything [0..*]

receiver : Occurrence

transitionLinkSource : StateAction {redefines transitionLinkSource, transitionLinkSource}

Constraints

None.

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

Element

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 Types

Action
Evaluation

Features

subcalculations : Calculation [0..*] {subsets subactions}

The subactions of this FunctionInvocation that are FunctionInvocations.

Constraints

None.

9.2.11.2.2 calculations

Element

CalculationUsage

Description

calculations is the base Feature for all CalculationUsages.

General Types

Calculation
actions
evaluations

Features

None.

Constraints

None.

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

Element

ConstraintDefinition

Description

ConstraintCheck is the most general class for constraint checking. ConstraintCheck is the base type of all ConstraintDefinitions.

General Types

BooleanEvaluation

Features

None.

Constraints

None.

9.2.12.2.2 constraintChecks

Element

ConstraintUsage

Description

constraintChecks is the base feature of all ConstraintUsages.

General Types

booleanEvaluations
ConstraintCheck

Features

None.

Constraints

None.

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

Element

ConcernDefinition

Description

ConcernCheck is the most general class for concern checking. ConcernCheck is the base type of all ConcernDefinitions.

General Types

RequirementCheck

Features

None.

Constraints

None.

9.2.13.2.2 concernChecks

Element

ConcernUsage

Description

concernChecks is the base feature of all ConcernUsages.

General Types

requirementChecks
ConcernCheck

Features

None.

Constraints

None.

9.2.13.2.3 DesignConstraintCheck

Element

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 Types

RequirementCheck

Features

part : Part {redefines subject}

Constraints

None.

9.2.13.2.4 FunctionalRequirementCheck

Element

ConstraintDefinition

Description

A FunctionalRequirementCheck specifies an action that a system, or part of a system, must perform.

General Types

RequirementCheck

Features

subject : Action {redefines subject}

Constraints

None.

9.2.13.2.5 InterfaceRequirementCheck

Element

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 Types

RequirementCheck

Features

subject : Interface {redefines subject}

Constraints

None.

9.2.13.2.6 PerformanceRequirementCheck

Element

ConstraintDefinition

Description

A PerformanceRequirementCheck quantitatively measures the extent to which a system, or a system part, satisfies a required capability or condition.

General Types

RequirementCheck

Features

subject : AttributeValue {redefines subject}

Constraints

None.

9.2.13.2.7 PhysicalRequirementCheck

Element

ConstraintDefinition

Description

A PhysicalRequirementCheck specifies physical characteristics and/or physical constraints of the system, or a system part.

General Types

RequirementCheck

Features

subject : Part {redefines subject}

Constraints

None.

9.2.13.2.8 RequirementCheck

Element

RequirementDefinition

Description

RequirementCheck is the most general class for requirements checking. RequirementCheck is the base type of all RequirementDefinitions.

General Types

ConstraintCheck

Features

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 check for satisfaction of the required constraints.

Constraints

[no name]

[no documentation]

allTrue(assumptions) implies allTrue(constraints)

9.2.13.2.9 requirementChecks

Element

RequirementUsage

Description

requirementChecks is the base feature of all RequirementUsages.

General Types

RequirementCheck

constraintChecks

Features

None.

Constraints

None.

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

Element

CaseDefinition

Description

Case is the most general class of performances of CaseDefinitions. Case is the base class of all CaseDefinitions.

General Types

Calculation

Features

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

None.

9.2.14.2.2 cases

Element

CaseUsage

Description

cases is the base feature of all CaseUsages.

General Types

calculations
Case

Features

None.

Constraints

None.

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

Element

ActionDefinition

Description

An AnalysisAction is a specialized kind of Action used intended to be used as a step in an AnalysisCase.

General Types

Action

Features

None.

Constraints

None.

9.2.15.2.2 AnalysisCase

Element

AnalysisCaseDefinition

Description

`AnalysisCase` is the most general class of performances of `AnalysisCaseDefinitions`. `AnalysisCase` is the base class of all `AnalysisCaseDefinitions`.

General Types

`Case`

Features

`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

None.

9.2.15.2.3 analysisCases

Element

`AnalysisCaseUsage`

Description

`analysisCases` is the base feature of all `AnalysisCaseUsages`.

General Types

`AnalysisCase`
`cases`

Features

None.

Constraints

None.

9.2.15.2.4 AnalysisAction <ActionDefinition>

Description

An AnalysisAction is a specialized kind of Action used intended to be used as a step in an AnalysisCase.

General Types

Action

Features

None.

Constraints

None.

9.2.15.2.5 AnalysisCase <AnalysisCaseDefinition>

Description

AnalysisCase is the most general class of performances of AnalysisCaseDefinitions. AnalysisCase is the base class of all AnalysisCaseDefinitions.

General Types

Case

Features

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

None.

9.2.15.2.6 analysisCases <AnalysisCaseUsage>

Description

analysisCases is the base feature of all AnalysisCaseUsages.

General Types

AnalysisCase
cases

Features

None.

Constraints

None.

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 PassIf

Element

CalculationDefinition

Description

PassIf returns a *pass* or fail *VerdictKind* depending on whether its argument is true or false.

General Types

None.

Features

isPassing : Boolean

in Whether or not a verification has passed.

verdict : VerdictKind

return *pass* if *isPassing* is true and *fail* otherwise.

Constraints

None.

9.2.16.2.2 VerificationCase

Element

VerificationCaseDefinition

Description

VerificationCase is the most general class of performances of VerificationCaseDefinitions. *VerificationCase* is the base class of all VerificationCaseDefinitions.

General Types

Case

Features

objective : VerificationCheck {redefines objective}

The objective of 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 VerificationCaseDefinition 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

None.

9.2.16.2.3 verificationCases

Element

VerificationCaseUsage

Description

verificationCases is the base feature of all VerificationCaseUsages.

General Types

VerificationCase
cases

Features

None.

Constraints

None.

9.2.16.2.4 VerificationCheck

Element

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 Types

RequirementCheck

Features

requirementVerifications : RequirementCheck [0..*] {subsets constraints}

A record of the evaluations of the *RequirementChecks* of requirements being verified.

Constraints

None.

9.2.16.2.5 VerificationMethod

Element

AttributeDefinition

Description

VerificationMethod can be used as metadata annotating a verification case or action.

General Types

None.

Features

kind : VerificationMethodKind [1..*]

The methods by which the annotated verification was carried out.

Constraints

None.

9.2.17 Use Cases

9.2.17.1 Use Cases Overview

This package defines the base types for use cases and related behavioral elements in the SysML language.

9.2.17.2 Elements

9.2.17.2.1 UseCase

Element

UseCaseDefinition

Description

UseCase is the most general class of performances of UseCaseDefinitions. UseCase is the base class of all UseCaseDefinitions.

General Types

Case

Features

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

None.

9.2.17.2.2 useCases

Element

UseCaseUsage

Description

useCases is the base feature of all UseCaseUsages.

General Types

UseCase
cases

Features

None.

Constraints

None.

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

Element

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 Types

TabularRendering

Features

`columnView` : View [0..*] {ordered}

The Views to be rendered in the column cells, in order, of each rows of the table.

Constraints

None.

9.2.18.2.2 asInterconnectionDiagram

Element

RenderingUsage

Description

`asInterconnectionDiagram` renders a View as an interconnection diagram, using the graphical notation defined in the SysML specification.

General Types

GraphicalRendering

Features

None.

Constraints

None.

9.2.18.2.3 asTextualNotation**Element**

RenderingUsage

Description

asTextualNotation renders a View into textual notation as defined in the KerML and SysML specifications.

General Types

TextualRendering

Features

None.

Constraints

None.

9.2.18.2.4 asTreeDiagram**Element**

RenderingUsage

Description

asTreeDiagram renders a View as a tree diagram, using the graphical notation defined in the SysML specification.

General Types

GraphicalRendering

Features

None.

Constraints

None.

9.2.18.2.5 GraphicalRendering**Element**

RenderingDefinition

Description

A GraphicalRendering is a Rendering of a View into a Graphical format.

General Types

Rendering

Features

None.

Constraints

None.

9.2.18.2.6 Rendering

Element

RenderingDefinition

Description

Rendering is the base type of all RenderingDefinitions.

General Types

Part

Features

subrenderings : Rendering [0..*]

Other Renderings used to carry out this Rendering.

Constraints

None.

9.2.18.2.7 renderings

Element

RenderingUsage

Description

renderings is the base feature of all RenderingUsages.

General Types

Rendering

parts

Features

None.

Constraints

None.

9.2.18.2.8 TabularRendering

Element

RenderingDefinition

Description

A TabularRendering is a Rendering of a View into a tabular format.

General Types

Rendering

Features

None.

Constraints

None.

9.2.18.2.9 TextualRendering

Element

RenderingDefinition

Description

A TextualRendering is a Rendering of a View into a textual format.

General Types

Rendering

Features

None.

Constraints

None.

9.2.18.2.10 View

Element

ViewDefinition

Description

View is the base type of all ViewDefinitions.

General Types

Part

Features

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

None.

9.2.18.2.11 ViewpointCheck

Element

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 Types

RequirementCheck

Features

subject : View {redefines subject}

The subject of this ViewpointCheck, which must be a View.

Constraints

None.

9.2.18.2.12 viewpointChecks

Element

ViewpointUsage

Description

viewpointChecks is the base feature of all ViewpointUsages.

General Types

ViewpointCheck
requirementChecks

Features

None.

Constraints

None.

9.2.18.2.13 viewpointConformance

Element

SatisfyRequirementUsage

Description

General Types

RequirementCheck

Features

viewpointSatisfactions : ViewpointCheck [0..*] {subsets constraints}

The required ViewpointChecks.

Constraints

None.

9.2.18.2.14 views

Element

ViewUsage

Description

views is the base feature of all ViewUsages.

General Types

parts
View

Features

None.

Constraints

None.

9.2.19 Standard View Definitions

9.2.19.1 Standard View Definitions Overview

This package defines the standard graphical view definitions for the SysML language. Each view definition specifies the kind of model elements that can be presented in a view usage and the method for rendering the elements. The standard views are generally rendered as a graph with nodes connected by edges. The nodes can have any number of compartments that contain selected members of the node. For example, a part definition or part usage may contain an attributes compartment that contains the attributes of the part. Definition and usage nodes may also contain connection points corresponding to their ports and parameters. The nodes and edges may be rendered with specialized syntax in different views.

The *StandardViewDefinitions* package contains the following normative standard view definitions:

- Containment View (cv)
- Member View (mv)
- Package View (pv)
- Definition and Usage View (duv)
- Tree View (tv)
- Interconnection View (iv)
- Action Flow View (acv)
- State Transition View (stv)
- Sequence View (sv)
- Use Case View (ucv)
- Requirement View (rv)
- Analysis Case View (acv)
- Verification Case View (vcv)
- View and Viewpoint View (vvv)
- Language Extension View (lev)
- Grid View (gv)
- Geometry View (gev)

These standard views establish a basic set of views that shall be supported by any conforming tool. Other kinds of views may also be defined beyond the standard views, using the view modeling capabilities of SysML (see [7.25](#)).

[Table 36](#) provides an overview of the element content supported by each view.

Table 36. Standard View Definitions

	Standard View Definition															
	Containment View (cv)	Member View (mv)	Package View (pv)	Definition and Usage View (duv)	Tree View (tv)	Interconnection View (iv)	Action Flow View (acv)	State Transition View (stv)	Sequence View (sv)	Use Case View (ucv)	Requirement View (rv)	Analysis Case View (acv)	Verification Case View (vcv)	View and Viewpoint View (vv)	Language Extension View (lev)	Grid View (gv)
Graphical Elements																
Action (includes control nodes, loop, send, accept, ...)	x		x	x	x	x	x	x	x		x	x				
Actor*	x		x						x		x	x				
Allocation	x		x	x	x				x							
Analysis	x		x	x	x						x	x				
Annotated element* (includes comment, documentation, textual representation, and metadata, rationale)	x		x	x	x	x	x	x	x	x	x	x	x	x	x	
Attribute	x		x	x	x											
Binding connection	x		x	x	x	x					x	x				
Calculation	x		x	x	x	x	x	x	x	x	x	x	x	x	x	
Causation	x		x	x	x											
Change and time triggers	x		x	x	x	x	x	x	x							
Concern	x		x	x	x					x		x				
Conjugate port definition*	x		x	x	x											
Connection	x		x	x	x											
Constraint	x		x	x	x	x	x	x	x	x	x	x	x	x	x	
Derive requirement										x						
Directed feature*	x		x	x	x											

	Standard View Definition																
	Containment View (cv)	Member View (mv)	Package View (pv)	Definition and Usage View (duv)	Tree View (tv)	Interconnection View (iv)	Action Flow View (acf)	State Transition View (stv)	Sequence View (sv)	Use Case View (ucv)	Requirement View (rv)	Analysis Case View (acv)	Verification Case View (vcv)	View and Viewpoint View (vvv)	Language Extension View (levr)	Grid View (gv)	Geometry View (gev)
Graphical Elements																	
Enumeration	x		x	x	x												
Event occurrence								x									
Expression	x																
Feature value																	
Flow connection	x	x	x	x	x	x	x	x	x		x	x					
Individual (flag on def/usage)	x	x	x	x													
Interface	x	x	x	x													
Item	x	x	x	x													
Lifeline*								x									
Message	x	x	x	x				x									
Metadata	x	x	x	x													
Objective	x	x	x	x				x			x	x					
Occurrence	x	x	x	x				x									
Package*	x	x															
Parameter	x	x				x	x	x	x		x						
Part	x	x	x	x				x									
Port	x	x	x	x				x									
Proxy connection point*					x	x	x	x									
Quantity value	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Requirement	x	x	x	x						x		x					
Snapshot*	x	x	x	x	x	x	x										
Stakeholder*	x	x	x	x				x	x			x					
State	x	x	x	x	x	x	x										
Subject*	x	x	x	x	x				x	x	x	x					

	Standard View Definition																
	Containment View (cv)	Member View (mv)	Package View (pv)	Definition and Usage View (duv)	Tree View (tv)	Interconnection View (iv)	Action Flow View (acf)	State Transition View (stv)	Sequence View (sv)	Use Case View (ucv)	Requirement View (rv)	Analysis Case View (acv)	Verification Case View (vcv)	View and Viewpoint View (vvv)	Language Extension View (levv)	Grid View (gv)	Geometry View (gev)
Graphical Elements																	
Succession*	x		x			x	x	x	x	x		x	x				
Timeslice*	x		x	x	x	x	x	x	x			x	x				
Transition*	x		x	x	x			x									
Use case	x		x	x	x					x							
Measurement reference (i.e., unit)	x		x														
Variant (flag)	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Variation (flag)	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Verification	x		x	x	x						x		x				
View	x		x	x	x								x				
Viewpoint	x		x	x	x								x				
Other relationships*																	
Annotation	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	
Dependency	x		x														
Expose	x		x										x				
Feature membership	x		x	x	x												
Import	x	x															
Membership	x	x		x	x												
Portion Membership	x		x	x	x												
Subclassification	x		x	x	x												
Defined by	x		x	x	x												
Redefine	x		x	x	x												
Refinement	x		x	x							x						
Subset	x		x	x	x							x	x				
Verify	x		x														

	Standard View Definition																
	Containment View (cv)	Member View (mv)	Package View (pv)	Definition and Usage View (duv)	Tree View (tv)	Interconnection View (iv)	Action Flow View (acv)	State Transition View (stv)	Sequence View (sv)	Use Case View (ucv)	Requirement View (rv)	Analysis Case View (acv)	Verification Case View (vcv)	View and Viewpoint View (vvv)	Language Extension View (levv)	Grid View (gv)	Geometry View (gev)
Graphical Elements																	
References*																	
Assert constraint	x		x	x	x												
Assume constraint	x		x	x	x												
Exhibit state	x		x	x	x			x									
Include use case	x		x	x	x					x							
Perform action	x		x	x	x	x	x	x		x							
Ref feature	x		x	x	x	x	x	x	x	x							
Require constraint	x		x	x	x												
Satisfy requirement	x		x	x	x					x							

Note. Elements without an asterisk are kinds of definition and usage elements. Elements with an asterisk are not kinds of a definition nor a usage element.

Release Note. The standard view definitions are expected to be further refined for the final submission. For example, filter expressions should be provided for each view definition to formalize what kinds elements such views may contain.

9.2.19.2 Elements

9.2.19.2.1 ActionFlowView

Element

ViewDefinition

Description

View definition to present connections between actions. Valid nodes and edges in an ActionFlowView are:

- Actions with nested actions
- Parameters with direction
- Flow connection usages (e.g., kinds of transfers from output to input)
- Binding connections between parameters (e.g., delegate a parameter from one level of nesting to another)
- Proxy connection points

- Swim lanes
- Conditional succession
- Control nodes (fork, join, decision, merge)
- Control structures, e.g., if-then-else, until-while-loop, for-loop
- Send and accept actions
- Change and time triggers
- Compartments on actions and parameters

Short name: afv

General Types

InterconnectionView

Features

None.

Constraints

None.

9.2.19.2.2 AnalysisCaseView

Element

ViewDefinition

Description

View definition to present exposed analysis cases and their relationships. Valid nodes and edges in an AnalysisCaseView are:

- Analysis case
- Actor
- Objective
- Action
- Calculation
- Compartments
- Binding connection between an actor and an input parameter with same name

Short name: acv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.3 ContainmentView

Element

ViewDefinition

Description

View definition to present the hierarchical membership structure of model elements starting from an exposed root element. The typical rendering in graphical notation is as an indented list of rows, consisting of dynamically collapsible-expandable nodes that represent branches and leaves of the tree.

Short name: cv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.4 DefinitionAndUsageView

Element

ViewDefinition

Description

View definition to present Definition and Usage members of exposed model element(s), as well as the relationships between them.

Short name: duv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.5 GeometryView

Element

ViewDefinition

Description

View definition to present a visualization of exposed spatial items in two or three dimensions Valid nodes and edges in a GeometryView are:

- Spatial item, including shape
- Coordinate frame
- Feature related to spatial item, such as a quantity (e.g. temperature) of which values are to be rendered on a color scale

The typical rendering in graphical notation would include a number of visualization parameters, such as:

- 2D or 3D view
- viewing direction
- zoom level
- light sources
- object projection mode, e.g., isometric, perspective, orthographic
- object rendering mode, e.g., shaded, wireframe, hidden line
- object pan (placement) and rotate (orientation) settings
- color maps

Short name: gev

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.6 GridView

Element

ViewDefinition

Description

View definition to present exposed model elements and their relationships, arranged in a rectangular grid. GridView is the generalization of the following more specialized views:

- Tabular view
- Data value tabular view
- Relationship matrix view

Short name: gv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.7 InterconnectionView

Element

ViewDefinition

Description

View definition to present exposed features as nodes, nested features as nested nodes, and connections between features as edges between (nested) nodes. Nested nodes may present boundary features (e.g., ports, parameters).

Short name: iv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.8 LanguageExtensionView

Element

ViewDefinition

Description

View definition to present an exposed metadata definition used to extend a library concept.

Short name: lev

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.9 MemberView**Element**

ViewDefinition

Description

View definition to present any members of exposed model element(s). This is the most general view, enabling presentation of any model element. The typical rendering in graphical notation is as a graph of nodes and edges.

