Date: 14 November 2005

Systems Modeling Language (SysML) Specification

version 1.0 alpha

SysML Partners **(www.sysml.org)**†

---

†Includes the following Partners who have submitted Letters of Intent for the OMG's *UML for Systems Engineering RFP*: Telelogic AB, Motorola, Inc., and Gentleware AG.

GENERAL NOTICE

This document describes a proposed language specification developed by an open source project using an open source license for redistribution and use. In accordance with the open source license of the previous version of this specification (*System Modeling Language (SysML) Specification* v. 0.9, 10 January 2005), the copyright notice for this revision is followed by the copyright notice, terms, conditions, notices and disclaimers of the previous version, all of which also apply to this revision. A summary of the modifications to this version of the specification can be found in the Change Summary section of the Preface.

COPYRIGHT NOTICE

© 2005 Gentleware AG
© 2005 Motorola, Inc.
© 2005 Northrop Grumman
© 2005 PivotPoint Technology Corporation
© 2005 Telelogic AB

COPYRIGHT NOTICE FOR SysML v. 0.9

© 2003-2005 American Systems Corporation
© 2003-2005 ARTISAN Software Tools
© 2003-2005 BAE SYSTEMS
© 2003-2005 The Boeing Company
© 2003-2005 Ceira Technologies
© 2003-2005 Deere & Company
© 2003-2005 EADS Astrium GmbH
© 2003-2005 EmbeddedPlus Engineering
© 2003-2005 Eurostep Group AB
© 2003-2005 Gentleware AG
© 2003-2005 I-Logix, Inc.
© 2003-2005 International Business Machines
© 2003-2005 International Council on Systems Engineering
© 2003-2005 Israel Aircraft Industries
© 2003-2005 Lockheed Martin Corporation
© 2003-2005 Motorola, Inc.
© 2003-2005 Northrop Grumman
© 2003-2005 oose.de Dienstleistungen für innovative Informatik GmbH
© 2003-2005 PivotPoint Technology Corporation
© 2003-2005 Raytheon Company
© 2003-2005 Telelogic AB
© 2003-2005 THALES

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

This document describes a proposed language specification developed by an informal partnership of vendors and users, with input from additional reviewers and contributors. This document does not represent a commitment to implement any portion of this specification in any company's products. See the full text of this document for additional disclaimers and acknowledgments. The information contained in this document is subject to change without notice.

The specification proposes to customize the Unified Modeling Language (UML) specification of the Object Management Group (OMG) to address the requirements of Systems Engineering. These include many of the requirements requested by

the UML for Systems Engineering RFP, OMG document number ad/03-03-41. This document includes references to and excerpts from the *UML 2.0 Superstructure Specification* (OMG document number ptc/2004-10-02) and *UML 2.0 Infrastructure Specification* (Final Adopted Specification; OMG document number ptc/2003-09-15) with copyright holders and conditions as noted in those documents.

LICENSES

Redistribution and use of this specification, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of this specification must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- The Copyright Holders listed in the above copyright notice may not be used to endorse or promote products derived from this specification without specific prior written permission.
- All modified versions of this specification must include a prominent notice stating how and when the specification was modified.

THIS SPECIFICATION IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SPECIFICATION, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

TRADEMARKS

Systems Modeling Language and SysML, which are used to identify this specification, are not usable as trademarks since SysML Partners has established their usage to identify this specification without any trademark status or restriction. Organizations that wish to establish trademarks related to this specification should distinguish them somehow from SysML and Systems Modeling Language, for example by adding a unique prefix (e.g., OMG SysML).

Unified Modeling Language and UML are trademarks of the OMG. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

Systems Modeling Language and SysML, which are used to identify this specification, are not usable as trademarks since SysML Partners has established their usage to identify this specification without any trademark status or restriction. Organizations that wish to establish trademarks related to this specification should distinguish them somehow from SysML and Systems Modeling Language, for example by adding a unique prefix (e.g., OMG SysML).

Unified Modeling Language and UML are trademarks of the OMG. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

Pluralitas non est ponenda sine necessitate.

Plurality should not be posited without necessity.

— *William of Ockham (1285–1349)*

Preface

The Systems Modeling Language (SysML) continues to evolve as tool vendors and users gain experience in implementing and applying it to solve pragmatic systems engineering problems. The following sections summarize changes since the last public version of this specification (*System Modeling Language (SysML) Specification* v. 0.9, 10 January 2005), and provide information required by the OMG submission process.

0.1 CHANGE SUMMARY

The following is a summary of the changes since the last published SysML specification, based on extensive vendor implementation and user application experience:

- **General Improvements**
  - **Less is more.** The SysML has been reduced in size and complexity, yet its expressive power is demonstrably increased, as is evidenced by the enhanced revised Sample Problem, which is more mature and pragmatic than its predecessor. For example, the Sample Problem now addresses modeling Measures of Effectiveness and Trade Studies, which are essential for practicing systems engineers. See Appendix B, "Sample Problem" for details.
  - **The language architecture is refined and clarified.** SysML is now specified as a strict UML profile, so it is clear to both vendors and users which subset of UML constructs is reused, and what constructs have been added or modified by SysML. In addition, both normative and non-normative model libraries have been added in a straightforward and consistent way that makes it easier for vendors and users to futher customize the language for their special needs. See Chapter 6, "Language Architecture" for an overview of the language architecture.
  - **Executable models "just work."** A key benefit of the reduced complexity and the stricter compliance with UML 2.0 profile semantics is that the new Sample Problem is not only more sophisticated, it is also demonstrably executable. Stated otherwise, SysML v. 1.0 is not just yet another engineering drawing notation; it is an architecturally complete and fully executable language that can drive system engineering simulations. See "Support Documents" on page 8 for information about obtaining executable models.
  - **Alignment with other standards and best practices is increased.** Concepts and examples are better aligned with other system engineering standards and best practices, such as IEEE-Std-1471-2000 (*IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*), IEEE Std. 1220-1998: *IEEE Standard for Application and Management of the Systems Engineering Process,* OMG ptc/04-04-02: *UML 2.0 Testing Profile Specification.* See "Relationships to Other Standards" on page 8 for information about alignment with other standards.
  - **General information accessiblity is significantly improved.** The specification has been reorganized and rewritten to improve readability and consistency, and indices and table lists have been added to improve navigation.
- **Improvements to Structural Constructs**
  - Classes and Assemblies have been unified using the Block structural construct, and Flow Ports and Flow specifications have been added to specify input and output items that may include data as well as physical entities, such as fluids, solids, gases, and energy. See Chapter 8, "Blocks" for details.
  - Parametric Constraints are defined by extending UML Collaborations, which provide more natural semantics and distinctive notation for this new diagram type. An additional benefit of this approach is that SysML can now support the specification of pattern structures. See Chapter 9, "Parametric Constraints" for details.
  - The definition/usage dichotomy for structural constructs is made expliciit and applied consistently, which makes defining and applying Blocks and Parametric Constraints more intuitive and straightforward. See Part II -, "Struc-

tural Constructs" for an explanation.

- **Improvements to Behavioral Constructs**
  - Activities have been refined to reduce their complexity and increase consistency with the rest of the specification. See Chapter 10, "Activities" for details.
  - Interactions has been reduced to a subset of Sequence diagrams, which decreases the semantic overlap with Activities. See Chapter 11, "Sequences" for details.
- **Improvements to Cross-Cutting Constructs**
  - Requirements have been enhanced so that users can customize and assign classification categories, risks and verification methods. See Chapter 14, "Requirements" for details.
  - Allocation trace dependencies have been unified and simplified, and the content for tabular format is described in a non-prescriptive manner. See Chapter 15, "Allocations" for details.
  - Model management constructs have been added and include support for views and viewpoints in a manner compatible with EEE-Std-1471-2000 (*IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*). See Chapter 16, "Model Management" for details.
  - Normative types and enumerations used by the SysML profile are modularly defined in a separate package. See Chapter 17, "Types" for details.
- **Other improvements**
  - The complete abstract syntax for the SysML profile and the UML 2.0 metamodel reused by the profile are provided to facilitate understanding, validate architecture integrity, and faciliate implementation and model interchange using XMI and AP-233. See "Support Documents" on page 8 for information about obtaining the complete abstract syntax for the SysML profile.
  - Non-normative extensions and model libraries are defined in separate appendices. See Appendix C, "Non-Normative Extensions" and Appendix D, "Non-Normative Model Library"for details.
  - The Requirements Traceabilty Matrix for tracking SysML compliance with the UML for SE RFP is now generated with a requirements management tool, so the information is more complete, accurate and consistent. See Appendix G, "Requirements Traceability Matrix" for details.

0.2 OMG RFP RESPONSE

This Systems Modeling Language (SysML) Specification draft is being submitted to the OMG in response to the UML for SE RFP (omg document ad/03-04-04). Material in this submission that responds directly to the format required by the OMG submission process is localized within this section. By separating the information unique to the OMG technical process and submission format, and referencing the applicable portions of the technical specification, we are able to organize the specification in a form that can facilitate further stages of the OMG technology adoption and ISO Publicly Available Specification (PAS) processes.

The following SysML Partners have submitted Letters of Intent to the OMG to respond to its UML for SE RFP: Telelogic AB, Motorola, Inc., and Gentleware AG. A complete list of submitters and supporters is available at www.SysML.org/Partners.htm.

The information required by Section 4.9.2 ("Required Outline") of the UML for SE RFP is provided in the following parts.

0.2.1 Part I of RFP Response

Copyright Waiver and Trademark Usage

An unlimited number of copies of this document may be made by OMG or by OMG members in accordance with the Berkeley-style open source license described in the Licenses section that precedes this Preface. Note that the copyrights for this specification are shared by a group of companies, some of whom are not OMG members.

As noted in the Trademarks section that precedes this preface, Systems Modeling Language and SysML, which are used to identify this specification, are not usable as trademarks since SysML Partners has established their usage to identify this specification without any trademark status or restriction. Organizations that wish to establish trademarks related to this specification should distinguish them somehow from SysML and Systems Modeling Language (for example, by adding a unique prefix such as OMG SysML.

Submission contact points

The following person may be contacted for information regarding this submission:

Cris Kobryn (Cris.Kobryn@sysml.org OR Cris.Kobryn@telelogic.com)

In addition, the following public mailing list is available for providing feedback and requesting information about this specification: SysMLforum@googlegroups.com.

Guide to material in the submission

An overview of the background, goals and technical content of this proposal is described in Chapter 1 “Scope” of this document.

Overall design rationale

The design rationales for the language architecture and the specification approach used by this proposal are explained in Chapter 6, “Language Architecture” and Chapter 7, “Language Formalism”.

Statement of proof of concept

This proposed specification is being prototyped or implemented by more than one of the submitting organizations.

Resolution of RFP requirements and requests

The proposed specification makes use of existing OMG specifications and follows OMG guidelines in conformance with Section 5 “General Requirements on Proposals” of the RFP.

Section 6.5 “Mandatory Requirements” of the RFP requires a specific form of matrix that indicates how the proposed solution satisfies each of numbered requirements in the “Specific Requirements on Proposals” section of the RFP. Appendix G, “Requirements Traceability Matrix” and Chapter 2, “Compliance” address this requirement.

Response to RFP issues to be discussed.

Section 6.7 of the RFP, “Issues to be discussed” contains a single issue, which requests a sample problem description as follows:

> Submissions shall include models of one or more sample problems to demonstrate how their customization of UML for SE addresses the requirements of this RFP. The submitter may select one or more sample problems of their choosing, or apply their proposed solution to the sample problem descriptions included on the RFP page at http://syseng.omg.org/UML_for_SE_RFP.htm. The compliance matrix referred to in Section 6.5, must include a reference to the portion of the sample problem, which demonstrates how each requirement is being addressed.

The response to this “Issue to be discussed” is provided in Appendix B, “Sample Problem” of this document.

0.2.2 Part II of RFP Response

Proposed specification

The proposed specification is contained in the body of this document (including appendices). This specification includes both normative and explanatory material in a format that is largely self-contained, and which could be adopted and published in conformance with the OMG process.

Proposed compliance points

Proposed compliance points are described in Chapter 2 “Compliance” of this specification.

0.2.3 Part III of RFP Response

Summary of requests versus requirements

See “Resolution of RFP requirements and requests” on page iii.

Changes or extensions required to adopted OMG specifications

No changes or extensions are required.

Table of Contents

Preface i
0.1 CHANGE SUMMARY i
0.2 OMG RFP RESPONSE ii
0.2.1 Part I of RFP Response iii
0.2.2 Part II of RFP Response iv
0.2.3 Part III of RFP Response iv
Part I. Introduction 1
1 Scope 3
2 Compliance 3
2.1 Compliance to the SysML specification 3
2.2 Compliance of SysML to UML 5
3 References 7
3.1 Normative References 7
3.2 Non-Normative References 7
4 Terms and definitions 7
5 Additional information 8
5.1 Support Documents 8
5.2 Relationships to Other Standards 8
5.3 How to Read this Specification 9
5.4 Acknowledgements 9
6 Language Architecture 11
6.1 Design Principles 11
6.2 Package structure 11
6.3 Extension Mechanisms 20
6.4 4-Layer Metamodel Architecture 21
6.5 Alignment with XMI and AP-233 21
7 Language Formalism 23
7.1 Level of Formalism 23
7.2 Chapter Specification Structure 23
7.3 Use of Constraints 24
7.4 Use of Natural Language 24
7.5 Conventions and Typography 24
Part II - Structural Constructs 25
8 Blocks 27
8.1 Overview 27
8.2 Diagram elements 29
8.3 Package structure 32
8.4 UML extensions 33
8.4.1 Stereotypes 33
Block 33
FlowPort 34
FlowSpecification 35
ServicePort 35
8.4.2 Diagram extensions 35
Block Definition diagram 35
Internal Block diagram 36
8.5 Usage examples 36

9 Parametric Constraints ... 39
9.1 Overview ... 39
9.2 Diagram elements ... 40
9.3 Package structure ... 41
9.4 UML extensions ... 41
9.4.1 Stereotypes ... 41
Binding ... 41
ParametricConstraint ... 42
ParametricConstraintUse ... 43
9.4.2 Diagram extensions ... 43
Parametric diagram ... 43
9.5 Usage examples ... 43
Part III - Behavioral Constructs ... 47
10 Activities ... 49
10.1 Overview ... 49
10.2 Diagram elements ... 50
10.3 Package structure ... 56
10.4 UML extensions ... 57
10.4.1 Stereotypes ... 57
Continuous ... 57
ControlValue (a predefined enumeration) ... 58
ControlOperator ... 58
Discrete ... 58
NoBuffer ... 59
Overwrite ... 59
Optional ... 59
Probability ... 59
Rate ... 60
10.4.2 Diagram extensions ... 60
Activity ... 60
ControlFlow ... 61
ObjectNode ... 62
10.5 Usage examples ... 63
11 Sequences ... 71
11.1 Overview ... 71
11.2 Diagram elements ... 72
11.3 Package structure ... 75
11.4 UML extensions ... 75
11.5 Usage examples ... 75
12 State Machines ... 81
12.1 Overview ... 81
12.2 Diagram elements ... 82
12.3 Package structure ... 84
12.4 UML extensions ... 85
12.5 Usage examples ... 85
13 Use Cases ... 87
13.1 Overview ... 87
13.2 Diagram elements ... 88
13.3 Package structure ... 89
13.4 UML extensions ... 89
13.5 Usage examples ... 90

Part IV - Crosscutting Constructs 91
14 Requirements 93
14.1 Overview 93
14.2 Diagram elements 94
14.3 Package structure 95
14.4 UML extensions 95
14.4.1 Stereotypes 96
Derive 96
Requirement 97
RequirementKind (user defined enumeration) 97
RiskKind (user defined enumeration) 97
Satisfy 98
TestCase 98
Verdict (a user defined enumeration) 98
Verify 98
VerifyMethodKind (user defined enumeration) 99
14.4.2 Table extensions 99
14.5 Usage examples 99
15 Allocations 107
15.1 Overview 107
15.2 Diagram elements 107
15.3 Package structure 109
15.4 UML extensions 109
15.4.1 Stereotypes 109
Allocate 109
Allocated 110
15.4.2 Diagram extensions 110
15.4.3 Table extensions 111
15.5 Usage examples 111
16 Model Management 115
16.1 Overview 115
16.2 Diagram elements 115
16.3 Package structure 117
16.4 UML extensions 117
16.4.1 Stereotypes 118
Conform 118
View 118
Viewpoint 118
16.4.2 Diagram extensions 119
16.5 Usage examples 119
17 Types 121
17.1 Overview 121
17.2 Diagram elements 122
17.3 Package structure 123
17.4 UML extensions 123
17.4.1 Enumerations 123
ControlValue 123
RequirementKind 123
RiskKind 124
Verdict 124
VerifyMethodKind 124

17.4.2 DataTypes ... 125
Complex ... 125
Real ... 125
17.5 Usage examples ... 125
18 Auxiliary Constructs ... 127
18.1 Overview ... 127
18.2 Diagram elements ... 128
18.3 Package structure ... 130
18.4 UML extensions ... 130
18.4.1 Stereotypes ... 131
DistributedValue ... 131
Problem ... 131
Rationale ... 131
ValueProperty ... 132
ValueType ... 132
18.4.2 Diagram extensions ... 133
18.5 Usage examples ... 133
19 Profiles & Model Libraries ... 135
19.1 Overview ... 135
19.2 Diagram elements ... 136
19.3 Package Structure ... 139
19.4 UML extensions ... 139
19.4.1 Metaclass extensions ... 139
19.4.2 Diagram extensions ... 139
Stereotype ... 139
19.5 Usage examples ... 141
19.5.1 Defining a Profile ... 141
19.5.2 Adding Stereotypes to a Profile ... 142
19.5.3 Defining a Model Library that uses a Profile ... 143
19.5.4 Guidance on whether to use a Stereotype or Class ... 143
19.5.5 Using a Profile ... 144
19.5.6 Using a Stereotype ... 144
19.5.7 Using a Model Library Element ... 145
Part V - Appendices ... 147
Appendix A. Diagrams ... 149
A.1 Overview ... 149
Appendix B. Sample Problem ... 151
B.1 Overview ... 151
B.2 Problem Summary ... 151
B.3 Diagrams ... 151
B.3.1 Requirements Diagram for the “Hybrid SUV” ... 151
B.3.2 Trade Study and Measures of Effectiveness ... 152
B.3.3 Requirements Derivation ... 158
B.3.4 Requirements Verification ... 159
B.3.5 Use Case Diagram ... 160
B.3.6 Sequence Diagrams ... 161
B.3.7 Activity Diagram for “Control Power” ... 165
B.3.8 External Block Diagram for the Hybrid SUV ... 169
B.3.9 Transmission Properties ... 169
B.3.10 Allocations ... 171
B.3.11 Block Diagram ... 173

B.3.12 Interfaces ... 174
B.3.13 State Machine Diagram for the Transmission “Shift” behavior ... 175
B.3.14 Parametric Block Diagram ... 177
B.3.15 Requirements Satisfaction ... 178
B.3.16 Complete Traceability ... 179
Appendix C. Non-Normative Extensions... 183
C.1 Activities ... 183
C.1.1 Overview ... 183
C.1.2 Diagram Elements ... 183
C.1.3 Package Structure ... 184
C.1.4 UML Extensions ... 184
Stereotypes ... 185
Diagram Extensions ... 186
C.1.5 Usage Examples ... 187
C.2 Requirements... 188
C.2.1 Overview ... 188
C.2.2 Diagram elements ... 189
C.2.3 Package Structure ... 189
C.2.4 UML Extensions ... 190
Stereotypes 190
Diagram Extensions 192
C.2.5 Compliance Level ... 192
C.2.6 Usage Example ... 192
Appendix D. Non-Normative Model Library ... 193
D.1 Pre-defined ValueTypes... 193
D.2 Pre-defined Enumeration Literals... 194
Appendix E. OMG XMI Model Interchange... 197
E.1 Overview ... 197
Appendix F. ISO AP233 Model Interchange ... 199
F.1 Overview... 199
F.2 Background ... 199
F.3 Approach ... 202
F.3.1 Capturing Express models in UML ... 203
F.3.2 Converting Express models to UML ... 205
F.4 Model Alignment... 206
F.4.1 SysML Requirements Model ... 206
F.4.2 AP233 Requirements Model ... 208
F.4.3 Mapping Module: Requirements ... 215
F.4.4 Mapping between AP233 and SysML ... 216
F.5 Proof of Concept... 216
Appendix G. Requirements Traceability Matrix... 219
G.1 Overview ... 219

List of Figures

SysML Reuse of UML 2.0 5
Relationship of SysML Profile Package to UML and Prototypical User Model 13
Package Contents of SysML Profile 14
Package Contents of Non-Normative Model Libraries 15
Package Contents of Non-Normative Extensions 16
Package Contents of Default Enumerations 17
Package Contents of Blocks 18
Package Contents of UML2 Ruse for System Blocks 19
package Contents of Activities 20
Abstract syntax for Blocks (1 of 2). 33
Abstract syntax for Blocks (2 of 2). 33
FlowPort notation. 34
Service Port definition. 36
Service Port usage. 37
Flow Port definition. 37
Flow Port usage. 38
Abstract syntax for Parametric Constraints. 41
Parametric constraint definition using internal structure notation. 44
Parametric constraint definition using collaboration notation. 44
Parametric constraints on a Block diagram. 45
Parametric constraints on a Parametric diagram. 46
Abstract syntax for Activities. 57
Class diagram with activities as classes. 61
CallBehaviorAction notation.in activty diagram 61
Class diagram with activities as blocks associated with types of object nodes. 62
ObjectNode notation in activity diagrams. 62
ObjectNode notation in activity diagrams. 63
Activity Diagram Example: Control Power (1 of 4) 64
Activity Diagram Example: Control Power (2 of 4) 65
Activity Diagram Example: Control Power (3 of 4) 66
Activity Diagram Example: Control Power (4 of 4) 67
Continuous system example 1. 68
Continuous system example 2. 69
Continuous system example 3. 69
Top-level Sequence diagram for Drive Car 76
Accelerate scenario 77
Acceleration with allocations 78
State-Centric State Machine Diagram: Transmission “Switch Gear” Behavior 85
Transition-Centric State Machine Diagram: Transmission “Switch Gear” Behavior 86
Top-Level Use Cases for HybridSUV system. 90
Package structure for Requirements. 95
Abstract syntax for Requirements. 96
Decomposition of a compound requirement. 100
Requirements derivation using compact stereotype notation. 101
Requirements satisfaction. 102
Requirements verification. 103
Requirements traceability 104
Example of requirements trace dependency table 105
Abstract syntax for Allocations. 109

Block Definition Diagram:Allocation of behavior to the Transmission 112
Properties of the Transmission Block showing allocations 113
Example Allocation Table 114
Abstract syntax for Model Management. 117
Functional view conforming to its viewpoint. 119
Traceability across views: black box perspective 119
Traceability between Functional and Design views: white box perspective 120
Package structure for Requirements. 123
Abstract syntax for Auxiliary Constructs. 131
Block Definition diagram: Transmission properties 133
Internal Block diagram: Transmission properties 134
Defining a stereotype 140
Using a stereotype 140
Using stereotypes and showing values 141
Other notational forms for showing values 141
Definition of a profile 141
Profile Contents 142
Model libraries example 143
A model with applied profile and imported model library 144
Using two stereotypes on one model element 144
Using model library elements 145
SysML Diagram Taxonomy 149
Requirement Diagram:Top-Level User Requirements 152
Measures of Effectiveness for the Hybrid SUV 153
Key Performance Parameters for the Hybrid SUV 154
Usage of constraints to map KPPs to MoE scores for Acceleration MoE 155
Constraint definitions for Newton's Law 156
Block Definition diagram of composite parametric constraint IntergrateAndNormalize 157
Internal Block diagram of composite parametric constraint IntergrateAndNormalize 158
Requirement Diagram:Requirements Derivation 159
Requirement Diagram:Requirements Verification 160
Use Case Diagram 161
Sequence Diagram: Top Level "black-box" 162
Sequence Diagram: Accelerate Scenario 163
Sequence Diagram: Accelerate Scenario allocated to components of Hybrid SUV 164
Activity Diagram Example: Control Power (1 of 4) 165
Activity Diagram Example: Control Power (2 of 4) 166
Activity Diagram Example: Control Power (3 of 4) 167
Activity Diagram Example: Control Power (4 of 4) 168
Block Definition Diagram: Equipment Breakdown Structure 169
Block Definition Diagram: Properties of Transmission 170
External Block Diagram: Allocation of Behavior to the Transmission 171
Example tabular format of allocation traces 172
Block Diagram: Internal structure of the Power Subsystem 173
External Block Diagram: Command and Telemetry Interface Definitions 174
External Block Diagram: Flow Specification Definitions 175
State-Centric State Machine Diagram: Transmission "Switch Gear" Behavior 176
Transition-Centric State Machine Diagram: Transmission "Switch Gear" Behavior 177
Mass Constraints 178
Requirement Diagram: Requirement Satisfaction 179
Requirement Diagram: Requirement Traceability 180

Sample Traceability Table 181
Package Structure for SysML Activities 184
Abstract Syntax for Activity Non-Normative extensions. 185
Example activity with «EFFBD» stereotype applied 187
Example activity with «streaming» and «nonStreaming» stereotypes applied to subactivities. 187
Package Structure for SysML Requirements 189
Abstract Syntax for Effectiveness Metamodel. 190
Measures of Effectiveness. 192
AP233 and related protocols 200
AP233 Toplevel Architecture 201
Models in use for the SysML to AP233 alignment 203
Excerpt of the Express meta model 204
UML Profile for Express 205
Activities to derive the AP233 UML model from the Express model 206
SysML requirements model 207
Basic pattern for AP233 208
Property assignment 210
Representation of properties in AP233 211
Breakdown structure in AP233 213
System breakdown hierarchy 214
Mapping Module Requirements 215
Model-derived APIs 217
API abstraction layers 218

List of Tables

Graphical nodes for Blocks. 29
Graphical paths for Blocks. 31
Graphical nodes for Parametric Constraints. 40
Graphical nodes for Activities. 50
Graphical paths for Activities. 52
Other graphical elements included in Activity diagrams. 54
Graphical nodes for State Machines. 82
Graphical paths for State Machines. 84
Graphical nodes for Use Cases. 88
Graphical paths for Use Cases. 88
Graphical nodes for Requirements. 94
Graphical paths for Requirements. 94
Graphical nodes for Allocations. 107
Graphical paths for Allocations. 108
Graphical nodes for Model Management. 115
Graphical paths for Model Management. 116
Graphical nodes for Requirements. 122
Graphical nodes for Auxiliary Constructs. 128
Graphical paths for Auxiliary Constructs. 129
Graphical nodes for Profiles 136
Graphical paths for Profiles 136
Graphical nodes for Profiles 138
Graphical nodes for non-normative extensions to Activities 183
Graphical nodes included in effectiveness element 189
Graphical paths for effectiveness. 189

Part I. Introduction

This specification defines a general-purpose modeling language for systems engineering applications, called the Systems Modeling Language (SysML). SysML supports the specification, analysis, design, verification and validation of a broad range of complex systems. These systems may include hardware, software, information, processes, personnel, and facilities.

The orgins of the SysML initiative can be traced to a strategic decision by the International Council on Systems Engineering's (INCOSE) Model Driven Systems Design workgroup in January 2001 to customize the Unified Modeling Language (UML) for systems engineering applications. This resulted in a collaborative effort between INCOSE and the Object Management Group (OMG), which maintains the UML specification, to jointly charter the OMG Systems Engineering Domain Special Interest Group (SE DSIG) in July 2001. The SE DSIG, with support from INCOSE and the ISO AP 233 workgroup, developed the requirements for the modeling language, which were subsequently issued by the OMG as part of the UML for Systems Engineering Request for Proposal (UML for SE RFP; OMG document ad/03-03-41) in March 2003.

Currently it is common practice for systems engineers to use a wide range of modeling languages, tools and techniques on large systems projects. In a manner similar to how UML unified the modeling languages used in the software industry, SysML is intended to unify the diverse modeling languages currently used by systems engineers.

Since SysML is being defined as a UML 2.0 Profile, it is able to reuse the relatively mature notation and semantics of a second generation modeling language. In addition, systems engineers modeling with SysML 1.0 and software engineers modeling with UML 2.0 will be able to collaborate when modeling software-intensive systems. This will improve communication among the various stakeholders who participate in the systems development process and promote interoperability among modeling tools. It is anticipated that SysML will be customized to model domain specific applications, such as automotive, aerospace, communications and information systems.

The next following chapters describe the SysML language architecture and the specification formalism used to define SysML.

1 Scope

The purpose of this document is to specify the Systems Modeling Language (SysML), a general-purpose modeling language for systems engineering. This specification documents the concrete syntax (notation), abstract syntax, semantics, and design rationales for SysML, and provides examples of how it can be used to solve common systems engineering problems. Its intent is to specify the language so that systems engineering modelers can learn to apply and use it, modeling tool vendors can implement and support it, and both can provide constructive feedback to improve future versions. The following public mailing list is available for providing feedback and requesting information about this specification: SysMLforum@googlegroups.com.

SysML is designed to provide simple but powerful constructs for modeling a wide range of systems engineering problems. **This first version of SysML is particularly effective in specifying requirements, system structure, functional behavior, allocations, basic testing and basic trade studies during the specification and design phases of systems engineering. It does not support decision trees, comprehensive testing, comprehensive trade studies or fully executable functional behavior, although we encourage users and vendors to experiment in these areas and let us know about their experiences.** We expect that these language shortcomings will be addressed in future version of SysML as we gain more experience implementing and applying it.

SysML is being aligned with two evolving interoperability standards: the OMG XMI model interchange standard for UML modeling tools and the ISO AP-233 data interchange standard for systems engineering tools. While the details of this alignment are beyond the scope of this specification, basic information about the XMI and AP-233 interoperability standards can be found in and relevant references are furnished in Appendix E, “OMG XMI Model Interchange” and Appendix F, “ISO AP233 Model Interchange”.

The following sections provide background information about this specification, including information about compliance and a glossary of terms. A guide for both systems engineers and vendors who read this specification is provided in Section 5.3, ’How to Read this Specification’. The main body of this document (Parts II-IV) describes the normative technical content of the specification. The appendices include non-normative technical content that will aid in the understanding, implementation, and application of this specification.