Short name: mv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.10 PackageView**Element**

ViewDefinition

Description

View definition to present package structure and content. Owned or imported packages are presented as nodes and their members can be presented in textual compartments.

Short name: pv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.11 RequirementView

Element

ViewDefinition

Description

View definition to present exposed requirements and their relationships. Valid nodes and edges in a UseCaseView are:

- Requirement, including possible metadata
- Nested requirements
- Objective
- Requirement allocation
- Requirement satisfaction (shown in a satisfy requirements compartment)
- Verified by

Short name: rv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.12 SequenceView

Element

ViewDefinition

Description

View definition to present time ordering of event occurrences on lifelines of exposed features. Valid nodes and edges in a SequenceView are:

- Features such as parts with their lifelines
- Event occurrences on the lifelines
- Messages sent from one part to another with and without a type of flow
- Succession between event occurrences
- Nested sequence view (e.g., a reference to a view)
- Compartments

The typical rendering in graphical notation depicts the exposed features horizontally along the top, with vertical lifelines. The time axis is vertical, with time increasing from top to bottom.

Short name: sv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.13 StateTransitionView

Element

ViewDefinition

Description

View definition to present states and their transitions. Valid nodes and edges in a StateTransitionView are:

- States with nested states
- Entry, do, and exit actions
- Transition usages with triggers, guards, and actions
- Compartments on states

Short name: stv

General Types

InterconnectionView

Features

None.

Constraints

None.

9.2.19.2.14 TreeView

Element

ViewDefinition

Description

View definition to present exposed features as nodes, and membership relationships as edges. The typical rendering in graphical notation is as a graph of nodes and edges.

Short name: tv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.15 UseCaseView

Element

ViewDefinition

Description

View definition to present exposed use cases. Valid nodes and edges in a UseCaseView are:

- Use case, with subject
- Actor
- Include relationship
- Objective
- Compartments
- Binding connection between an actor and an input parameter with same name

Short name: ucv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.16 VerificationCaseView

Element

ViewDefinition

Description

View definition to present exposed verification cases and their relationships. Valid nodes and edges in a VerificationCaseView are:

- Verification case, with subject, i.e., unit/article under verification

- Actor
- Objective
- Action, e.g., steps needed to perform the verification
- Requirement
- Verify relationship
- Compartments
- Binding connection between an actor and an input parameter with the same name

Short name: vcv

General Types

None.

Features

None.

Constraints

None.

9.2.19.2.17 ViewAndViewpointView

Element

ViewDefinition

Description

View definition to present exposed View and Viewpoint. Valid nodes and edges in a ViewAndViewpointView are:

- View definition
- View usage
- Viewpoint definition
- Viewpoint Usage
- Stakeholder
- Concern definition
- Concern Usage
- Expose
- Exposed element
- Subclassification
- Defined by
- Subset
- Redefinition

Short name: vvv

General Types

None.

Features

None.

Constraints

None.

9.2.20 Metadata

9.2.20.1 Metadata Overview

This package defines the base types for metadata definitions and related metadata annotations in the SysML language.

9.2.20.2 Elements

9.2.20.2.1 Metadataltem

Element

MetadataDefinition

Description

Metadataltem is the most general class of *Items* that represent *Metaobjects*. *Metadataltem* is the base type of all MetadataDefinitions.

General Types

Metaobject

Item

Features

None.

Constraints

None.

9.2.20.2.2 metadataltems

Element

ItemUsage

Description

metadataltems is the base feature of all MetadataUsages.

Note: It is not itself a MetadataUsage, because it is not being used as an AnnotatingElement here.

General Types

metaobjects

items

MetadataItem

Features

None.

Constraints

None.

9.2.21 SysML

This package contains a reflective SysML model of the SysML abstract syntax. It is generated from the normative MOF abstract syntax model (see [8.3](#)) as follows.

1. The *SysML* model imports all elements from the reflective *KerML* package (see [*KerML*, 9.2.17]) and directly contains all metaclasses mapped from the SysML abstract syntax, without any subpackaging.
2. A metaclass from the MOF model is mapped into a `MetadataDefinition` in the *KerML* package.
 - The MOF metaclass name is mapped unchanged.
 - Generalizations of the MOF metaclass are mapped to `ownedSpecializations`.
 - All properties from the MOF metaclass are mapped to `usages` of the corresponding `MetadataDefinition` (see below). All non-association-end properties are grouped before association-end properties.
3. A property from the MOF model is mapped into an `AttributeUsage` or `ItemUsage`, depending on whether the MOF property type is a data type or a class.
 - The following feature properties are set as appropriate:
 - `isAbstract` = `true` if the MOF property is a derived union
 - `isReadonly` = `true` if the MOF property is read-only.
 - `isDerived` = `true` if the MOF property is derived.
 - `isReferential` = `true` if the MOF property is *not* composite.
 - The MOF property name is mapped unchanged.
 - The MOF property type is mapped to an `ownedTyping` relationship.
 - If the MOF property type is a primitive type, the relationship is to the corresponding type from the *ScalarValues* package (see [*KerML*, 9.3.2]).
 - If the MOF property type is a metaclass, the relationship is to the corresponding reflective `MetadataDefinition`.
 - The MOF property multiplicity is mapped to an owned `MultiplicityRange` with bounds given by `LiteralExpressions`.
 - Subsetted properties from the MOF property are mapped to `ownedSubsettings` of the corresponding reflective `Features` or `Usages`.
 - Redefined properties from the MOF property are mapped to `ownedRedefinitions` of the corresponding reflective `Features` or `Usages`.
 - If the MOF property is `annotatedElement`, then `Metaobject::annotatedElement` is added to the list of redefined properties for the mapping.
4. An enumeration from the MOF model is mapped into an `EnumerationDefinition`.
 - The MOF enumeration name is mapped unchanged.
 - Each enumeration literal from the MOF enumeration is mapped into an `enumeratedValue` of the `EnumerationDefinition`, with the same name as the MOF enumeration literal.

Note that associations are not mapped from the MOF model and, hence, non-navigable association-owned end properties are not included in the reflective model.

9.3 Metadata Domain Library

9.3.1 Metadata Domain Library Overview

The Metadata Domain Library contains library models of generally useful metadata that can be used to annotate model elements (see [7.3](#)).

9.3.2 Modeling Metadata

9.3.2.1 Modeling Metadata Overview

This package contains definitions of metadata generally useful for annotating models.

9.3.2.2 Elements

9.3.2.2.1 Issue

Element

MetadataDefinition

Description

Issue is used to record some issue concerning the annotated element.

General Types

MetadataItem

Features

text : String

A textual description of the issue.

Constraints

None.

9.3.2.2.2 Rationale

Element

MetadataDefinition

Description

Rationale is used to explain a choice or other decision made related to the annotated element.

General Types

MetadataItem

Features

explanation : Anything [0..1]

A reference to a `Feature` that provides a formal explanation of the rationale. (For example, a trade study whose result explains the choice of a certain alternative).

`text : String`

A textual description of the rationale (required).

Constraints

None.

9.3.2.2.3 Refinement

Element

`MetadataDefinition`

Description

Refinement is used to identify a `Dependency` as modeling a refinement relationship. In such a relationship, the source elements of the relationship provide a more precise and/or accurate representation than the target elements.

General Types

`MetadataItem`

Features

`annotatedElement : Dependency {redefines annotatedElement}`

Constraints

None.

9.3.2.2.4 StatusInfo

Element

`MetadataDefinition`

Description

StatusInfo is used to annotate a model element with status information.

General Types

`MetadataItem`

Features

`originator : String [0..1]`

The originator of the annotated element.

`owner : String [0..1]`

The current owner of the annotated element.

risk : Risk [0..1]

An assessment of risk for the annotated element.

status : StatusKind

The current status of work on the annotated element (required).

Constraints

None.

9.3.2.2.5 StatusKind

Element

EnumerationDefinition

Description

StatusKind enumerates the possible statuses of work on a model element.

General Types

AttributeValue

Features

closed

Status is closed.

done

Status is done.

open

Status is open.

tbc

Status is to be confirmed.

tbd

Status is to be determined.

tbr

Status is to be resolved.

Constraints

None.

9.3.3 Risk Metadata

9.3.3.1 Risk Metadata Overview

This package defines metadata for annotating model elements with assessments of risk.

9.3.3.2 Elements

9.3.3.2.1 Level

Element

AttributeDefinition

Description

A *Level* is a *Real* number in the interval 0.0 to 1.0, inclusive.

General Types

AttributeValue

Real

Features

level : Level {redefines self}

Constraints

levelRange

[no documentation]

level >= 0.0 and level <= 1.0

9.3.3.2.2 LevelEnum

Element

EnumerationDefinition

Description

LevelEnum provides standard probability Levels for low, medium and high risks.

General Types

Level

Features

high

High level, taken to be 75%.

low

Low level, taken to be 25%.

medium

Medium level, taken to be 50%.

Constraints

None.

9.3.3.2.3 Risk

Element

MetadataDefinition

Description

Risk is used to annotate a model element with an assessment of the risk related to it in some typical risk areas.

General Types

MetadataItem

Features

costRisk : RiskLevel [0..1]

The risk that work on the annotated element will exceed its planned cost.

scheduleRisk : RiskLevel [0..1]

The risk that work on the annotated element will not be completed on schedule.

technicalRisk : RiskLevel [0..1]

The risk of unresolved technical issues regarding the annotated element.

totalRisk : RiskLevel [0..1]

The total risk associated with the annotated element.

Constraints

None.

9.3.3.2.4 RiskLevel

Element

AttributeDefinition

Description

RiskLevel gives the probability of a risk occurring and, optionally, the impact if the risk occurs.

General Types

AttributeValue

Features

impact : Level [0..1]

The impact of the risk if it occurs (with 0.0 being no impact and 1.0 being the most severe impact).

probability : Level

The probability that a risk will occur.

Constraints

None.

9.3.3.2.5 RiskLevelEnum

Element

EnumerationDefinition

Description

RiskLevelEnum enumerates standard *RiskLevels* for low, medium and high risks (without including impact).

General Types

RiskLevel

Features

high

Risk level with high probability.

low

Risk level with low probability.

medium

Risk level with medium probability.

Constraints

None.

9.3.4 Parameters of Interest Metadata

9.3.4.1 Parameters of Interest Metadata Overview

This package contains definitions of metadata to identify key parameters of interest, including measures of effectiveness (MOE) and other key measures of performance (MOP).

9.3.4.2 Elements

9.3.4.2.1 MeasureOfEffectiveness

Element

MetadataDefinition

Description

MeasureOfEffectiveness (short name *moe*) is semantic metadata for identifying an attribute as a measure of effectiveness.

General Types

MetadataItem

SemanticMetadata

Features

annotatedElement : Usage {redefines annotatedElement}

baseType : Type {redefines baseType}

Base type is *measuresOfEffectiveness*.

Constraints

None.

9.3.4.2.2 MeasureOfPerformance

Element

MetadataDefinition

Description

MeasureOfPerformance (short name *mop*) is semantic metadata for identifying an attribute as a measure of performance.

General Types

MetadataItem

SemanticMetadata

Features

annotatedElement : Usage {redefines annotatedElement}

baseType : Type {redefines baseType}

Base type is *measuresOfPerformance*.

Constraints

None.

9.3.4.2.3 measuresOfEffectiveness

Element

AttributeUsage

Description

Base feature for attributes that are measures of effectiveness.

General Types

attributeValues

Features

None.

Constraints

None.

9.3.4.2.4 measuresOfPerformance

Element

AttributeUsage

Description

Base feature for attributes that are measures of performance.

General Types

attributeValues

Features

None.

Constraints

None.

9.3.5 Image Metadata

9.3.5.1 Image Metadata Overview

This package provides attributive data and metadata to allow a model element to be annotated with an image to be used in its graphical rendering or as a marker to adorn graphical or textual renderings.

9.3.5.2 Elements

9.3.5.2.1 Icon

Element

MetadataDefinition

Description

Icon metadata can be used to annotate a model element with an image to be used to show render the element on a diagram and/or a small image to be used as an adornment on a graphical or textual rendering. Alternatively, another metadata definition can be annotated with an *Icon* to indicate that any model element annotated by the containing metadata can be rendered according to the *Icon*.

General Types

MetadataItem

Features

fullImage : Image [0..1]

A full-sized image that can be used to render the annotated element on a graphical view, potentially as an alternative to its standard rendering.

smallImage : Image [0..1]

A smaller image that can be used as an adornment on the graphical rendering of the annotated element or as a marker in a textual rendering.

Constraints

None.

9.3.5.2.2 Image

Element

AttributeDefinition

Description

Image provides the data necessary for the physical definition of a graphical image.

General Types

AttributeValue

Features

content : String [0..1]

Binary data for the image according to the given MIME type, encoded as given by the encoding.

encoding : String [0..1]

Describes how characters in the content are to be decoded into binary data. At least "base64", "hex", "identify", and "JSONEscape" shall be supported.

location : String [0..1]

A URI for the location of a resource containing the image content, as an alternative for embedding it in the content attribute.

type : String [0..1]

The MIME type according to which the content should be interpreted.

Constraints

None.

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](#)) and related analysis tasks.

9.4.2 Analysis Tooling

9.4.2.1 Analysis Tooling Overview

This package contains definitions for metadata annotations related to analysis tool integration.

9.4.2.2 Elements

9.4.2.2.1 ToolExecution

Element

MetadataDefinition

Description

ToolExecution metadata identifies an external analysis tool to be used to implement the annotated action.

General Types

MetadataItem

Features

toolName : String

uri : String

Constraints

None.

9.4.2.2 ToolVariable

Element

MetadataDefinition

Description

ToolVariable metadata is used in the context of an action that has been annotated with *ToolExecution* metadata. It is used to annotate a parameter or other feature of the action with the name of the variable in the tool that is to correspond to the annotated feature.

General Types

MetadataItem

Features

name : String

Constraints

None.

9.4.3 Sampled Functions

9.4.3.1 Sampled Functions Overview

This package provides a library model of discretely sampled mathematical functions.

A *SampledFunction* can be used for many engineering purposes. For example, it can represent a time series observation, where the domain consists of time instants expressed on a given timescale and the range consists of observed quantity values. It can also capture a physical property of a substance that depends on temperature or pressure or both.

A *SampledFunction* with numerical domain and range types can be used as input to an interpolation algorithm in order to obtain range values for domain values that fall in-between the discretely sampled domain values.

9.4.3.2 Elements

9.4.3.2.1 Domain

Element

CalculationDefinition

Description

Domain returns the sequence of the *domainValues* of all samples in a *SampledFunction*.

General Types

Calculation

Features

fn : SampledFunction

input

result : Anything [0..*]

output

Constraints

None.

9.4.3.2.2 Interpolate

Element

CalculationDefinition

Description

An *Interpolate* calculation returns an interpolated range value from a given *SampledFunction* for a given domain *value*. If the input domain *value* is outside the bounds of the *domainValues* of the *SampleFunction*, null is returned.

General Types

Calculation

Features

fn : SampledFunction

input

result : Anything [0..1]

output

value : Anything

input

Constraints

None.

9.4.3.2.3 interpolateLinear

Element

CalculationUsage

Description

interpolateLinear is an *Interpolate* calculation assuming a linear functional form between *SamplePairs*.

General Types

Interpolate
calculations

Features

fn : SampledFunction

input

result : Anything [0..1] {redefines result}

output

value : Anything {redefines value}

input

Constraints

None.

9.4.3.2.4 Range

Element

CalculationDefinition

Description

Range returns the sequence of the *rangeValues* of all samples in a *SampledFunction*.

General Types

Calculation

Features

fn : SampledFunction

input

result : Anything [0..*]

output

Constraints

None.

9.4.3.2.5 Sample

Element

CalculationDefinition

Description

Sample returns a *SampledFunction* that samples a given calculation over a sequence of *domainValues*.

General Types

Calculation

Features

calculation : CalculationUsage [0..*]

input

domainValues : Anything [0..*]

input

sampling : SampledFunction

output

Constraints

None.

9.4.3.2.6 SampledFunction

Element

AttributeDefinition

Description

SampledFunction is a variable-size, ordered collection of *SamplePair* elements that represents a generic, discretely sampled, uni-variate or multi-variate mathematical function. The function must be monotonic, either strictly increasing or strictly decreasing.

It maps discrete domain values to discrete range values. The domain of the function is represented by the sequence of *domainValues* of each *SamplePair* in *samples*, and the range of the function is represented by the sequence of *rangeValues* of each *SamplePair* in *samples*.

General Types

OrderedMap

Features

samples : SamplePair [0..*] {redefines elements, ordered}

Constraints

[no name]

Note: Assumes the functions '<' and '>' are defined for the domain type.

```
(1..size(samples)-1)->forAll { in i;
    (samples.domainValue[i] < samples.domainValue[i+1]) } or // Strictly increasing
(1..size(samples)-1)->forAll { in i;
    (samples.domainValue[i] > samples.domainValue[i+1]) }      // Strictly decreasing
```

9.4.3.2.7 SamplePair

Element

AttributeDefinition

Description

SamplePair is a key-value pair of a *domainValue* and a *rangeValue*, used as a sample element in *SampledFunction*.

General Types

KeyValuePair

Features

domainValue : Anything {redefines key}

rangeValue : Anything {redefines val}

Constraints

None.

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 \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

$$\mathbf{x}' = f(\mathbf{x}, \mathbf{u}),$$

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

[Fig. 107](#) shows the action definitions in the State Space Representation library. This library defines *StateSpaceDynamics* as the base abstract action definition, which provides the basic structure of input, output, and state space. This definition has *getNextState* and *getOutput* calculations as well to calculate the next state and the current output, which corresponds to the math functions of $f()$ and $g()$, respectively.

ContinuousStateSpaceDynamics gives the continuous extension of *StateSpaceDynamics* by redefining *getNextState* and adding *getDerivative* that calculates the derivative of the state space, corresponding to \mathbf{x}' . Likewise, *DiscreteStateSpaceDynamics* gives the discrete extension by adding *getDifference* that calculates $\Delta\mathbf{x}$, that is the difference between the current state and the next state, and *getNextState* adds it to *stateSpace*.