2 Compliance

As with UML, the basic units of compliance for SysML are the packages that define the SysML profile, which are described in Chapter 6, “Language Architecture”. Since SysML is defined as a strict Profile of UML, there are two kinds of compliance that need to be addressed:

- Compliance to the SysML specification. The kind of compliance is concerned with defining the extent to which a SysML tool implements this specification.
- Compliance of the SysML specification to the UML specification. The kind of compliance is concerned with defining the extent to which the SysML specification reuses the UML specification.

2.1 Compliance to the SysML specification

All SysML language constructs are categorized into two levels of compliance, Basic and Advanced, as is explained in Chapter 7, “Language Formalism”. Consequently, it is natural to organize SysML compliance into Basic and Advanced levels, as is shown in Table 1 and Table 2, respectively. Compliance within a level is further decomposed by the packages that define the major diagram types. The following compliance options are valid for each level or package:

- **no** compliance: Implementation does not comply with the concrete syntax, abstract syntax, well-formedness rules, semantics of the package.

- **full** compliance: Implementation fully complies with the concrete syntax, abstract syntax, well-formedness rules, semantics of the package.
- **XMI** compliance: Implementation complies with XMI model interchange of the package.
- **AP233** compliance: Implementation complies with AP-233 model interchange of the package.

For an implementation of SysML to comply with a particular SysML package requires complying with any packages on which the particular package depends. Since SysML is defined as a strict profile of UML, this includes not only other SysML packages, but all UML packages on which the SysML package depends.

Table 1 Summary of SysML Compliance Points: Basic

Compliance Level	Compliance Point	Valid Options
Basic SysML	All packages, all Basic level constructs	no, full, XMI, AP233
Basic	Activities, Basic level constructs	no, full, XMI, AP233
Basic	Allocations, Basic level constructs	no, full, XMI, AP233
Basic	Auxiliary Constructs, Basic level constructs	no, full, XMI, AP233
Basic	Blocks, Basic level constructs	no, full, XMI, AP233
Basic	Sequences, Basic level constructs	no, full, XMI, AP233
Basic	Parametric Constraints, Basic level constructs	no, full, XMI, AP233
Basic	Requirements, Basic level constructs	no, full, XMI, AP233
Basic	State Machines, Basic level constructs	no, full, XMI, AP233
Basic	Use Cases, Basic level constructs	no, full, XMI, AP233

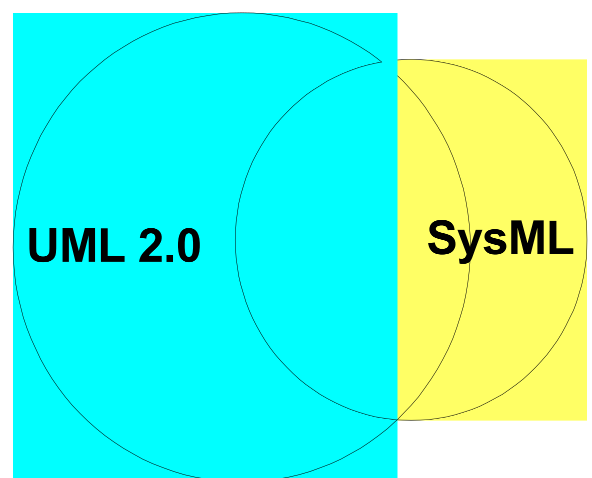
.

Table 2 Summary of SysML Compliance Points: Advanced

Compliance Level	Compliance Point	Valid Options
Advanced	All packages, all Advanced level constructs	no, full, XMI, AP233
Advanced	Activities, Advanced level constructs	no, full, XMI, AP233
Advanced	Allocations, Advanced level constructs	no, full, XMI, AP233
Advanced	Axuiliary Constructs, Advanced level constructs	no, full, XMI, AP233
Advanced	Blocks, Advanced level constructs	no, full, XMI, AP233
Advanced	Sequences, Advanced level constructs	no, full, XMI, AP233
Advanced	Parametric Constraints, Advanced level constructs	no, full, XMI, AP233
Advanced	Requirements, Advanced level constructs	no, full, XMI, AP233
Advanced	State Machines, Advanced level constructs	no, full, XMI, AP233
Advanced	Use Cases, Advanced level constructs	no, full, XMI, AP233

2.2 Compliance of SysML to UML

In order to better understand the relationship between the UML and SysML languages, consider the Venn diagram shown in Figure 1-1, where the sets of language constructs that comprise the UML and SysML languages are shown as the circles marked “UML 2.0” and “SysML”, respectively. The intersection of the two circles indicates the common diagrams that SysML and UML share, which are listed in the legend portion of the figure. The region marked “New Diagrams” in the figure indicates the new diagram types defined for SysML that have no counterparts in UML. Note that a significant part of UML is not required to implement SysML, which results in a smaller language that is easier to learn, implement and apply.



Common diagrams: Activities, Block Definitions (UML2::Classes), Internal Blocks (UML2::Composite Structures), Sequences, State Machines, Use Cases

New diagrams: Allocations, Parametric Constraints, Requirements

Figure 1-1. SysML Reuse of UML 2.0

The compliance matrix in Table 3 below specifies the UML 2 Superstructure packages that a SysML tool must reuse in order to implement SysML. Stated otherwise, these UML 2 Superstructure packages must be available for any SysML implementation. The valid options are shown below. The Package Structure in the individual chapters also shows which packages are required for SysML.

- **no**: SysML does not require this UML package. However, SysML is intended to be compatible with the package if it is used.
- **partial**: SysML only requires selected classes from this UML package.

- **complete:** SysML requires this complete UML package.

Table 3 UML 2.0 Superstructure Packages Required for SysML

UML Compliance Level	Compliance Point	UML Package Required for SysML
Basic (Level 1)	All packages	complete
Intermediate (Level 2)	Actions::IntermediateActions	partial
Intermediate (Level 2)	Activities:: IntermediateActivities	partial
Intermediate (Level 2)	Activities:: StructuredActivities	partial
Intermediate (Level 2)	CommonBehaviors:: Communications	partial
Intermediate (Level 2)	CommonBehaviors::Time	partial
Intermediate (Level 2)	Components::BasicComponents	no
Intermediate (Level 2)	CompositeStructures::InvocationActions	partial
Intermediate (Level 2)	CompositeStructures::Ports	partial
Intermediate (Level 2)	CompositeStructures:: StructuredClasses	partial
Intermediate (Level 2)	Deployments::Artifacts	no
Intermediate (Level 2)	Deployments::Nodes	no
Intermediate (Level 2)	Interactions::Fragments	partial
Intermediate (Level 2)	Profiles	partial
Intermediate (Level 2)	StateMachines:: BehaviorStateMachines	partial
Intermediate (Level 2)	StateMachines:: MaximumOneRegion	no
Complete (Level 3)	Actions::CompleteActions	partial
Complete (Level 3)	Activities::CompleteActivities	partial
Complete (Level 3)	Activities:: CompleteStructuredActivities	partial
Complete (Level 3)	Activities:: ExtraStructuredActivities	partial
Complete (Level 3)	AuxiliaryConstructs:: InformationFlows	partial
Complete (Level 3)	AuxiliaryConstructs:: Models	partial
Complete (Level 3)	AuxiliaryConstructs::Templates	no

Table 3 UML 2.0 Superstructure Packages Required for SysML

Complete (Level 3)	Classes:: AssociationClasses	no
Complete (Level 3)	Classes:: PowerTypes	no
Complete (Level 3)	CompositeStructures:: Collaborations	partial
Complete (Level 3)	Components:: PackagingComponents	no
Complete (Level 3)	Deployments:: ComponentDeployments	no
Complete (Level 3)	StateMachines::ProtocolStateMachines	no

3 References

3.1 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references, subsequent amendments to, or revisions of, any of these publications do not apply.

- [UML 2005] OMG formal/05-07-04: *Unified Modeling Language: Superstructure* version 2.0

3.2 Non-Normative References

The following non-normative references have proven useful in developing concepts and examples for this specification:

- [IEEE-Std-1471 2000] IEEE-Std-1471-2000: *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*
- [IEEE-Std-1220 1998] IEEE Std. 1220-1998: *IEEE Standard for Application and Management of the Systems Engineering Process*
- [UMLforTesting 2004] OMG ptc/04-04-02: *UML 2.0 Testing Profile Specification*

4 Terms and definitions

For the purposes of this specification, the terms and definitions given in the following apply. The lexicon is a synthesis of terms and definitions from various sources, including, but not limited to, previous UML glossaries, the UML for SE RFP, and the INCOSE MDSD Concept Model Semantic Dictionary.

The following conventions are used in the term definitions below:

- The entries usually begin with a lowercase letter. An initial uppercase letter is used when a word is usually capitalized in standard practice. Acronyms are all capitalized, unless they traditionally appear in all lowercase.

- When one or more words in a multi-word term is enclosed in brackets, it indicates that those words are optional when referring to the term. For example, *use case [class]* may be referred to as simply *use case*.
- A phrase of the form "Contrast: <term>" refers to a term that has an opposed or substantively different meaning.
- A phrase of the form "See: <term>" refers to a related term that has a similar, but not synonymous meaning.
- A phrase of the form "Synonym: <term>" indicates that the term has the same meaning as another term, which is referenced.
- A phrase of the form "Acronym: <term>" indicates that the term is an acronym. The reader is usually referred to the spelled-out term for the definition, unless the spelled-out term is rarely used.)

Note: SysML v. 0.9 reviewer feedback found the previous version of this glossary to contain many definitions that were not meaningful to practicing system engineers, and numerous others that were inconsistent with the normative part of this specification, as well as itself. Consequently, the glossary is being rewritten so that all definitions will be meaningful to domain experts, and are consistent with the rest of the specification. The updated glossary will be included in the SysML v. 1.0 beta update.

---

5 Additional information

5.1 Support Documents

The following support documents and models are relevant to this specification:

- *Compliance and Requirements Traceability for SysML v. 1.0a*: A statement of compliance and a Requirments Traceability Matrix (RTM) that shows how this specification satisfies the requirements of the UML for Systems Engineering RFP. See Appendix G, "Requirements Traceability Matrix". [.DOC format]
- *Abstract Syntax for SysML v. 1.0a:* Complete abstract syntax for the SysML profile and the UML 2.0 metamodel reused by the profile is provided to facilitate understanding, validate architecture integrity, and faciliate implementation and model interchange using XMI and AP-233. Available in multiple formats [HTML, .XMI, tool-specific].
- *Sample Problem Model for SysML v. 1.0a*: Executable and non-executable implementations of SysML v. 1.0 Sample Problem in multiple formats [tool-specific, HTML]. See Appendix B, "Sample Problem".

These suppport documents are available for download from the Artifacts page of the SysML web (www.SysML.org/artifacts.htm).

5.2 Relationships to Other Standards

SysML is defined as an extension of the OMG *UML 2.0 Superstructure Specification* (OMG document number ptc/2004-10-02). If SysML requires any changes to this UML specification, they will be described in a future version of this document.

SysML is also being aligned with two evolving interoperability standards: the ISO AP-233 data interchange standard for systems engineering tools and the OMG XMI 2.0 model interchange standard for UML 2.0 modeling tools. While the details of this alignment are beyond the scope of this specification, overviews of alignment issues and relevant references are furnished in Appendix E, "OMG XMI Model Interchange" and Appendix F, "ISO AP233 Model Interchange".

SysML supports the OMG's Model Driven Architecture initiative by its reuse of the UML standard, and its architectural alignment with the OMG XMI 2.0 and ISO AP-233 interoperability standards.

A partial listing of other system engineering standards and best practices that have influenced the development of this specification is included in 3.2, 'Non-Normative References'.

5.3 How to Read this Specification

This specification is intended to be read by systems engineers so that they may learn and apply SysML, and by modeling tool vendors so that they may implement and support SysML. As background all readers are encouraged to first read Part I "Introduction".

After reading the introduction, readers should be prepared to explore the user-level constructs defined in the next three parts: Part II - "Structural Constructs", Part III - "Behavioral Constructs", and Part IV - "Crosscutting Constructs". Systems engineers should read the Overview, Diagram Elements and Usage Examples sections in each chapter, and explore the Package Structure and UML Extension sections as they see fit. Modeling tool vendors should read all sections. In addition, Systems engineers who want to understand how to apply the language and assess its coverage should read the Appendix B, "Sample Problem" and Appendix G, "Requirements Traceability Matrix" respectively.

Although the chapters are organized into logical groupings that can be read sequentially, this is a reference specification and is intended to be read in a non-sequential manner.

5.4 Acknowledgements

The following companies and organizations have contributed to the development of this specification:

- Industry
  - BAE SYSTEMS
  - Boeing
  - Deere & Company
  - EADS Astrium
  - Eurostep
  - Israel Aircraft Industries
  - Lockheed Martin Corporation
  - Motorola
  - Northrop Grumman
  - oose.de Dienstleistungen für innovative Informatik GmbH
  - Raytheon
  - THALES
- Government
  - NASA/Jet Propulsion Laboratory
  - National Institute of Standards and Technology (NIST)
  - DoD/Office of the Secretary of Defense (OSD)
- Vendors

  - ARTISAN Software Tools
  - Ceira Technologies
  - EmbeddedPlus Engineering
  - Gentleware
  - IBM
  - I-Logix
  - Mentor Graphics
  - PivotPoint Technology
  - Telelogic
  - Structured Software Systems Limited
  - Vitech
- Liaisons
  - Consultative Committee for Space Data Systems (CCSDS)
  - Embedded Architecture and Software Technologies (EAST)
  - International Council on Systems Engineering (INCOSE)
  - ISO STEP AP-233
  - Systems Level Design Language (SLDL) and Rosetta

The following persons were members of the specification team that designed and wrote this specification: Vincent Arnould, Laurent Balmelli, Ian Bailey, James Baker, Conrad Bock, Carolyn Boettcher, Roger Burkhart, Murray Cantor, Bruce Douglass, Harald Eisenmann, Anders Ek, Brenda Ellis, Marilyn Escue, Sanford Friedenthal, Eran Gery, Drora Goshen, Hal Hamilton, Dwayne Hardy, James Hummel, Cris Kobryn, Michael Latta, Robert Long, Alan Moore, Veronique Normand, Salah Obeid, Dave Oliver, David Price, Chris Sibbald, Joseph Skipper, Rick Steiner, Robert Thompson, Lars Tufvesson, Jim U'Ren, Thomas Weigert, Tim Weilkiens, and Brian Willard.

In addition, the following persons contributed valuable ideas and feedback that significantly improved the content and the quality of this specification: Perry Alexander, Randy Bruce, Michael Jesse Chonoles, Mike Dickerson, Orazio Gurrieri, Julian Johnson, Dave Kosan, Bryan Lima, Jim Long, Henrik Lönn, Jamel Marzouki, Jim Schier, Matthias Weber, Bran Selic, and Peter Shames.

6 Language Architecture

This first version of the SysML specification is defined as strict Profile of UML. The advantage of this approach is that SysML reuses and extends proven modeling constructs from UML, the industry standard for modeling software intensive systems. Consequently, a large number of system and software engineers who already know UML can easily learn SysML, and the large installed base of UML tools can be readily adapted to support the SysML profile. The disadvantage of this approach is that, in some limited cases, constructs that were originally designed for software intensive systems may be less intuitive for system engineers who lack UML or object paradigm experience. When considering the tradeoffs, the SysML language architects determined that the advantages far outweighed the disadvantages, and decided to design the initial version as a strict UML Profile. This design decision will be revisited in the future if, after more experience with SysML implementations and applications it becomes apparent that SysML requires its own metamodel (cf. reusing the UML metamodel via the Profile mechanism).

This following sections describe the design principles and package structure of the SysML language.

6.1 Design Principles

The following fundamental design principles have guided the development of SysML:

- **Parsimony**: SysML is based on a subset of UML that economically satisfies the basic requirements of the systems engineering community as defined in the UML for SE RFP. Additional constructs and diagram types are added to this UML subset only as needed to address derived system engineering requirements discovered during the language specification process. This disciplined application of Occam's razor results in a more concise, yet more semantically expressive language which is easier to learn, implement and apply.
- **Reuse**: SysML strictly reuses UML constructs wherever practical, and when modifications to UML are required, they are done in a manner that strives to minimize changes to the underlying language. Consequently, SysML is intended to be straightforward for UML vendors to implement.
- **Modularity**: The principle of strong cohesion and loose coupling is applied to organize normative and non-normative language constructs into stereotype extension and model library packages.
- **Layering**: Layering is used to organize the SysML profile in two ways. First, since SysML is defined as strict UML Profile, all SysML packages may be considered an extension layer of the underlying UML metamodel. Second, a SysML language constructs are organized into two levels of compliance, Basic and Advanced, which constitutes an additional layering.
- **Partitioning**: Partitioning is used to organize conceptual areas within the same layer. SysML's package structure, which is explained in the following section, partitions the SysML profile into packages that correspond to the language's major diagram types. This partitioning is largely isomorphic with UML's package structure, and is intended to facilitate reuse and implementation.
- **Extensibility**: SysML supports the same extension mechanisms furnished by UML (metaclasses, stereotypes, model libraries), so that the language can be further extended for specific systems engineering domains, such as automotive, aerospace, manufacturing and communications.
- **Interoperability**: SysML is aligned with the semantics of the ISO AP-233 data interchange standard to support interoperability among engineering tools, and inherits the XMI interchange from UML.

6.2 Package structure

The relationship of the *SysML* «profile» package to other packages, such as the *UML 2 Reused* «metamodel» package, which represents the subset of UML 2.0 Superstructure model elements that are imported and reused, is shown in Figure 6-2. The prototypical *UserModel* package can apply the *SysML* «profile» package directly, and it can also import a *Non-Normative*

Model Libraries package that applies the the *SysML* «profile» package. Furthermore, in addition to applying the normative *SysML* «profile» package, the *UserModel* package can optionally apply the *Non-Normative Extensions* «profile» package.

The subpackages contained in the *SysML* «profile», the *Non-Normative Model Libraries* «modelLibrary», and the *Non-Normative Extensions* «profile» packages are shown in Figure 6-3, Figure 6-4 and Figure 6-5, respectively. Figure 6-6 shows a further drill-down into the contents of the DefaultEnumerations package, which is a subpackage of the the *Non-Normative Model Libraries* «modelLibrary» shown in Figure 6-4.

Each of the subpackages of the the *SysML* «profile» package shown in Figure 6-3 is further decomposed into *Basic* and *Advanced* subpackages, which correspond to the Basic and Advanced compliance levels of this specification. (See Section 2, “Compliance,” on page 3). In those cases where the package contains only Basic constructs , the Advanced subpackage is omitted. Figure 6-7 shows the contents of the Blocks package, which contains only a Basic subpackage that imports the *UML2 Reuse for SysML Blocks* package, the contents of which are shown in Figure 6-8. The *UML2 Reuse for SysML Blocks* subpackages constain the UML 2 metamodel elements that are stereotyped by the Blocks package. Figure 6-9 shows the subpackages of the Activities package, which contains both Basic and Advanced constructs.

In this manner the *SysML* «profile» package is organized and applied. Although a further exploration of the SysML package structure is outside the scope of this overview, the reader who is interested in learning more details is encouraged to explore the package structure of the *Abstract Syntax for SysML v. 1.0a* support document. See Section 5.1, “Support Documents,” on page 8.

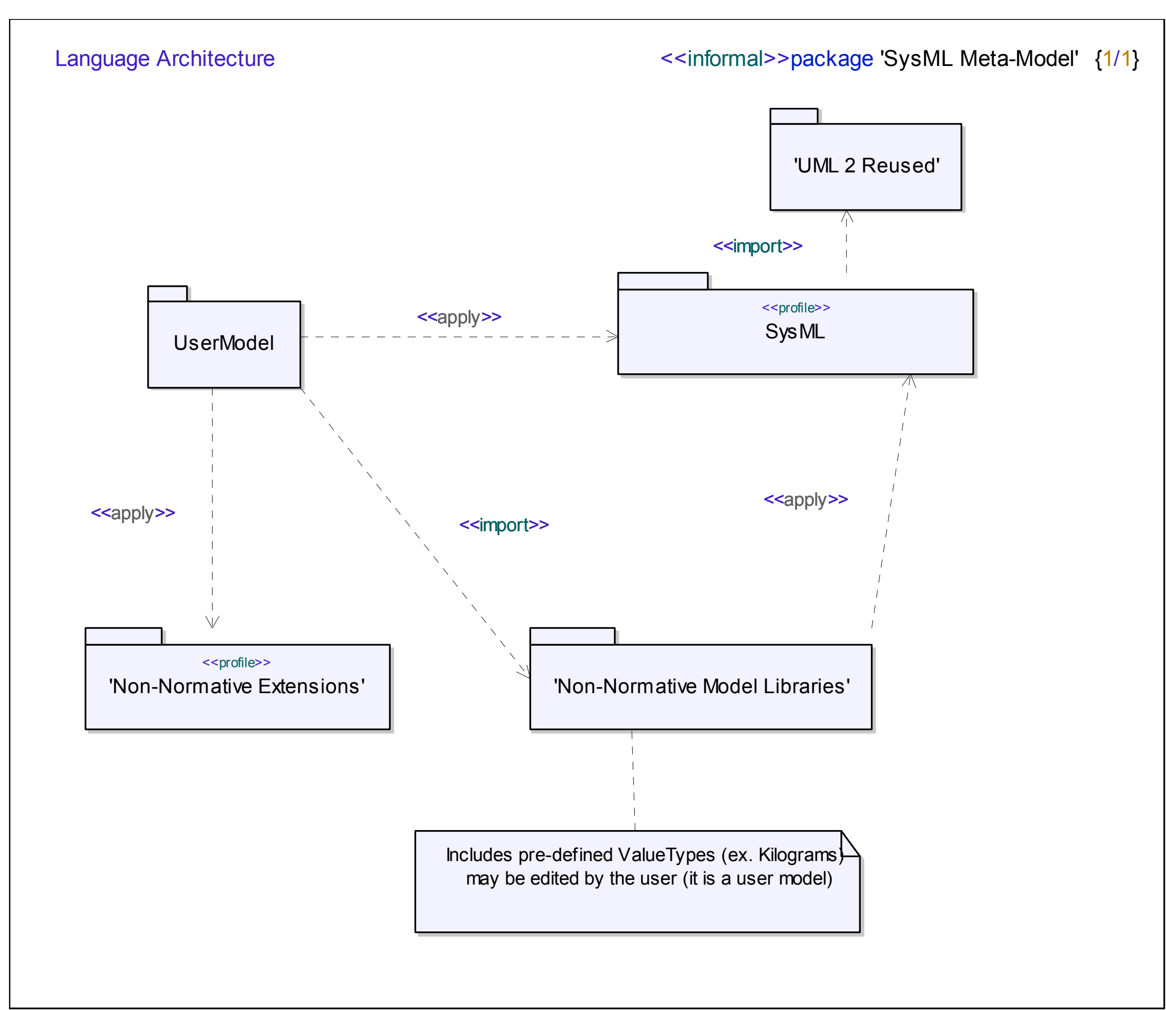


Figure 6-2. Relationship of SysML Profile Package to UML and Prototypical User Model

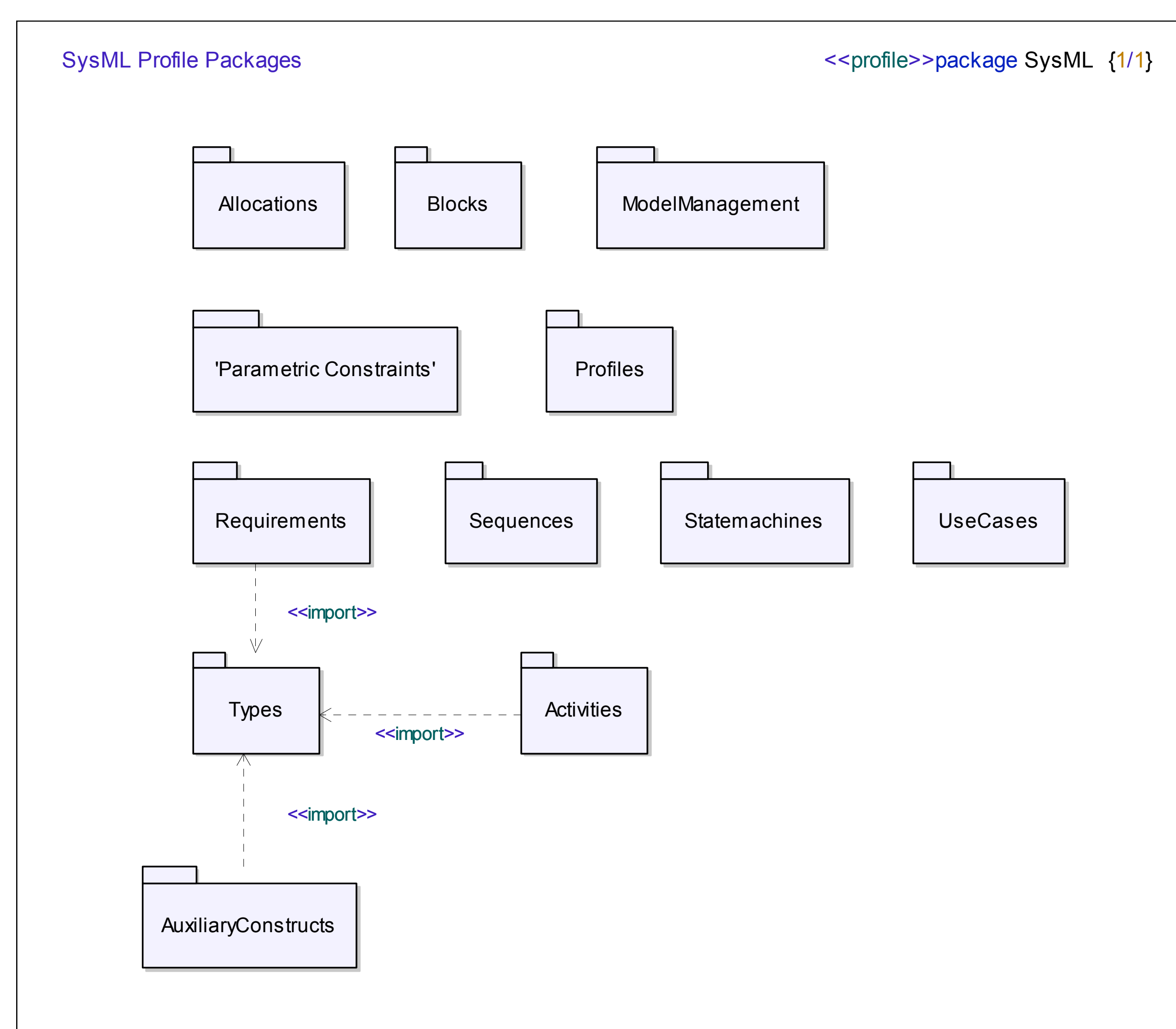


Figure 6-3. Package Contents of SysML Profile

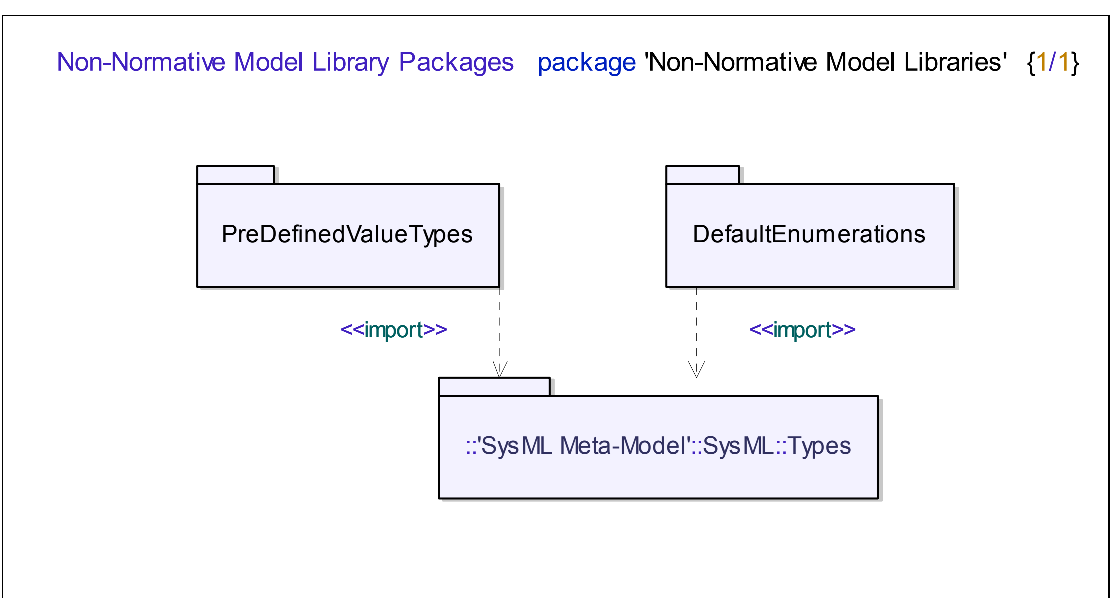


Figure 6-4. Package Contents of Non-Normative Model Libraries

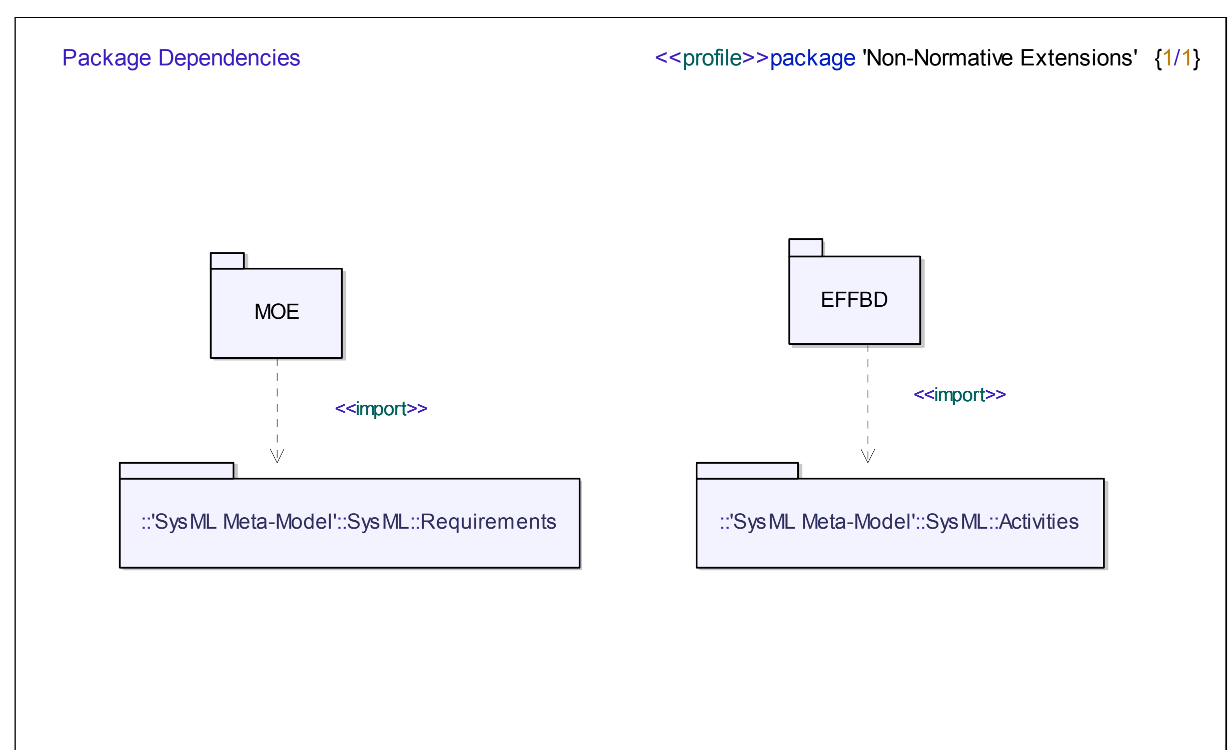


Figure 6-5. Package Contents of Non-Normative Extensions

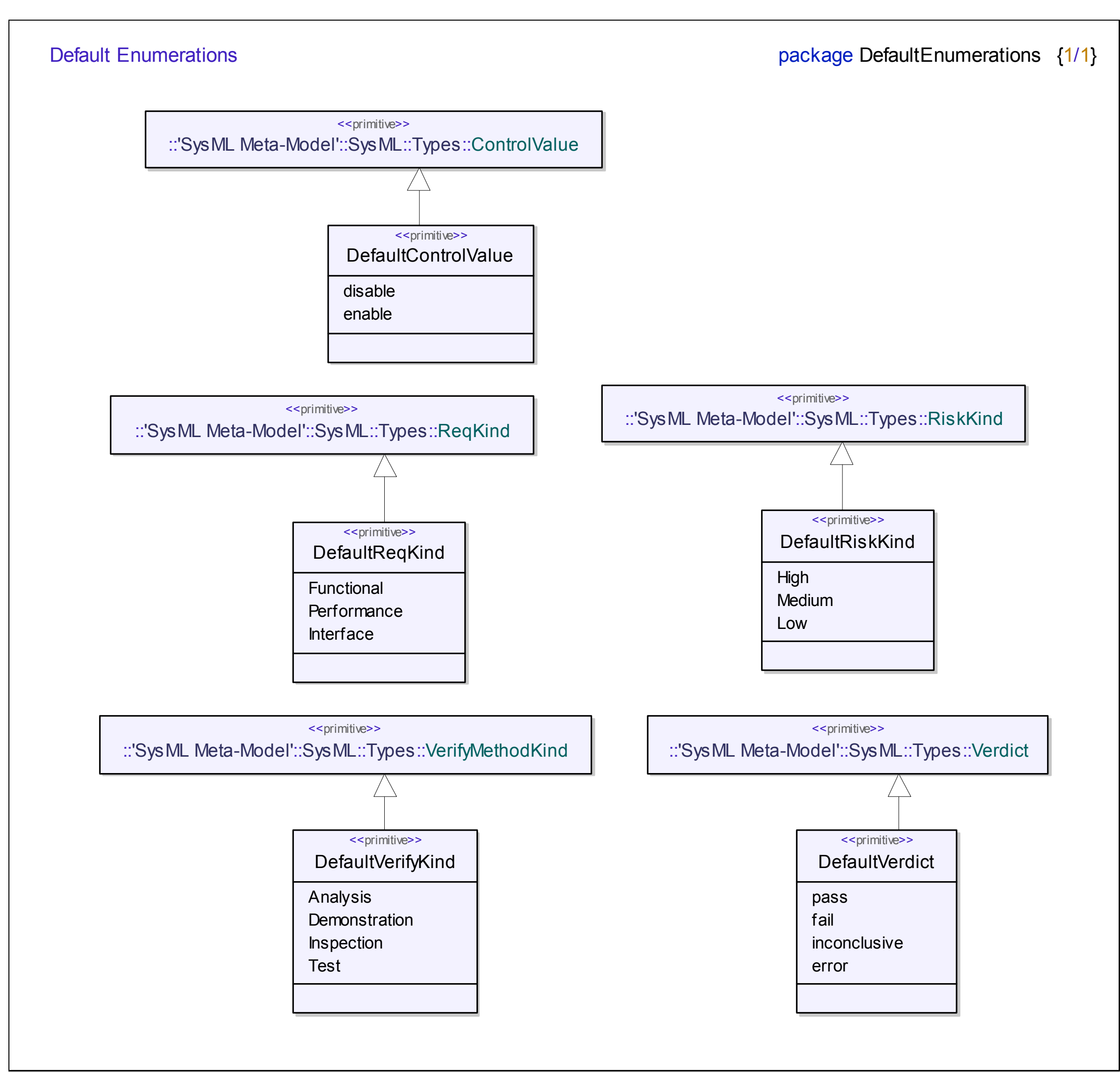


Figure 6-6. Package Contents of Default Enumerations

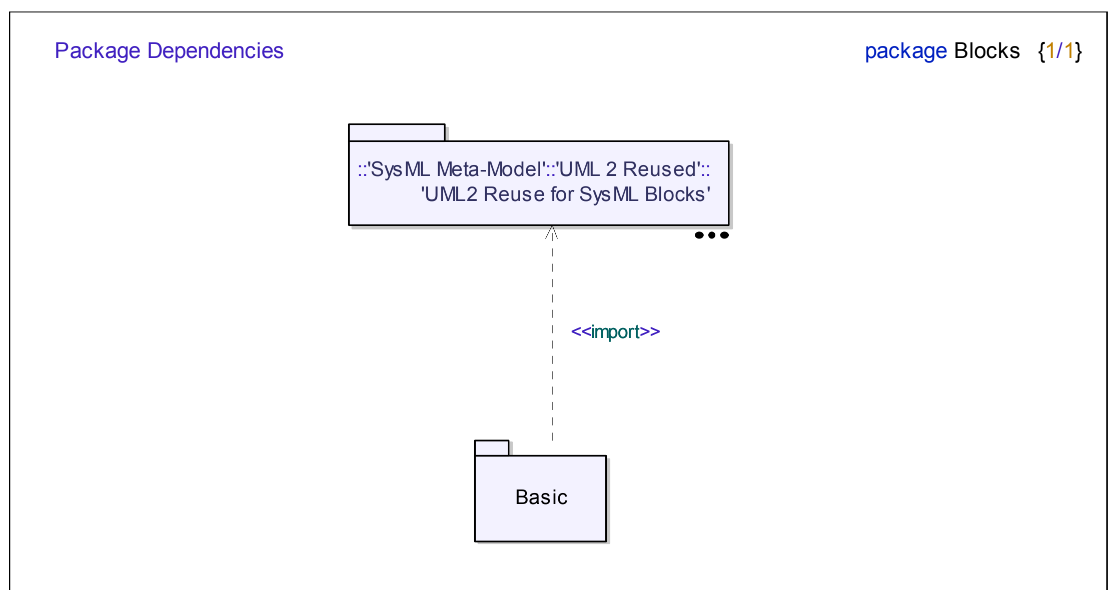


Figure 6-7. Package Contents of Blocks

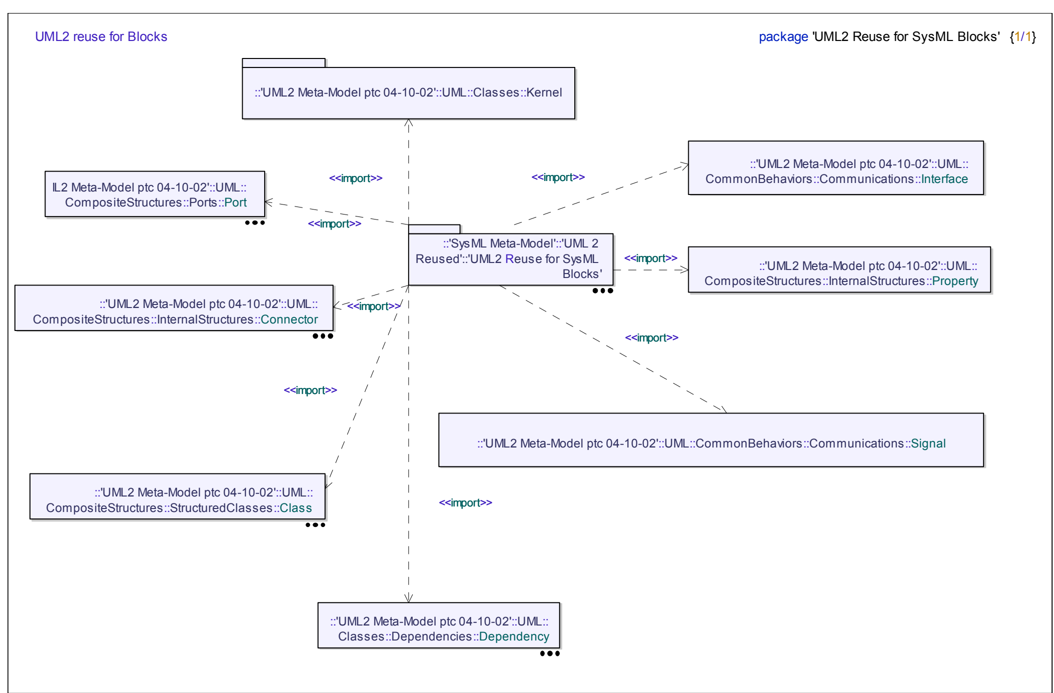


Figure 6-8. Package Contents of UML2 Ruse for System Blocks

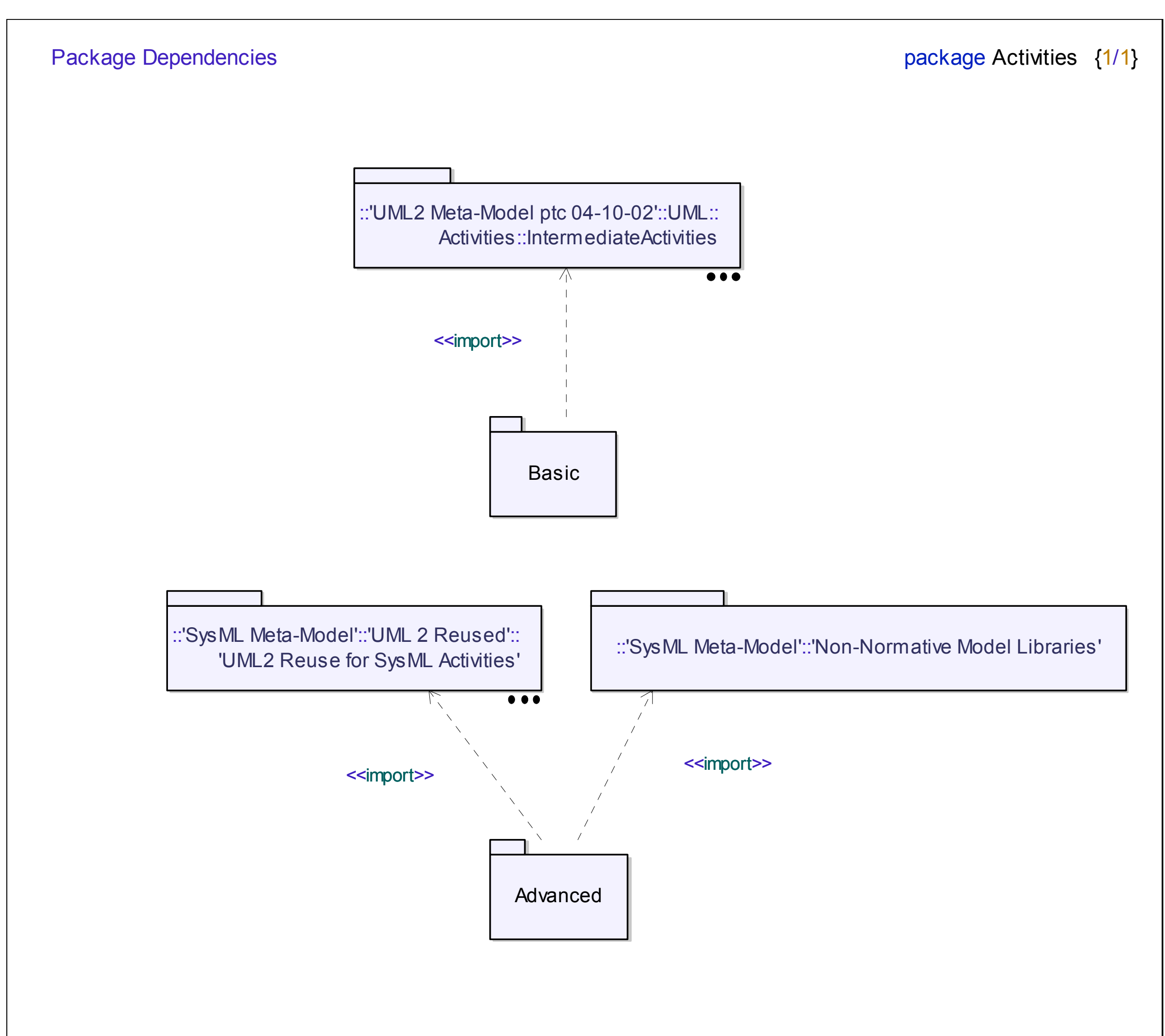


Figure 6-9. package Contents of Activities

6.3 Extension Mechanisms

This specification uses several extension mechanisms for defining SysML:

- stereotypes
- diagram extensions
- model library elements

SysML defines UML stereotypes that extend the semantics and notation of UML metamodel elements in a manner consistent with the Profile mechanism defined in the UML Superstructure specification [UML2 2005]. SysML diagram extensions define new diagram notations that supplement diagram notations reused from UML. Model library elements define model elements

that can be reused and modifed by the user, without special knowledge of the underlying metamodel.

Chapter 19 “Profiles & Model Libraries”, Appendix C, “Non-Normative Extensions” and Appendix D, “Non-Normative Model Library” show examples how systems engineers can further customize SysML using stereotypes and model libraries.

6.4 4-Layer Metamodel Architecture

Like the UML 2.0 metamodel on which the SysML profile is based, the SysML language architecture conforms to a 4-layer metmodel architecture pattern. In particular, the SysML profile extends the UML metamodel layer (a.k.a., the “M2” layer). so that both the SysML profile and the UML metamodel may be considered instances of the Meta Object Facility meta-meta-model (a.k.a., the “M3” layer). SysML model library elements are defined at the modeling layer (a.k.a., the “M1” layer).

6.5 Alignment with XMI and AP-233

The SysML profile is architecturally aligned with the OMG XMI and ISO AP-233 model interchange standards, which is explained further in Appendix E, “OMG XMI Model Interchange” and Appendix F, “ISO AP233 Model Interchange”, respectively.

7 Language Formalism

This chapter explains the specification techniques used to define SysML. These specification techniques have the following goals:

- Correctness.
- Precision.
- Conciseness.
- Consistency.
- Understandability.

The following sections explain the level of formalism, the chapter specification structure, constraint specification, and the specification technique used in this specification describes SysML as a UML extension that is defined using stereotypes and metaclasses.

7.1 Level of Formalism

SysML is defined using metamodeling and profiling techniques that use precise natural language (English) to specify contraints and semantics. In general, the syntax of the language is specified precisely so that SysML will support tool interoperability using OMG XMI and via ISO AP-233 model interchange formats.

SysML's detailed semantics are described using natural language, striking a difficult balance between formal rigor and understandability. As executable SysML modeling tools become more mainstream, it is also likely that more formal techniques will be applied to improve the precsion of SysML.

7.2 Chapter Specification Structure

This section provides information about how the top-level SysML packages are defined in each chapter. Each chapter has one or more of the following sections:

Overview

This section provides an overview of the modeling constructs defined in the subject package, which are usually associated with one or more SysML diagram types.

Diagram elements

This section provides tables that summarize the concrete syntax (notation) and abstract syntax references for the graphic nodes and paths associated with the relevant diagram types.

Package structure

This section specifies the package import dependencies on the UML metamodel.

UML extensions

This section specifies how the UML metamodel is extended via stereotypes, diagram extensions, and table extensions to produce semantics and notation for new SysML constructs. Stereotypes are specified with the following sections:

- Definition: A brief statement conveying the fundamental meaning of the stereotype. This definition is reused in the SysML glossary.

- Description: An explanation of the meaning and significance of the stereotype in isolation, and in relation to other constructs.
- Constraints: The well-formedness rules for applying stereotype.
- Notation: A description of the concrete syntax used for visualizing the stereotype.

Usage examples

This section shows how the SysML modeling constructs can be applied to solve pragmatic systems engineering problems.

7.3 Use of Constraints

SysML constraints are expressed using precise natural language (English).

7.4 Use of Natural Language

SysML uses natural language (English) for much of the specification, including the specification of constraints, and providing general descriptive text for stereotypes and other model elements.

7.5 Conventions and Typography

In the description of SysML, the following conventions have been used:

- While referring to stereoptypes, metaclasses, metaassociations, metaattributes, etc. in the text, the exact names as they appear in the model are always used.
- No visibilities are presented in the diagrams, since all elements are public.
- If a mandatory section does not apply for an extension, use the text: 'N/A' ('Not Applicable'). If an optional section is not applicable, it is not included.
- Stereotype, metaclass and metassociation names: initial embedded capitals are used (e.g., 'ModelElement', 'ElementReference').
- Boolean metaattribute names: always start with 'is' (e.g., 'isComposite').
- Enumeration types: always end with "Kind" (e.g., 'DependencyKind').

Part II - Structural Constructs

This Part of the specification defines the static, skeletal constructs used in SysML structure diagrams, which include Block diagrams and the Parametric Constraint diagrams. Both of these structural diagram types enforce a separation of concerns between definitions (specifications) and usages (applications), which is comonly referred to as the defintion/usage dichotomy. In the case of Block diagrams, *blocks* are used for definitions, and *parts* are used for applications. In the case of Parametric Constraints, *parametric constraints* are used for definitions, and *parametric constraint usages* are used for applications.

8 Blocks

8.1 Overview

A Block is a modular unit of system that encapsulates its contents, which include attributes, operations and constraints. Blocks can be connected to other Blocks to form composite structures, and can be decomposed into parts to expose internal structures. Blocks can be used to specify the structure of a system as a collection of properties that fulfill specific roles (parts) within a larger whole. A block also shows the connections between its roles (parts) that enable their interoperation within the context of the larger whole. Blocks may own ports, and thus can be connected to external parts in a larger context in which they are used. Each part is specified by a block with its own properties, ports, and internal structure, so a uniform set of elements is used to represent multiple levels of a system hierarchy.

SysML enforces a separation of concerns between structural definitions (specifications) and usages (applications), which is comonly referred to as the defintion/usage dichotomy. In the case of block diagrams, blocks are used for definitions, and parts are used for applications.

Blocks provides a general-purpose capability to model systems as trees of modular components. The specific kinds of components, the kinds of connections between them, and the ways these elements combine to define the total system can be chosen according to the goals of a particular system model. The SysML block model can be used throughout all phases of system specification and design, and can be applied to many different kinds of systems. These systems may be logical or physical, and may include software, hardware or human organizations. The parts in these systems may interact by many different means, such as software operations, discrete state transitions, flows of inputs and outputs, or continuous interactions.

Blocks and their associated diagrams are based on the UML concept of composite structures. UML composite structures provide the essential mechanisms to define a block in terms of its structural features. These include its internal parts, ports that can be used to connect it to other parts in its environment, and connections between parts as well as parts and ports that enable their interaction within a containing whole. UML composite structure diagrams can be used to show either a “black box view,” in which only the externally visible elements are visible, or a “white box view,” which exhibits the internal details of its parts and connections. They go further than the “architectural block diagrams” common to many engineering disciplines by specifying patterns of occurrences of their internal parts and connections, using structural features.

Blocks

To distinguish UML structured classes that adopt the SysML conventions for modeling system architecture, SysML defines a stereotype of UML classes called «block». Because SysML blocks may be applied to a wide variety of system types, SysML blocks include only a subset of the modeling elements that UML defines for composite structures. In particular, SysML does not include the UML component concepts; components are modeled as blocks in SysML.

SysML blocks build on structured classes to define a basic set of modeling elements, in order to enable the modeling of system structures early in a development cycle before any commitment has been made as to how a modeled element may be realized. Blocks can be coupled with other Blocks via Connectors to form systems of increasing complexity: components, subsystems, systems, and systems-of-systems.

Service Ports

A ServicePort specifies the services that the owning Block provides (offers) to its environment as well as the services that the owning Block requires of its environment. ServicePorts are specified (typed) using Interfaces. For example, a Block representing an automatic transmission in a car might have a ServicePort that specifies, via its provided interface, that the Block can accept a command to change gears. The engine control unit in the car might have a ServicePort that specifies, via its required interface, that it can send a command to change gears. Service Ports reuse UML::Port notation and semantics.

Flow Ports

A FlowPort specifies the input and output items that may flow between a Block and its environment. Input and output items may include data as well as physical entities, such as fluids, solids, gases, and energy. FlowPorts are defined using FlowSpecifications. For example, a Block representing an automatic transmission in a car might have a FlowPort that specifies torque as an input and another flow port that specifies torque as an output. FlowPorts are a stereotype of the UML metaclass Port and extend its notation and semantics.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Blocks.

8.2 Diagram elements

Table 4. *Graphical nodes for Blocks.*

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
BlockDefinition Diagram	***Block Definition* DiagramName** **Block1** ◆—— **Block2**	**none**	**Basic**
Block	«block»**BlockName** *Attributes* prop1: pTypeA prop2: Real = 500.0 {unit = meter} *Operations* op1(param1: PT): return Type1 *Constraints* { prop2 > 1000.0 }	**SysML::Blocks::Block**	**Basic**
DataType	**DataTypeName** *Attributes* prop1: pTypeA prop2: Real = 500.0 {unit = meter} *Operations* op1(param1: PT): return Type1 *Constraints* { prop2 > 1000.0 }	**UML::Classes::Kernel::Datatype**	**Basic**
FlowPort	iTrIn iTrOut **«block» Transmission** p1 **«part» trans:Transmission** p1	**SysML::Blocks::FlowPort**	**Basic**

Table 4. *Graphical nodes for Blocks.*

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
FlowSpecification	«flowSpecification» **iEng** t: Torque; iEng; iTrans	**SysML::Blocks::FlowSpecification**	**Basic**
Interface	«interface» **iTransCmd**; iTransCmd; iTransData	**UML::CommonBehaviors::Communications::Interface**	**Basic**
Non-Composite Property	**propertyName: TypeName**	**UML::Classes::Kernel:: Property with isComposite equal False**	**Basic**
Part	**partName : TypeName**	**UML::Classes::Kernel:: Property with isComposite equal True**	**Basic**

Table 4. *Graphical nodes for Blocks.*

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
ServicePort	iTransCmd «block» Transmission p3 iTransData «part» trans:Transmission p3	**SysML::Blocks::ServicePort**	**Basic**

Table 5. *Graphical paths for Blocks.*

PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Aggregation		**UML::Classes::Kernel:: Property with aggregation equal shared**	**Basic**
Association		**UML::Classes::Kernel::Association**	**Basic**
Composition		**UML::Classes::Kernel:: Property with aggregation equal composite**	**Basic**
Connector	connectorName: associationName	**UML::CompositeStructures::InternalStructures::Connector**	**Basic**
Containment		**UML::Classess::Kernel::Class:: nestedClassifier**	**Basic**
Generalization		**UML::Classes::Kernel::Generalization**	**Basic**
Realization		**UML::Classes::Kernel::Realization**	**Basic**

8.3 Package structure

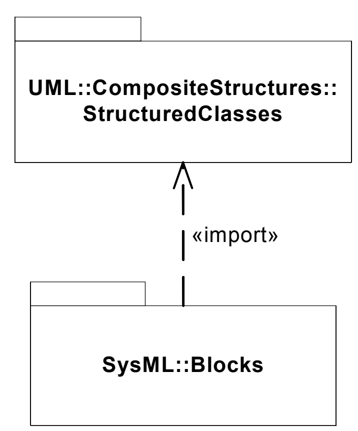


Figure 8-1. Package structure for Blocks.

8.4 UML extensions

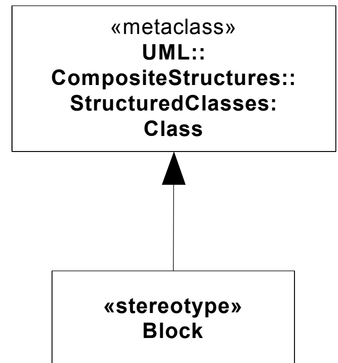


Figure 8-2. Abstract syntax for Blocks (1 of 2).

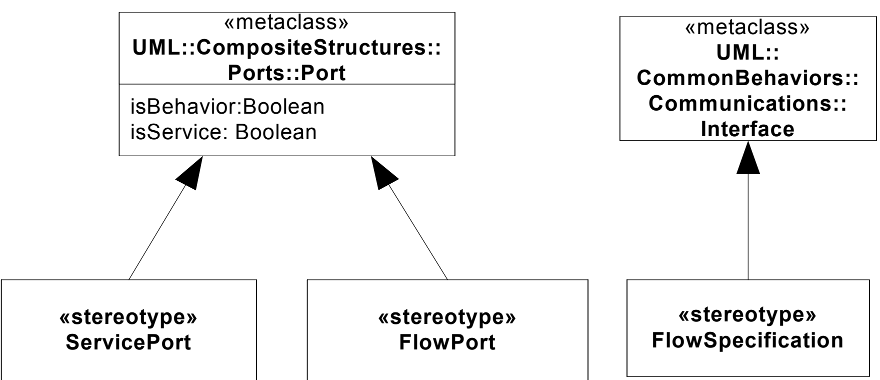


Figure 8-3. Abstract syntax for Blocks (2 of 2).

8.4.1 Stereotypes

8.4.1.1 Block

Definition

A Block is a modular unit of system that encapsulates its contents, which include attributes, operations and constraints. Blocks can be connected to other Blocks to form composite structures, and can be decomposed into parts.

Description

A Block is a stereotype of structured class that describes a structure of interconnected parts. A «block» may be used to model the structure of any kind of system, regardless of whether the components of the system consist of logical, physical, software, hardware, human, or other kinds of entities.

Besides the stereotypes introduced in this chapter, additional user-defined stereotypes may be defined as part of a user profile to categorize different kinds of systems and the roles they fill in particular contexts. Such distinctions may indicate specific stages of refinement (e.g., functional, physical), or a particular context in which a system appears (e.g., internal, external). The «block» stereotype should be used as an initial common root for such user-defined system types.

Notation

A Block is shown using a UML classifier symbol with the stereotype «block». The internal structure of a block is defined in a block definition diagram.

8.4.1.2 FlowPort

Definition

A FlowPort is an interaction point that specifies the input and/or output items that may flow between a Block and its environment. Input and output items may include data as well as physical entities, such as fluids, solids, gases, and energy.

Description

A FlowPort is a stereotype of Port that specifies an interaction point between a block and its environment for the flow of material, energy or data items. FlowPorts may be associated with a provided and a required FlowSpecifications. These FlowSpecifications determine what properties can flow between a Block and its environment.

Constraints

[1] The type of a FlowPort must be a FlowSpecification.

[2] The required interfaces of a FlowPort must be FlowSpecifications.

[3] The provided interfaces of a FlowPort must be FlowSpecifications.

[4] FlowPorts may only be connected if their required and provided flow specifications are compatible.

[5] A FlowPort must only be connected to another FlowPort.

Semantics

FlowPorts are distinct from Service Ports in that they specify an interaction point for the flow of properties, as opposed to signals or operation calls, between the owning Block and its environment. By virtue of the fact that they extend UML Ports they share the same dynamic semantics. That is to say that the dynamic behavior of the FlowPort is to simply forward signals or method calls arriving at the port along the connectors connected to the port.

Notation

A FlowPort is shown following the notation for UML ports with a small arrow extending from one side of the small square symbol to the opposite side. If the flow port has only provided flow specifications, the arrow head is shown on the inner side parallel to the side of the block symbol the port is attached to. If the flow port has only required flow specifications, the arrow head is shown on the outer side. Otherwise, arrow heads are shown on both sides, indicating that a bidirectional flow is possible across this port.

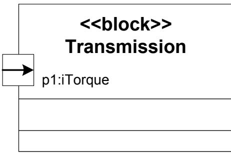


Figure 8-4. FlowPort notation.

Figure 8-4 shows that the block *Transmission* has a FlowPort *p1* over which only inbound items will flow, i.e., *p1* has only provided FlowSpecifications. The items that may flow across this port are defined by the FlowSpecification *iTorque*.

8.4.1.3 FlowSpecification

Definition

A FlowSpecification defines the entities which may flow across a FlowPort.

Description

A FlowSpecification specifies the entities which may flow across a FlowPort. They are restricted UML interfaces whose attributes are typed by a ValueType, a DataType or a Block.

Constraints

[1] All the properties of a given FlowSpecification are either inputs or outputs.

[2] The properties owned by a FlowSpecification must be typed by a ValueType, DataType or a Block.

Semantics

A FlowSpecification represents a contract that the block realizing the FlowSpecification must fulfill. For example, a provided FlowSpecification states that the block or blocks that realize the FlowSpecification must accept the flow of material, energy of information specified by the attributes of the FlowSpecification. Similarly, a required FlowSpecification states that the block or blocks that realize the FlowSpecification must source the flow of material, energy or information specified by the attributes of the FlowSpecification.

A FlowSpecification does not specify how the realizing Block sources, sinks or otherwise processes the material, energy or information but merely what needs to be sourced, sunk by the realizing Block.