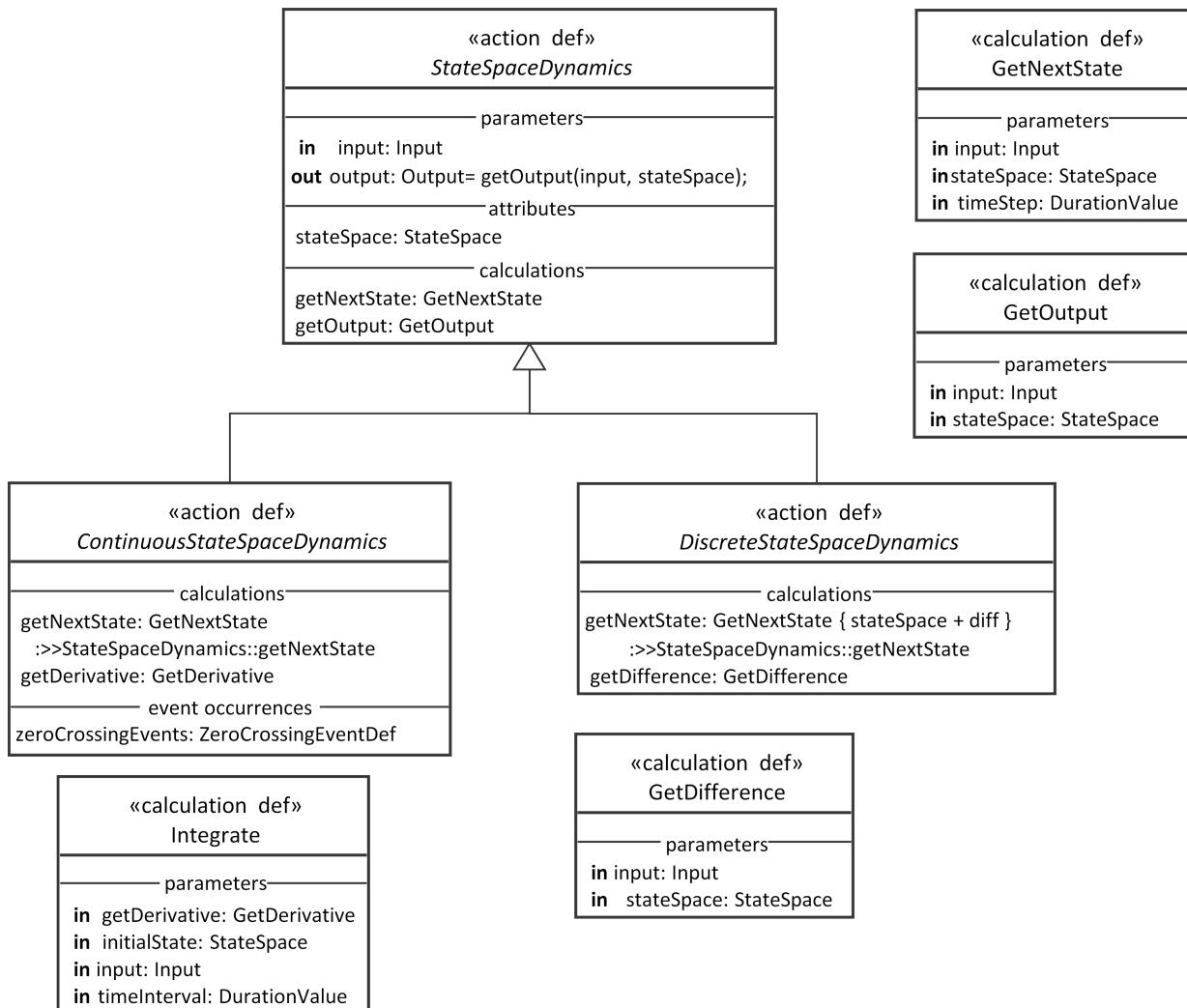


Figure 107. State Space Representation action and calculation definitions

`zeroCrossingEvents` defined in `ContinuousStateSpaceDynamics` give event occurrences when the derivative cross the zero. Some solvers, especially variable-step ones, need to identify such points for precise integration and thus implementations of `ContinuousStateSpaceDynamics` may notify zero-crossings with these event occurrences.

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 EvaluationFunction

Element

CalculationDefinition

Description

An `EvaluationFunction` is a calculation that evaluates a `TradeStudy` alternative, producing a `ScalarValue` that can be compared with the evaluation of other alternatives.

General Types

Calculation

Features

alternative : Anything

The alternative to be evaluated

result : ScalarValue {redefines result}

A `ScalarValue` representing the evaluation of the given alternative.

Constraints

None.

9.4.5.2.2 MaximizeObjective

Element

RequirementDefinition

Description

A `MaximizeObjective` is a `TradeStudyObjective` that requires that the `selectedAlternative` have the maximum `EvaluationFunction` value of all the given alternatives.

General Types

TradeStudyObjective

Features

best : ScalarValue {redefines best}

For a *MaximizeObjective*, the best value is the maximum one.

Constraints

None.

9.4.5.2.3 MinimizeObjective

Element

RequirementDefinition

Description

A *MinimizeObjective* is a *TradeStudyObjective* that requires that the `selectedAlternative` have the minimum `EvaluationFunction` value of all the given alternatives.

General Types

TradeStudyObjective

Features

best : ScalarValue {redefines best}

For a *MinimizeObjective*, the best value is the minimum one.

Constraints

None.

9.4.5.2.4 TradeStudy

Element

AnalysisCaseDefinition

Description

A *TradeStudy* is an analysis case whose subject is a set of alternatives (at least one) and whose result is a selection of one of those alternatives. The alternatives are evaluated based on a given `ObjectiveFunction` and the selection is made such that it satisfies the objective of the *TradeStudy* (which must be a *TradeStudyObjective*).

General Types

AnalysisCase

Features

`evaluationFunction : EvaluationFunction [0..*]`

The *EvaluationFunction* to be used to evaluate the alternatives.

In a *TradeStudy* usage, redefine this feature to provide the desired calculation (or bind it to a calculation usage that does so).

`selectedAlternative : Anything {redefines result}`

The alternative selected by this *TradeStudy*, which is the one that meets the requirement of the *tradeStudyObjective*.

`studyAlternatives : Anything [1..*] {redefines subject}`

The set of alternatives being considered in this *TradeStudy*.

In a *TradeStudy* usage, bind this feature to the actual collection of alternatives to be considered.

`tradeStudyObjective : TradeStudyObjective {redefines objective}`

The objective of this TradeStudy.

Redefine this feature to give it a definition that is a concrete specialization of *TradeStudyObjective*. That can either be one of the specializations provided in this package, or a more specific user-defined one.

Constraints

None.

9.4.5.2.5 TradeStudyObjective

Element

`RequirementDefinition`

Description

A *TradeStudyObjective* is the base definition for the objective of a *TradeStudy*. The requirement is to choose from a given set of alternatives the *selectedAlternative* for that has the best evaluation according to a given *EvaluationFunction*. What value is considered "best" is not defined in the abstract base definition but must be computed in any concrete specialization.

General Types

`RequirementCheck`

Features

`alternatives : Anything [1..*]`

The alternatives being considered in the *TradeStudy* for which this *TradeStudyObjective* is the objective.

`best : ScalarValue`

Out of the evaluation results of all the given alternatives, the one that is considered "best", in the sense that it is the value the *selectedAlternative* should have. This value must be computed in any concrete specialization of *TradeStudyObjective*/em>.

fn : EvaluationFunction

The EvaluationFunction to be used in evaluating the given alternatives.

selectedAlternative : Anything {redefines subject}

The alternative that should be selected, as evaluated using the given *EvaluationFunction*.

Constraints

[no name]

[no documentation]

fn(selectedAlternative) == best

9.5 Cause and Effect Domain Library

9.5.1 Cause and Effect Domain Library Overview

9.5.2 Causation Connections

9.5.2.1 Causation Connections Overview

This package provides a library model modeling causes, effects, and causation connections between them.

9.5.2.2 Elements

9.5.2.2.1 causes

Element

OccurrenceUsage

Description

Occurrences that are causes.

General Types

occurrences

Features

None.

Constraints

None.

9.5.2.2.2 effects

Element

OccurrenceUsage

Description

Occurrences that are effects.

General Types

occurrences

Features

None.

Constraints

None.

9.5.2.2.3 Multicausation

Element

ConnectionDefinition

Description

A *Multicausation* connection models the situation in which one set of occurrences causes another. All causes must at least exist before all effects.

To create a *Multicausation* connection, specialize this connection definition adding specific end features of the relavent types. Ends representing causes should subset *causes*, while ends representing effects should subset *effects*. There must be at least one cause and at least one effect.

General Types

Connection

Features

causes : Occurrence [1..*] {subsets participant}

The causing occurrences.

Redefines *CausationConnections::causes* in the context of *Multicausation*.

effects : Occurrence [1..*] {subsets participant}

The effect occurrences caused by the causing occurrences.

Redefines *CausationConnections::effects* in the context of *Multicausation*.

Constraints

disjointCauseEffect

causes must be disjoint from *effects*.

```
isEmpty(intersection(causes, effects))
```

9.5.2.2.4 multicausations

Element

ConnectionUsage

Description

multicausations is the base feature for *Multicausation ConnectionUsages*.

General Types

connections
Multicausation

Features

None.

Constraints

None.

9.5.3 Cause and Effect

9.5.3.1 Cause and Effect Overview

This package provides language-extension metadata for cause-effect modeling.

9.5.3.2 Elements

9.5.3.2.1 CausationMetadata

Element

MetadataDefinition

Description

CausationMetadata allows for the specification of additional metadata about a cause-effect ConnectionDefinition or ConnectionUsage..

General Types

MetadataItem

Features

annotatedElement1 : ConnectionDefinition {subsets annotatedElement}

annotatedElement2 : ConnectionUsage {subsets annotatedElement}

isNecessary : Boolean

Whether all the causes are necessary for all the effects to occur. If this is false (the default), then some or all of the effects may still have occurred even if some of the causes did not.

isSufficient : Boolean

Whether the causes were sufficient for all the effects to occur. If this is false (the default), then it may be the case that some other occurrences were also necessary for some or all of the effects to have occurred.

probability : Real [0..1]

The probability that the causes will actually result in effects occurring.

Constraints

None.

9.5.3.2.2 CausationSemanticMetadata

Element

MetadataDefinition

Description

CausationMetadata (short name *causation*) is *SemanticMetadata* for a *Causation* connection.

General Types

CausationMetadata

SemanticMetadata

Features

baseType : Type {redefines baseType}

Base type is *CausationConnections::causations*.

Constraints

None.

9.5.3.2.3 CauseMetadata

Element

MetadataDefinition

Description

CauseMetadata (short name *cause*) identifies a `Usage` as being a cause occurrence. It is intended to be used to tag the cause ends of a *Multicausation*.

General Types

MetadataItem
SemanticMetadata

Features

annotatedElement : Usage {redefines annotatedElement}

baseType : Type {redefines baseType}

Base type is *CausationConnections::causes*.

Constraints

None.

9.5.3.2.4 EffectMetadata

Element

MetadataDefinition

Description

EffectMetadata (short name *effect*) identifies a `Usage` as being a effect occurrence. It is intended to be used to tag the effect ends of a *Multicausation*.

General Types

MetadataItem
SemanticMetadata

Features

annotatedElement : Usage {redefines annotatedElement}

baseType : Type {redefines baseType}

Base type is *CausationConnections::effects*.

Constraints

None.

9.5.3.2.5 MulticausationSemanticMetadata

Element

MetadataDefinition

Description

MulticausationMetadata (short name *multicausation*) is *SemanticMetadata* for a *Multicausation* connection.

General Types

CausationMetadata
SemanticMetadata

Features

baseType : Type {redefines baseType}

Base type is *CausationConnections*::*multicausations*.

Constraints

None.

9.6 Requirement Derivation Domain Library

9.6.1 Requirement Derivation Domain Library Overview

9.6.2 Derivation Connections

9.6.2.1 Derivation Connections Overview

This package provides a library model for derivation connections between requirements.

9.6.2.2 Elements

9.6.2.2.1 Derivation

Element

ConnectionDefinition

Description

A *Derivation* connection asserts that one or more *derivedRequirements* are derived from a single *originalRequirement*. This means that any subject that satisfies the *originalRequirement* should, in itself or through other things related to it, satisfy each of the *derivedRequirements*.

A *ConnectionUsage* typed by *Derivation* must have *RequirementUsages* for all its ends. The single end for the original requirement should subset *originalRequirement*, while the rest of the ends should subset *derivedRequirements*.

General Types

Connection

Features

derivedRequirements : RequirementCheck [1..*] {subsets participant}

The one or more requirements that are derived from the original requirement.

Redefines *DerivationConnections::derivedRequirements* in the context of *Derivation*.

originalRequirement : RequirementCheck {subsets participant}

The single original requirement.

Redefines *DerivationConnections::originalRequirement* in the context of *Derivation*.

participant : RequirementCheck [2..*] {redefines participant}

All the *participants* in a *Derivation* must be requirements.

Constraints

originalNotDerived

The original requirement must not be a derived requirement.

derivedRequirements->excludes (originalRequirement)

originalImpliesDerived

Whenever the *originalRequirement* is satisfied, all of the *derivedRequirements* must also be satisfied.

originalRequirement.result implies allTrue(derivedRequirements.result)

9.6.2.2 derivations

Element

ConnectionUsage

Description

derivations is the base feature for *Derivation ConnectionUsages*.

General Types

connections
Derivation

Features

None.

Constraints

None.

9.6.2.2.3 derivedRequirements

Element

RequirementUsage

Description

derivedRequirements are the derived requirements in *Derivation* connections.

General Types

occurrences

Features

None.

Constraints

None.

9.6.2.2.4 originalRequirements

Element

RequirementUsage

Description

originalRequirements are the original requirements in *Derivation* connections.

General Types

occurrences

Features

None.

Constraints

None.

9.6.3 Requirement Derivation

9.6.3.1 Requirement Derivation Overview

This package provides language-extension metadata for modeling requirement derivation.

9.6.3.2 Elements

9.6.3.2.1 DerivationMetadata

Element

MetadataDefinition

Description

DerivationMetadata (short name *derivation*) is *SemanticMetadata* for a *Derivation* connection.

General Types

MetadataItem
SemanticMetadata

Features

annotatedElement1 : ConnectionDefinition {subsets annotatedElement}

annotatedElement2 : ConnectionUsage {subsets annotatedElement}

baseType : Type

Base type is *CausationConnections::multicausations*.

Constraints

None.

9.6.3.2.2 DerivedRequirementMetadata

Element

MetadataDefinition

Description

DerivedRequirementMetadata (short name *derive*) identifies a Usage as a derived requirement. It is intended to be used to tag the derived requirement ends of a *Derivation*.

General Types

MetadataItem
SemanticMetadata

Features

annotatedElement : Usage

baseType : Type

Base type is *CausationConnections::effects*.

Constraints

None.

9.6.3.2.3 OriginalRequirementMetadata

Element

MetadataDefinition

Description

OriginalRequirementMetadata (short name *original*) identifies a *Usage* as an original requirement. It is intended to be used to tag the original requirement end of a *Derivation*.

General Types

MetadataItem
SemanticMetadata

Features

annotatedElement : Usage

baseType : Type

Base type is *CausationConnections::causes*.

Constraints

None.

9.7 Geometry Domain Library

9.7.1 Geometry Domain Library Overview

9.7.2 Spatial Items

9.7.2.1 Spatial Items Overview

This package models physical items that have a spatial extent and act as a spatial frame of reference for obtaining position and displacement vectors of points within them.

9.7.2.2 Elements

9.7.2.2.1 CompoundSpatialItem

Element

ItemDefinition

Description

A *CompoundSpatialItem* is a *SpatialItem* that is a spatial union of a collection of component *SpatialItems*.

General Types

SpatialItem

Features

componentItems : SpatialItem [0..*] {subsets subitems}

The composite *SpatialItems* that together make up this *CompoundSpatialItem*. The *coordinateFrames* of the *compositeItems* are nested in the *coordinateFrame* of the *CompoundSpatialItem* and, by default, they have the same *localClock* as the *CompoundSpatialItem*.

Constraints

None.

9.7.2.2 CurrentDisplacementOf

Element

CalculationDefinition

Description

The *CurrentDisplacementOf* two *Points* relative to a *SpatialItem* and a *Clock* is the *DisplacementOf* the *Points* relative to the *SpatialItem*, at the *currentTime* of the *Clock*.

General Types

CurrentDisplacementOf

Features

clock : Clock {redefines clock}

displacementVector : VectorQuantityValue {redefines displacementVector}

frame : SpatialItem {redefines frame}

point1 : Point {redefines point1}

point2 : Point {redefines point2}

Constraints

None.

9.7.2.2.3 CurrentPositionOf

Element

CalculationDefinition

Description

The *CurrentPositionOf* a *Point* relative to a *SpatialItem* and a *Clock* is the *PositionOf* the *Point* relative to the *SpatialItem* at the *currentTime* of the *Clock*.

General Types

CurrentPositionOf

Features

clock : Clock {redefines clock}

point : Point {redefines point}

```
positionVector : VectorQuantityValue {redefines positionVector}  
spatialItem : SpatialItem {redefines frame}
```

Constraints

None.

9.7.2.2.4 DisplacementOf

Element

CalculationDefinition

Description

The *DisplacementOf* two *Points* relative to a *SpatialItem*, at a specific *TimeInstantValue* relative to a given *Clock*, is the *displacementVector* computed as the difference between the *PositionOf* the first *Point* and *PositionOf* the second *Point*, relative to that *SpatialItem*, at that *timeInstant*.

General Types

DisplacementOf

Features

```
clock : Clock {redefines clock}  
displacementVector : VectorQuantityValue {redefines displacementVector}  
frame : SpatialItem {redefines frame}  
point1 : Point {redefines point1}  
point2 : Point {redefines point2}  
time : TimeInstantValue {redefines time}
```

Constraints

zeroDisplacementConstraint

If either *point1* or *point2* occurs within the other, then the *displacementVector* is the zero vector.

```
(point1.spaceTimeEnclosedOccurrences->includes(point2) or  
point2.spaceTimeEnclosedOccurrences->includes(point1)) implies  
isZeroVector(displacementVector)
```

9.7.2.2.5 PositionOf

Element

CalculationDefinition

Description

The *PositionOf* a *Point* relative to a *SpatialItem*, at a specific *TimeInstantValue* relative to a given *Clock*, is a *positionVector* that is a *VectorQuantityValue* in the *coordinateFrame* of the *SpatialItem*. The default *Clock* is the *localClock* of the *SpatialItem*.

General Types

PositionOf

Features

clock : Clock {redefines clock}

point : Point {redefines point}

positionVector : VectorQuantityValue {redefines positionVector}

spatialItem : SpatialItem {redefines frame}

time : TimeInstantValue {redefines time}

Constraints

spacePositionConstraint

The result *positionVector* is equal to the *PositionOf* the *Point* *spaceShot* of the frame that encloses the given *point*, at the given *time*.

```
(frame.spaceShots as Point)->forAll{in p : Point;
    p.spaceTimeEnclosedOccurrences->includes(point) implies
    positionVector == PositionOf(p, time, frame)
}
```

positionTimePrecondition

The given *point* must exist at the given *time*.

```
TimeOf(point.startShot) <= time and
time <= TimeOf(point.endShot)
```

9.7.2.2.6 SpatialItem

Element

ItemDefinition

Description

A *SpatialItem* is an *Item* with a three-dimensional spatial extent that also acts as a *SpatialFrame* of reference.

General Types

SpatialFrame

Item

Features

`coordinateFrame : VectorMeasurementReference`

The three-dimensional `VectorMeasurementReference` to be used as the measurement reference for position and displacement vector values relative to this `SpatialItem`.

`localClock : Clock {redefines localClock}`

A local `Clock` to be used as the corresponding time reference within this `SpatialItem`. By default this is the singleton `Time::universalClock`.

`originPoint : Point`

The `Point` at the origin of the `coordinateFrame` of this `SpatialItem`.