8.4.1.4 ServicePort

Definition

A ServicePort is a unique interaction point for software or service communications between a Block and its environment or between a Block and its internal Parts.

Description

A ServicePort is a stereotype of Port and has the same semantics and notation. The name has been changed to differentiate software and service Ports from FlowPorts.

Constraints

[1] A ServicePort must only be connected to another ServicePort.

8.4.2 Diagram extensions

8.4.2.1 Block Definition diagram

Description

A diagram type, indicated on the diagram frame by a diagram kind of “block definition” or the abbreviation “bd” (see Appendix A “Diagrams”), is available to show block definitions and follows the graphical conventions of a UML class diagram showing block, their properties and their relationships.

8.4.2.2 Internal Block diagram

Description

A diagram type, indicated on the diagram frame by a diagram kind of “internal block” or the abbreviation “ibd” (see Appendix A “Diagrams”), is available to show the internal structure of a block and follows the graphical conventions of a UML composite structure diagram showing internal structure (parts, ports and connectors) of the subject block.

8.5 Usage examples

The following diagrams illustrate how Blocks diagrams are used. A complete sample problem that includes Blocks diagram can be found in Appendix B “Sample Problem”.

Figure 8-5 and Figure 8-6 show the use of service ports. The example defines a block *Car* that has two parts: *ecu* (an *EngineControlUnit*) and *trans* (the *Transmission)*. Figure 8-5 shows the definition of two interfaces: *iTransCmd* which defines the commands that the transmission can receive, and *iTransData* which defines the telemetry data that the transmission sends. The engine control unit *ecu* has a service port *p1* which provides the *iTransData* interface and requires the *iTransCmd* interface to function properly. The transmission has one service port *p3* which provides the *iTransCmd* interface and requires the *iTransData* interface.

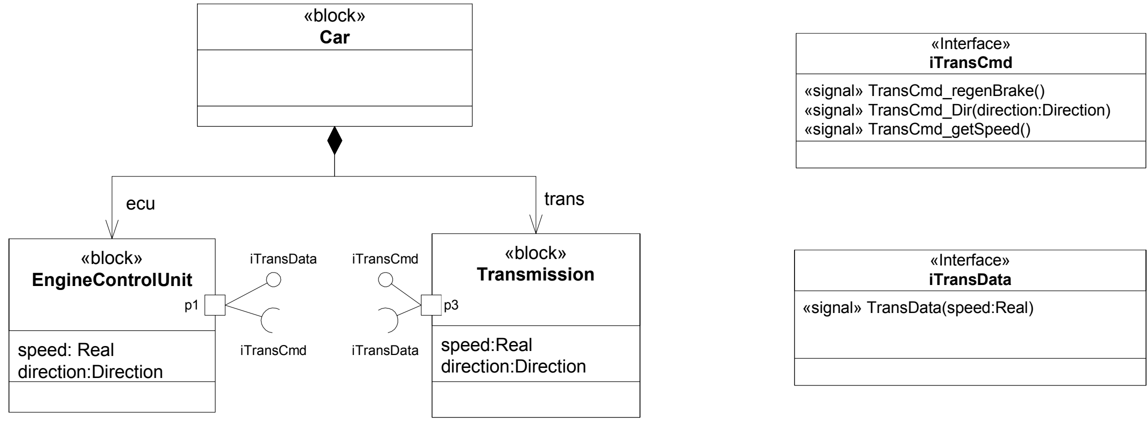


Figure 8-5. Service Port definition.

Figure 8-6 shows the internal structure of the *Car* as an internal block diagram. In this diagram the engine control unit *ecu* part is connected via port *p1* to the transmission *trans* part via port *p3* through the connector *CANbus*. Provided and required interfaces on *p1* are elided. In this structure, the behavior of the engine control unit *ecu* will send commands to the transmission,

which will receive these commands on the *p3* port. The behavior of the transmission will respond appropriately to the commands received, and will send a signal representing the current speed back to the engine control unit.

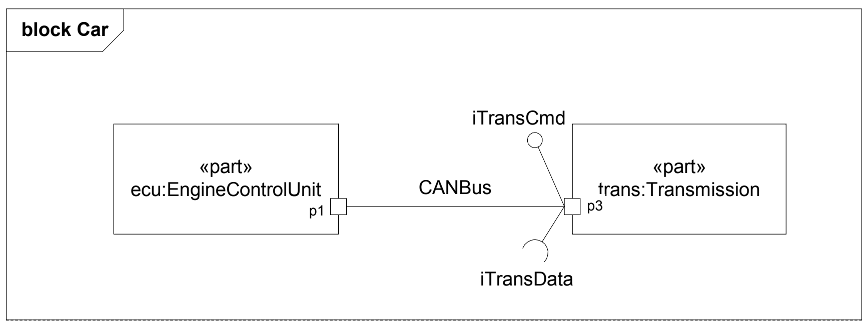


Figure 8-6. Service Port usage.

Figure 8-7 and Figure 8-8 show the use of FlowPorts. This example defines a Car block that has three parts: the engine *eng*, the transmission *trans*, and the differential *diff*. Figure 8-7 also defines the physical quantity *Torque* and a FlowSpecification *iTorque* which carries the sole item *Torque* (the quantity *Torque* is a user defined ValueType with unit Newton Meters and dimension ML^2/T^2. See Chapter 18, “Auxiliary Constructs” for the definition of ValueType itself and the definition of *Torque*. Appendix D also contains a list of pre-defined ValueTypes.). The Engine has a flow port *p1* with required flow specification *iTorque*. The Transmission has two ports *p1* and *p2*, the former having the provided flow specification *iTorque* while the latter has *iTorque* as the required flow specification. The Differential has one flow port *p1* with provided flow specification *iTorque.*.

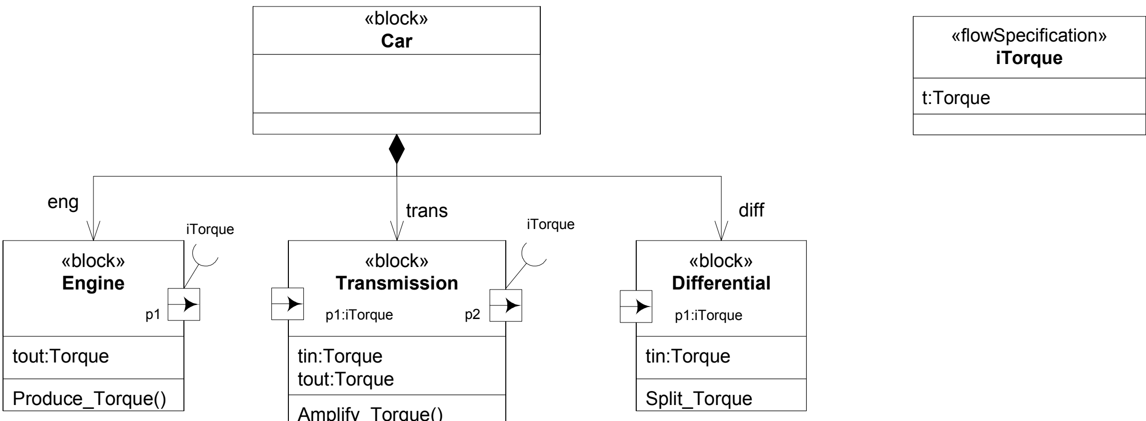


Figure 8-7. Flow Port definition.

Figure 8-8 shows the internal structure of the *Car* as internal block diagram. In this structure, the engine *eng* part is connected via FlowPort *p1* to the transmission *trans* part via its FlowPort *p1* through the connector *shaft1*. The transmission part is in turn connected via FlowPort *p2* to the *p1* port of the differential *diff* part through connector *shaft2*. This structure specifies that the engine can transmit torque via p1, which will be carried via shaft1 to the transmission port p1. The transmission can accept

this torque via p1 and, as a result of the behavior allocated to the transmission, amplify and re-transmit the amplified torque to the differential port p1 via p2 and shaft 2.

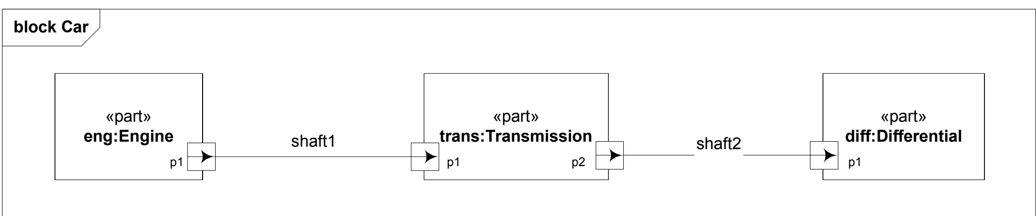


Figure 8-8. Flow Port usage.

9 Parametric Constraints

9.1 Overview

A parametric constraint specifies how a change to the value of one structural property of a system impacts the value of other system structural properties. Usually the constrained properties express quantitative characteristics of a system, but parametric models may also be used on non-quantitive properties. Parametric constraints are non-directional, so they have no notion of causality. Parametric constraints are typically used in combination with block diagrams.

Parametric constraint diagrams complement block diagrams, and are typically used in conjuntion with them. The definition/usage dichotomy applies to parametric constraints just as it does to blocks. Consequently, there is a semantic and notational difference between parametric constraint definition and parametric constraint use, where the former emphasizes specification and the latter emphasizes application. The main benefits of this separations of concerns is re-use and consistency. By separating definition from usage, the same specification can be re-used in many different contexts and any updates to the definition are reflected in all usages.

Parametric constraints are frequently used to specify performance and reliablity requirements during system analysis. Since parametric constraints are frequently interrelated, they often form a network of constraints among a system's structural properties. Although SysML provides a concrete syntax for naming and visualizing parametric constraints, it does not define or designate a particular language for expressing the constraint itself. Consequently, the constraint portion of a parametric constraint may be defined using a constraint language of the modeler's choosing, including mathematical equations (e.g., *Force = mass * acceleration*), logical rules (e.g., *IF altitude GTE 60,000 meters THEN ...*) and precise natural language (e.g., *The Pressure is proportional to Area.*"). Model libraries can be used to specify reusable parametric constraints for generic and specific purposes (e.g., Ohm's law for the electrical engineering domain).

Parametric constraints can be used to support tradeoff analysis. A modeler can specify a parametric constraint that represents an evaluation function for assessing alternative solutions. The evaluation function produces one or more outputs that represent a general measure of effectiveness. The evaluation function may include a weighting of utility functions associated with various criteria used to evaluate the alternatives. These criteria may be associated with selected system performance, cost, and physical properties. The corresponding properties from each alternative is put into the evaluation function to determine the overall measure of effectiveness. These properties may have probability distributions associated with them that are also fed into the evaluation function to compute a probabilistic or expected measure of goodness. An example that shows how parametric constraints can be used to support tradeoff analysis can be found in Appendix B, "Sample Problem" and Appendix C, "Non-Normative Extensions".

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Parametric Constraints.

9.2 Diagram elements.

Table 6. *Graphical nodes for Parametric Constraints.*

Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Parametric Constraint	«parametric» ParametricName	**SysML::ParametricCon-straints::ParametricConstraint**	**Advanced**
Parametric Constraint Use	BlockName «parametricUse» Name: ParametricName	**SysML::ParametricCon-straints:: ParametricConstraintUse**	**Advanced**
Binding	«binding» Path parameterName	**SysML::ParametricCon-straints::Binding**	**Advanced**

9.3 Package structure

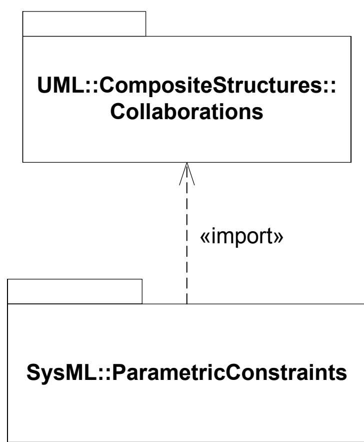


Figure 9-1. Package structure for Parametric Constraints.

9.4 UML extensions

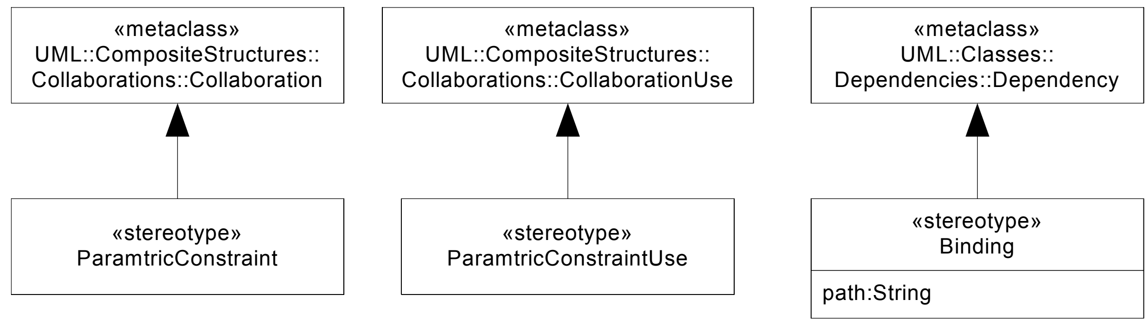


Figure 9-2. Abstract syntax for Parametric Constraints.

9.4.1 Stereotypes

9.4.1.1 Binding

Definition

A Binding relationship is a dependency between a parameter role in a parametric constraint and a property or parameter role in another parametric constraint. The value of the parameter role and the property are bound to be equivalent within the context of the parametric constraint.

Description

A Binding is a stereotype of Dependency. As with other dependencies, the arrow direction points from the (client/source) property (or parameter role in the case where one parametric constraint use is bound to another) to the (supplier/target) parameter role if the parametric constraint. Parametric constraints define the contexts for bindings, and apply across levels of the structural hierarchy. For those cases where binding includes also includes a path, the binding applies to the property that can be reached by navigating the binding path.

- path: String [1]: A string resulting from the concatenation of a sequence of property names that begins with the name of a property of the classifier typing the property at the client end, and ends with the name of the represented property. Each name in the concatenated string is the name of the role of the entity represented by the preceding name. Each property name in the sequence is separated by a single dot ("."). (For example, the name "Engine.SparkPlug.voltage" represents the property named *voltage* of the part *SparkPlug* of the part *Engine* of the enclosing block.)

Notation

A binding is shown as a dashed line between a collaboration use symbol and a property, or a collaboration use symbol and another collaboration use symbol, adorned with the keyword «binding». The path is shown following the stereotype.

Presentation option

Symbols representing a hierarchically nested property may be shown within the outline of the symbol representing the outermost property. The dashed line representing the binding is drawn to that symbol, and the path is shown within that symbol. See Figure 9-5 for an example of this presentation option.

9.4.1.2 ParametricConstraint

Definition

A Parametric Constraint defines how the values of one or more Block properties are restricted.

Description

The ParametricConstraint is a stereotyped collaboration that restricts the values of one or more Block properties. A parametric constraint definition may contain other parametric constraints to define a constraint as a composition of other constraints, as well as other properties that may define the internal state of the parametric constraint (such as constants used in defining this constraint). A parametric constraint definition does not specify any direction of causality by which the relation of its part values is established, but only specifies that a given relation is required to hold across its bound values. The specific relation that a parametric constraint defines may be specified either by a UML constraint on the properties of the constraint definition, or by informal specification in the documentation for the constraint. The roles of the parametric constraint definition are typically parameters in the system of equations or rules that defines the constraint.

Notation

The specific relation that a parametric constraint defines between its properties is shown using UML constraint notation, either in a compartment of the classifier symbol representing the ParametricConstraint, or with the internal structure of the constraint. A parametric constraint definition is shown by preceding the name with the keyword «parametric» and the dashed ellipse symbol inherited from Collaboration.

9.4.1.3 ParametricConstraintUse

Definition

A Parametric Constraint Use specifies how a Parametric Constraint is applied to restrict the values one or more Block properties.

Description

A ParametricConstraintUse is a stereotyped CollaborationUse in which the parameter roles are bound to system structural properties. Once bound, the parametric constraint describes the relationship between the bound properties.

Constraint

[1] The type of a ParametricConstraintUse must be a ParametricConstraint.

Notation

A ParametricConstraintUse is shown by preceding the name with the keyword «parametricUse» and the dashed ellipse symbol inherited from collaboration use. The binding of the parameter roles of a parametric constraint to a property in a block diagram is shown by a dashed line from the symbol representing the occurrence of the parametric constraint use to the property. The dashed line represents a dependency, as with a collaboration use, where the client end is the property and the supplier end is the role of the parametric constraint that types the parametric constraint use. The dashed line is labeled on the client end with the name of the supplier element (the role of the parametric constraint typing the parametric constraint use).

Two parametric constraint uses may make reference to the same parameter role. A dashed line between the two occurrences of the respective parametric constraint uses, labeled on each end with the name of the supplier role, represents an implicit part with an anonymous name and a type that is compatible with the supplier roles in the collaborations that type each parametric constraint use identified by the labels. This implicit part is bound to the respective roles in each parametric constraint. For an example, see Figure 9-5, where both Newton's law and the HP2Force relation reference the same parameter role, *force*, that represents the force exerted on the object.

9.4.2 Diagram extensions

9.4.2.1 Parametric diagram

Description

A diagram type, indicated on the diagram frame by a diagram kind of "parametric" or the abbreviation "par" (see Appendix A, Diagrams), is available to show parametric constraints and follows the graphical conventions of a UML internal structure diagram showing a collaboration.

9.5 Usage examples

The following diagrams illustrate how Parametric Constraint diagrams are used. A complete sample problem that includes Parametric Constraint diagrams can be found in Appendix B, "Sample Problem".

Figure 9-3 and Figure 9-4 shows two parametric constraints that are used in subsequent examples. The parametric constraint in Figure 9-3 shows three parameter roles (*force*, *mass*, and *acceleration)* that are parts of the internal structure of the parametric constraint. A constraint, demarcated by set braces, relates these three parameter roles. Alternatively, in Figure 9-4 the three

parameter roles (force, hp, and k1) are attributes of a collaboration, and the constraint relating these parameter roles is shown in a special compartment.

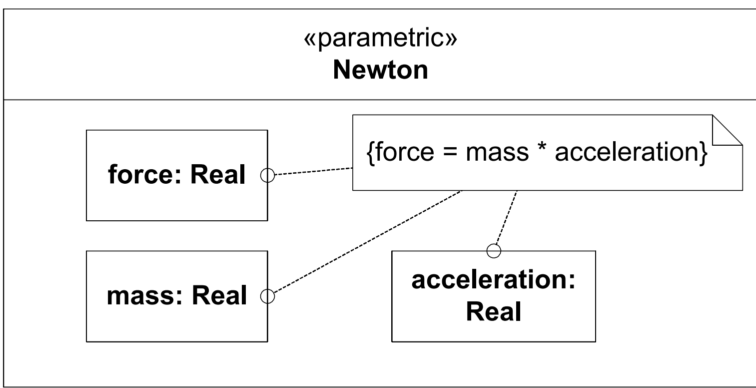


Figure 9-3. Parametric constraint definition using internal structure notation.

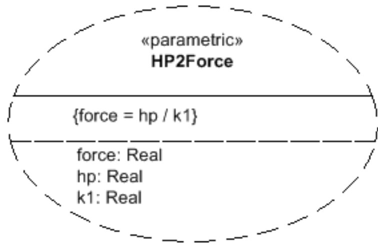


Figure 9-4. Parametric constraint definition using collaboration notation.

Parametric constraint uses are typically shown on a parametric block diagram or an internal block diagram, where they are shown with other aspects of the block specification. Figure 9-5 shows an internal block diagram for the *HybridSUV* block from Appendix B (the external block diagram for the *HybridSVU* is shown in Figure B-16). In this diagram we see several properties of the HybridSUV: the electric motor *em*, the internal combustion engine *ice*, *mass*, and *acceleration.* Using the presentation option discussed in Section 9.4.1.1 to show nested properties in a compartment of the symbol representing the outermost property, the *maxHp* property of *em* and the *displacement* property of *ice* are shown as well. These properties are related by parametric constraints *maxHP* (*Sum2Real*), *disp2hp* (*Disp2HP*), *hp2force* (*HP2Force*), and *accel* (*Newton*). For example, *accel* relates the *mass* property of the *HybridSUV* to its *acceleration* and to a *force*. Similarly, the use of parametric constraint *HP2Force*, *hp2force* relates a *force*, *hp*, and a constant *k1*, as shown in Figure 9-4. Note that the same force appears in both

Newton and *HP2Force* and is represented using a parameter role binding connecting the two uses of these parametric constraints.

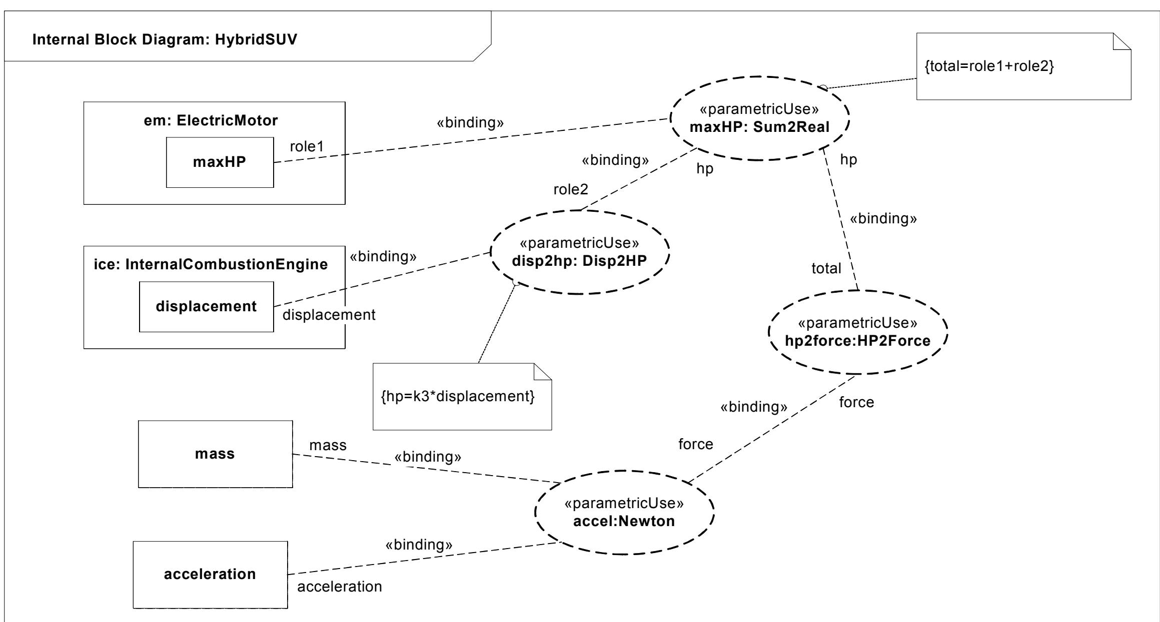


Figure 9-5. Parametric constraints on a Block diagram.

Figure 9-6 shows a more complex example of a parametric constraint. Here all constraints of the previous example are captured in a single parametric constraint *AccelerationRelations*. In a use of this constraint, the four roles *hp, mass, displacement*, and *acceleration* have to be bound to properties within the structure where the constraint is applied.

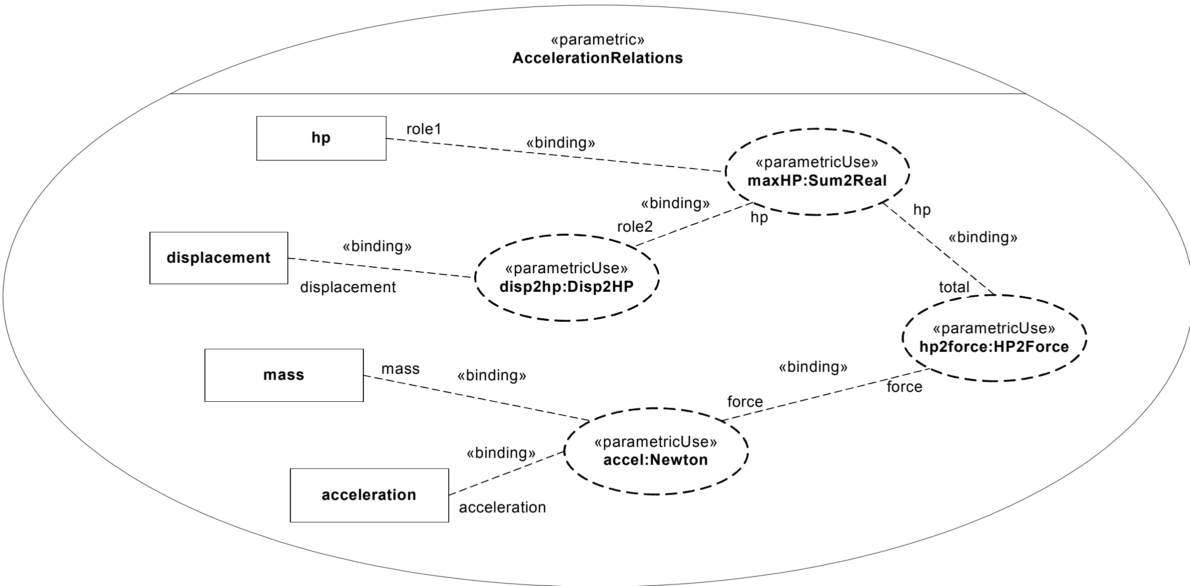


Figure 9-6. Parametric constraints on a Parametric diagram.

See Sample Problems B.4.2 and B.4.14 for other examples of parametric constraints.

Part III - Behavioral Constructs

This Part specifies the dynamic, behavioral constructs used in SysML behavioral diagrams, such as Activity diagrams, Sequence diagrams, and State Machine diagrams. The Activity represents the basic unit of behavior that is used in all behavioral diagrams. An activity is a behavior that is composed of actions, some of which may invoke other activities. The State Machine diagram includes activities that are invoked during transition between states, upon entry or exit from a state, or while in a state. The Sequence diagram includes activities as methods of operations that are invoked by messages. Use Case diagrams, which specify system usages and are both behavioral and structural, are also defined in this Part.

10 Activities

10.1 Overview

Activities specify sequential and concurrent behaviors that are connected by control flows and object flows. Activities can be nested or atomic; in the latter case they are referrred to as actions. Activity diagrams are used to specify the functional behavior of a system, and are analogous to Extended Functional Flow Block Diagrams (EFFBDs). See Appendix C, “Non-Normative Extensions” for a non-normative mapping of Activity notation to EFFBD notation.

The following sections describe how SysML extends UML Activity diagrams to support system engineering applications:

Control as data

In UML activities controls can only enable actions to start. In SysML controls can also disable actions that are already executing via Control Values. SysML also provides Control Operators to control other actions.

Continuous systems

SysML provides extensions for continuous behaviors that are generally applicable to any sort of distributed flow of information and physical items through a system. These extensions include:

- In UML modelers can only specify discrete flows of information. SysML allows modelers to specify both continuous and discrete flows, where the flows can be involve matter and energy as well as information. SysML also allows modelers to define the rate at which entities flow. Discrete and continuous flows are unified under the rate of flow, as is traditionall for mathematical models of continuous change.
- SysML extends object nodes, including pins, with the option for newly arriving values to replace values that are already in the object nodes. It also extends object nodes with the option to discard values if they do not immediately flow downstream. These extensions are useful for ensuring that the most recent information is available to actions by indicating when old values should not be kept in object nodes, and for preventing fast or continuously flowing values from collecting in an object node, as well as modeling transient values (e.g., electrical signals).

Probability assignment

SysML allows modelers to assign expressions to flows that evaluate to probabilities for the likelihood that a value leaving a decision node or object node will traverse a flow. SysML also extends output parameter sets with probabilities for the likelihood that values will be output on a parameter set.

Activities as Blocks

In UML activities and all other behaviors are classes and their instances are executions. Consequently, in UML all behaviors can appear on class diagrams and participate in generalization and association relationships. Since SyML Block Defintion diagrams are analogous to UML Class diagrams, SysML extends Block Defintion diagram notation to include activities, clarifies the semantics of composition associations, and defines consistency rules between Activity diagrams and Block diagrams.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Activities.

10.2 Diagram elements

This section describes the concrete syntax for graphical nodes and paths in Activity diagrams..

Table 7. *Graphical nodes for Activities.*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
AcceptEventAction		**UML::Actions::CompeleteActions**	**Basic**
Action		**UML::Activities::CompeleteActivities::Action**	**Basic**
ActivityFinal		**UML::Activities::IntermediateActivities::ActivityFinalNode**	**Basic**
ActivityNode	**See ExecutableNode, ControlNode, and ObjectNode.**	**UML::Activities::CompleteActivities::ActivityNode**	**Basic**
ControlNode	**See DecisionNode, FinalNode, ForkNode, InitialNode, JoinNode, and MergeNode.**	**UML::Activities::BasicActivities::ControlNode**	**Basic**
ControlOperator	«controlOperator» Function	**SysML::«controlOperator»**	**Advanced**
DecisionNode		**UML::Activities::IntermediateActivities::DecisionNode**	**Basic**
FinalNode	**See ActivityFinal and FlowFinal.**	**UML::Activities::IntermediateActivities::FinalNode**	**Basic**
FlowFinal		**UML::Activities::IntermediateActivities::FlowFinalNode**	**Advanced**
ForkNode		**UML::Activities::IntermediateActivities::ForkNode**	**Basic**

Table 7. *Graphical nodes for Activities.*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
InitialNode		**UML::Activities::BasicActivities::InitialNode**	**Basic**
JoinNode		**UML::Activities::CompleteActivities::JoinNode**	**Basic**
MergeNode		**UML::Activities::Intermediate-Activities::MergeNode**	**Basic**
ObjectNode		**UML::Activities::CompleteActivities::OjectNode and its children.**	**Basic**
SendSignalEvent		**UML::Actions::BasicActions**	

Table 8. *Graphical paths for Activities.*

Path Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Rate	«continuous» «discrete» «continuous» Object Node «discrete» Object Node { rate = constant } { rate = distribution } { rate = constant } { rate = distribution } Object Node Object Node «rate» rate = constant rate = distribution	**SysML::«rate», SysML::«continuous», SysML::«discrete»**	**Advanced**
Rate	{ rate = constant } { rate = distribution }	**SysML::«rate»**	**Advanced**
OverWrite	«overwrite» Function «overwrite» «overwrite» Object Node	**SysML::«overwrite»**	**Advanced**
NoBuffer	«noBuffer» Function «noBuffer» «noBuffer» Object Node	**SysML::«noBuffer»**	**Advanced**

Table 8. *Graphical paths for Activities.*

PATH NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Probability	Function { probability = valueSpecification } { probability = valueSpecification }	**SysML::«probability»**	**Advanced**
Optional	«optional» Function «optional»	**SysML::«optional»**	**Basic**
isControl	{ control } Function { control }	**UML::Activities::CompleteActivities::Pin.isControl**	**Advanced**
isStream	{ stream } Function { stream }	**UML::Activities::CompleteActivities::Parameter.isStream**	**Advanced**
ActivityEdge	**See ControlFlow and ObjectFlow.**	**UML::Activities::CompleteActivities::ActivityEdge**	**Basic**
ControlFlow		**UML::Activities:: BasicActivities:: ControlFlow** **SysML::ControlFlow**	**Basic**

Table 8. *Graphical paths for Activities.*

PATH NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
ObjectFlow		**UML::Activities::CompleteActivities::ObjectFlow and its children.**	**Basic**

Table 9. *Other graphical elements included in Activity diagrams.*

ELEMENT NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Activity, ObejctNode, Association on Block Diagram	«activity» activity name; action name; «activity» activity name; «activity» activity name; object node name; «block» block name	**SysML::Activity, SysML:ObjectNode**	**Advanced**
Activity	Activity name Parameter name: type	**UML::Activities::CompleteActivities::Activity**	**Basic**
ActivityPartition	Partition Name; (Partition Name) invocation	**UML::Activities:: IntermediateActivities:: ActivityPartition**	**Basic**

Table 9. *Other graphical elements included in Activity diagrams.*

ELEMENT NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
InterruptibleActivity-Region		**UML:Activities:: CompleteActivities:: InterruptibleActivityRegion**	**Advanced**
Local pre- and post-conditions.	«localPrecondition» constraint name «localPostcondition» constraint	**UML:Activities:: CompleteActivities::Action**	**Advanced**
ParameterSet		**UML::Activities:: CompleteActivities:: ParameterSet**	**Advanced**

10.3 Package structure

The package structure for SysML Activities is shown in Figure 10-1. .

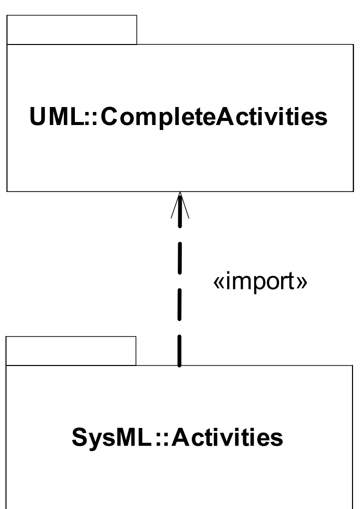


Figure 10-1. Package structure for Activities.

10.4 UML extensions

This section describes the UML stereotypes and diagram extensions required to define SysML Activity diagrams..

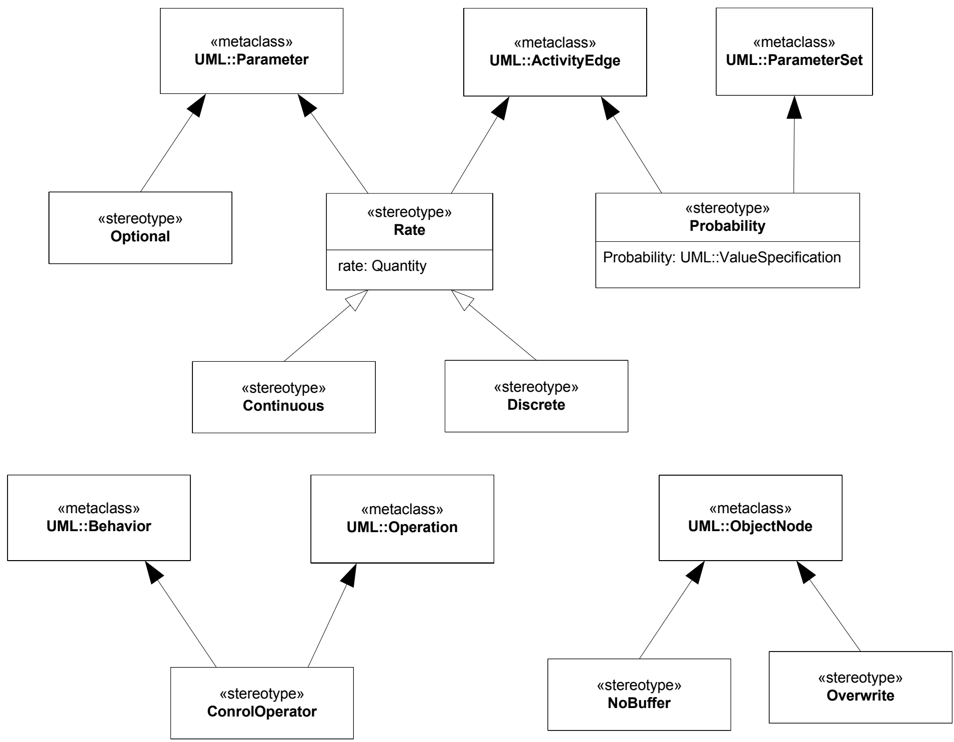


Figure 10-2. Abstract syntax for Activities.

10.4.1 Stereotypes

10.4.1.1 Continuous

This is a kind of Rate stereotype representing a rate of flow for items treated as infintesimals (e.g., water flowing a pipe) or entities sufficiently small enough to treat as continuously flowing (e.g., ball bearings in a factory). It is independent from UML streaming (see "Rate"). A streaming parameter may result in continuous flows or not, and a continuous flow may involve streaming parameters or not.

UML places no restriction on the rate at which tokens flow. In particular, the time between tokens can approach as close to zero as needed, for example to simulate continuous flow. There is also no restriction in UML on the kinds of values that flow through an activity. In particular, the value may represent as small a number as needed, for example to simulate continuous material or energy flow. Finally, the exact timing of token flow is not completely prescribed in UML. In particular, token flow on different edges may be coordinated to occur in a clocked fashion, as in time march algorithms for numerical solvers of ordinary differential equations, such as Runge-Kutta.

Constraints

[1] The «nobuffer» stereotype always applies to object nodes that have an incoming edge stereotyped by «continuous».

10.4.1.2 ControlValue (a predefined enumeration)

Description

The ControlValue enumeration is a type available for modelers to apply when control is to be treated as data (see section 10.4.1.3) and for UML control pins. It can be used for behavior and operation parameters, object nodes, and attributes, and so on. The ControlValue enumeration is defined in Chapter 17, "Types" and subclassed in Appendix C, "Non-Normative Extensions" to provide a set of default enumeration literals. Modelers can extend the enumeration with additional literals, such as suspend, resume, with their own semantics. See Appendix D for customizing the enumeration literals.

Constraints

1. UML::ObjectNode::isControlType is true for object nodes with type ControlValue.

10.4.1.3 ControlOperator

Description

When the «controlOperator» stereotype is applied to behaviors, the behavior takes control values as inputs or provides them as outputs, that is, it treats control as data. The control values do not enable or disable the control operator execution based on their value, they only enable based on their presence as data. Pins for control parameters are regular pins, not UML control pins. This is so the control value is passed into or out of the action and the invoked behavior, rather than control the starting of the action, or indicating the ending of it. When the «controlOperator» stereotype is not applied, the behavior may not have a parameter typed by ControlValue. The «controlOperator» stereotype also applies to operations, with the same semantics.

Constraints

[1] When the «controlOperator» stereotype is applied, the behavior or operation must have at least one parameter typed by ControlValue. If the stereotype is not applied, the behavior or operation may not have any parameter typed by ControlValue.

[2] A method behavior must have the «controlOperator» stereotype applied if its operation does.

10.4.1.4 Discrete

This is a kind of Rate stereotype representing a rate of flow for items treated as individuals for the purpose of the application, for example, cars in a car factory.

Constraints

[1] The «discrete» and «continuous» stereotypes cannot be applied to the same element at the same time.

10.4.1.5 NoBuffer

Description

When the «nobuffer» stereotype is applied to object nodes, tokens arriving at the node that are refused by outgoing edges, or refused by actions for object nodes that are input pins, are discarded. This is typically used with fast or continuously flowing data values, to prevent buffer overrun, or to model transient values, such as electrical signals. For object nodes that are the target of continuous flows, «nobuffer» and «overwrite» have the same effect. When the stereotype is not applied, the semantics is as in UML, specifically, tokens arriving at an object node that are refused by outgoing edges, or action for input pins, are held until they can leave the object node.

Constraints

[1] The «nobuffer» and «overwrite» stereotypes cannot be applied to the same element at the same time.

10.4.1.6 Overwrite

Description

When the «overwrite» stereotype is applied to object nodes, a token arriving at a full object node replaces the ones already there (a full object node has as many tokens as allowed by its upper bound). This is typically used on an input pin with an upper bound of 1 to ensure that stale data is overridden at an input pin. For upper bounds greater than one, the token replaced is nondeterministic. A null token removes all the tokens already there. The number of tokens replaced is equal to the weight of the incoming edge, which defaults to 1. For object nodes that are the target of continuous flows, «overwrite» and «nobuffer» have the same effect. When the stereotype is not applied, the semantics is as in UML, specifically, tokens arriving at object nodes do not replace ones that are already there.

Constraints

[1] The «overwrite» and «nobuffer» stereotypes cannot be applied to the same element at the same time.

10.4.1.7 Optional

Description

When the «optional» stereotype is applied to parameters, the lower multiplicity must be equal to zero. Otherwise, the lower multiplicity must be greater than zero, which is called "required".

Constraints

[1] A parameter with the «optional» stereotypes applied must have multiplicity.lower equal to zero, otherwise multiplicity.lower must be greater than zero.

10.4.1.8 Probability

Description

When the «probability» stereotype is applied to edges coming out of decision nodes and object nodes, it provides an expression for the probability that the edge will be traversed. These must be between zero and one inclusive, and add up to one for edges with same source at the time the probabilities are used.

When the «probability» stereotype is applied to output parameter sets, it gives the probability the parameter set will be given values at runtime. These must be between zero and one inclusive, and add up to one for output parameter sets of the same behavior at the time the probabilities are used.

Constraints

[1] The «probability» stereotype can only be applied to activity edges that have decision nodes or object nodes as sources, or to output parameter sets.

[2] When the «probability» stereotype is applied to an activity edge, then it must be applied to all edges coming out of the same source.

[3] When the «probability» stereotype is applied to an output parameter set, it must be applied to all the parameter sets of the behavior or operation owning the original parameter set, and all the output parameters must be in some parameter set.

10.4.1.9 Rate

Description

When the «rate» stereotype is applied to an activity edge, it specifies the rate over time that objects and values traverse the edge, that is, the rate they leave the source node and arrive at the target. It does not refer to the rate at which a value changes over time. When the stereotype is applied to a parameter, the parameter must be streaming, and the stereotype gives the rate over time that objects or values are expected to flow in or out of the parameter while the behavior or operation is executing. Streaming is a characteristic of UML behavior parameters that supports the input and output of items while a behavior is executing, rather than only when the behavior starts and stops. The flow may be continuous or discrete, see the specialized rates in section 10.4.1.1, and section 10.4.1.4. The quantity must have units and dimensions appropriate to rates of flow.

Constraints

[1] When the «rate» stereotype is applied to parameter, the parameter must be streaming.

[2] The denominator for units used in the rate property must be time units.

[3] Rates on edges that come from or go to streaming parameters must be more than or equal to the rate of the parameters.

10.4.2 Diagram extensions

The entries below give notational extensions for some UML elements.

10.4.2.1 Activity

Notation

In UML 2, all behaviors are classes, including activities, and their instances are executions of the activity. This follows the general practice that classes define the constraints under which the instances must operate. Creating an instance of an activity causes the activity to start executing, and vice versa. Destroying an instance of an activity terminates the corresponding execution, and vice versa. Terminating an execution also terminates the execution of any other activities that it invoked synchronously, that is, expecting a reply.

Activities as classes can have associations between each other, including strong composition associations. Strong composition means that destroying an instance at the whole end destroys instances at the part end. When strong composition is used with activity classes, the termination of execution of an activity on the whole end will terminate executions of activities on the part end of the links.

Combining the two aspects above, when an activity invokes other activities, they can can be associated by a strong composition association, with the invoking activity on the whole end, and the invoked activity on the part end. If an execution of an activity on the whole end is terminated, then the executions of the activities on the part end are also terminated. The upper multiplicity on the part end restricts the number of concurrent synchronous executions of the behavior that can be invoked by the containing activity. The lower multiplicity on the part end is always zero, because there will be some time during the execution of the containing activity that the lower level activity is not executing. See Constraints, below.

Activities in class diagrams appear as regular classes, using the «activity» keyword for clarity, as shown in Figure 10-3. See example in section 10.5. The names of the CallBehaviorActions that correspond to the association can be used as end names of the association on the part end. Activities in class diagrams can also appear with the same notation as CallBehavior-Action, except the rake notation can be omitted, if desired. Also see use of activities in class diagrams at ObjectNode.

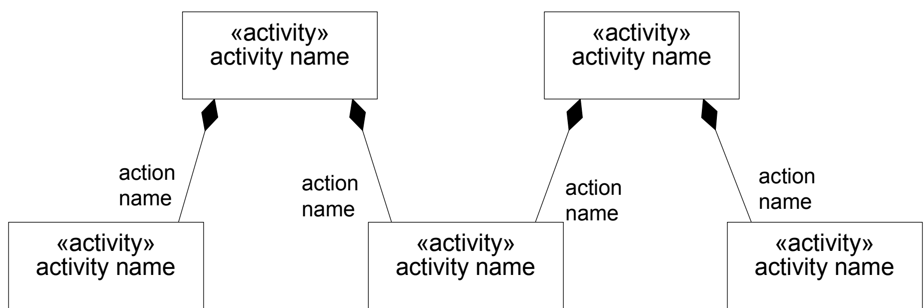


Figure 10-3. Class diagram with activities as classes.

CallBehaviorActions in activity diagrams can optionally show the action name with the name of the invoked behavior using the colon notation shown in Figure Figure 10-4.

action name : behavior name

Figure 10-4. CallBehaviorAction notation.in activty diagram

Constraints

The following constraints apply when composite associations in class diagrams are defined between activities:

[1] The part end name must be the same as the name of a synchronous CallBehaviorAction in the composite activity. If the action has no name, and the invoked activity is only used once in the calling activity, then the end name is the same as name of the invoked activity.

[2] The part end activity must be the same as the activity invoked by the corresponding CallBehaviorAction.

[3] The lower multiplicity at the part end must be zero.

[4] The upper multiplicity at the part end must be 1 if the corresponding action invokes a nonreentrant behavior.

10.4.2.2 ControlFlow

Presentation Option

Control flow may be notated with a dashed line and stick arrowhead.

10.4.2.3 ObjectNode

Notation

See Section 10.4.2.1 concerning activities appearing in class diagrams. Associations can be used between activities and classes that are the type of object nodes in the activity, as shown in Figure 10-5. This supports linking the execution of the activity with items that are flowing through the activity and happen to be contained by the object node at the time the link exists. The names of the object node that correspond to the association can be used as end names of the association on the end towards the object node type. The upper multiplicity on the object node end restricts the number of instances of the item type can can reside in the object node at one time, which must be lower than the maximum amount allowed by the object node itself.. The lower multiplicity on the object node end is always zero, because there will be some time during the execution of the containing activity that there is no item in the object node. The associations may be composite if the intention is to delete instances of the class flowing the activity when the activity is terminated. See example in Section 10.5 .

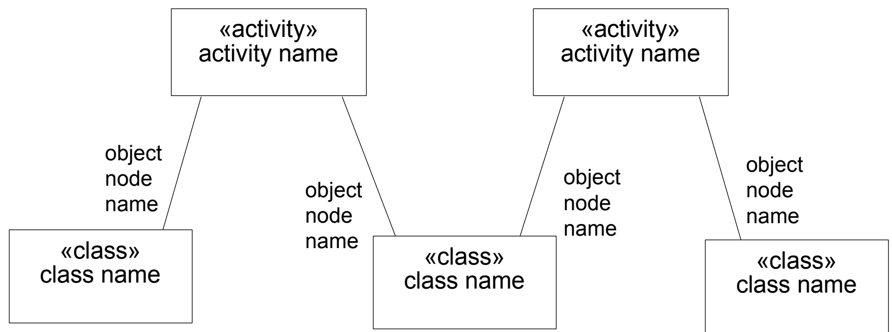


Figure 10-5. Class diagram with activities as blocks associated with types of object nodes.

Object nodes in activity diagrams can optionally show the node name with the name of the type of the object node as shown in Figure 10-6.

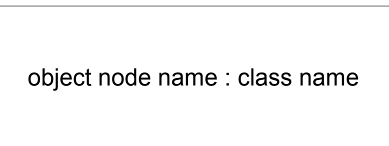


Figure 10-6. ObjectNode notation in activity diagrams.

Stereotypes applying to parameters can appear on object nodes in activity diagrams, as shown in Figure 10-7, when the object node notation is used as a shorthand for pins. The stereotype applies to all parameters corresponding to the pins notated by the object node.

«stereotype name»

object node name

Figure 10-7. ObjectNode notation in activity diagrams.

Constraints

The following constraints apply when associations in class diagrams are defined between activities and classes typing object nodes:

[1] The end name towards the object node type is the same as the name of an object node in the activity at the other end.

[2] The class must be the same as the type of the corresponding object node.

[3] The lower multiplicity at the object node type end must be zero.

[4] The upper multiplicity at the object node type end must be equal to the upper bound of the corresponding object node.

10.5 Usage examples

The following diagrams illustrate how Activity diagrams are used. A complete sample problem that includes Activity diagrams can be found in Appendix B, “Sample Problem”.

Figure 10-8 through Figure 10-9 shows the Activity Diagram for the ControlPower Activity scenario also shown in Figure 11-3. These diagrams are fully elaborated, including partitions (commonly called “swimlanes”) to illustrate the allocation of behavior to structure. The initial versions of these diagrams may not have partitions, but may instead focus on behavior alone. Once the equipment breakdown structure is defined, partitions may be added or explicit allocations can be performed. See Chapter 15, “Allocations”.

Note that many of the elements displayed on these diagrams are representations of the same model element shown on other diagrams. for example the ApplyAccelerator SendSignalAction is the same underlying model element as the message ApplyAccelerator on Figure B-12 and Figure B-13.

The circles with letters inside represent off page connectors used to connect flows across pages of large diagrams.

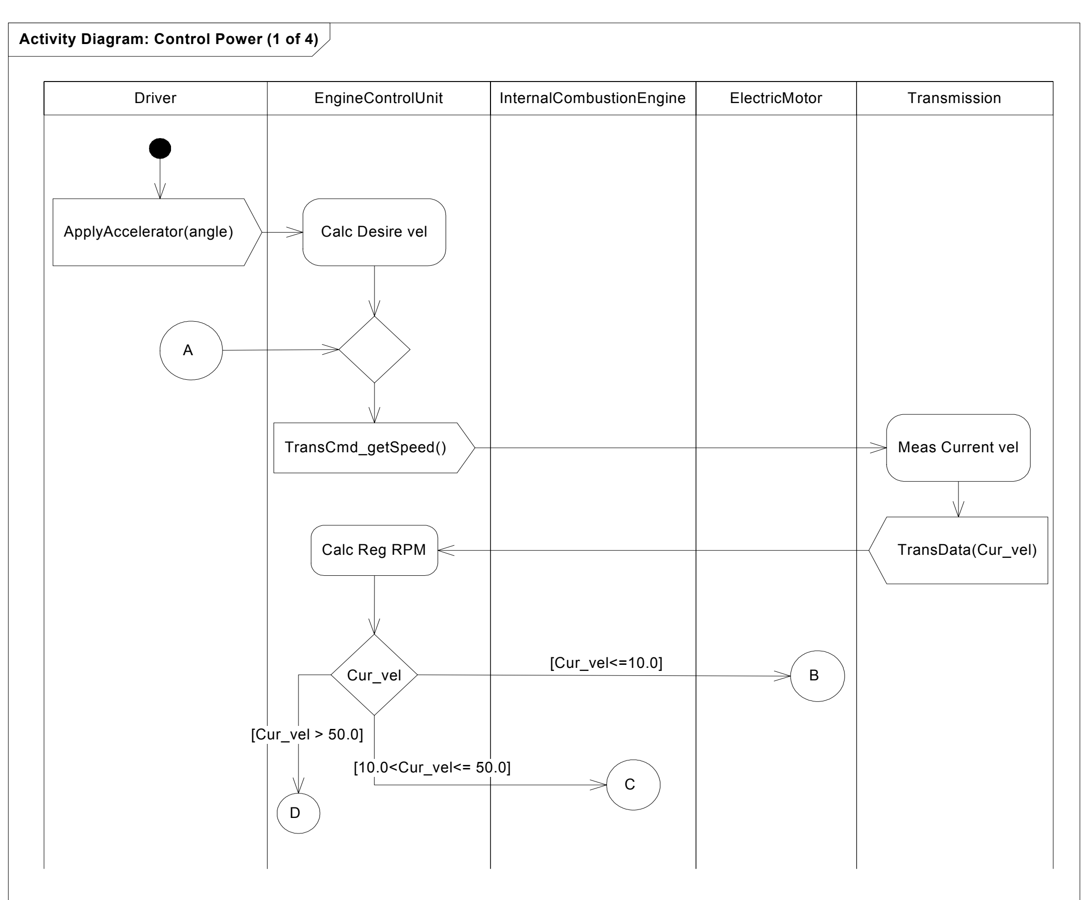


Figure 10-8. Activity Diagram Example: Control Power (1 of 4)

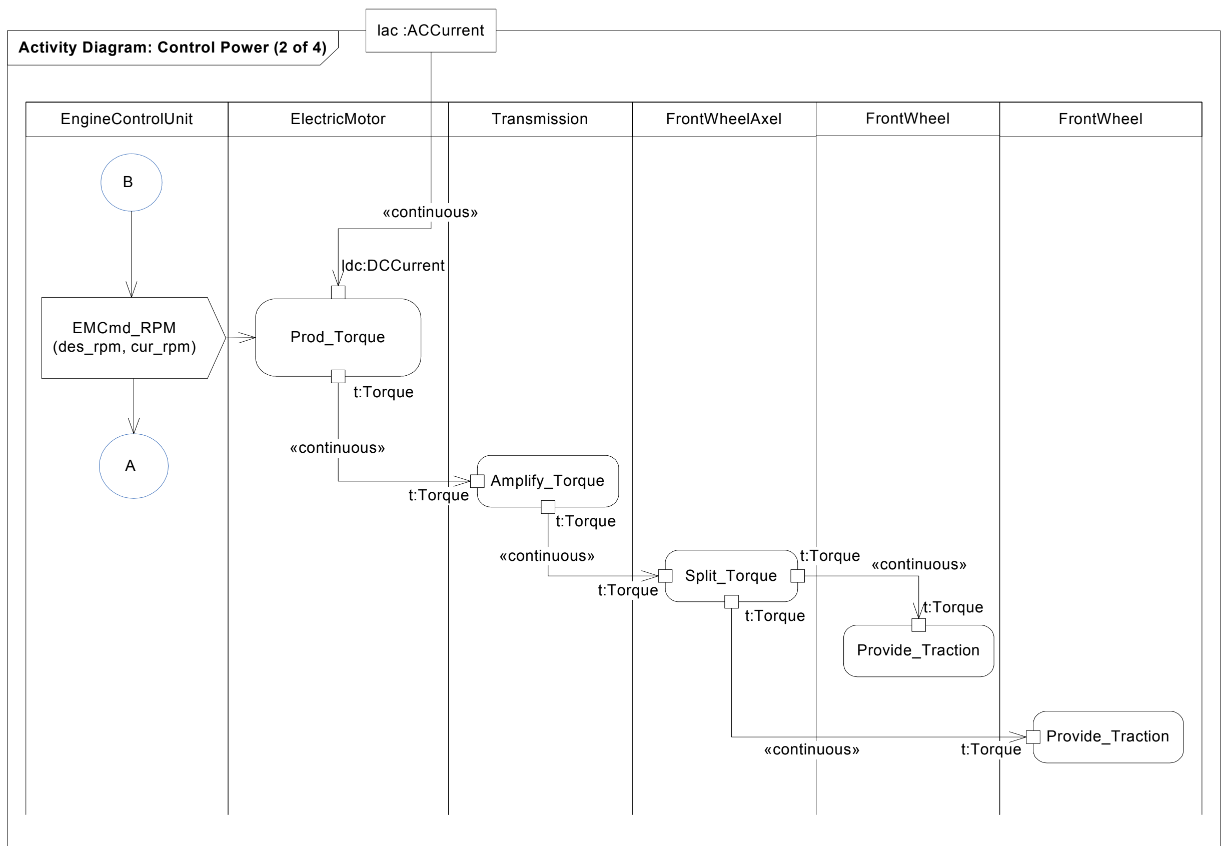


Figure 10-9. Activity Diagram Example: Control Power (2 of 4)

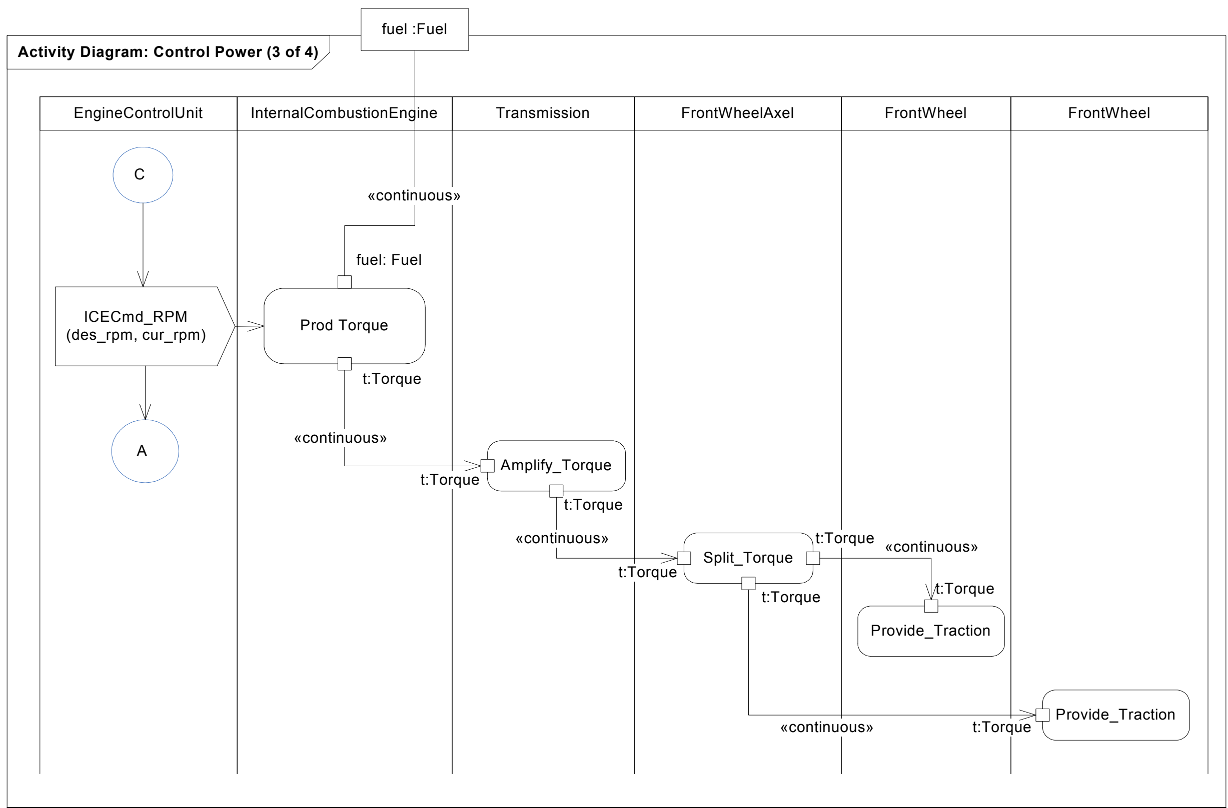


Figure 10-10. Activity Diagram Example: Control Power (3 of 4)

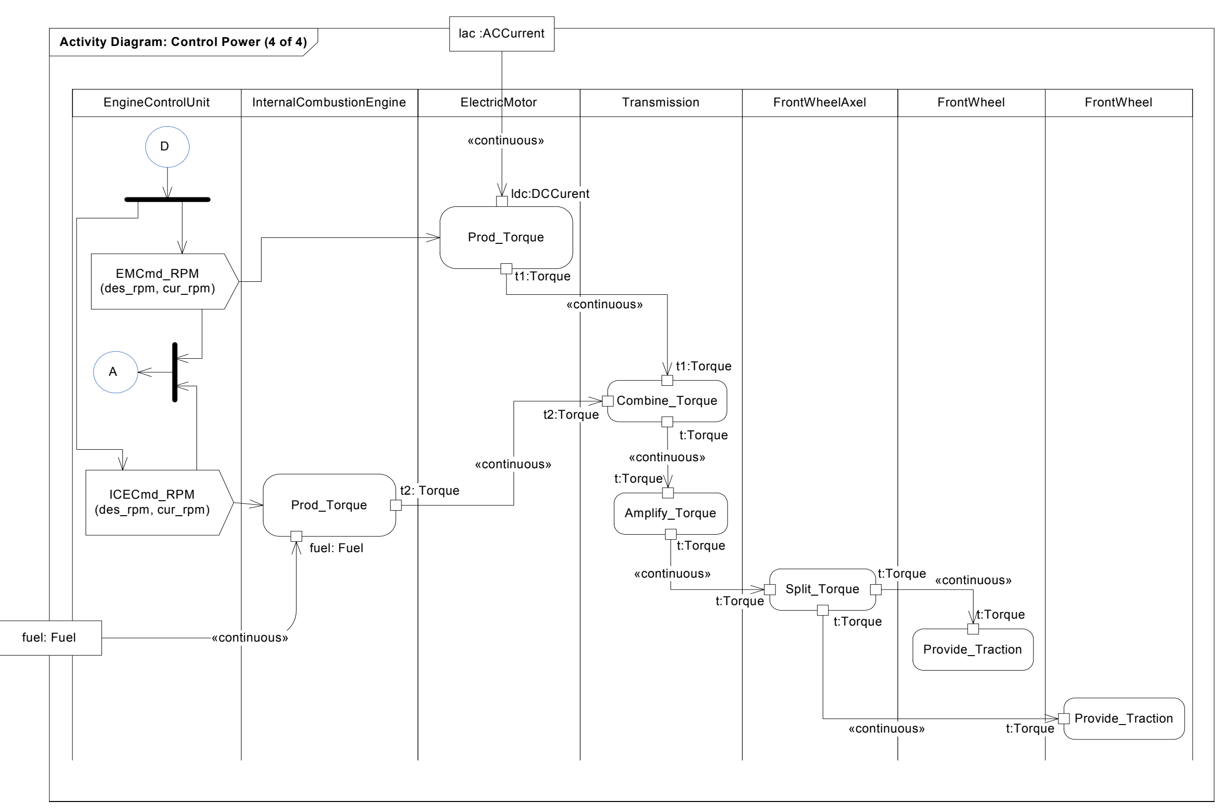


Figure 10-11. Activity Diagram Example: Control Power (4 of 4)

The following examples illustrate modeling continuous systems. Figure 10-12 shows a simplified model of driving and braking in a car that has an automatic braking system. Turning the key on starts two behaviors, Driving and Braking, which are the responsibility of the Driver and Brake System respectively. These behaviors execute until the key is turned off, using streaming parameters to communicate with other functions. The Driving behavior outputs a brake pressure continuously to the Braking behavior while both are executing, as indicated by the «continuous» rate and streaming properties (streaming is a characteristic of UML behavior parameters that supports the input and output of items while a behavior is executing, rather than only when the behavior starts and stops). Brake pressure information also flows to a control operator that outputs a control value to enable or disable the Monitor Traction behavior. No control pins are used on Monitor Traction, so once it is enabled, the continuously arriving enable control values from the control operator have no effect, per UML semantics. When the brake pressure goes to zero, disable control values are emitted from the control operator. The first one disables the monitor, and the rest have no effect. While the monitor is enabled, it outputs a modulation frequency for applying the brakes as determined by the ABS system. The rake notations on the control operator and Monitor Traction indicate they are further defined by activities, as shown in Figure 10-13 and Figure 10-14.