Constraints

`originPointConstraint`

The `CurrentPositionOf` the `originPoint` must always be a zero vector.

```
isZeroVector(CurrentPositionOf(originPoint, SpatialItem::self))
```

9.7.3 Shape Items

9.7.3.1 Shape Items Overview

This package defines basic geometric `Items`, most of which can be used as `shapes` of other `Items` (see [7.10](#)). Their `innerSpaceDimensions` are either 1 (`Curves`) or 2 (`Surfaces`), which is the number of variables needed to identify any space point occupied by an Item, without regard to higher dimensional spaces in which it might be embedded (see `Occurrences` and `Objects` in Kernel Modeling Language). All are `StructuredSpaceObjects` (`Paths` and `Shells` for `Curves` and `Surfaces`, respectively), except for `Lines`. This enables them to divide their `spaceSlices` into `faces`, `edges`, and `vertices`, identifying `Surfaces`, `Curve`, and `Points`, respectively. `Paths` have no `faces`, but `Shells` do. All the `Paths` and `Shells` are closed (`isClosed=true`, they have no `shape`), except for `Discs`, enabling them to be `shapes` of other `Items`.

9.7.3.2 Elements

9.7.3.2.1 Circle

Element

`ItemDefinition`

Description

A Circle is an Ellipse with semiaxes equal to its `radius`.

General Types

`Ellipse`

Features

`radius : LengthValue`

semiMajorAxis {redefines semiMajorAxis}

Constraints

None.

9.7.3.2.2 CircularCone

Element

ItemDefinition

Description

A CircularCone is a Cone with a circular base.

General Types

Cone

Features

base : CircularDisc {redefines base}

radius : LengthValue

semiMajorAxis {redefines semiMajorAxis}

Constraints

None.

9.7.3.2.3 CircularCylinder

Element

ItemDefinition

Description

A CircularCylinder is a Cylinder with two circular sides.

General Types

Cylinder

Features

af : CircularDisc {redefines af}

base : CircularDisc {redefines base}

radius : LengthValue

semiMajorAxis {redefines semiMajorAxis}

Constraints

None.

9.7.3.2.4 CircularDisc

Element

Description

A CircularDisc is a Disc bound by a Circle.

General Types

Disc

Features

edges : Circle {redefines edges}

radius : LengthValue

shape : Circle {redefines shape}

Constraints

None.

9.7.3.2.5 Cone

Element

ItemDefinition

Description

A Cone has one elliptical sides joined to a point by a curved side.

General Types

ConeOrCylinder

Features

af {redefines af}

apex {subsets vertices}

edges {redefines edges}

faces {redefines faces}

Constraints

None.

9.7.3.2.6 ConeOrCylinder

Element

ItemDefinition

Description

A ConeOrCylinder is a Cone or a Cylinder with a given elliptical base, height, width (perpendicular distance from the base to the center of the top side or vertex), and offsets of this perpendicular at the base from the center of the base.

General Types

Shell

Features

ae [0..2] {subsets edges}

af : Disc [0..1] {subsets faces}

base : Disc {subsets faces}

be {subsets edges}

cf {subsets faces}

edges [2..4] {redefines edges}

faces : Surface [2..3] {redefines faces}

height : LengthValue

semiMajorAxis : LengthValue

semiMinorAxis : LengthValue

vertices [0..1] {redefines vertices}

xoffset : LengthValue

yoffset : LengthValue

Constraints

None.

9.7.3.2.7 ConicSection

Element

ItemDefinition

Description

A ConicSection is a closed PlanarCurve, possibly disconnected, see Hyperbola.

General Types

PlanarCurve

Path

Features

edges [1..2] {redefines edges}

isClosed {redefines isClosed}

vertices {redefines vertices}

Constraints

None.

9.7.3.2.8 ConicSurface

Element

ItemDefinition

Description

A ConicSurface is a Surface that has ConicSection cross-sections.

General Types

Shell

Features

edges {redefines edges}

faces [1..2] {redefines faces}

genus {redefines genus}

vertices {redefines vertices}

Constraints

None.

9.7.3.2.9 Cuboid

Element

ItemDefinition

Description

A Cuboid is a Polyhedron with six sides, all quadrilateral.

General Types

CuboidOrTriangularPrism

Features

edges {redefines edges}

faces {redefines faces}

ff : Quadrilateral {redefines ff}

rf : Quadrilateral {redefines rf}

Constraints

None.

9.7.3.2.10 CuboidOrTriangularPrism

Element

ItemDefinition

Description

A CuboidOrTriangularPrism is a Polyhedron that is either a Cuboid or TriangularPrism.

General Types

Polyhedron

Features

bf : Quadrilateral {subsets faces}

bfe {subsets edges}

bflv {subsets vertices}

bfrv {subsets vertices}

bre {subsets edges}

brlv {subsets vertices}

brrv {subsets vertices}

bsle {subsets edges}

bsre {subsets edges}

edges [0..*] {redefines edges}

```
faces [5..6] {redefines faces}
ff : Polygon {subsets faces}
rf : Polygon {subsets faces}
slf : Quadrilateral {subsets faces}
srf : Quadrilateral [0..1] {subsets faces}
tf : Quadrilateral {subsets faces}
tfe {subsets edges}
tflv {subsets vertices}
tfrv [0..3] {subsets vertices}
tre {subsets edges}
trlv {subsets vertices}
trrv [0..3] {subsets vertices}
tsle {subsets edges}
tsre [0..2] {subsets edges}
ufle {subsets edges}
ufre [0..2] {subsets edges}
urle {subsets edges}
urre [0..2] {subsets edges}
vertices {redefines vertices}
```

Constraints

None.

9.7.3.2.11 Cylinder

Element

ItemDefinition

Description

A Cylinder has two elliptical sides joined by a curved side.

General Types

ConeOrCylinder

Features

ae {redefines ae}
af {redefines af}
edges {redefines edges}
faces {redefines faces}
vertices {redefines vertices}

Constraints

None.

9.7.3.2.12 Disc

Element

ItemDefinition

Description

A Disc is a Shell bound by an Ellipse.

General Types

PlanarSurface
Shell

Features

edges : Ellipse {redefines edges}
faces : PlanarSurface {redefines faces}
semiMajorAxis : LengthValue
semiMinorAxis : LengthValue
shape : Ellipse {redefines shape}
vertices {redefines vertices}

Constraints

None.

9.7.3.2.13 EccentricCone

Element

ItemDefinition

Description

An EccentricCone is a Cone with least one positive offset.

General Types

Cone

Features

None.

Constraints

None.

9.7.3.2.14 EccentricCylinder

Element

ItemDefinition

Description

An EccentricCylinder is a Cylinder with least one positive offset.

General Types

Cylinder

Features

None.

Constraints

None.

9.7.3.2.15 Ellipse

Element

ItemDefinition

Description

An Ellipse is a ConicSection in the shape of an ellipse of given semiaxes.

General Types

ConicSection

Features

edges {redefines edges}

semiMajorAxis : LengthValue
semiMinorAxis : LengthValue

Constraints

None.

9.7.3.2.16 Ellipsoid

Element

ItemDefinition

Description

An Ellipsoid is a ConicSurface with only elliptical cross-sections.

General Types

ConicSurface

Features

faces {redefines faces}

semiAxis1 : LengthValue

semiAxis2 : LengthValue

semiAxis3 : LengthValue

Constraints

None.

9.7.3.2.17 Hyperbola

Element

ItemDefinition

Description

A Hyperbola is a planar Path in the shape of a hyperbola with given axes.

General Types

ConicSection

Features

conjugateAxis : LengthValue

tranverseAxis : LengthValue

Constraints

None.

9.7.3.2.18 Hyperboloid

Element

ItemDefinition

Description

A Hyperboloid is a ConicSurface with only hyperbolic cross-sections.

General Types

ConicSurface

Features

conjugateAxis : LengthValue

transverseAxis : LengthValue

Constraints

None.

9.7.3.2.19 Line

Element

ItemDefinition

Description

A Line is a PlanarCurve that is straight.

General Types

PlanarCurve

Features

outerSpaceDimension {redefines outerSpaceDimension}

Constraints

None.

9.7.3.2.20 Parabola

Element

ItemDefinition

Description

A Parabola is a planar Path in the shape of a parabola of a given focal length.

General Types

ConicSection

Features

edges {redefines edges}

focalDistance : LengthValue

Constraints

None.

9.7.3.2.21 Paraboloid

Element

ItemDefinition

Description

A Paraboloid is a ConicSurface with only parabolic cross-sections.

General Types

ConicSurface

Features

faces {redefines faces}

focalDistance : LengthValue

Constraints

None.

9.7.3.2.22 Path

Element

ItemDefinition

Description

Path is the most general structured Curve.

General Types

StructuredSpaceObject

Item
Curve

Features

edges [1..*] {redefines edges}

faces {redefines faces}

vertices {redefines vertices}

Constraints

None.

9.7.3.2.23 PlanarCurve

Element

ItemDefinition

Description

A PlanarCurve is a Curve with a given `length` embeddable in a plane.

General Types

Item
Curve

Features

`length` : LengthValue

outerSpaceDimension {redefines outerSpaceDimension}

Constraints

None.

9.7.3.2.24 PlanarSurface

Element

ItemDefinition

Description

A PlanarSurface is a Surface with given `area` that is flat.

General Types

Surface
Item

Features

area : LengthValue

outerSpaceDimension {redefines outerSpaceDimension}

Constraints

None.

9.7.3.2.25 Polygon

Element

ItemDefinition

Description

A Polygon is a closed planar Path with straight edges.

General Types

PlanarCurve
Path

Features

edges : Line {redefines edges}

isClosed {redefines isClosed}

Constraints

None.

9.7.3.2.26 Polyhedron

Element

ItemDefinition

Description

A Polyhedron is a closed Shell with polygonal sides.

General Types

Shell

Features

edges {redefines edges}

faces : Polygon [2..*] {redefines faces}

genus {redefines genus}
isClosed
outerSpaceDimension {redefines outerSpaceDimension}
vertices [0..*] {redefines vertices}

Constraints

None.

9.7.3.2.27 Pyramid

Element

ItemDefinition

Description

p>A Pyramid is a Polyhedron with the sides of a polygon (base) forming the bases of triangles that join at an apex point. Its height is the perpendicular distance from the base to the apex, and its offsets are between this perpendicular at the base and the center of the base.

General Types

Polyhedron

Features

apex [0..*] {redefines vertices}
base {subsets faces}
edges [0..*] {redefines edges}
faces [0..*]
height : LengthValue
wall : Triangle [0..*] {subsets faces}
wallNumber : Positive
xoffset : LengthValue
yoffset : LengthValue

Constraints

None.

9.7.3.2.28 Quadrilateral

Element

ItemDefinition

Description

A Quadrilateral is a four-sided Polygon.

General Types

Polygon

Features

e1

e2

e3

edges {redefines edges}

v12 {subsets vertices, ordered}

v23 {subsets vertices, ordered}

v34 {subsets vertices, ordered}

v41 {subsets vertices, ordered}

vertices {redefines vertices}

Constraints

None.

9.7.3.2.29 Rectangle

Element

ItemDefinition

Description

A Rectangle is a Quadrilateral four right angles and given `length` and `width`.

General Types

Quadrilateral

Features

`length` : LengthValue

`width` : LengthValue

Constraints

None.

9.7.3.2.30 RectangularCuboid

Element

ItemDefinition

Description

A RectangularCuboid is a Cuboid with all Rectangular sides.

General Types

Cuboid

Features

bf : Rectangle {redefines bf}

ff : Rectangle {redefines ff}

height : LengthValue

length : LengthValue

rf : Rectangle {redefines rf}

slf : Rectangle {redefines slf}

srf : Rectangle {redefines srf}

tf : Rectangle {redefines tf}

width : LengthValue

Constraints

None.

9.7.3.2.31 RectangularPyramid

Element

ItemDefinition

Description

A RectangularPyramid is Pyramid with a rectangular base.

General Types

Pyramid

Features

base : Rectangle {redefines base}

baseLength : LengthValue

baseWidth : LengthValue

Constraints

None.

9.7.3.2.32 RectangularToroid

Element

ItemDefinition

Description

A RectangularToroid is a revolution of a Rectangle.

General Types

Toriod

Features

rectangleLength : LengthValue

rectangleWidth : LengthValue

revolvedCurve : Rectangle {redefines revolvedCurve}

Constraints

None.

9.7.3.2.33 RightCircularCone

Element

ItemDefinition

Description

A RightCircularCone is a CircularCone with zero offsets.

General Types

CircularCone

Features

xoffset {redefines xoffset}

yoffset {redefines yoffset}

Constraints

None.

9.7.3.2.34 RightCircularCylinder

Element

ItemDefinition

Description

A RightCircularCylinder is a CircularCylinder with zero offsets.

General Types

CircularCylinder

Features

xoffset {redefines xoffset}

yoffset {redefines yoffset}

Constraints

None.

9.7.3.2.35 RightTriangle

Element

ItemDefinition

Description

A RightTriangle is a Triangle with edges opposite the `hypotenuse` at right angles.

General Types

Triangle

Features

hypotenuse {redefines e3}

xoffset {redefines xoffset}

Constraints

None.

9.7.3.2.36 RightTriangularPrism

Element

ItemDefinition

Description

A RightTriangularPrism is a TriangularPrism with two right triangular sides, with given length, width, and height.

General Types

TriangularPrism

Features

bf : Rectangle {redefines bf}

ff : RightTriangle {redefines ff}

height : LengthValue

length : LengthValue

rf : RightTriangle {redefines rf}

slf : Rectangle {redefines slf}

tf : Rectangle {redefines tf}

width : LengthValue

Constraints

None.

9.7.3.2.37 Shell

Element

ItemDefinition

Description

Shell is the most general structured Surface.

General Types

StructuredSpaceObject

Surface

Item

Features

None.

Constraints

None.

9.7.3.2.38 Sphere

Element

ItemDefinition

Description

A Sphere is an Ellipsoid with all the same semiaxes.

General Types

Ellipsoid

Features

radius : LengthValue

semiAxis1 {redefines semiAxis1}

semiAxis2

Constraints

None.

9.7.3.2.39 Tetrahedron

Element

ItemDefinition

Description

A Tetrahedron is Pyramid with a triangular base.

General Types

Pyramid

Features

base : Triangle {redefines base}

baseLength : LengthValue

baseWidth : LengthValue

Constraints

None.

9.7.3.2.40 Toriod

Element

ItemDefinition

Description

A Toroid is a surface generated from revolving a planar closed curve about an line coplanar with the curve. It is single sided with one hole.

General Types

Shell

Features

edges {redefines edges}

faces {redefines faces}

genus {redefines genus}

revolutionRadius : LengthValue

revolvedCurve : PlanarCurve

vertices {redefines vertices}

Constraints

None.

9.7.3.2.41 Torus

Element

ItemDefinition

Description

A Torus is a revolution of a Circle.

General Types

Toroid

Features

majorRadius {redefines revolutionRadius}

minorRadius : LengthValue

revolvedCurve : Circle {redefines revolvedCurve}

Constraints

None.

9.7.3.2.42 Triangle

Element

ItemDefinition

Description

A Triangle is three-sided Polygon with given length (base), width (perpendicular distance from base to apex), and offset of this perpendicular at the base from the center of the base.

General Types

Polygon

Features

apex {subsets vertices, ordered}

base {subsets edges}

e2 {subsets edges}

e3 {subsets edges}

edges {redefines edges}

length : LengthValue

v12 {subsets vertices, ordered}

v31 {subsets vertices, ordered}

vertices {redefines vertices}

width : LengthValue

xoffset : LengthValue

Constraints

None.

9.7.3.2.43 TriangularPrism

Element

ItemDefinition

Description

A TriangularPrism is a Polyhedron with five sides, two triangular and the others quadrilateral.

General Types

Features

edges {redefines edges}

faces {redefines faces}

ff : Triangle {redefines ff}

rf : Triangle {redefines rf}

Constraints

None.

9.8 Quantities and Units Domain Library

9.8.1 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 projects 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 coordinate transformations. It also provides for the definition of coherent systems of quantities and systems of units, as well as operators and functions to support unambiguous quantity expressions, automated unit or scale conversion, and coordinate transformations.

The most widely accepted, scrutinized, and globally used 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. 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 specify the actual quantities, units and scales as standardized in parts 3 to 14 of the ISO/IEC 80000 series as SysML AttributeDefinitions and AttributeUsages. These packages 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].

9.8.2 Quantities

9.8.2.1 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 PowerValue`) as a `power` expressed as 1500 milliwatt. This is an improvement over SysML v1, where the choice of measurement unit or scale 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 a one-dimensional 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: `false` specifies a free vector space (product), and `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. It is the displacement to get from the position defined by the first vector to that defined by 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 instant values can only be specified with respect to a measurement reference that is a 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.

9.8.2.2 Elements

9.8.2.2.1 scalarQuantities

Element

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 Types

`vectorQuantities`
`ScalarQuantityValue`

Features

None.

Constraints

None.

9.8.2.2.2 ScalarQuantityValue

Element

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 Types

`VectorQuantityValue`

NumericalValue

Features

mRef : ScalarMeasurementReference {redefines mRef}

Specification of the ScalarMeasurementReference for the value of the scalar quantity.

Constraints

oneElement

[no documentation]

dimensions[1] == 1

9.8.2.2.3 tensorQuantities

Element

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 Types

TensorQuantityValue

dataValues

Features

None.

Constraints

None.

9.8.2.2.4 TensorQuantityValue

Element

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. In engineering the name 'tensor' is typically 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 `AttributeUsage mRef`.

It is possible to specify the contravariant and covariant order of a tensor quantity through the `AttributeUsages contravariantOrder and covariantOrder`, the sum of which must be equal to `order`. In applications where it is not important to distinguish between contravariant and covariant tensors (or vectors), the convention is to use contravariant by default and therefore set `contravariantOrder` equal to `order`, and `covariantOrder` equal to zero.

General Types

`Array`

Features

`contravariantOrder : Positive`

The number of contravariant indices of the tensor quantity.

`covariantOrder : Positive`

The number of covariant indices of the tensor quantity.

`dimensions : Positive [0..*] {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 [1..*] {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 `order` of the associated `TensorMeasurementReference mRef`.

Constraints

orderSum

[no documentation]

```
contravariantOrder + covariantOrder == order
```

matchingDimensions

[no documentation]

```
dimensions == mRef.dimensions
```

9.8.2.2.5 vectorQuantities

Element

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 Types

`VectorQuantityValue`
`tensorQuantities`

Features

None.

Constraints

None.

9.8.2.2.6 VectorQuantityValue

Element

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 `TensorQuantityValue` and has `order` one.

A `VectorQuantityValue` can be free (`isBound == false`) or bound (`isBound == 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 `VectorMeasurementReference` that includes a specified choice of origin. In both cases the `VectorMeasurementReference` typically defines a coordinate system.

General Types

TensorQuantityValue
NumericalVectorValue

Features

mRef : VectorMeasurementReference {redefines mRef}

Specification of the VectorMeasurementReference for the value of the vector quantity.

Constraints

[no name]

[no documentation]

order == 1

9.8.3 Measurement References

9.8.3.1 Measurement References Overview

This package defines the general AttributeDefinitions and AttributeUsages to construct measurement references. This includes:

- measurement units,
- ordinal, logarithmic and cyclic measurement scales,
- unit conversions,
- coordinate frames and coordinate transformations,
- 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 dimensional analysis of quantity expressions.