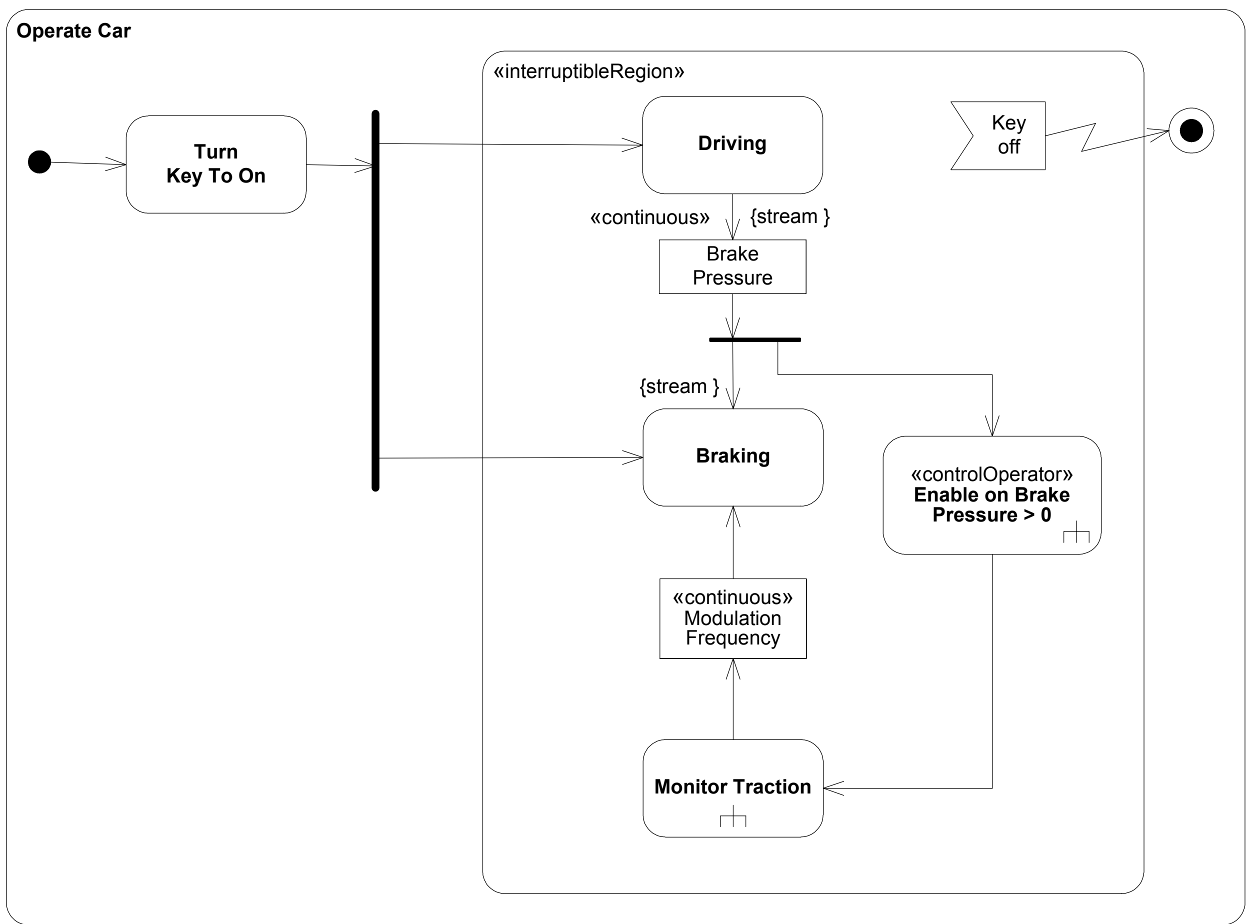


Figure 10-12. Continuous system example 1.

The activity diagram for Monitor Traction is shown in Figure 10-13. When Monitor Traction is enabled, it begins listening for signals coming in from the wheel and accelerometer, as indicated by the signal receipt symbols on the left, which begin listening automatically when the activity is. A traction index is calculated every 10 ms, which is the slower of the two signal rates. The accelerometer signals come in continuously, which means the input to Calculate Traction does not buffer values. The

result of Calculate Traction is filtered by a decision node for a threshold value and Calculate Modulation Frequency determines the output of the activity.

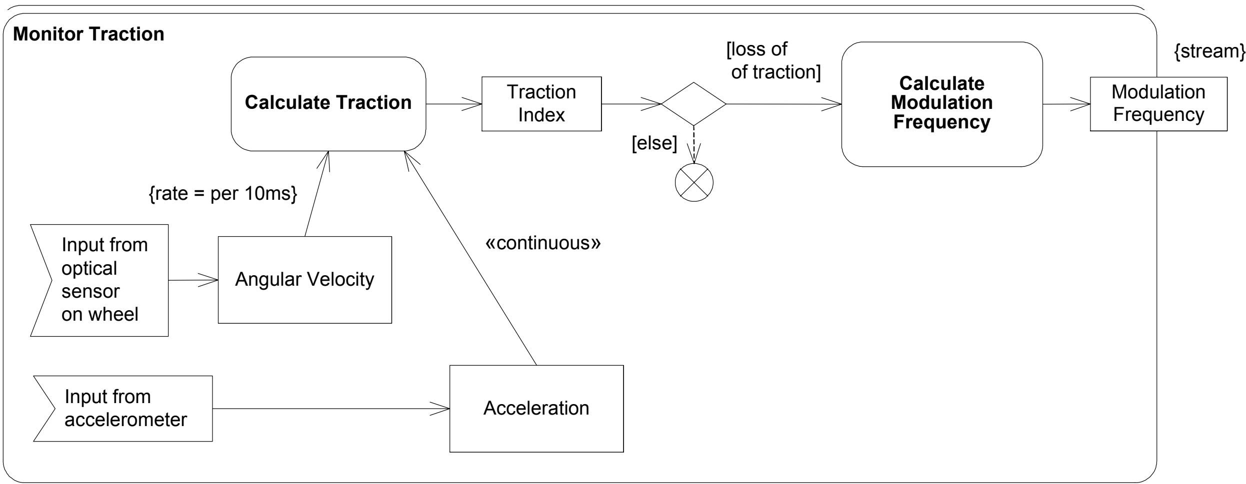


Figure 10-13. Continuous system example 2.

The activity diagram for the control operator Enable on Brake Pressure > 0 is shown in Figure 10-14. The decision node and guards determine if the brake pressure is greater than zero, and flow is directed to value specification actions that output an enabling or disabling control value from the activity. The edges coming out of the decision node indicate the probability of each branch being taken.

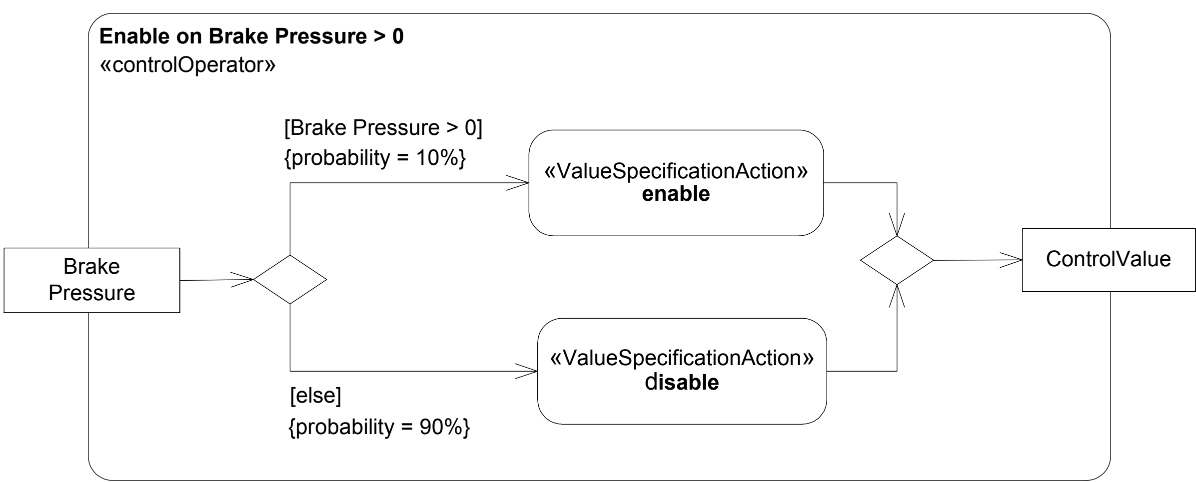


Figure 10-14. Continuous system example 3.

11 Sequences

11.1 Overview

The Sequences package defines a set of constructs for modeling communnications among block structures arranged in time order. A Sequence diagram specifies a series of interactions in terms of message flows. A message combines control and dataflow. It initiates behavior in the object receiving the message and passes inputs to the behavior. The time ordering of the messages is associated with the vertical placement of the message on the diagram. Complex sequences can be decomposed into interaction uses and combined fragments.. Conditional logic can be included to represent alternative flows, parallel flows, and loops. Gates provide interaction points with external lifelines. Lifelines can be decomposed into their constituent parts.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Sequences.

11.2 Diagram elements

The graphical nodes that can be included in Sequence diagrams are shown in Table 14.

Table 10 - *Graphical nodes for Sequences***.**

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Sequence Diagram Frame	**seq** EventOccurrence	The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner. See "Interaction (from BasicInteraction, Fragments)" on page 419.	**Basic**
Lifeline	:Lifeline	**UML::Interactions::Fragments**	**Basic**
ExecutionSpecification	ob2:C2 Ob3:C3 doIt	**UML::Interactions::BasicInteractions**	**Basic**
InteractionUse	**ref** N	**UML::Interactions::Fragments**	**Basic**

Table 10 - *Graphical nodes for Sequences*.

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
CombinedFragment	alt	**UML::Interactions::Fragments**	**Basic**
StateInvariant / Continuation	:Y p==15	**UML::Interactions:: BasicInteractions** **UML::Interactions::Fragments**	**Advanced**
Coregion	s[u]:B m3 m2	See explanation under *parallel* in "CombinedFragment (from Fragments)" on page 409	**Advanced**
DestructionEvent		**UML::Interactions:: BasicInteractions**	**Basic**

Table 10 - *Graphical nodes for Sequences*.

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Duration Constraint **Duration Observation**	:User Code d=**duration** {d..3*d} CardOut {0..13} OK	**UML::CommonBehaviors: SimpleTime**	**Basic**
Time Constraint **Time Observation**	CardOut {0..13} t=**now** OK {t..t+3}	**UML::CommonBehaviors: SimpleTime**	**Basic**

The graphic paths between the graphic nodes are given in Table 15.

Table 11 - *Graphic paths for Sequences*.

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Message	Code doit(z)	**UML::Interactions:: BasicInteractions**	**Basic**
Lost Message	lost	**UML::Interactions:: BasicInteractions**	**Advanced**

Table 11 - *Graphic paths for Sequences.*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Found Message	found	**UML::Interactions:: BasicInteractions**	**Advanced**

11.3 Package structure

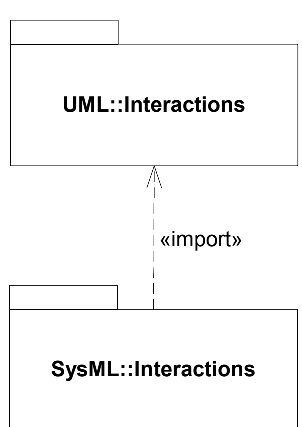


Figure 11-1. Package structure for Sequences.

11.4 UML extensions

No UML extensions are defined for Sequence diagrams.

11.5 Usage examples

The following diagrams illustrate how Sequence diagrams are used. A complete sample problem that includes Sequence diagrams can be found in Appendix B, “Sample Problem”.

Figure 11-2 shows a “black-box” sequence diagram describing the “Drive Car” use case. The diagram is considered a “black-box” diagram as it shows a single lifeline for the Hybrid SUV and does not show any internal parts. The diagram has

interaction occurrences for each of the «included» use cases shown in Figure 13-2 of Chapter 13, “Use Cases”, and shows a time ordering of executions (other time orderings are possible, i.e. this is a partial specification of behavior).

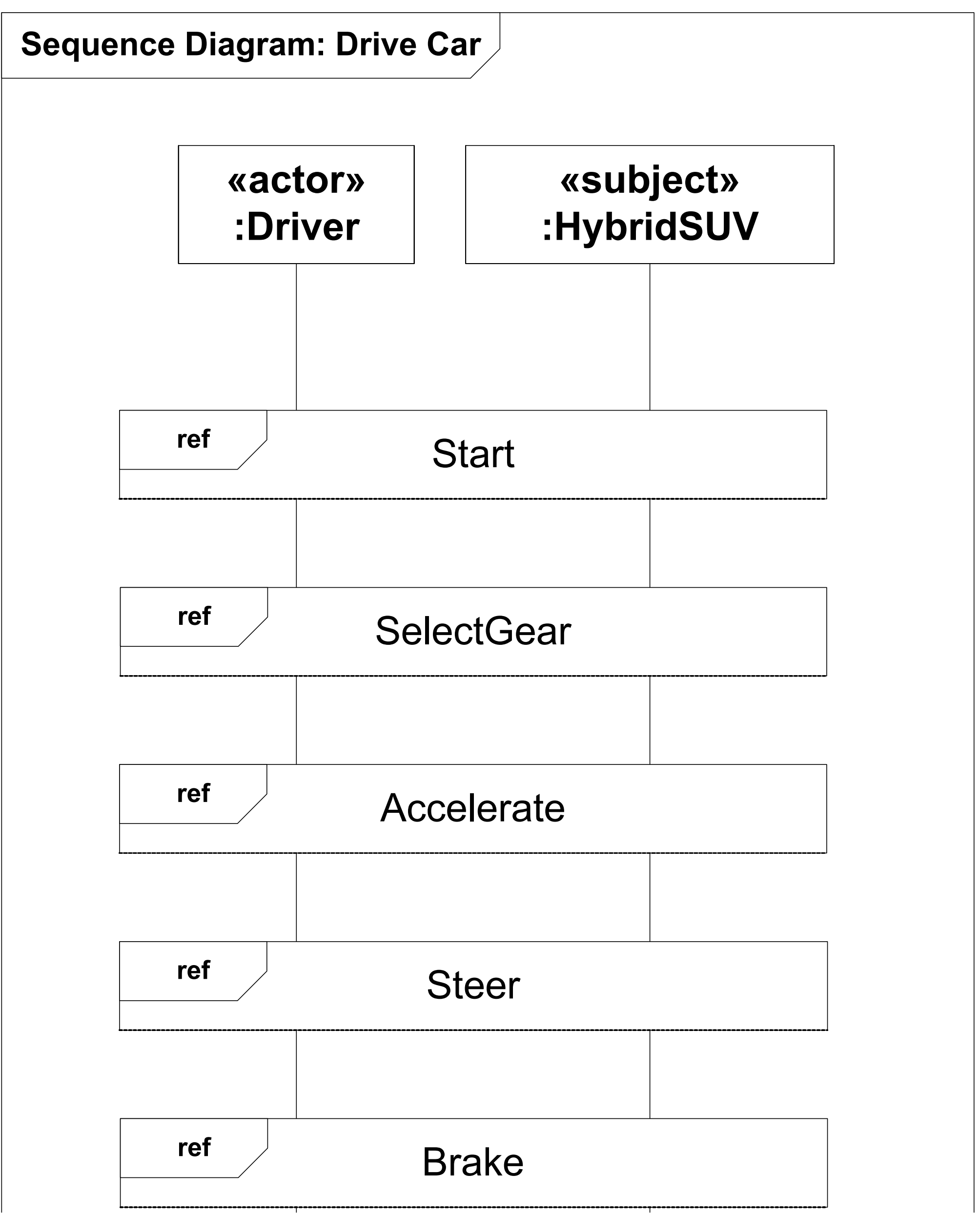


Figure 11-2. Top-level Sequence diagram for Drive Car

Figure 11-3 shows the details of the Accelerate interaction occurrence. When the :Driver presses the accelerator the :HybridSUV receives the ApplyAccelerator message with a parameter: “angle”. In response to this event, the :HybridSUV performs its ControlPower activity.

This is still a “black-box” view of the system, however the “ref: Accelerate Allocate” text on the :HybridSUV lifeline, known as a part decomposition, indicates that a white-box (or allocated) view exists that shows the internal interactions of the :HybridSUV for this scenario.

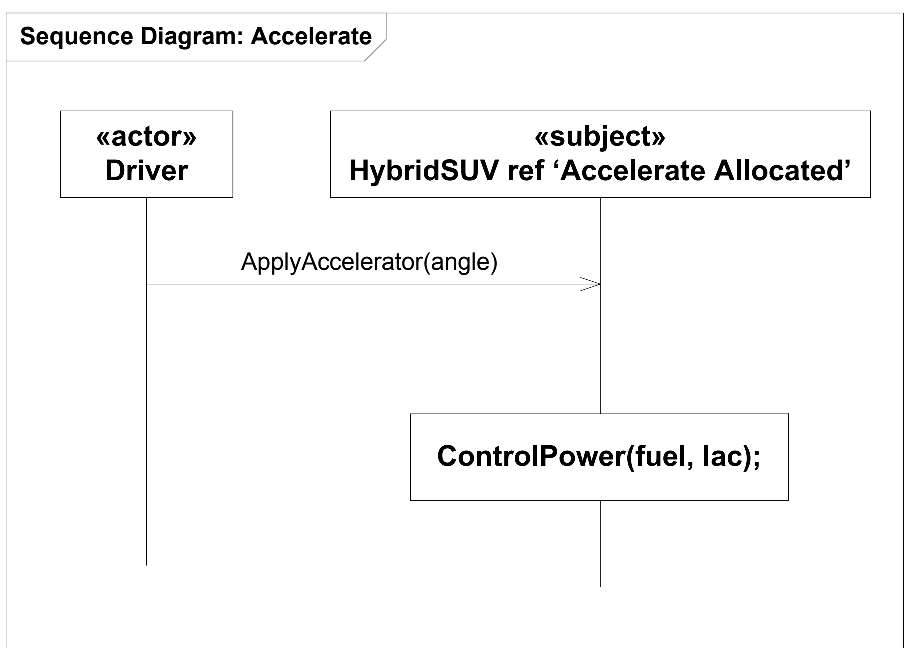


Figure 11-3. Accelerate scenario

Figure 11-4 shows the “white-box” sequence diagram for the Accelerate scenario. The choice of Activity Diagrams, Sequence Diagrams, or a combination of the two for defining scenarios is a matter of personal taste. Of course a diagram with this level of detail cannot be created until candidate parts (units, subsystems, etc. of the equipment breakdown structure) have been identified. See Section B.3.8 of Appendix B for the Block Definition diagram describing the blocks referenced in Figure 11-4.

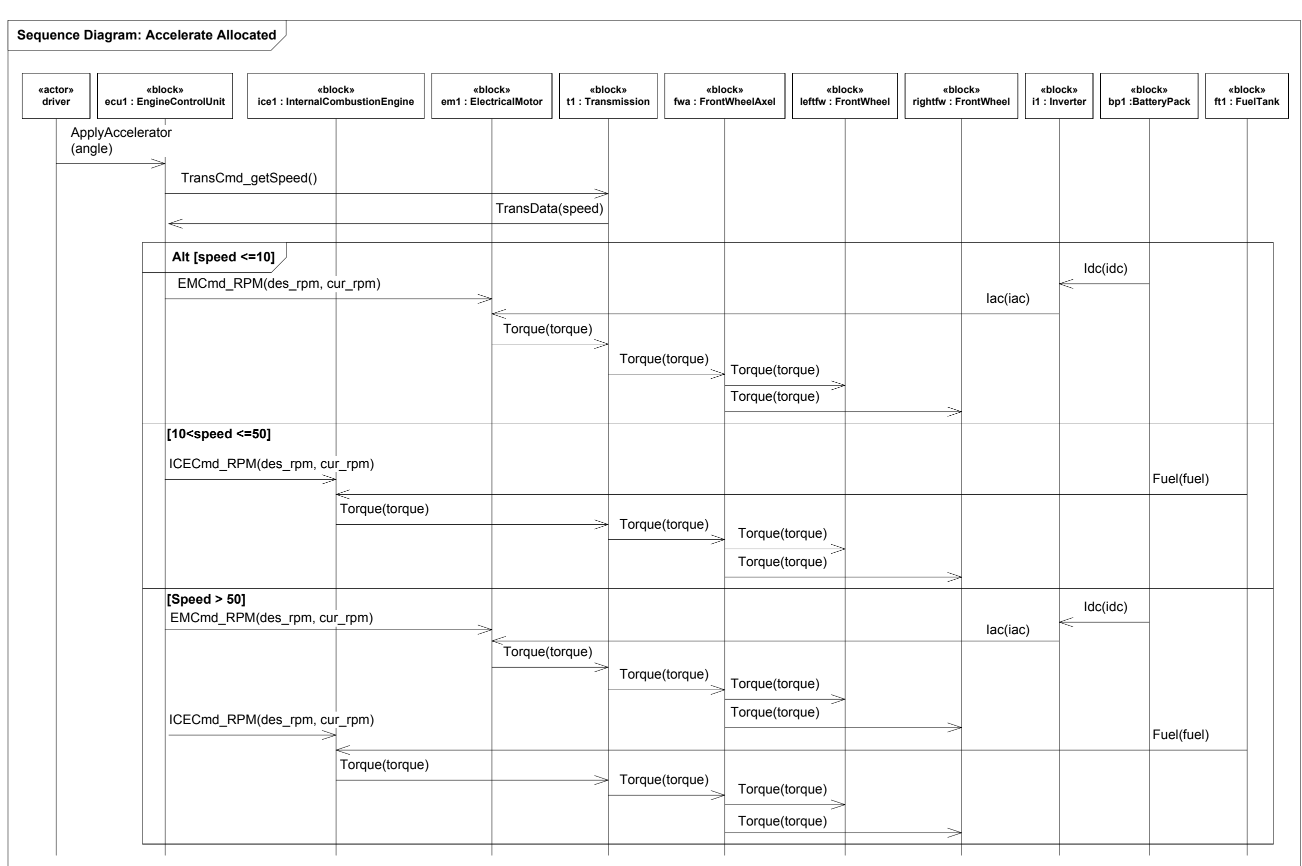


Figure 11-4. Acceleration with allocations

12 State Machines

12.1 Overview

The StateMachine package defines a set of concepts for modeling discrete behavior through finite state transition systems. A state machine represents behavior as the state history of an object in terms of its transitions and states. The activities that are invoked during the transition, entry, and exit of the states are specified along with the associated event and guard conditions. Activities that are invoked while in the state are specified as *do Activities*, and can be either continuous or discrete. A composite state has nested states that can be sequential or concurrent.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for state machines.

12.2 Diagram elements

The following tables describe the graphical nodes and paths for State Machines.

Table 12: Graphical nodes for State Machines.

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERANCE*	*COMPLIANCE*
Action	MinorReq := Id;	**UML::Statemachines:: BehavioralStatemachines:: Transition**	Basic
Choice pseudo state	[Id>10] d<=10]	**UML::Statemachines:: BehavioralStatema-chines::PseudoState**	Basic
Composite state	S Sb2 Sb1 Sb3	**UML::Statemachines:: BehavioralStatema-chines::State**	Basic
Entry point	again	**UML::Statemachines:: BehavioralStatema-chines::PseudoState**	Basic
Exit point	aborted	**UML::Statemachines:: BehavioralStatema-chines::PseudoState**	Basic
Final state		**UML::Statemachines:: BehavioralStatema-chines::PseudoState**	Basic
History, Shallow pseudo state	H	**UML::Statemachines:: BehavioralStatema-chines::PseudoState**	Basic

Table 12: Graphical nodes for State Machines.

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERANCE	COMPLIANCE
Initial pseudo state		**UML::Statemachines:: BehavioralStatemachines::PseudoState**	**Basic**
Junction pseudo state		**UML::Statemachines:: BehavioralStatemachines::PseudoState**	**Basic**
Receive signal action	Req(Id)	**UML::Statemachines:: BehavioralStatemachines::Transition**	**Basic**
Region	S	**UML::Statemachines:: BehavioralStatemachines::PseudoState**	**Basic**
Send signal action	TurnOn	**UML::Statemachines:: BehavioralStatemachines::Transition**	**Basic**
Simple state	S	**UML::Statemachines:: BehavioralStatemachines::State**	**Basic**
State Machine	ReadAmountSM aborted	**UML::Statemachines:: BehavioralStatemachines::Statemachine**	**Basic**

Table 12: Graphical nodes for State Machines.

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERANCE	COMPLIANCE
Submachine state	ReadAmount : ReadAmountSM ⊗ aborted	**UML::Statemachines:: BehavioralStatemachines::State**	**Basic**

Table 13: Graphical paths for State Machines.

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERANCE	COMPLIANCE
Transition	**Event [Guard]/ Action** →	**UML::Statemachines:: BehavioralStatemachines::Transition**	**Basic**

12.3 Package structure

Figure 12-1 shows the package structure for State Machines.

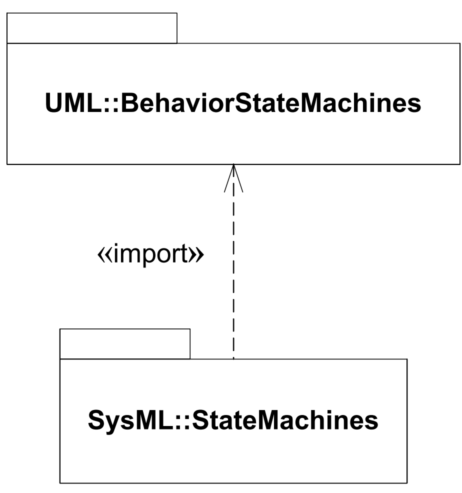


Figure 12-1. Package structure for State Machines.

12.4 UML extensions

No UML extensions are defined for State Machines.

12.5 Usage examples

The following diagrams illustrate how state machine diagrams are used. A complete sample problem that includes State Machine diagrams can be found in .

Figure 12-2 shows the state machine diagram that describes the dynamic behavior of a Transmission for the Activity "Shift". This version of the state machine diagram uses the notation commonly refered to as "state centric". In this notation the nodes represent the states and the trigger, guard, action associated with transitions are specified using text associated with the transition. The inverted fork notation in the lower left hand corner of the Reverse and Forward states indicates that these states have sub-states defined.

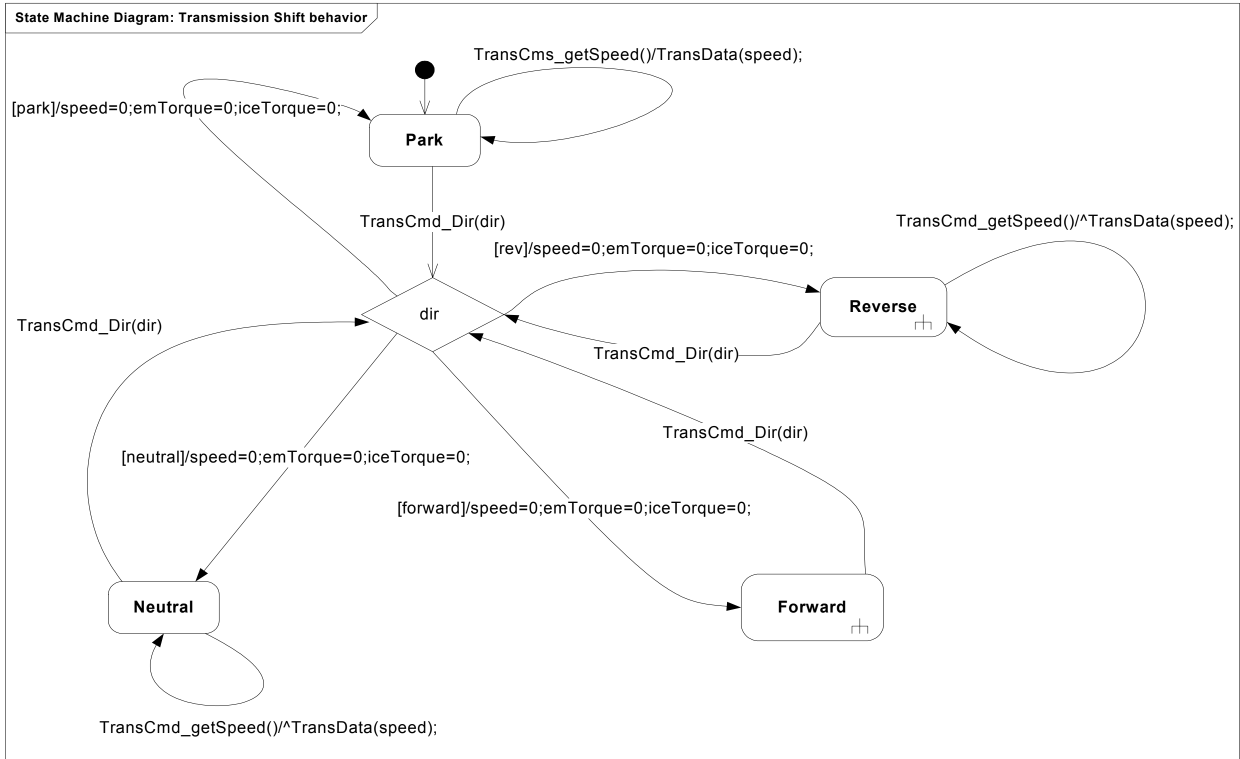


Figure 12-2. State-Centric State Machine Diagram: Transmission "Switch Gear" Behavior

Figure 12-3 shows the same state machine diagram drawn using the "transition centric" notation. Using this notation, triggers, actions and guards are shown as nodes on the diagram. Both diagrams are semantically equivalent. The choice of state centric or transition centric statemachine notation is a matter of modeler preference.