9.8.3.2 Elements

9.8.3.2.1 AffineTransformMatrix3d

Element

AttributeDefinition

Description

AffineTransformMatrix3d is a three dimensional CoordinateTransformation specified via an affine 4x4 transformation matrix. The interpretation of the matrix is as follows: - the upper left 3x3 matrix represents the rotation matrix - the upper right 3x1 column vector represents the translation vector - the bottom row must be the row vector (0, 0, 0, 1). I.e. the matrix has the following form: (R, R, R, T, R, R, R, T, R, R, R, T, 0, 0, 0, 1) where the cells marked R form the rotation matrix and the cells marked T form the translation vector. Note: See https://en.wikipedia.org/wiki/Transformation_matrix, under affine transformations for a general explanation.

General Types

CoordinateTransformation
Array

Features

dimensions : Positive [0..*] {redefines dimensions, ordered, nonunique}

elements : Real {redefines elements, ordered, nonunique}

Constraints

validSourceDimensions

[no documentation]

source.dimensions = 3

9.8.3.2.2 ConversionByConvention

Element

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 Types

UnitConversion

Features

None.

Constraints

None.

9.8.3.2.3 ConversionByPrefix

Element

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".

See also SIPrefixes.

General Types

UnitConversion

Features

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

None.

9.8.3.2.4 CoordinateFrame

Element

AttributeDefinition

Description

A CoordinateFrame is a VectorMeasurementReference with the specific purpose to quantify (i.e., coordinatize) a vector space, and optionally locate and orient it with respect to another CoordinateFrame. The optional attribute `transformation` enables specification of the location and orientation of this CoordinateFrame as with respect to another (reference) coordinate frame. If the `target` of the `transformation` is this CoordinateFrame, the `transformation` specifies a CoordinateFrame that is nested inside the CoordinateFrame that is the `source` of the `transformation`. The primary use of this is to specify a chain of nested coordinate frames follows the composite structure of a physical architecture. Typically the `source` CoordinateFrame of the `transformation` is the frame is the frame of the next higher containing Item or Part in a composite structure.

General Types

VectorMeasurementReference

Features

transformation : CoordinateTransformation [0..1]

Constraints

None.

9.8.3.2.5 CoordinateFramePlacement

Element

AttributeDefinition

Description

CoordinateFramePlacement is a CoordinateTransformation by placement of the target frame in the source frame. Attribute `origin` specifies the location of the origin of the target frame through a vector in the source frame. Attribute `basisDirections` specifies the orientation of the target frame by specifying the directions of the respective basis vectors of the target frame via direction vectors in the source frame. An empty sequence of `basisDirections` signifies no change of orientation of the target frame with respect to the source frame.

General Types

CoordinateTransformation

Features

`basisDirections` : VectorQuantityValue [1..*]

`origin` : VectorQuantityValue [0..1]

Constraints

`validOriginDimensions`

[no documentation]

`origin.dimensions = source.dimensions`

`validBasisDirectionsMRef`

[no documentation]

`basisDirections->forAll(bd | bd.mRef = source)`

`validBasisDirectionsDimensions`

[no documentation]

`basisDirections->forAll(bd | bd.dimensions = source.dimensions)`

9.8.3.2.6 CoordinateTransformation

Element

AttributeDefinition

Description

A CoordinateTransformation is an AttributeDefinition that defines the transformation relationship between two coordinate systems, that are represented by VectorMeasurementReferences, typically CoordinateFrames.

General Types

None.

Features

source : VectorMeasurementReference

target : VectorMeasurementReference

Constraints

[no name]

[no documentation]

forall (bd: basisDirections | bd.mRef == source)

matchingSourceAndTarget

[no documentation]

source.dimensions == target.dimensions

validOriginMRef

[no documentation]

origin.mRef == source

numberOfBasisDirections

[no documentation]

size(basisDirections) == source.dimensions[1]

9.8.3.2.7 CyclicRatioScale

Element

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 Types

MeasurementScale

Features

modulus : Number

Attribute `modulus` is a Number that defines the modulus, i.e. periodic cycle, of this CyclicRatioScale.

Constraints

None.

9.8.3.2.8 DefinitionalQuantityValue

Element

AttributeDefinition

Description

Representation of a particular quantity value that is used in the definition of a *MeasurementReference*. Typically such a particular value is defined by convention. It can be used to define a selected reference value, such as the meaning of zero on a measurement scale or the origin of a top-level coordinate system.

General Types

None.

Features

definition : String

num : Number [1..*]

Constraints

None.

9.8.3.2.9 DerivedUnit

Element

AttributeDefinition

Description

DerivedUnit is a MeasurementUnit that represents a measurement unit that depends on one or more powers of other measurement units.

General Types

MeasurementUnit

Features

None.

Constraints

None.

9.8.3.2.10 IntervalScale

Element

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 Types

MeasurementScale

Features

`isBound : Boolean {redefines isBound}`

For an IntervalScale `isBound` is always `true`, since the scale must include a definition of what zero means.

Constraints

None.

9.8.3.2.11 LogarithmicScale

Element

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:

$$v = f \cdot \log_b(x/x_{\text{ref}})^a$$

where: f is a multiplication factor, \log_b is the log function for the given logarithm base b , x is the actual quantity, x_{ref} is a reference quantity, a is an exponent.

General Types

MeasurementScale

Features

exponent : Number

Attribute `exponent` is the exponent $\backslash(a\backslash)$ in the logarithmic value expression.

factor : Number

Attribute `factor` is the multiplication factor $\backslash(f\backslash)$ 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) $\backslash(x_{\{ref\}}\backslash)$ in the logarithmic value expression.

Constraints

None.

9.8.3.2.12 MeasurementScale

Element

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 Types

ScalarMeasurementReference

Features

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

None.

9.8.3.2.13 MeasurementUnit

Element

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 Types

ScalarMeasurementReference

Features

`unitConversion : UnitConversion [0..1]`

`AttributeUsage unitConversion` optionally specifies a UnitConversion that specifies a linear conversion relationship with respect to another MeasurementUnit. This can be used to support automated quantity value 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

None.

9.8.3.2.14 OrdinalScale

Element

AttributeDefinition

Description

An OrdinalScale is a MeasurementScale that represents an ordinal measurement scale, i.e. a scale on which only the ordering of quantity values is meaningful, not the intervals between any pair of values and neither their ratio.

Note: In order to keep the library simple the associated `unit` of an OrdinalScale shall be set to the unit of dimension one, although a unit is meaningless for an OrdinalScale.

Implementation note: The unit attribute of MeasurementScale should be made optional, and its multiplicity be redefined in the specialization of MeasurementScale.

General Types

MeasurementScale

Features

isBound : Boolean {redefines isBound}

Constraints

None.

9.8.3.2.15 Rotation

Element

AttributeDefinition

Description

Representation of a rotation about an axis over an angle. Attribute `axisDirection` specifies the direction of the rotation axis. Attribute `angle` specifies the angle of rotation, where a positive value implies right-handed rotation. Attribute `isIntrinsic` asserts whether the intermediate coordinate frame moves with the rotation or not, i.e. whether an intrinsic or extrinsic rotation is specified. See https://en.wikipedia.org/wiki/Davenport_chained_rotations for details.

General Types

TranslationOrRotation

Features

angle : AngularMeasureValue

axisDirection : VectorQuantityValue

isIntrinsic : Boolean

Constraints

None.

9.8.3.2.16 ScalarMeasurementReference

Element

AttributeDefinition

Description

A ScalarMeasurementReference is a specialization of VectorMeasurementReference. It represents a single measurement reference for a ScalarQuantityValue or for a component of a tensor or vector quantity. Its `order` is

zero. ScalarMeasurementReference is also a generalization of MeasurementUnit and MeasurementScale, which in turn can be regarded as the basis vector of for respectively a free or bound scalar quantity. 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 Types

VectorMeasurementReference

Features

dimensions : Positive [0..1] {redefines dimensions, ordered, nonunique}

mRefs : ScalarMeasurementReference [1..*] {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.

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.

scaleValueDefinitions : ScaleValueDefinition [0..*]

Attribute `scaleValueDefinition` specifies zero or more ScaleValueDefinition that represent particular essential values on a measurement unit or scale.

Constraints

None.

9.8.3.2.17 ScaleValueDefinition

Element

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 Types

None.

Features

description : String

num : Number

Constraints

None.

9.8.3.2.18 ScaleValueMapping

Element

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 Types

None.

Features

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

None.

9.8.3.2.19 SimpleUnit

Element

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 Types

MeasurementUnit

Features

None.

Constraints

exponentIsOne

[no documentation]

```
this.unitPowerFactor1.exponent == 1
```

ExponentIsOne

[no documentation]

```
self.unitPowerFactor.exponent = 1
```

9.8.3.2.20 TensorMeasurementReference

Element

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 Types

Array

Features

definitionalQuantityValues : DefinitionalQuantityValue [0..*] {ordered}

isBound : Boolean

mRefs : ScalarMeasurementReference [1..*] {redefines elements, ordered, nonunique}

order : Natural {redefines rank}

Constraints

None.

9.8.3.2.21 Translation

Element

AttributeDefinition

Description

Representation of a translation with respect to a coordinate frame Attribute `translationVector` specifies the displacement vector that constitutes the translation.

General Types

TranslationOrRotation

Features

translationVector : VectorQuantityValue

Constraints

None.

9.8.3.2.22 TranslationOrRotation

Element

AttributeDefinition

Description

TranslationOrRotation is an abstract union of Translation and Rotation.

General Types

None.

Features

None.

Constraints

None.

9.8.3.2.23 TranslationRotationSequence

Element

AttributeDefinition

Description

Coordinate frame transformation specified by a sequence of translations and/or rotations Note: This is a coordinate transformation that is convenient for interpretation by humans. In particular a sequence of rotations about the principal axes of a coordinate frame is much more easy understandable than a rotation about an arbitrary axis. Any sequence can be reduced to a single combination of a translation and a rotation about a particular axis, but in general the original sequence cannot be retrieved as there are infinitely many sequences representing the reduced transformation.

General Types

CoordinateTransformation

List

Features

elements : TranslationOrRotation [1..*] {redefines elements, ordered, nonunique}

Constraints

None.

9.8.3.2.24 UnitConversion

Element

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 Types

None.

Features

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

None.

9.8.3.2.25 UnitPowerFactor

Element

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 Types

None.

Features

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

None.

9.8.3.2.26 UnitPrefix

Element

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 Types

None.

Features

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

None.

9.8.3.2.27 VectorMeasurementReference

Element

AttributeDefinition

Description

A VectorMeasurementReference is a specialization of TensorMeasurementReference for vector quantities that are typed by a VectorQuantityValue. Its `order` is typically one, but can be zero in case of a specialization to ScalarMeasurementReference. It implicitly defines a vector space of dimension N equal to `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 frame for a vector space. See the CoordinateFrame specialization.

The attribute `isOrthogonal` indicates whether the inner products of the basis vectors of the vector space specified by VectorMeasurementReference are orthogonal or not.

General Types

TensorMeasurementReference

Features

dimensions : Positive [0..1] {redefines dimensions, ordered, nonunique}

isOrthogonal : Boolean

Constraints

placementCheck

[no documentation]

```
size(placement) == 0 | placement.target == self
```

9.8.3.2.28 VectorQuantityValue[1]

Element

Description

General Types

None.

Features

None.

Constraints

None.

9.8.4 ISQ

9.8.4.1 ISQ Overview

The ISQ package specifies a complete set of predefined quantity types for the International System of Quantities (ISQ). The ISQ itself is specified as a SystemOfQuantities. The quantity types are specified as specializations of TensorQuantityValue, VectorQuantityValue, ScalarQuantityValue, that capture all quantities defined in ISO/IEC 80000 parts 3 to 13. It also defines all TensorMeasurementReference, VectorMeasurementReference and ScalarMeasurementReference specializations needed to define concrete MeasurementReference 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"
- `ISQChemistryMolecular` 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.

9.8.4.2 Elements

9.8.4.2.1 amountOfSubstance

Element

`AttributeUsage`

Description**General Types**

AmountOfSubstanceValue
scalarQuantities

Features

None.

Constraints

None.

9.8.4.2.2 AmountOfSubstanceUnit**Element**

AttributeDefinition

Description**General Types**

SimpleUnit

Features

None.

Constraints

None.

9.8.4.2.3 AmountOfSubstanceValue**Element**

AttributeDefinition

Description**General Types**

ScalarQuantityValue

Features

mRef : AmountOfSubstanceUnit {redefines mRef}

num : Real {redefines num}

Constraints

None.

9.8.4.2.4 AngularMeasureValue

Element

Description

General Types

None.

Features

None.

Constraints

None.

9.8.4.2.5 Cartesian3dSpatialCoordinateSystem

Element

AttributeDefinition

Description

General Types

VectorMeasurementReference

Features

dimensions : Positive {redefines dimensions}

isBound : Boolean

isOrthogonal : Boolean {redefines isOrthogonal}

mRefs : LengthValue {redefines mRefs}

Constraints

None.

9.8.4.2.6 duration

Element

AttributeUsage

Description

General Types

scalarQuantities
DurationValue

Features

None.

Constraints

None.

9.8.4.2.7 DurationUnit

Element

AttributeDefinition

Description

General Types

SimpleUnit

Features

None.

Constraints

None.

9.8.4.2.8 DurationValue

Element

AttributeDefinition

Description

General Types

ScalarQuantityValue

Features

mRef : DurationUnit {redefines mRef}

num : Real {redefines num}

Constraints

None.

9.8.4.2.9 electricCurrent

Element

AttributeUsage

Description

General Types

scalarQuantities

ElectricCurrentValue

Features

None.

Constraints

None.

9.8.4.2.10 ElectricCurrentUnit

Element

AttributeDefinition

Description

General Types

SimpleUnit

Features

None.

Constraints

None.

9.8.4.2.11 ElectricCurrentValue

Element

AttributeDefinition

Description

General Types

ScalarQuantityValue

Features

```
mRef : ElectricCurrentUnit {redefines mRef}  
num : Real {redefines num}
```

Constraints

None.

9.8.4.2.12 length

Element

AttributeUsage

Description

General Types

LengthValue
scalarQuantities

Features

None.

Constraints

None.

9.8.4.2.13 LengthUnit

Element

AttributeDefinition

Description

General Types

SimpleUnit

Features

None.

Constraints

None.

9.8.4.2.14 LengthValue

Element

AttributeDefinition

Description

General Types

ScalarQuantityValue

Features

mRef : LengthUnit {redefines mRef}

num : Real {redefines num}

Constraints

None.

9.8.4.2.15 LuminousIntensity

Element

AttributeUsage

Description

General Types

scalarQuantities

LuminousIntensityValue

Features

None.

Constraints

None.

9.8.4.2.16 LuminousIntensityUnit

Element

AttributeDefinition

Description

General Types

SimpleUnit

Features

None.

Constraints

None.

9.8.4.2.17 LuminousIntensityValue

Element

AttributeDefinition

Description

General Types

ScalarQuantityValue

Features

mRef : LuminousIntensityUnit {redefines mRef}

num : Real {redefines num}

Constraints

None.

9.8.4.2.18 mass

Element

AttributeUsage

Description

General Types

scalarQuantities

MassValue

Features

None.

Constraints

None.

9.8.4.2.19 MassUnit

Element

AttributeDefinition

Description

General Types

SimpleUnit

Features

None.

Constraints

None.

9.8.4.2.20 MassValue

Element

AttributeDefinition

Description

General Types

ScalarQuantityValue

Features

mRef : MassUnit {redefines mRef}

num : Real {redefines num}

Constraints

None.

9.8.4.2.21 Position3dVector

Element

AttributeDefinition

Description

General Types

ThreeVectorValue

Features

None.

Constraints

None.

9.8.4.2.22 thermodynamicTemperature

Element

AttributeUsage

Description

General Types

scalarQuantities

ThermodynamicTemperatureValue

Features

None.

Constraints

None.

9.8.4.2.23 ThermodynamicTemperatureUnit

Element

AttributeDefinition

Description

General Types

SimpleUnit

Features

None.

Constraints

None.

9.8.4.2.24 ThermodynamicTemperatureValue

Element

AttributeDefinition

Description

General Types

ScalarQuantityValue

Features

mRef : ThermodynamicTemperatureUnit {redefines mRef}

num : Real {redefines num}

Constraints

None.

9.8.5 SI Prefixes

9.8.5.1 SI Prefixes Overview

This package specifies the SI unit prefixes as defined in [ISO 80000-1], so that they can be used in automated quantity value 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
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
gibi	Gi	1024^3
tebi	Ti	1024^4
pebi	Pi	1024^5
exbi	Ei	1024^6
zebi	Zi	1024^7
yobi	Yi	1024^8

9.8.5.2 Elements

9.8.6 SI

9.8.6.1 SI Overview

This package specifies the measurement units 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 all SI units available in a user model.

9.8.6.2 Elements

9.8.7 US Customary Units

9.8.7.1 US Customary Units Overview

This package specifies all US Customary measurement units, with conversion factors to compatible SI units, as defined in [NIST SP-811], The NIST Guide for the use of the International System of Units (in particular its Appendix B "Conversion Factors").

The statement `import USCUSTOMARYUNITS::*`; suffices to make all US Customary measurement units available in a user model.

9.8.7.2 Elements

9.8.8 Time

9.8.8.1 Time Overview

This package specifies concepts to support time-related quantities and metrology, beyond the quantities duration and time as defined in [ISO 80000-3]. Representations of the Gregorian calendar date and time of day as specified by the [ISO 8601-1] standard are used.

9.8.8.2 Elements

9.8.8.2.1 Clock

Element

PartDefinition

Description

A *Clock* provides a *currentTime* as a TimeInstantValue that advances monotonically over its lifetime.

General Types

Clock

Features

currentTime : TimeInstantValue {redefines currentTime}

Constraints

None.

9.8.8.2.2 Date

Element

AttributeDefinition

Description

Generic representation of a time instant as a calendar date.

General Types

TimeInstantValue

Features

None.

Constraints

None.

9.8.8.2.3 DateTime

Element

AttributeDefinition

Description

Generic representation of a time instant as a calendar date and time of day.

General Types

TimeInstantValue

Features

None.

Constraints

None.

9.8.8.2.4 DurationOf

Element

CalculationDefinition

Description

DurationOf returns the duration of a given *Occurrence* relative to a given *Clock*, which is equal to the *TimeOf* the end snapshot of the *Occurrence* minus the *TimeOf* its start snapshot.

General Types

DurationOf

Features

clock : Clock {redefines clock}

Default is inherited *Occurrence*::*localClock*.

duration : DurationValue {redefines duration}

o : Occurrence {redefines o}

Constraints

None.

9.8.8.2.5 Iso8601DateTime

Element

AttributeDefinition

Description

Representation of an ISO 8601-1 date and time in extended string format.

General Types

UtcTimeInstantValue

Features

num : Real {redefines num}

val : Iso8601DateTimeEncoding

Constraints

None.

9.8.8.2.6 Iso8601DateTimeEncoding

Element

AttributeDefinition

Description

Extended string encoding of an ISO 8601-1 date and time.

General Types

String

Features

None.

Constraints

None.

9.8.8.2.7 Iso8601DateTimeStructure

Element

AttributeDefinition

Description

Representation of an ISO 8601 date and time with explicit date and time component attributes.