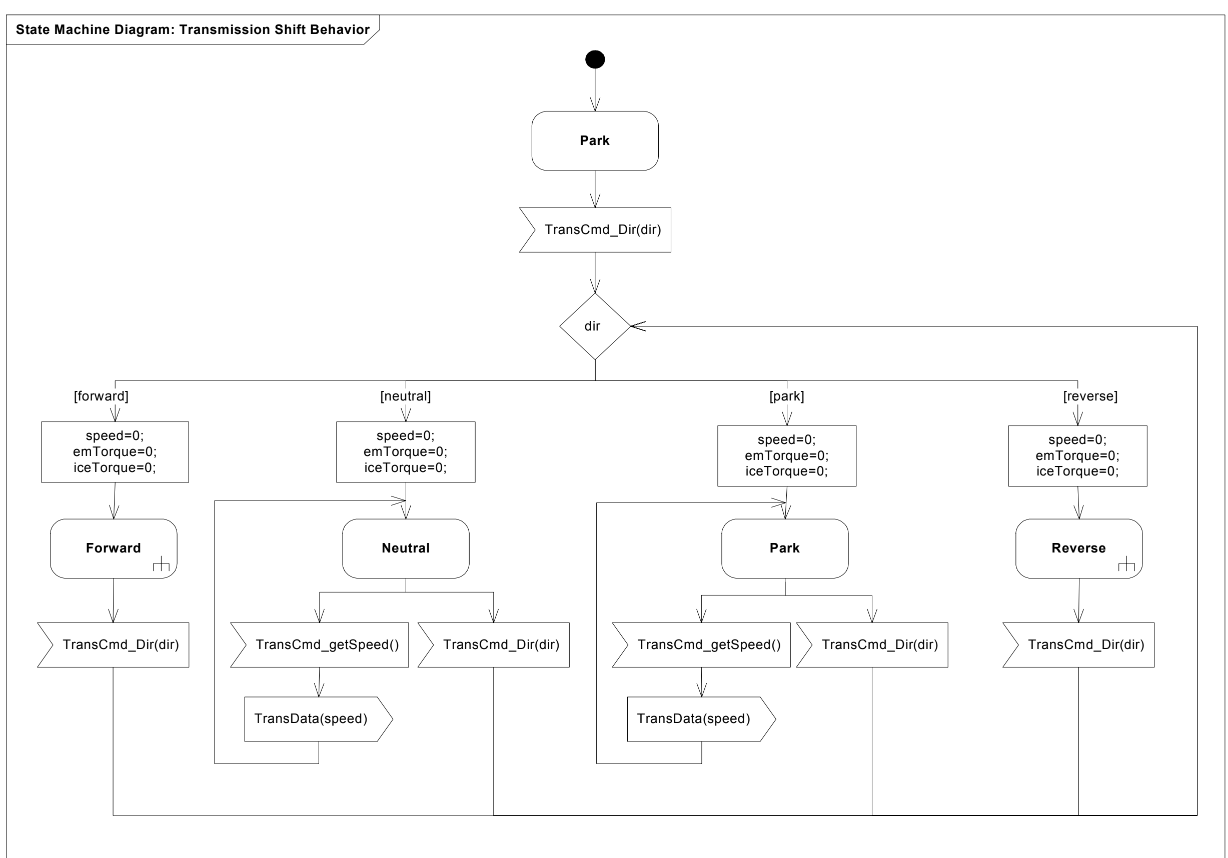


Figure 12-3. Transition-Centric State Machine Diagram: Transmission "Switch Gear" Behavior

13 Use Cases

13.1 Overview

The UseCases package defines a set of constructs for modeling required usages of a system. A use case diagram specifies the sequences of actions that a system can perform by interacting with outside agents (actors) to provide service transactions (use cases). Use case diagrams define use cases, actors and the associated communications between them. Actors may represent users, external systems, or other environmental entities. They may interact either directly or indirectly with the system. Actors may be specialized to represent a taxonomy of user types or external systems.

The subject of the use case can be represented via a system boundary. The use cases that are enclosed in the system boundary represent functionality that is realized by behaviors such as activity diagrams, sequence diagrams, and state machine diagrams.

The primary use case relationships are: *include*, *extend*, and *generalization*. The include relationship provides a mechanism for specifying common functionality which is shared among multiple use cases, and is always performed as part of the base use case. The extend relationship furnishes optional functionality that extends the base use case at a particular point under specific conditions. The generalization relationship provides a mechanism for specializing use cases.

Use Cases are powerful analysis aids during early analysis and design. They permit one to quickly identify external systems and/or users that interact with the system, to identify the associated external interfaces and to perform the initial, high-level functional decomposition (via the «include» relationship). Use Cases thus permit the high level description of behavior while bounding the system and providing context. They can be compared to the Context Diagram of the Data Flow Diagram (DFD) notation.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Use Cases.

13.2 Diagram elements

Table 14. *Graphical nodes for Use Cases.*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Use Case	UseCaseName	**UML::UseCases**	**Basic**
Actor	<<actor>> systemName	**UML::UseCases**	**Basic**
Subject	SubjectName	**Role name on Classifier**	**Basic**

Table 15. *Graphical paths for Use Cases.*

PATH TYPE	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Communication path		**UML::Classes::Kernel::Association**	**Basic**
Include	<<include>>	**UML::UseCases**	**Basic**
Extend	<<extend>>	**UML::UseCases**	**Basic**

13.3 Package structure

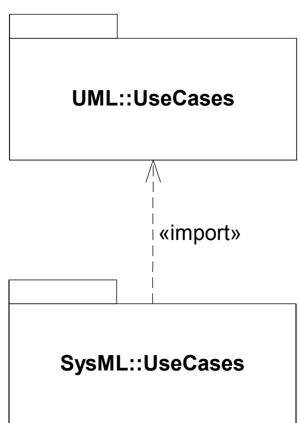


Figure 13-1. Package structure for SysML Use Cases.

13.4 UML extensions

No UML extensions are defined for Use Cases.

13.5 Usage examples

The following diagrams illustrate how Use Case diagrams are used. A complete sample problem that includes Use Case diagrams can be found in Appendix B.

Figure 13-2 shows the use case diagram for Hybrid SUV. The top level “Drive” use case is decomposed using «include» use cases. The subject (the HybridSUV) and the actors (Driver, Maintenance, IsuranceService and DMV) interact with the system to realize the use case.

This diagram aids in analysis by establishing the scope and context of the system under development (:HybridSUV), identifying key external entities (people, external systems, etc.) that interact with the system along with the associated external interfaces, and providing the initial high level decomposition of behavior according to key system threads or scenarios.

One example of re-use, the fact that the Brake use case is included in the Park and Drive use cases, is shown.

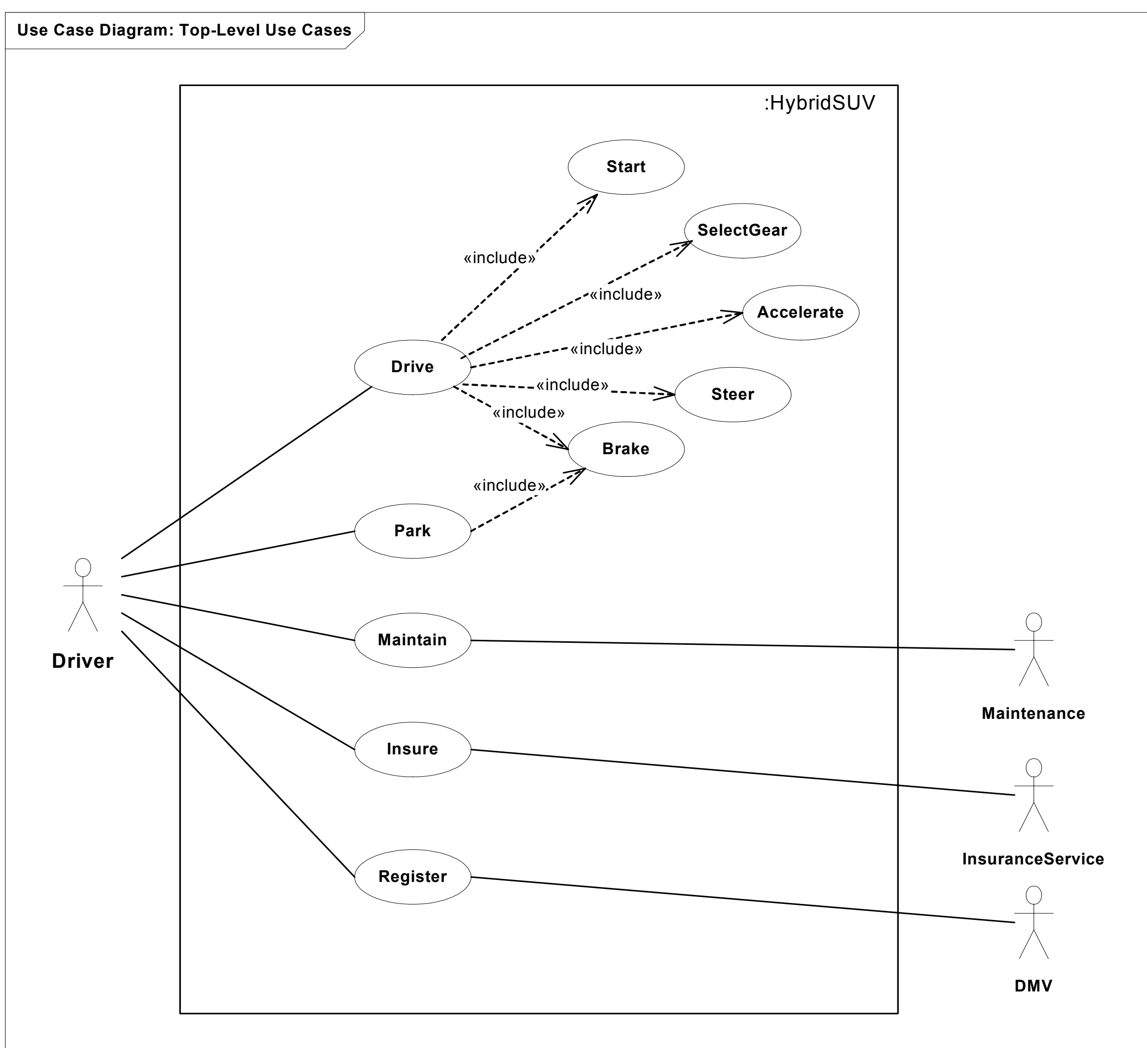


Figure 13-2. Top-Level Use Cases for HybridSUV system.

Part IV - Crosscutting Constructs

This Part specifies generic constructs that apply to both structure and behavior. It includes Requirements, Allocations, Model Management, Types, Auxiliary Constructs, and Profiles.

14 Requirements

14.1 Overview

A requirement specifies a capability or condition that a system must satisfy. A requirement may specify a function that a system must perform or a performance condition that a system must fulfill. SysML provides modeling constructs to represent requirements and relate them to other modeling elements.

A requirement can be decomposed into subrequirements, so that multiple requirements can be organized as a tree of compound requirements. Requirements can be related to each other, as well as to analysis, design, implementation and testing elements. A requirement can be generated or deduced from another requirement using the «derive» relationship. A requirement can be fulfilled by other model elements using the «satisfy» relationship. A requirement can be verified by various behaviors using the «verify» relationship. All of these are specializations of the UML «trace» relationship, which is used to track requirements and changes across models.

Modelers can categorize requirements by modifying their predefined properties, which include *id*, *source*, *text*, *kind*, *verifyMethod*, and *risk*. The latter three properties are defined via enumerations in Chapter 17, “Types”, and can be customized by SysML vendors and users using the non-normative enumerations defined in See Appendix D, “Non-Normative Model Library”.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Requirements.

14.2 Diagram elements

This section describes the concrete syntax for graphical nodes and paths in Requirement diagrams.

Table 1. *Graphical nodes for Requirements.*

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Requirement	«requirement» RequirementName «requirement» id = "MyID" source = "Marketing" text = "Requirement description" kind = "Performance" verifyMethod = "Analysis" risk = "High"	**SysML::Requirements:: Requirement**	**Basic**
TestCase	«testCase» TestCaseName	**SysML::Requirements:: TestCase**	**Basic**

Table 2. *Graphical paths for Requirements.*

PATH TYPE	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Composition		**UML::Classes::Kernel::Property with aggregation equal composite**	**Basic**
Derive	«derive»	**SysML::Requirements**	**Basic**
Satisfy	«satisfy»	**SysML::Requirements**	**Basic**
Verify	«verify»	**SysML::Requirements**	**Basic**

14.3 Package structure

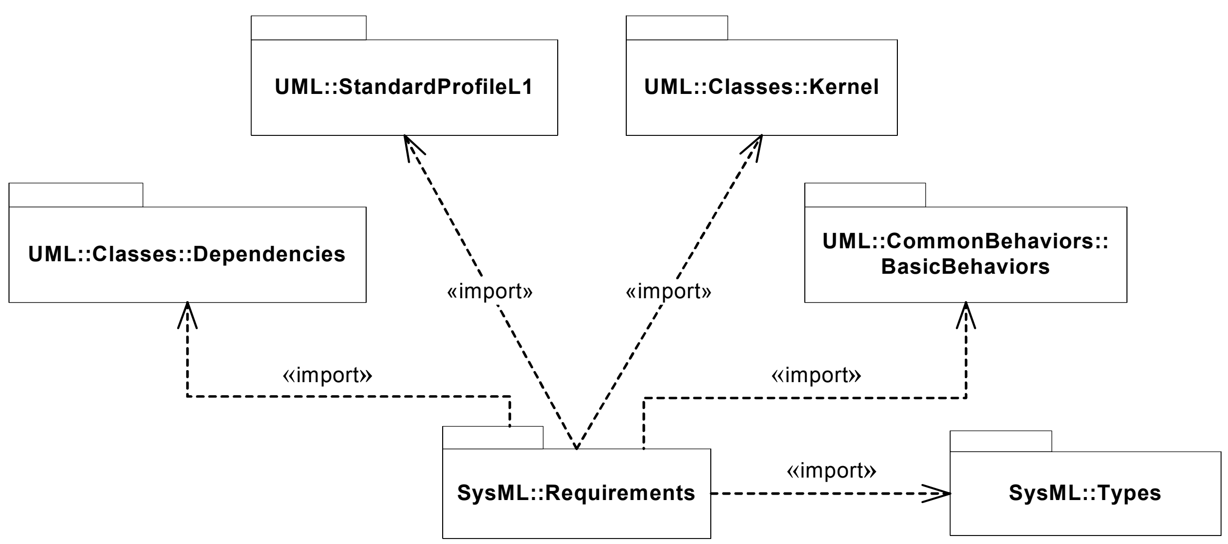


Figure 14-1. Package structure for Requirements.

14.4 UML extensions

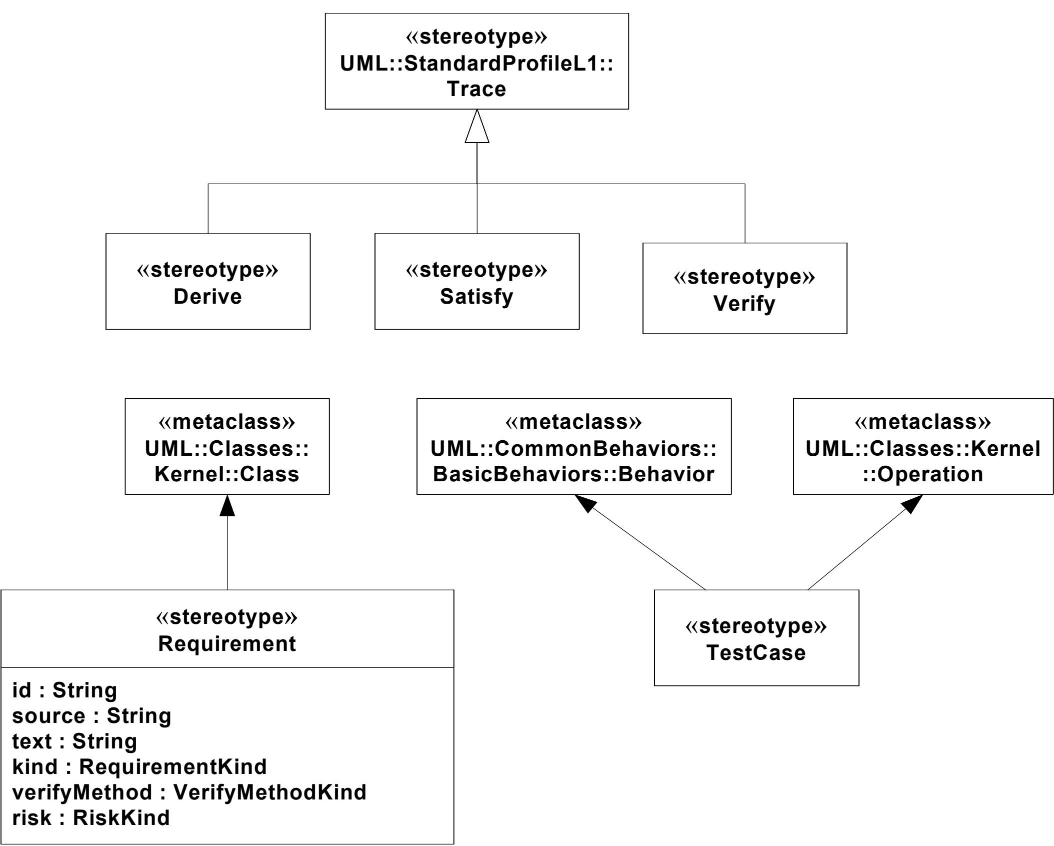


Figure 14-2. Abstract syntax for Requirements.

14.4.1 Stereotypes

14.4.1.1 Derive

Definition

A Derive relationship is a trace dependency between a derived requirement and a source requirement, where the derived requirement is generated or inferred from the source requirement.

Description

Derive is a specialization of the UML trace dependency. For example, a design requirement might derive from an analysis requirement, or a measure of effectiveness might derive from an analysis requirement. The arrow direction points from the derived requirement to the original requirement from which it is derived.

Constraints

[1] The source element must be an element stereotyped by «requirement».

[2] The target element must be an element stereotyped by «requirement».

14.4.1.2 Requirement

Definition

A Requirement specifies a capability or condition that a system must satisfy. A requirement may define a function that a system must perform or a performance condition that a system must fulfill. Requirements are used to establish a contract between the customer (or other stakeholder) and those responsible for designing and implementing the system.

Description

Requirement is a stereotype of Class. Composite requirements can be created by using the composition association. The default interpretation of a composite requirement, unless stated differently by the composite requirement itself, is that all its component requirements must be satisfied for the composite requirement to be satisfied.

Attributes

id	: String	The identifier of the requirement.
source	: String	The origin of the requirement, which can be expressed as a document title, a reference to document title, a reference to a stakeholder, or a stakeholder type (i.e., Marketing).
text	: String	The textual description or a reference to the textual description of the requirement.
kind	: RequirementKind	The kind/classification of the requirement.
risk	: RiskKind	The exposure severity (product of probability of the risk occurring and consequence of the risk occurring).
verifyMethod	:VerifyMethodKind	Specifies the applicable method for verifying the requirement.

Constraints

[1] The property *isAbstract* must be set to *true*.

[2] The only associations allowed for classes stereotyped by «requirement» are aggregation associations with other classes stereotyped by «requirement» where *isComposite* must be set to *true* and the lower bound for multiplicity must be set to zero.

[3] The property *ownedOperation* must be empty.

[4] The subtypes of a class stereotyped by «requirement» must also be stereotyped by «requirement».

[5] A nested classifier of a class stereotyped by «requirement» must also be a requirement.

14.4.1.3 RequirementKind (user defined enumeration)

A RequirementKind is a enumeration specifying classification categories for requirements. This enumeration is defined in the chapter SysML::Types (and corresponding SysML package) to facilitate user customization.

14.4.1.4 RiskKind (user defined enumeration)

A RiskKind is a user defined enumeration specifying risk categories for requirements. This enumeration is defined in the chapter SysML::Types (and corresponding SysML package) to facilitate user customization.

14.4.1.5 Satisfy

Definition

A Satisfy relationship is dependency between a supplier requirement and a client model element that fulfills the requirement.

Description

Satisfy is a specialization of the UML trace dependency. As with other dependencies, the arrow direction points from the satisfying (client/source) model element to the (supplier/target) requirement that is satisfied.

Constraints

[1] The target must be an element stereotyped by «requirement» or by one of «requirement» subtypes.

[2] The source must be an element that is not stereotyped by «requirement».

14.4.1.6 TestCase

Definition

A test case is a behavior or operation that specifies how a requirement is verified. A test case can address one or more verification methods. A test case always returns a verdict.

Description

Test Case is a stereotype of Behavior and Operation. A SysML test case is compatible with test case as it is defined in the *UML Profile for Testing*, but is not equivalent to it, since the semantics of SysML test case are more limited.

Constraints

[1] The type of the return result parameter of a test case must be Verdict.

14.4.1.7 Verdict (a user defined enumeration)

A verdict is a user defined enumeration specifying the set of possible evaluation results for a test case. This enumeration is defined in the chapter SysML::Types (and corresponding SysML package) to facilitate user customization.

14.4.1.8 Verify

Definition

A Verify relationship is a trace dependency between a supplier requirement and a client test case that determines whether a system fulfills the requirement.

Description

Verify is a specialization of the UML trace dependency. As with other dependencies, the arrow direction points from the (client/source) model element to the (supplier/target) requirement.

Constraints

[1] The target must be an element stereotyped by «requirement» or by one of «requirement» subtypes.

[2] The source must be an element stereotyped by «testCase».

14.4.1.9 VerifyMethodKind (user defined enumeration)

A VerifyMethodKind is a user defined enumeration specifying the methods that can be used for verifying a requirement. This enumeration is defined in the chapter SysML::Types (and corresponding SysML package) to facilitate user customization.

14.4.2 Table extensions

In order to increase the density of information, Requirements trace dependencies can also be shown using a compact tabular presenation format called a Requirements Traceability Table. A Requirements Traceability Table should contain, but is not limited to, the following information:

- Trace [Dependency] Name - the name of the trace dependency
- Trace [Dependency] Kind - the type of the trace dependency (e.g., derive, satisfy, verify)
- Source Name - the name of the source of the trace dependency
- Source Kind - the type of the source (e.g., requirement, activity, block)
- Target Name - the name of the target of the trace dependency
- Target Kind - the type of the target (e.g., requirement, block)

14.5 Usage examples

The following diagrams illustrate how Requirements diagrams are used. A complete sample problem that includes Requirements diagrams can be found in Appendix B, “Sample Problem”.

Figure 14-3 shows how a compound requirement can be decomposed into multiple sub-requirements.

.

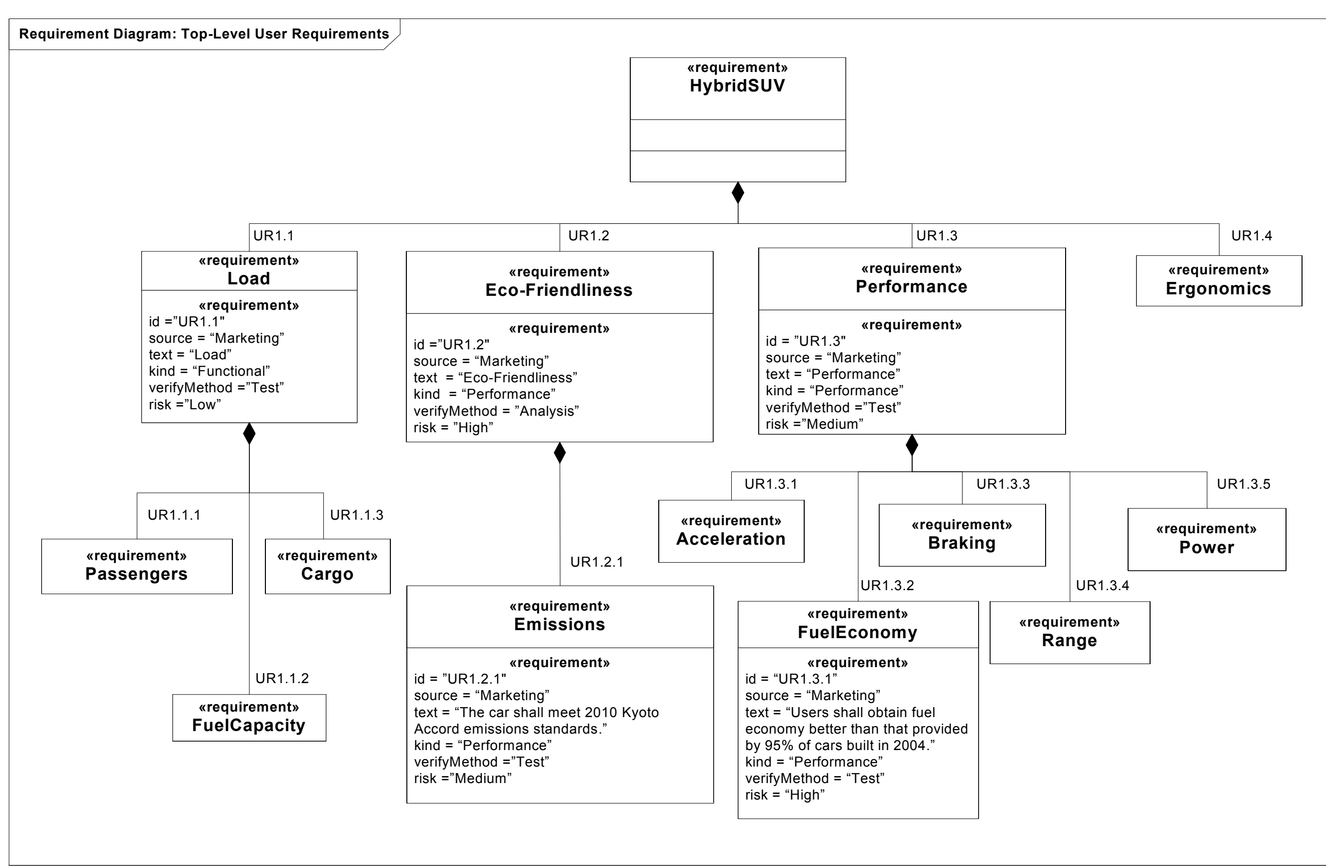


Figure 14-3. Decomposition of a compound requirement.

Figure 14-4 shows how requirements can be derived from other requirements.

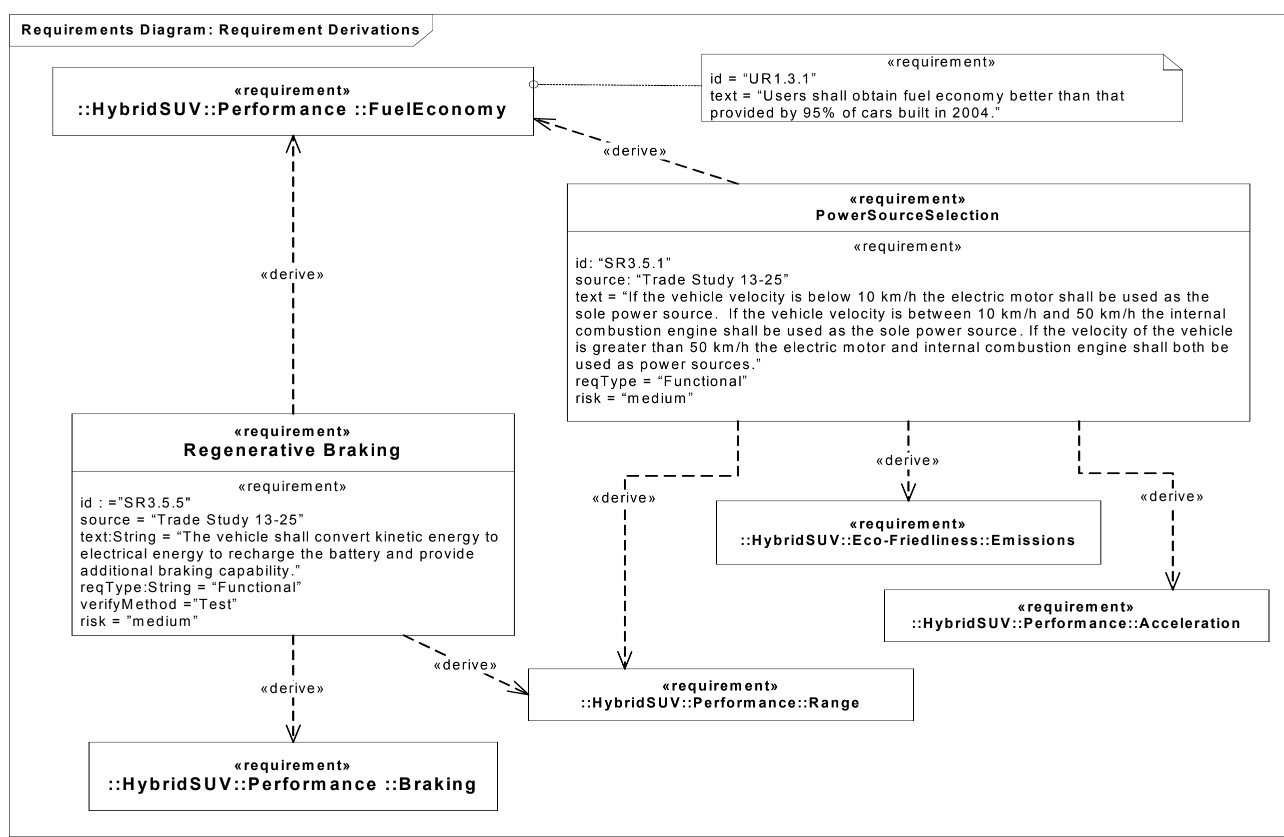


Figure 14-4. Requirements derivation using compact stereotype notation.