General Types

UtcTimeInstantValue

Features

day : Natural

hour : Natural

hourOffset : Integer

microsecond : Natural

minute : Natural

minuteOffset : Integer

month : Natural

num : Real {redefines num}

second : Natural

year : Integer

Constraints

None.

9.8.8.2.8 TimeInstant

Element

AttributeUsage

Description

General Types

scalarQuantities

TimeInstantValue

Features

None.

Constraints

None.

9.8.8.2.9 TimeInstantValue

Element

AttributeDefinition

Description

Representation of a time instant quantity. Also known as instant (of time) or point in time.

General Types

ScalarQuantityValue

Features

mRef : TimeScale {redefines mRef}

num : Real {redefines num}

Constraints

None.

9.8.8.2.10 TimeOf

Element

CalculationDefinition

Description

TimeOf returns a *TimeInstantValue* for a given *Occurrence* relative to a given *Clock*. This *TimeInstantValue* is the time of the start of the *Occurrence*, which is considered to be synchronized with the snapshot of the *Clock* with a *currentTime* equal to the returned *timeInstant*.

General Types

Evaluation

TimeOf

Features

clock : Clock {redefines clock}

Default is inherited *Occurrence*::*localClock*.

o : Occurrence {redefines o}

timeInstant : TimeInstantValue {redefines timeInstant}

Constraints

timeContinuityConstraint

If one *Occurrence* happens immediately before another, then the *TimeOf* the end snapshot of the first *Occurrence* equals the *TimeOf* the second *Occurrence*.

timeOrderingConstraint

If one *Occurrence* happens before another, then the *TimeOf* the end snapshot of the first *Occurrence* is no greater than the *TimeOf* the second *Occurrence*.

startTimeConstraint

The *TimeOf* an *Occurrence* is equal to the time of its start snapshot.

9.8.8.2.11 TimeOfDay

Element

AttributeDefinition

Description

Generic representation of a time instant as a time of day.

General Types

TimeInstantValue

Features

None.

Constraints

None.

9.8.8.2.12 TimeScale

Element

AttributeDefinition

Description

Generic time scale to express a time instant, including a textual definition of the meaning of zero time instant value. Attribute *definitionalEpoch* captures the specification of the time instant with value zero, also known as the (reference) epoch.

General Types

IntervalScale

Features

definitionalEpoch : DefinitionalQuantityValue

definitionalQuantityValues : DefinitionalQuantityValue {redefines definitionalQuantityValues}

unit : DurationUnit

Constraints

None.

9.8.8.2.13 universalClock

Element

PartUsage

Description

universalClock is a single *Clock* that can be used as a default universal time reference.

General Types

Clock
objects
Clock
universalClock

Features

None.

Constraints

None.

9.8.8.2.14 UTC

Element

AttributeDefinition

Description

Representation of the Coordinated Universal Time (UTC) time scale.

UTC is the primary time standard by which the world regulates clocks and time. It is within about 1 second of mean solar time at 0° longitude and is not adjusted for daylight saving time. UTC is obtained from International Atomic Time (TAI) by the insertion of leap seconds according to the advice of the International Earth Rotation and Reference Systems Service (IERS) to ensure approximate agreement with the time derived from the rotation of the Earth.

General Types

TimeScale

Features

definitionalEpoch : DefinitionalQuantityValue {redefines definitionalEpoch}

unit : DurationUnit {redefines unit}

Constraints

None.

9.8.8.2.15 utcTimeInstant

Element

AttributeUsage

Description

General Types

timeInstant
UtcTimeInstantValue

Features

None.

Constraints

None.

9.8.8.2.16 UtcTimeInstantValue

Element

AttributeDefinition

Description

Representation of a time instant expressed on the Coordinated Universal Time (UTC) time scale.

General Types

DateTime

Features

mRef : UTC {redefines mRef}

Constraints

None.

A Annex: Conformance Test Suite

(Normative)

Release Note. A conformance test suite will be provided in the final submission.

B Annex: Example Model

(Informative)

B.1 Introduction

The example presented in this Annex is intended to illustrate how SysML can be used to model a system. The example is a simple vehicle model that highlights selected language features. Both the graphical and corresponding textual notation are presented.

Release Note. The graphical views of the model are generated using prototype SysML v2 visualization applications. There are some inconsistencies between the diagrams in this annex and the graphical notation that is specified in the main body of this document.

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 that contain its parts, actions, and requirements. This model includes separate packages for *VehicleAnalysis*, *VehicleVerification*, *Individuals*, and *View_Viewpoints*. The *VehicleAnalysis* package contains analysis cases to analyze the system, and the *VehicleVerification* package contains verification cases and the verification system to verify the system. The *Views_Viewpoints* package specifies user-defined views. Additional packages that are not shown include a *MissionContext* package that contains use cases for how the vehicle is used in a particular context, and a *VehicleFamily* package that contains a model with variation points from which specific vehicle configurations can be derived. The package for the International System of Quantities (*ISQ*) is imported into this model from the SysML model library so it's content can be used to specify standard quantities and units. Imported packages can be shown with a dashed outline package symbol as shown in the figure or with an import relationship between the importing package and the imported package.

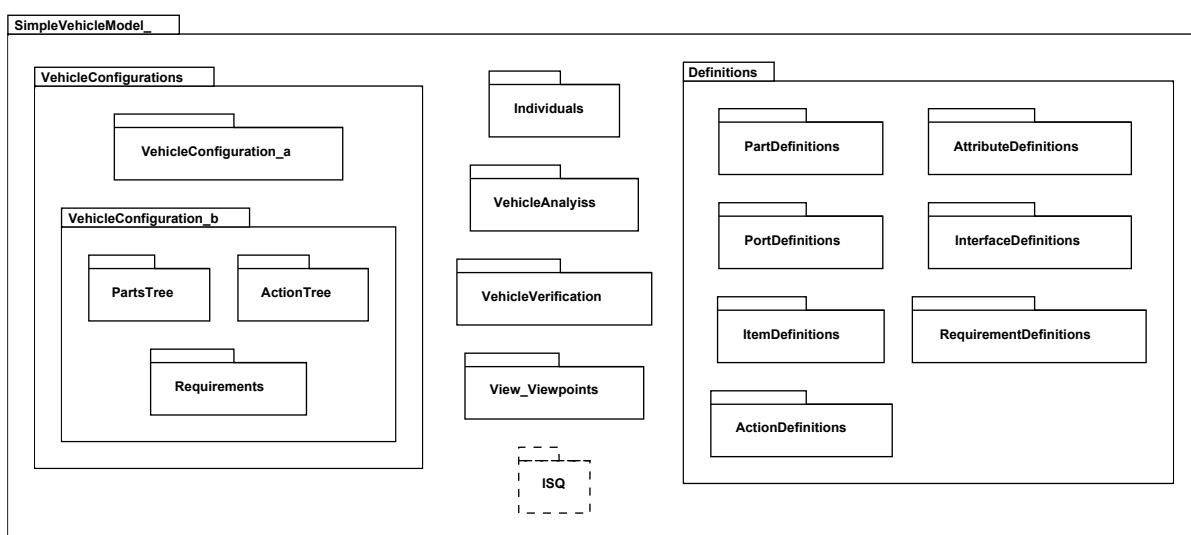


Figure 108. Model Organization for SimpleVehicleModel

```

package SimpleVehicleModel{
    import ISQ::.*;
    package Definitions { ... }
    package VehicleConfigurations{
        package VehicleConfiguration_a{ ... }
        package VehicleConfiguration_b{
            package PartsTree{ ... }
            package ActionTree{ ... }
            package Requirements{ ... }
        }
    }
    package VehicleAnalysis{ ... }
    package VehicleVerification{ ... }
    package Individuals{ ... }
    package Views_Viewpoints{ ... }
    ...
}

```

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. The *VehicleConfigurations* package would typically import packages for each major system element (e.g., subsystem) to aid in collaborative development, although this was not done for this simple example.

B.3 Definitions

The *Definitions* package contains a nested *PartDefinitions* 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 drayMass:> 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 vehicleStates

Figure 109. 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 [9.8.4](#)). Values of the attribute quantities contained in this library can then be assigned standard units from the *SI* (see [9.8.6](#)) or *USCustomaryUnits* (see [9.8.7](#)) 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 attribute called *fuelKind*. The attribute *fuelKind* is defined by the enumeration *FuelKind* that contains literal values for different kinds of fuel such as *gas* and *diesel*. The *FuelTank* also contains an item called *fuel*. An item is often 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).

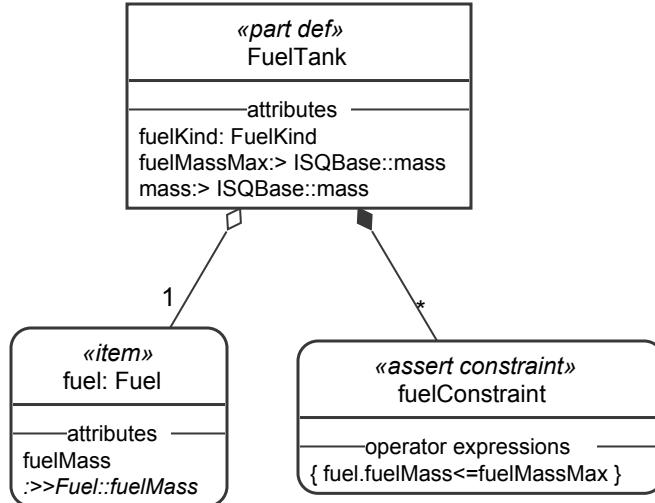


Figure 110. Part Definition for FuelTank Referencing Fuel it Stores

```

part def FuelTank{
    attribute mass :> ISQ::mass;
    attribute fuelKind:FuelKind;
    ref item fuel:Fuel{
        attribute redefines fuelMass;
    }
    attribute fuelMassMax:>ISQ::mass;
    assert constraint {fuel.fuelMass<=fuelMassMax}
    port fuelOutPort:FuelPort;
    port fuelInPort:~FuelPort;
}

```

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. 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 inconsistent 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.

The *FuelTank* also contains a *fuelInPort* and a *fuelOutPort*. The *fuelOutPort* is defined by *FuelPort* that contains a directed feature to represent the fuel flowing out of this port. The *fuelInPort* is defined by a port definition that is the conjugate of the *FuelPort*. The conjugate is denoted with a tilde (~) in front of the port definition name. The conjugate reverses the direction of each directed feature of the port that it conjugates, which in this case reverses the direction of the fuel to flow in to the port instead of out from the port. Note that the directed features are not shown in the figure but are specified as part of the definition of *FuelPort*.

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*.

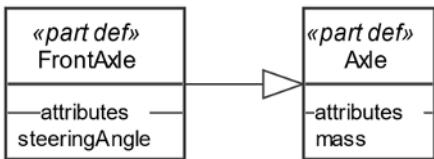


Figure 111. Axle and its Subclass FrontAxle

```

part def Axle{
    attribute mass:>ISQ::mass;
}
part def FrontAxle:>Axle{
    attribute steeringAngle:>ISQ::planeAngle;
}

```

The *Definitions* package also contains several other kinds of definition elements. The port definition *FuelPort* contains an item called *fuel* that can flow out of the port. The *fuel* is defined by the item definition *Fuel* that contains a *mass* attribute. The item def *FuelCmd* contains an attribute called *throttleLevel* that is defined by a *Real*. The action definition *ProvidePower* contains an input item *fuelCmd* that is defined by *FuelCmd*. It also contains an output attribute called *wheelToRoadTorque* that has multiplicity of 2, and is defined by the attribute definition *Torque*.

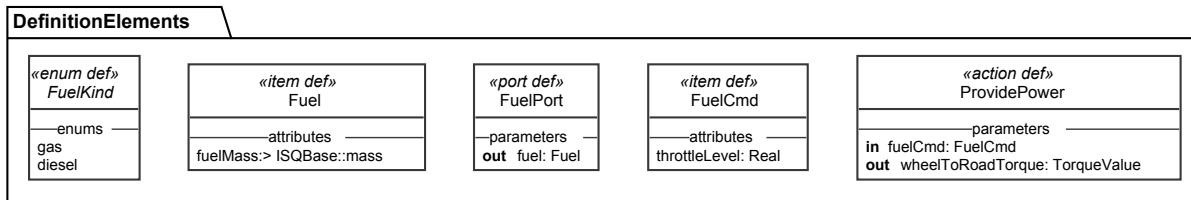


Figure 112. Example Definition Elements

```

action def ProvidePower {
    in item fuelCmd:FuelCmd;
    out wheelToRoadTorque:Torque[2];
}

item def FuelCmd{
    attribute throttleLevel:Real;
}

port def FuelPort{
    out item fuel:Fuel;
}

item def Fuel{
    attribute fuelMass:>ISQ::mass;
}

enum def FuelKind {gas; dielsel;}

```

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* inherits features from its part def *Vehicle*. It can then redefine or subset its inherited features and add new features. As an example, *vehicle_b* redefines the *mass* attribute it inherited from *Vehicle* and further constrains its mass to be the sum of its *dryMass*, *cargoMass*, and *fuelMass*. It redefines other features including other attributes, ports, actions, and states in a similar manner. Its actions are redefined to perform actions that are contained in the *ActionTree*. For example, the inherited action from *Vehicle* to *providePower* is redefined by *ActionTree:::providePower*. As described in B.6, the *providePower* contained in *ActionTree* is decomposed into other actions.

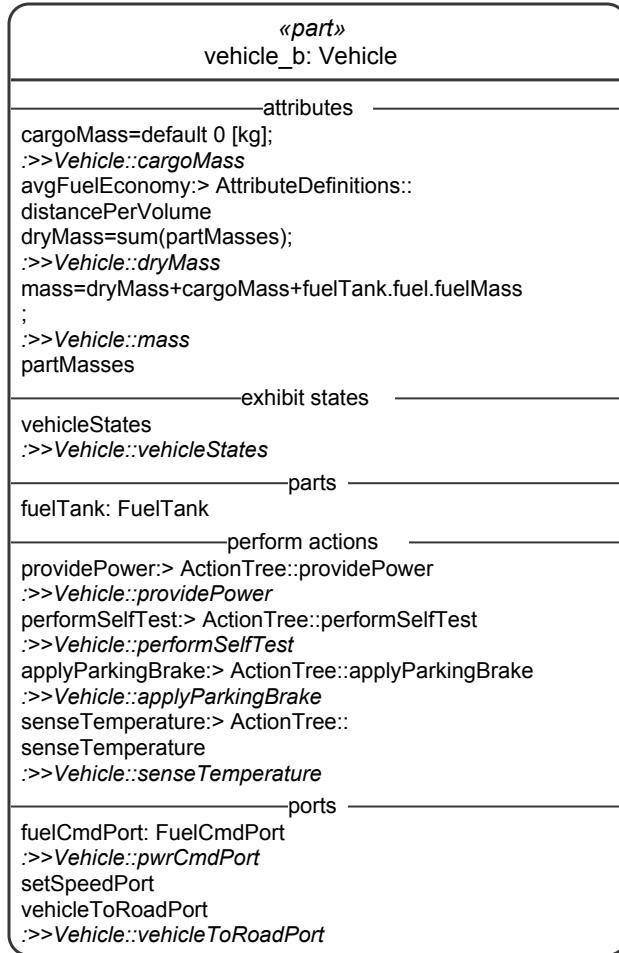


Figure 113. Part Usage for vehicle_b

```
part vehicle_b:Vehicle{
    attribute mass redefines mass=
        dryMass+cargoMass+fuelTank.fuel.fuelMass;
    attribute dryMass redefines dryMass=sum(partMasses);
    attribute redefines cargoMass default 0 [kg];
    attribute partMasses=(...); // collection of part.mass
    attribute fuelEconomy :> distancePerVolume;
    port fuelCmdPort:FuelCmdPort redefines pwrCmdPort{
        in item fuelCmd redefines pwrCmd;
    port setSpeedPort:~SetSpeedPort;
    port vehicleToRoadPort redefines vehicleToRoadPort{
        port wheelToRoadPort1:WheelToRoadPort;
        port wheelToRoadPort2:WheelOtRoadPort;
    }
    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;
}
part fuelTank:FuelTank{
...
}
```

```

    ...
}
```

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*, *frontAxleAssembly*, *rearAxleAssembly*, *fuelTank*, and *vehicleSoftware*. The *frontAxleAssembly* contains a *frontAxle* and two *frontWheels* as designated by the multiplicity [2]. The *rearAxleAssembly* contains a *rearAxle*, *differential*, *rearWheel1*, and *rearWheel2*. Note that some of the definition elements are elided from the figure but are visible in the textual notation. As always, views of the model only show selected aspects of the model.

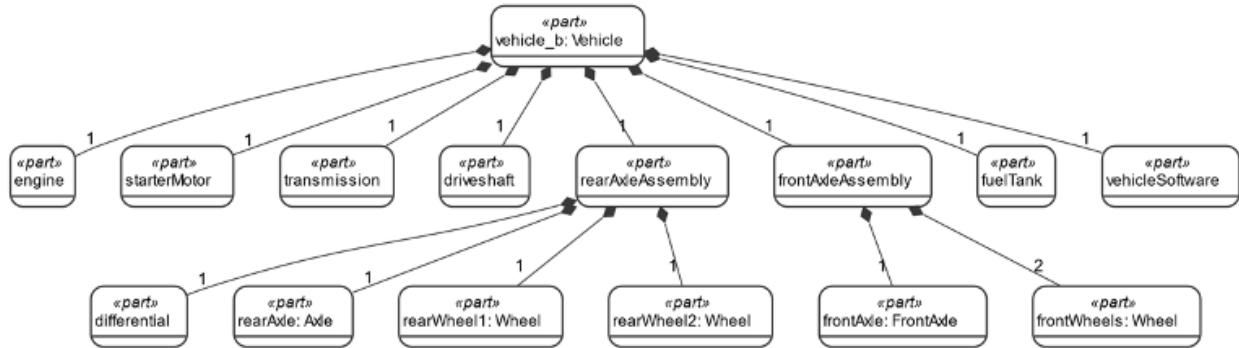


Figure 114. Parts Tree for vehicle_b

```

part vehicle_b:Vehicle {
    ...
    part starterMotor:StarterMotor;
    part fuelTank:FuelTank;
    part engine:Engine;
    part transmission:Transmission;
    part driveshaft:Driveshaft;
    part rearAxleAssembly{
        part differential:Differential;
        part rearAxle:Axe;
        part rearWheel1:Wheel;
        part rearWheel2:Wheel;
    }
    part frontAxleAssembly{
        part frontAxle:FrontAxle;
        part frontWheels:Wheel[2];
    }
    part vehicleSoftware: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 redefines the set of 4..8 cylinders to be exactly 4 *cylinders*, and then subsets the set of 4 cylinders to create *cylinder1*, *cylinder2*, *cylinder3*, and *cylinder4*. (See also the example of variability modeling in [B.12](#).)

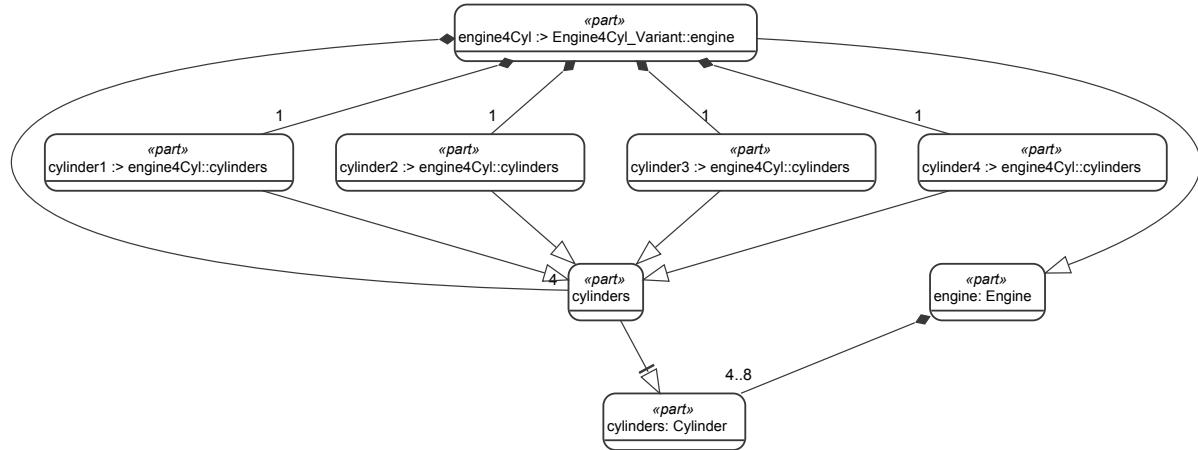


Figure 115. Variant engine4Cyl

```

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 the conjugate of the *DrivePwrPort*. The *DrivePwrPort* contains the directed feature *out engineTorque:Torque*. The conjugate of the *DrivePwrPort* 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.

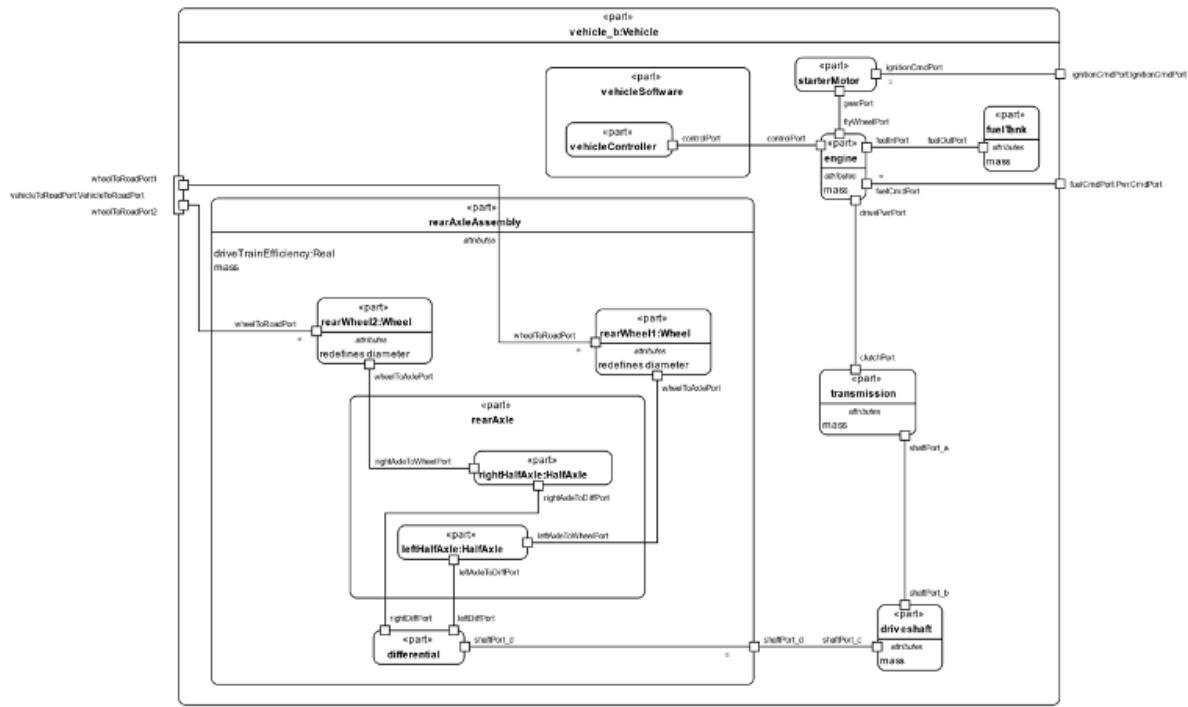


Figure 116. Parts Interconnection for vehicle_b

```

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 :>>wheelToAxePort;
        }
        part differential{
            port shaftPort_d;
            port leftDiffPort;
            port rightDiffPort;
        }
    }
}

```

```

        }

    part rearAxle{
        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.wheelToAxePort;
    connect rearAxe.rightHalfAxe.rightAxeToWheelPort
        to rearWheel2.wheelToAxePort;
}

part starterMotor{
    port ignitionCmdPort;
    port gearPort;
}
part engine{
    attribute mass;
    port fuelCmdPort;
    port drivePwrPort:DrivePwrPort;
    port fuelInPort;
    port flyWheelPort;
    port controlPort;
}
part transmission{
    attribute mass;
    port clutchPort:~DrivePwrPort;
    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. The actions inherit their input and output parameters from their definition and redefine them as necessary (Note: the inherited parameters are not shown.)

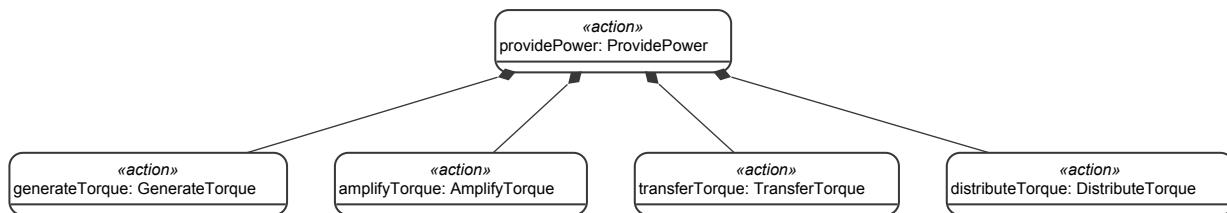


Figure 117. Action providePower

```

action providePower:ProvidePower{
    action generateTorque:GenerateTorque;
    action amplifyTorque:AmplifyTorque;
    action transferTorque:TransferTorque;
    action distributeTorque:DistributeTorque;
    ...
}

```

As shown in [Fig. 113](#), 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 flow connection to the input of the next subaction, except for *distributeTorque*, whose outputs are bound to the outputs of *providePower*. In SysML v2, the input and output parameters are streaming unless designated as succession flows, meaning that the inputs continue to be consumed and the outputs continue to be produced as the action executes.

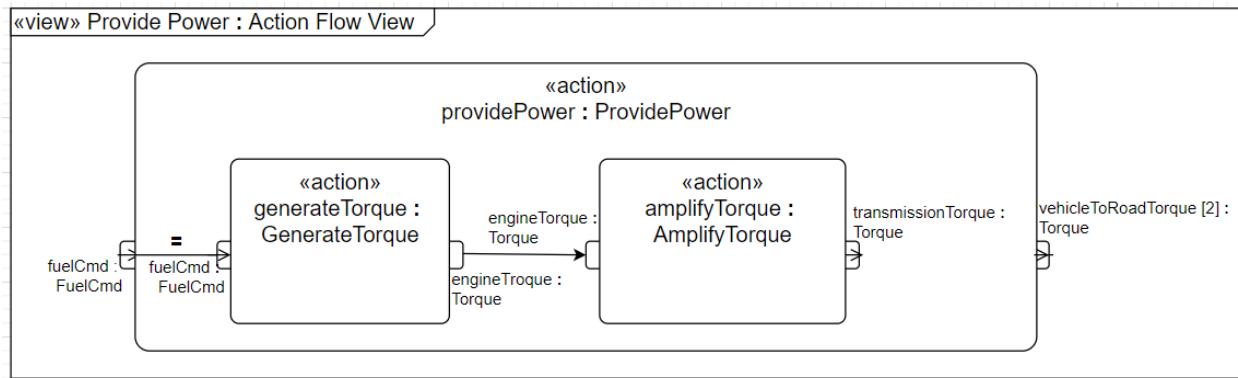


Figure 118. Action flow for providePower

```

action providePower:ProvidePower{
    ...
    bind fuelCmd = generateTorque.fuelCmd;
    flow generateTorque.engineTorque to amplifyTorque.engineTorque;
    flow amplifyTorque.transmissionTorque to transferTorque.transmissionTorque;
    flow transferTorque.driveshaftTorque to distributeTorque.driveshaftTorque;
    bind distributeTorque.wheelToRoadTorque = wheelToRoadTorque;
}
  
```

The *transportPassenger* action flow models the sequence of actions for a *Vehicle* to transport passengers. This action is defined by a use case called *TransportPassenger*, enabling the action to realize the use case. The 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 performed concurrently, after which the *transportPassenger* action is done. The fork and join control nodes and the successions (i.e., first and then) are used to control the action sequence.

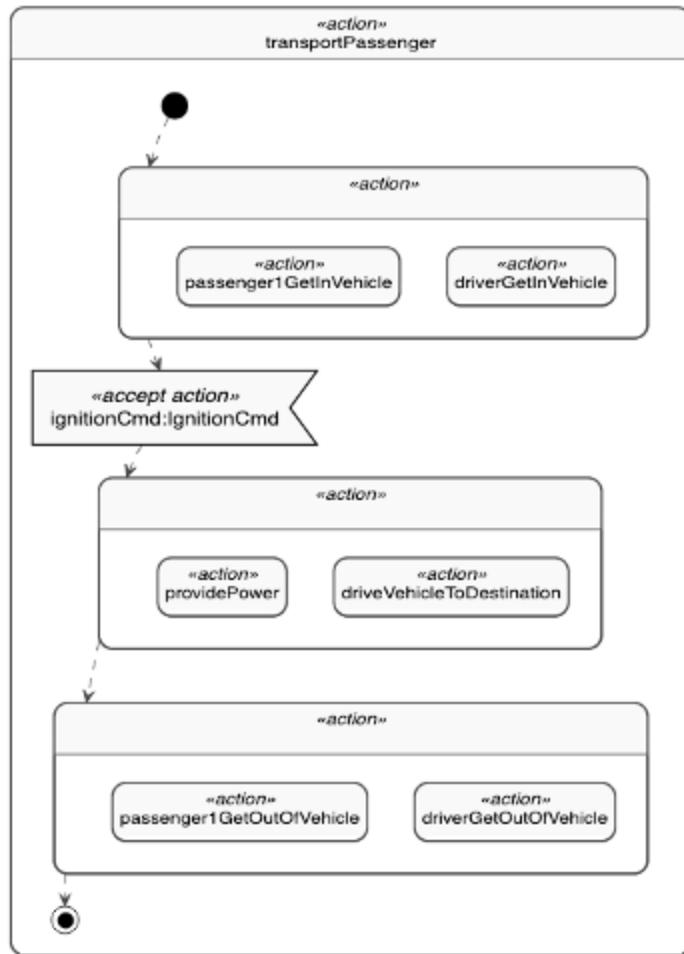


Figure 119. Action flow for transportPassenger

```

action transportPassenger{
    //action declarations
    ...
    first start;
    then fork fork1;
        then driverGetInVehicle;
        then passenger1GetInVehicle;
    first driverGetInVehicle then join1;
    first passenger1GetInVehicle then join1;
    first join1 then trigger;
    first trigger then fork2;
    fork fork2;
        then driveVehicleToDestination;
        then providePower;
    first driveVehicleToDestination then join2;
    first providePower then join2;
    first join2 then fork3;
    fork fork3;
        then driverGetOutOfVehicle;
        then passenger1GetOutOfVehicle;
    first driverGetOutOfVehicle then join3;
    first passenger1GetOutOfVehicle then join3;
    first join3 then done;
}

```

B.7 States

The states of a *Vehicle* enable selected actions to be performed. The *Vehicle* exhibits its state *vehicleStates*. This state is a parallel state, so its substates *operatingStates* and *healthStates* are concurrent. The states *operatingStates* and *healthStates* are not parallel, so only one of each of their substates can be active at any given time.

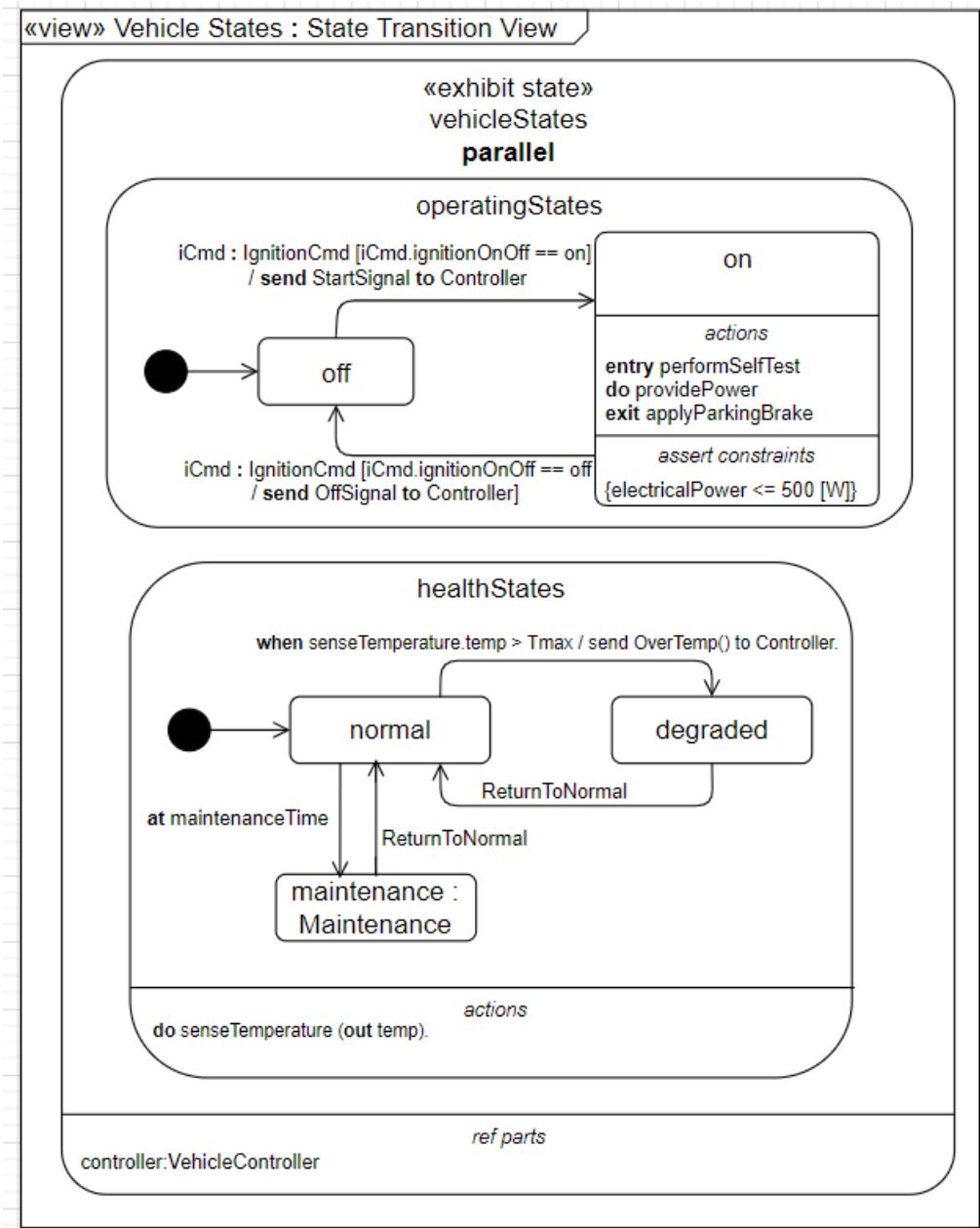


Figure 120. Vehicle States

```

exhibit state vehicleStates parallel {
    ref controller;

    state operatingStates {
        ...
    }
}

```

```

    }

    state healthStates {
        ...
    }
}

```

Note that the state `vehicleStates` has a referential feature `controller:VehicleController`. This allows the substates of `vehicleStates` to send a signal to the `controller` or one of its ports, or to access any other feature of `controller`.

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 performSelfTest;
        do providePower;
        exit applyParkingBrake;
        constraint {electricalPower<=500[W] }
    }

    transition initial then off;

    transition 'off-starting'
        first off
        accept ignitionCmd:IgnitionCmd via ignitionCmdPort
            if ignitionCmd.ignitionOnOff==IgnitionOnOff::on and brakePedalDepressed
        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* when 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 senseTemperature{
        out temp;
    }

    state normal;
    state maintenance;
    state degraded;

    transition initial then normal;

    transition 'normal-maintenance'
        first normal
        accept at maintenanceTime
        then maintenance;

    transition 'normal-degraded'
        first normal
        accept when senseTemperature.temp > 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 <= massRequired}*.

«requirement def» MassRequirement	
The actual mass shall be less than the required mass	
— attributes —	
massActual massRequired	
— constraints —	
require { massActual<=massRequired }	

Figure 121. 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 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 *massRequired* attribute 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. The *vehicleSpecification* also contains *vehicleFuelEconomyRequirements* for both city and highway.

Although not shown, the mass requirement is allocated to the mass of the vehicle using the allocate relationship, and the engine mass requirement is derived from the vehicle mass requirement using the derivation relationship.

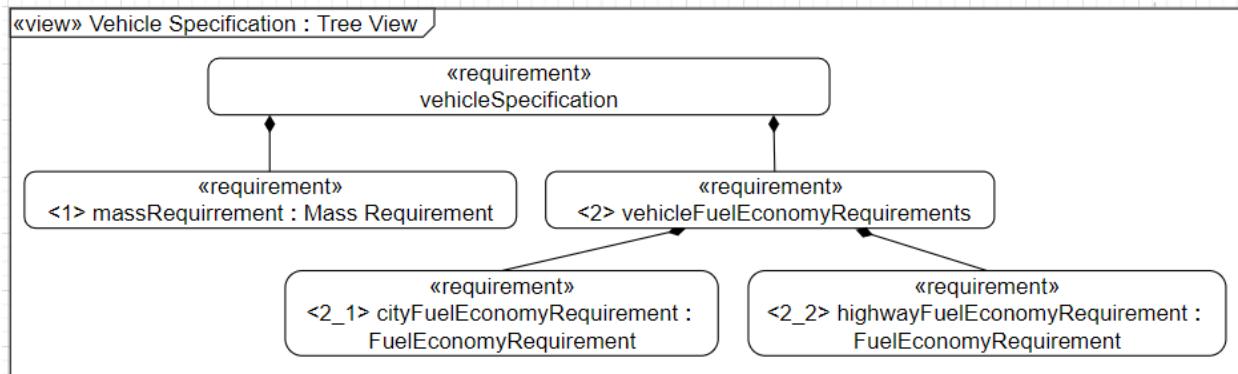


Figure 122. Requirements Group vehicleSpecification

```
requirement vehicleSpecification{
    subject vehicle:Vehicle;
    requirement <'1'> vehicleMassRequirement : MassRequirement {
        doc /* The total mass of a vehicle shall be less than or equal to the required mass.
        Assume total mass includes a full tank of gas of 60 kg*/
        attribute redefines massRequired=2000 [kg];
        attribute redefines massActual = vehicle.dryMass + fuelMassActual;
        attribute fuelMassActual:>ISQ::mass;
        attribute fuelMassMax:>ISQ::mass = 60 [kg];
    }
}
```

```

        assume constraint {fuelMassActual==fuelMassMax}
    }
    requirement <'2'> vehicleFuelEconomyRequirements {
        doc /* fuel economy requirements group */
        attribute assumedCargoMass:>ISQ::mass;
        requirement <'2_1'> cityFuelEconomyRequirement : FuelEconomyRequirement {
            redefines requiredFuelEconomy= 10 [km / L];
            assume constraint {assumedCargoMass>=500 [kg]}
        }
        requirement <'2_2'> highwayFuelEconomyRequirement : FuelEconomyRequirement {
            redefines requiredFuelEconomy= 12.75 [km / L];
            assume constraint {assumedCargoMass>=500 [kg]}
        }
    }
}
}

```

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*.

```

satisfy vehicleSpecification by vehicle_b {
    requirement vehicleMassRequirement :>> vehicleMassRequirement {
        attribute redefines massActual = vehicle_b.mass;
        attribute redefines fuelMassActual = vehicle_b.fuelTank.fuel.fuelMass;
    }
}

```

B.9 Analysis

The *FuelEconomyAnalysisModel* package contains an analysis case called *fuelEconomyAnalysis*. The objective for this analysis case is to estimate the fuel economy of the vehicle. 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*)

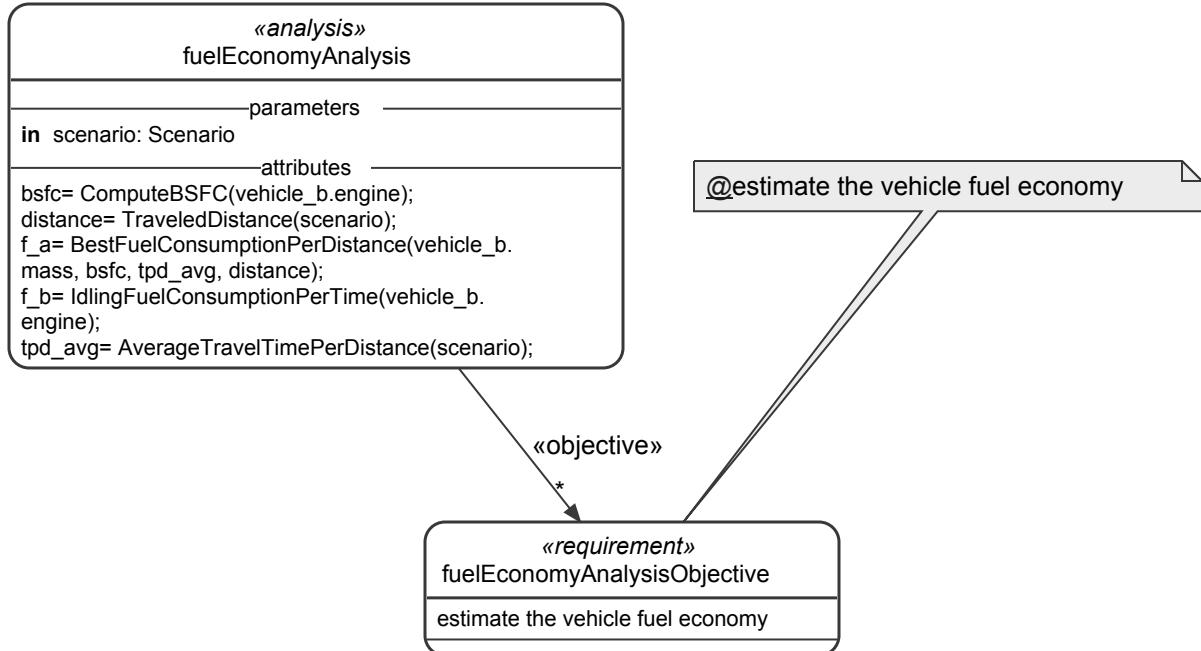


Figure 123. Analysis Case fuelEconomyAnalysis

```

analysis fuelEconomyAnalysis {
    in attribute scenario: Scenario;

    subject = vehicle_b;

    objective fuelEconomyAnalysisObjective {
        doc /* estimate the vehicle fuel economy */
    }

    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.

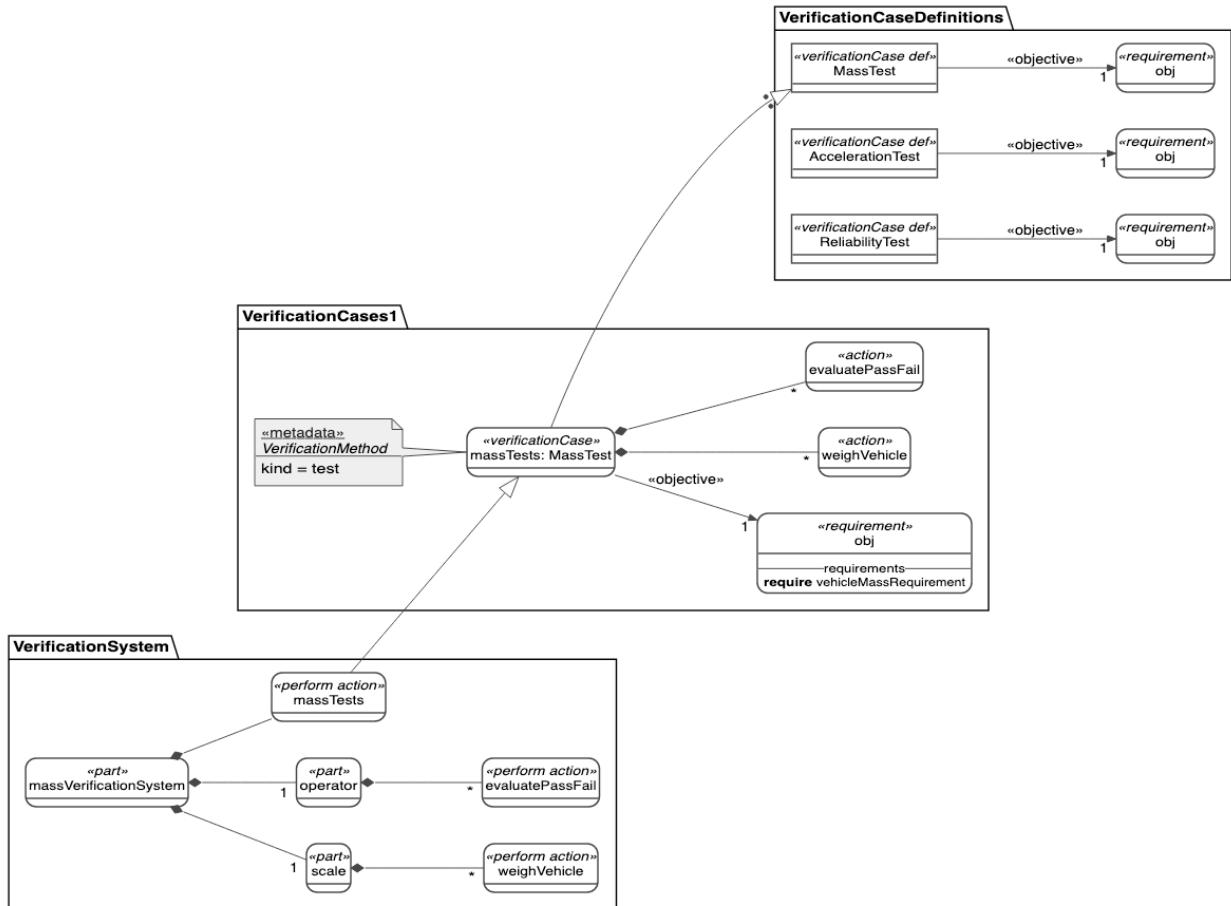


Figure 124. Vehicle Mass Verification Test

```

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 = VerificationMethodKind::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.

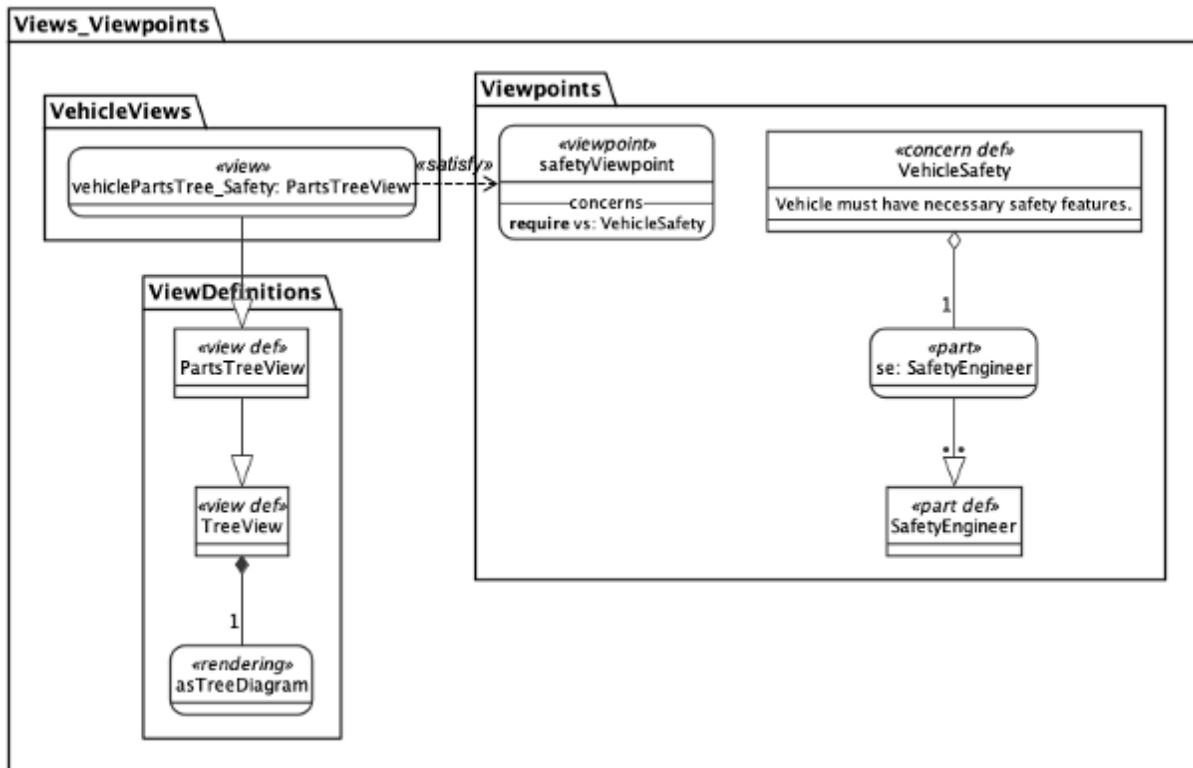


Figure 125. Vehicle Safety View

```

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::**;
    }
}

```

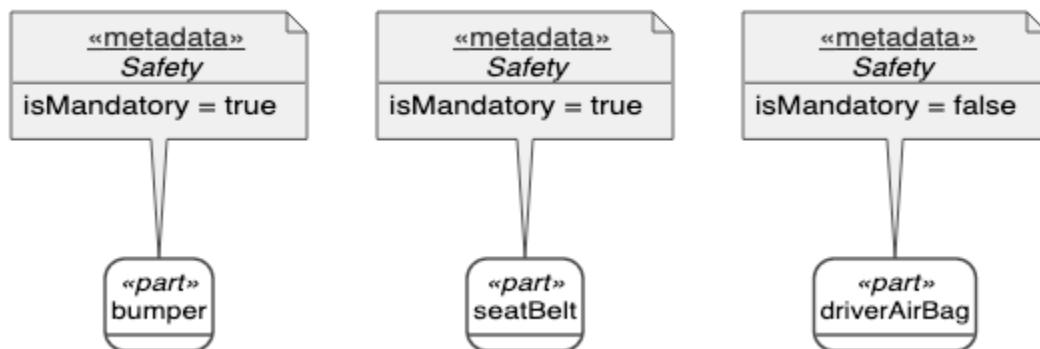


Figure 126. Rendering of view vehiclePartsTree_Safety

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. The choice of a selected variant at one variation point can constrain the available choices at another variation point. For this example, the choices are constrained to be a 4 cylinder engine with an manual transmission or a 6 cylinder engine with an automatic transmission.

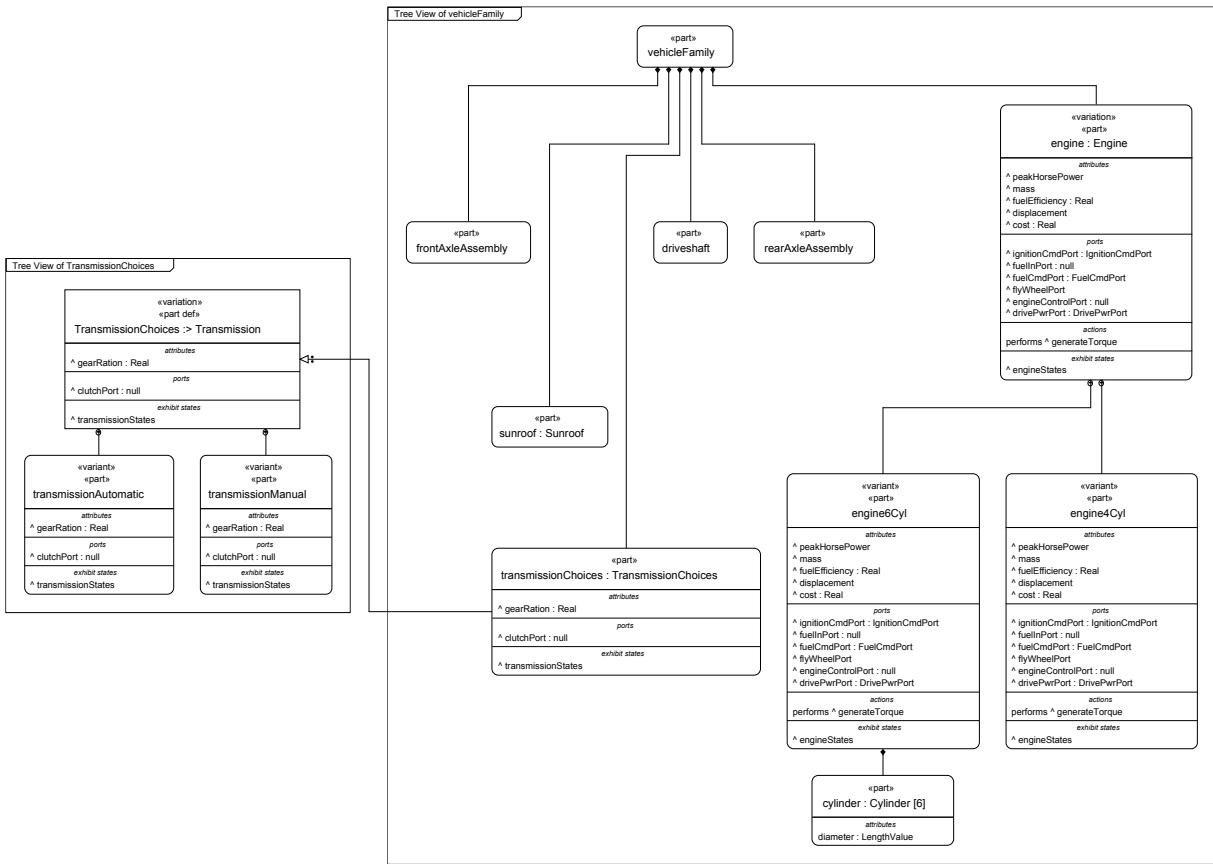


Figure 127. Variability Model for vehicleFamily

```

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` and inherit 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 `vehicle_1.frontAxleAssembly` is a usage of `FrontAxleAssembly_1`, whose `frontAxle` is a usage of `FrontAxle_1`, whose `wheels` are `Wheel_1` and `Wheel_2`. In this way, `vehicle_1` can decompose into a hierarchy of individual parts.

An individual definition and usage 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 context snapshot 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.

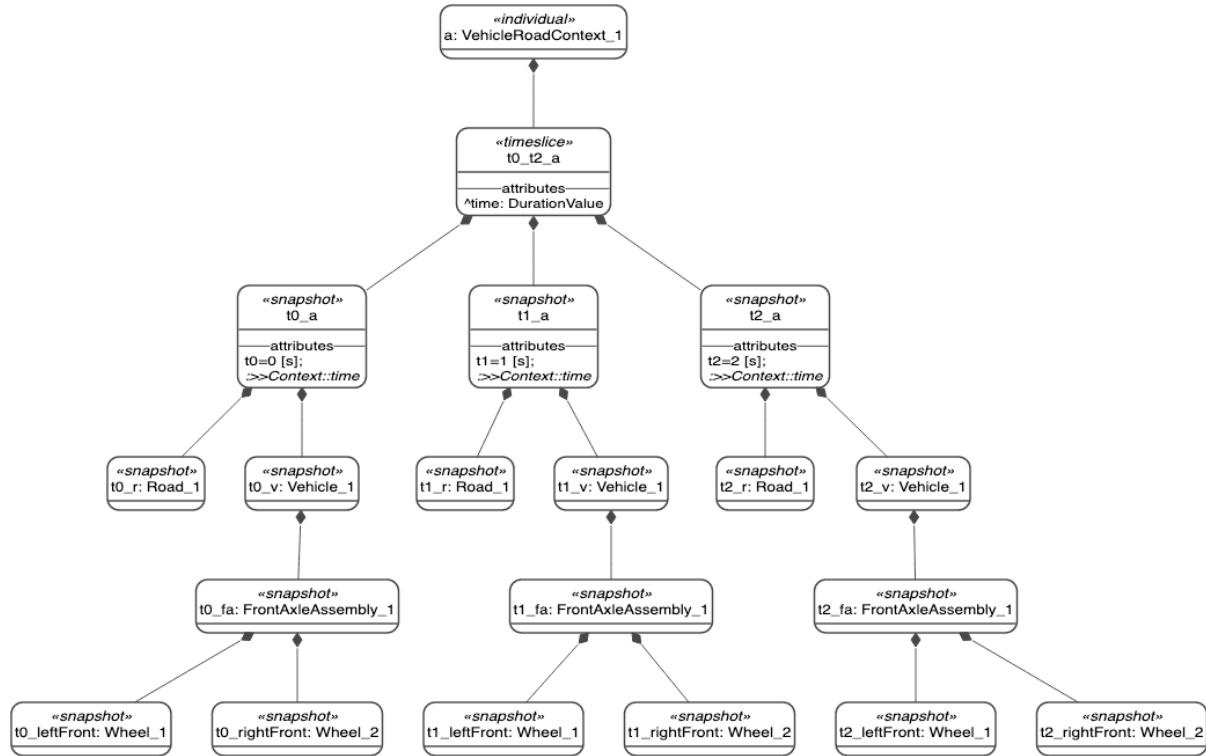


Figure 128. Vehicle Individuals and Snapshots

```

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:FrontAxleAssembly_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:FrontAxleAssembly_1{
            snapshot t2_leftFront:Wheel_1;
            snapshot t2_rightFront:Wheel_2;
        }
    }
}
}
}

```