Figure 14-5 shows how model elements can satisfy requirements..

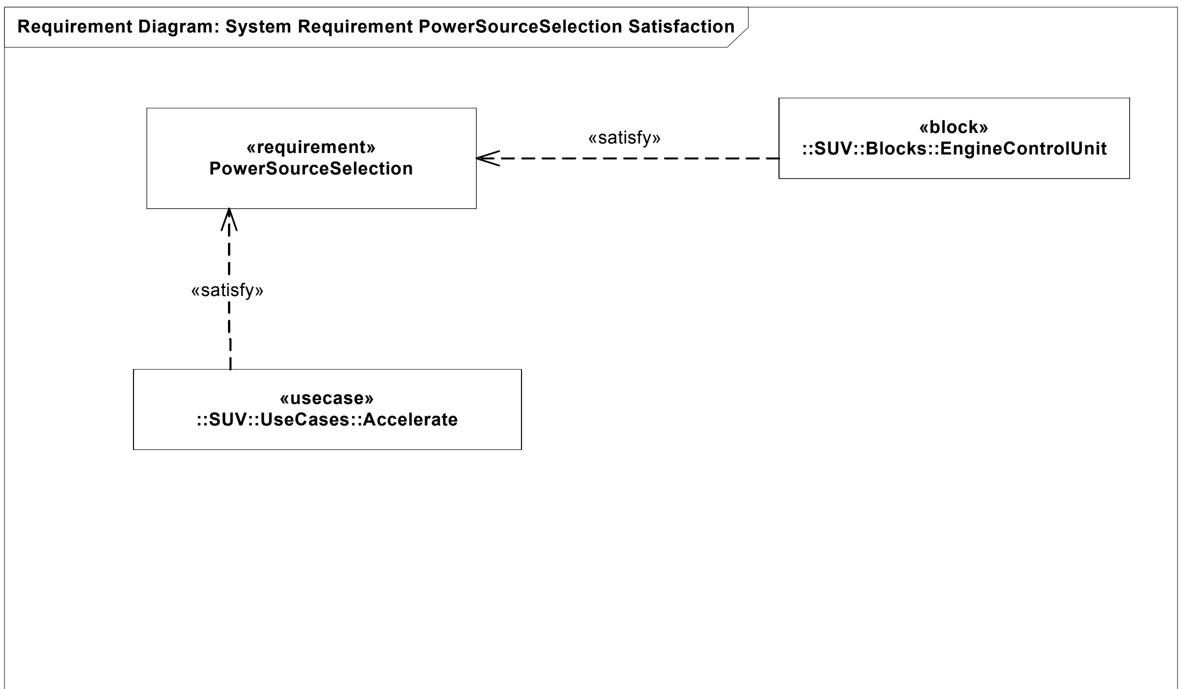


Figure 14-5. Requirements satisfaction.

Figure 14-6 shows how requirements can be verified by test case behaviors.

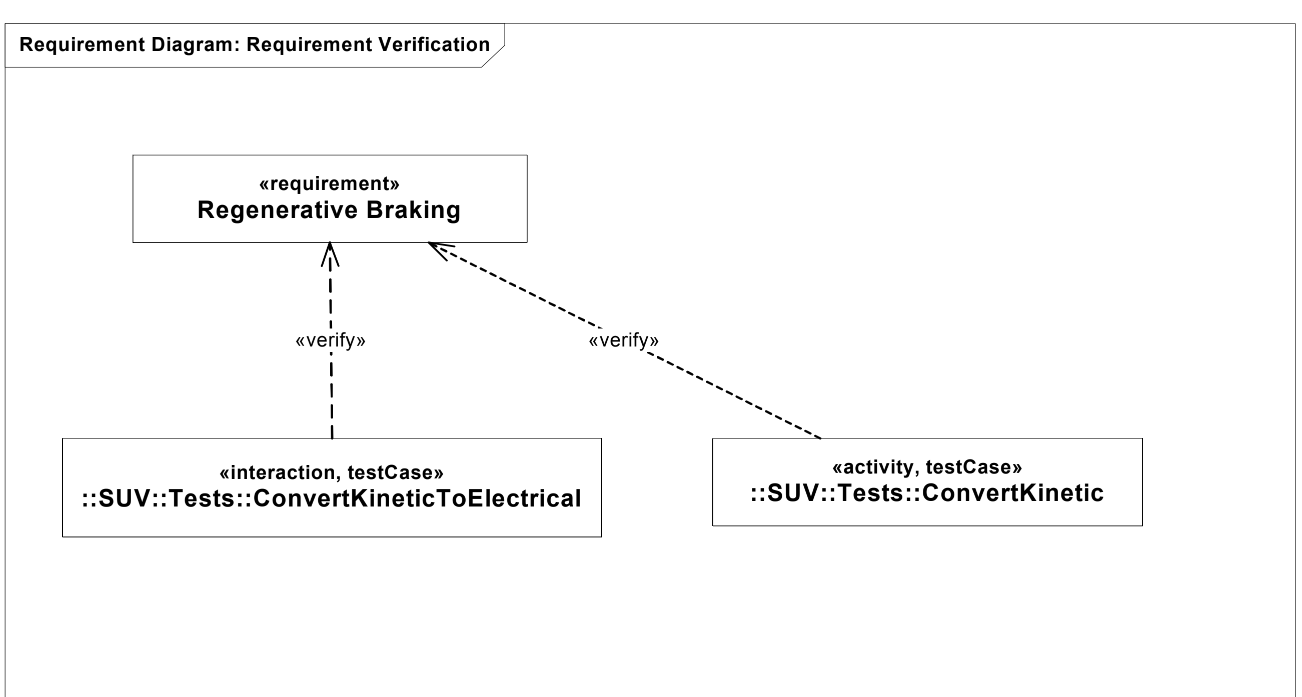


Figure 14-6. Requirements verification.

Figure 14-7 shows a wide range of requirement traces, including derive, satisfy and verify trace dependencies, as well as allocation relationships.

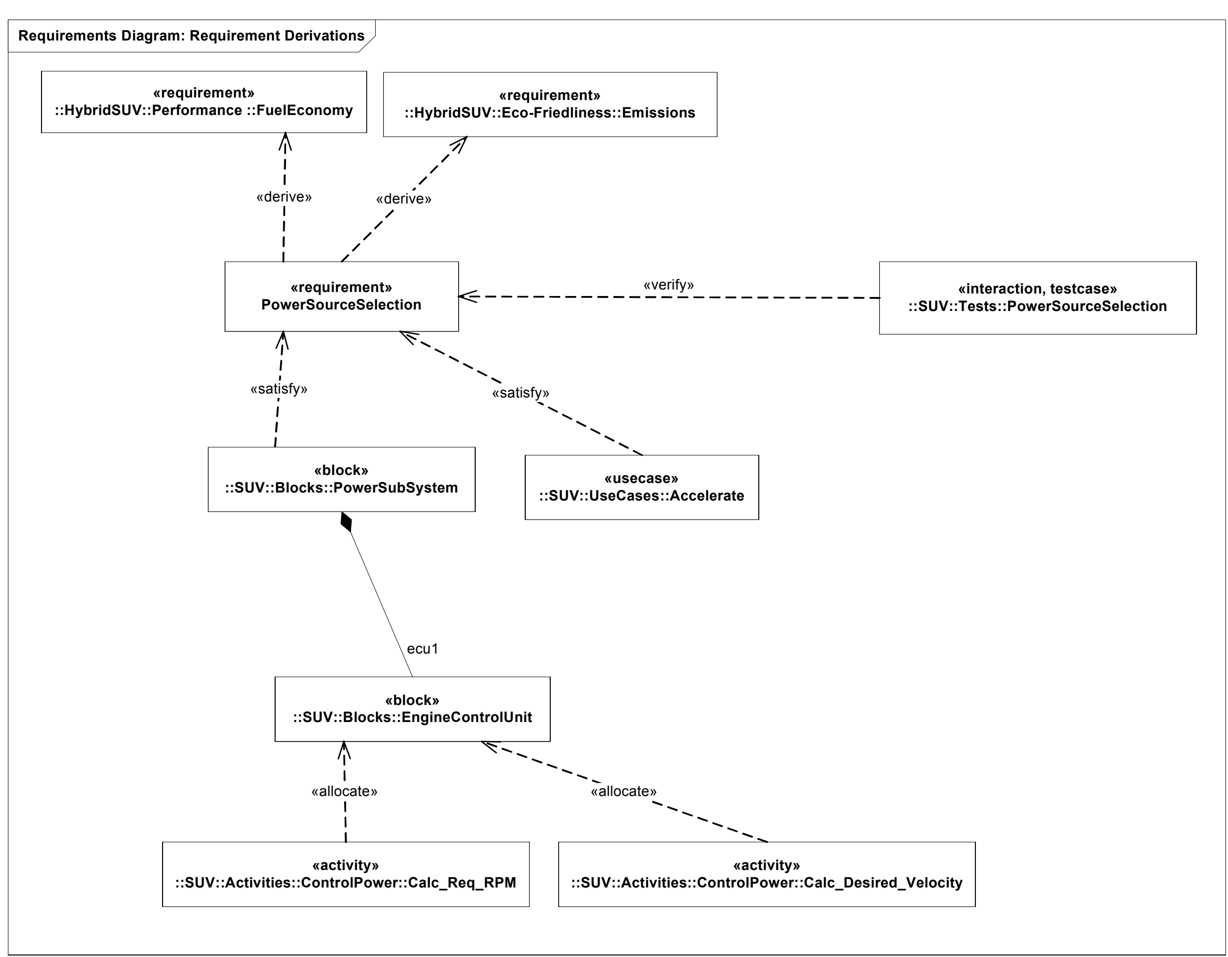


Figure 14-7. Requirements traceability

Figure 14-8 shows an example of a requirement trace dependency table generated from Figure 16-7.

Dependency	TraceKind	SourceName	SourceKind	TargetName	TargetKind
of ::SUV::Blocks::EngineControlUnit	allocate	Calc_Desire_vel	Activity	EngineControlUnit	Block
of ::SUV::Blocks::EngineControlUnit	allocate	Calc_Req_RPM	Activity	EngineControlUnit	Block
of ::SUV::Requirements::HybridSUV::Performance::FuelEconomy	derive	PowerSourceSelection	Requirement	FuelEconomy	Requirement
of HybridSUV::'Eco-Friendliness'::Emissions	derive	PowerSourceSelection	Requirement	Emissions	Requirement
of ::SUV::Tests::PowerSourceSelection	satisfy	PowerSubSystem	Block	PowerSourceSelection	Requirement
of ::SUV::Tests::PowerSourceSelection	satisfy	Accelerate	Usecase	PowerSourceSelection	Requirement
of ::SUV::Tests::PowerSourceSelection	verify	PowerSourceSelection	Interaction	PowerSourceSelection	Requirement

Figure 14-8. Example of requirements trace dependency table

15 Allocations

15.1 Overview

It is a systems engineering best practice to separate structure (form) from behavior (function) so that designs can be optimized by considering several different structures that provide the desired emergent behavior and properties. This approach provides the required degrees of freedom (in particular, how to decompose structure, how to decompose behavior, and how to relate the two) to optimize designs based upon trade studies among alternatives. The implication is that an explicit set of relationships must be maintained between form and function for each althernative. These relationships are known as allocation relationships. A more formal definition of allocation follows: Allocation is the term used by systems engineers to describe a design decision that assigns responsibility for meeting a requirement (requirements allocation) or implementing a behavior (functional allocation) to structural elements of the system.

In object oriented software design the engineer also performs allocation of function to form, both explicitly and implicitly. On a class diagram (cf. Block Definition diagram) the operations defined on a class (cf. Block) explicitly define the allocation of responsibility to the class for providing the associated behavior (see Chapter 9 for more on Blocks). In a sequence diagram, a message sent to a lifeline implicitly defines that that the receiving part will provide the associated behavior (see Chapter 12 for more on Sequences). In activity diagrams the placement of an activity in a partition implicitly defines that the part represented by the partition will provide the associated behavior (see Chapter 11 for more on Activities).

However, the systems engineer's concept of "allocation" requires a flexibility of expression suitable for abstract system specification and analysis. When searching for optimum designs, systems engineers are often required to associate behavior and structure in abstract, preliminary, and sometimes tentative ways. It is inappropriate to force the systems engineer into the detail of rigorous specification of object oriented methods too early in the development of a system architecture or design. The application of rigorous methods will surely follow, once the design decisions and made and behavior and structure models are more fully expressed. In the meantime, it is important and appropriate for systems engineers to use the notion of allocation to assess just how well the system "hangs together".

The various types of elements generally associated with one another in practice have given rise to various uses of the word "allocation". This chapter does not try to limit the use of the term "allocation", but to provide a basic capability to support allocation in the broadest sense, that of capturing an early design decisions.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Allocations.

15.2 Diagram elements

Table 3. *Graphical nodes for Allocations.*

NODE TYPE	CONCRETE SYNTAX	ABSTRACT SYNTAX	COMPLIANCE
Allocation derived properties displayed in compartment of Block.	BlockName «allocated» {allocatedFrom= ElementName} {allocatedTo= ElementName}	**SysML::Allocations:: Allocated**	**Basic**

Table 3. *Graphical nodes for Allocations.*

Node Type	*Concrete Syntax*	*Abstract Syntax*	*Compliance*
Allocation derived properties displayed in Comment.	«allocated» {allocatedFrom= ElementName} {allocatedTo= ElementName} ElementName	**SysML::Allocations::Allocated**	**Basic**

Table 4. *Graphical paths for Allocations.*

Path Type	*Concrete Syntax*	*Abstract Syntax*	*Compliance*
Allocation	«allocate»	**SysML::Allocations::Allocate**	**Basic**

15.3 Package structure

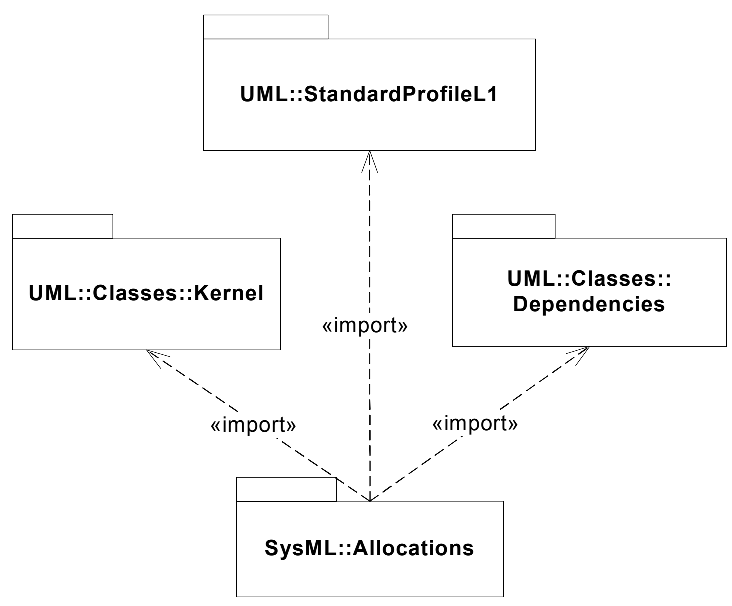


Figure 15-1. Package structure for Allocations.

15.4 UML extensions

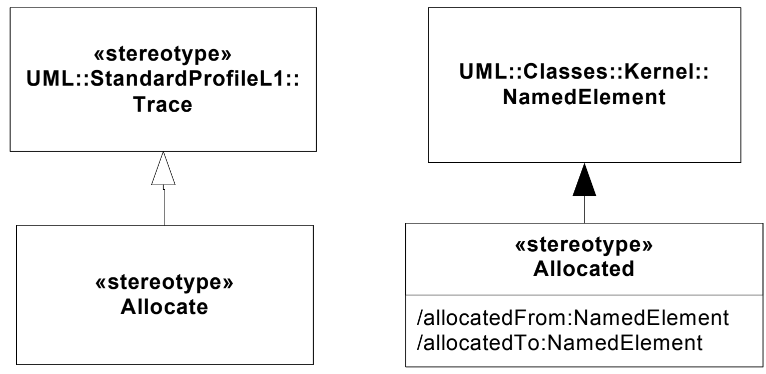


Figure 15-2. Abstract syntax for Allocations.

15.4.1 Stereotypes

15.4.1.1 Allocate

Description

Allocation is a mechanism for associating elements of different types, or in different hierarchies, at an abstract level. Allocation is used for assessing user model consistency, completeness (eg. is all behavior allocated to at least one block) and directing future design activity. It is expected that an «allocate» relationship between model elements is a precursor to a more concrete relationship between the elements, their properties, operations, attributes, or sub-classes.

Allocation is a subtype of the UML «trace» dependency permissable between any two NamedElements. It is directional - one NamedElement is the source, and one NamedElement is the target of an allocation.

Per systems engineering convention, the arrowhead end (target) of the «allocate» trace must be the element "allocated to".

The «allocate» trace may be further subtyped by the user with particular constraints regarding element type and attributes (see Chapter 19 Profiles and Model Libraries for information on extending SysML).

Constraints

[1] A single «allocate» trace shall have only one source (no arrowhead) and one target (arrowhead).

[2] Subtypes of the «allocate» trace should have constraints applied to constrain source and target types as appropriate.

15.4.1.2 Allocated

Description

Allocated applies to model elements that have at least one allocation relationship with another model element. Allocated elements may be either the source or target of an «allocate» trace.

The «allocated» stereotype provides a mechanism for a particular model element to conveniently retain and display the element at the opposite end of any «allocate» trace. This stereotype provides for the properties "allocatedFrom" and "allocatedTo", which are derived from the «allocate» trace.

Attributes

The following properties are derived from any «allocate» trace:

/allocatedTo – the set of elements that are the targets of an «allocate» trace whose source is extended by this stereotype (instance). This property is the union of all targets to which this instance is the source, i.e. there is only one /allocatedTo property per allocated model element.

/allocatedFrom – reverse of allocatedTo: the set of elements that are sources of an «allocate» whose target is extended by this stereotype (instance). The same characteristics apply as to /allocatedTo.

15.4.2 Diagram extensions

An «allocate» relationship is represented diagrammatically by a dependency with the keyword «allocate».

The properties /allocatedFrom and /allocatedTo may be displayed in a compartment, or in a comment. In both cases, curly braces {} are used to express the property as shown below:

{allocatedFrom= ElementName}

{allocatedTo= ElementName}

When applied to a classifier, an additional compartment may be used to display the «allocated» stereotype and its properties. The allocation compartment may be elided from the diagram. Basic compliance requires the allocation compartment for Blocks only. Advanced compliance requires the allocation compartment for Actions and Parts.

A comment may be used with any NamedElement. When used, the stereotype «allocated» will be used to distinguish the it from a constraint. Comments may be used on associations, including ActivityEdges and Connectors. (Basic compliance)

15.4.3 Table extensions

In order to increase the density of information, allocation trace dependencies can also be shown using a compact tabular presentation format called an Allocation Traceability Table. An Allocation Traceability Table must contain, but is not limited to, the following information:

- Source Name - the name of the source of the allocation
- Source Kind - the type of the source
- Target Name - the name of the target of the allocation
- Target Kind - the type of the target

See Section 15.5 for an example of a tabular representation of allocations.

15.5 Usage examples

The following examples are provided as an overview for representing allocation in SysML diagrams.

Figure 15-3 shows the allocation of sub-activities of the ControlPower and BrakeCar activities defined in Chapter 11 to the Transmission Block of the Hybrid SUV defined in Chapter 9. An «allocate» trace is drawn from from each activity to the Transmission block. Also shown is the «allocated» stereotype applied to the source and target of each «allocate» trace as a result of the allocation. The presence of the «allocated» keyword provides a useful means of initial coverage analysis as one can quickly scan diagrams to assess if all activities have been allocated and that all blocks have at least one behavior allocated to them.

It is important to note that «allocate» traces can be created between elements in the model without the need to show them on a diagram. This is important for two reasons: 1) in some cases it is not possible to put both elements involved in the trace relationship on the same diagram, for example connectors and activity edges; and 2) in many cases it will be more convenient to create the traces by other means (ex. tools could provide a matrix view to establish allocations, or right-click menues, or other means).

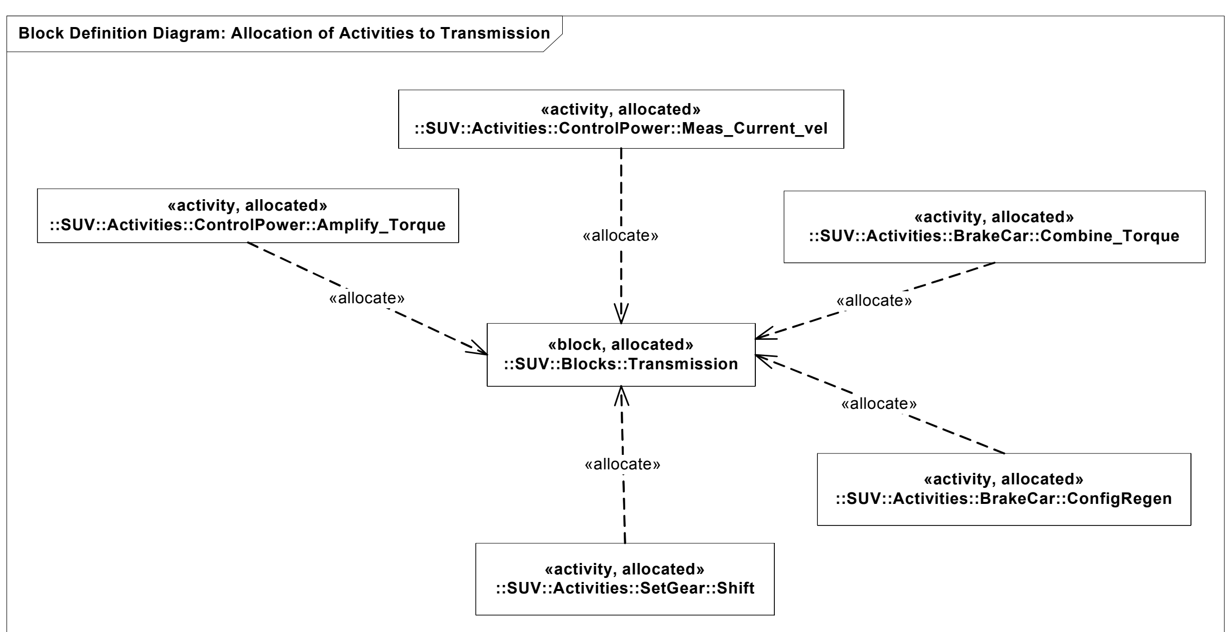


Figure 15-3. Block Definition Diagram:Allocation of behavior to the Transmission

Figure 15-4 shows the properties of the Transmission Block in a separate Block Definition diagram. In this case the «allocate» dependencies are not visible, however the «allocated» properties are displayed in a comment symbol.

As a result of the allocations, the model has been further elaborated to show operations of the Transmission Block which correspond to the allocations made.

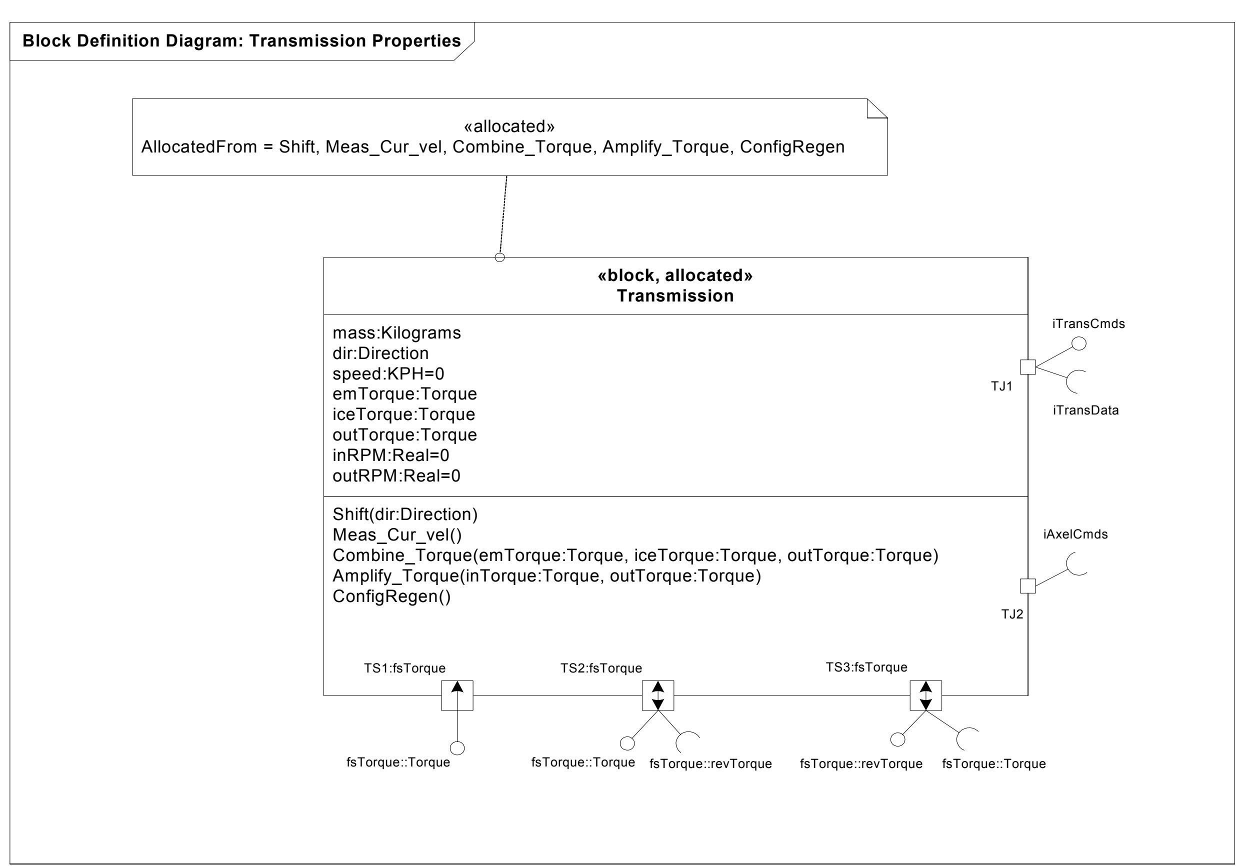


Figure 15-4. Properties of the Transmission Block showing allocations

Figure 15-5 shows an Allocation Traceability Table, which is an alternate tabular presentation of the allocation traces in the model. This table could easily be generated by tools via queries on the model.

Source Activity	Target Block								
	EngineControlUnit	InternalCombustionEngine	Transmission	FrontWheelAxel	FrontWheel	ElectricalMotor	BrakingSubsystem	Inverter	BatteryPack
InitializeICE	allocate								
StartICE		allocate							
SwitchGear	allocate								
Shift			allocate						
Calc_Desire_vel	allocate								
Meas_Current_vel			allocate						
Calc_Req_RPM	allocate								
Amplify_Torque			allocate						
Split_Torque				allocate					
Provide_Traction					allocate				
Prod_Torque		allocate							
Prod_Torque						allocate			
Brake							allocate		
SetRegenBrake	allocate								
ConfigRegen			allocate						
Combine_Torque			allocate	allocate					
GenerateAC						allocate			
RectifyCurrent								allocate	
StoreElectricalEnergy									allocate
ConvertDCtoAC								allocate	

Figure 15-5. Example Allocation Table

16 Model Management

16.1 Overview

The Model Management package defines a set of constructs for managing the complexity of SysML models. The primary constructs used for managing model complexity are packages and views. A package is a generic mechanism for grouping modeling constructs, whereas a view is a stereotype of package that is an abstraction of a whole system, and that addresses one or more concerns of the system stakeholders. A view conforms to its viewpoint, which specifies the purpose, stakeholders, stakeholder concerns, language selections and method selections associated with the view.

The defintions of view and viewpoint used by SysML are intended to be compatible with the *IEEE 1471 Recommended Practice for Architecture Description*. See See "Non-Normative References" on page 7.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Model Management.

16.2 Diagram elements

Table 5. *Graphical nodes for Model Management.*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
PackageDiagram	Package AcmeSystem Package1 <<import>> Package2	**N/A**	**Basic**
Package	PackageName	**UML::Classes::Kernel::Package**	**Basic**
View	<<view>> ViewName	**SysML::ModelManagement::View**	**Basic**

Table 5. *Graphical nodes for Model Management.*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Viewpoint	«viewpoint» Distribution «viewpoint» purpose = "To define ..." stakeholders = "Planners, ..." concerns = "What talks to ..." languages = "SysML" methods = "transport paradigms"	**SysML::ModelManagement::Viewpoint**	**Basic**

Table 6. *Graphical paths for Model Management.*

PATH NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Conform	«conform»	**SysML::ModelManagement::Conform**	**Basic**
PackageImport	«import»	**UML::Classes::Kernel::ElementImport with visibility = public**	**Basic**
PackageAccess	«access»	**UML::Classes::Kernel::ElementImport with visibility = private**	**Basic**
PackageContainment		**UML::Classesss::Kernel::Package::ownedElement**	**Basic**
Trace	«trace»	**UML::StandardProfileL1::«trace»**	**Basic**

16.3 Package structure

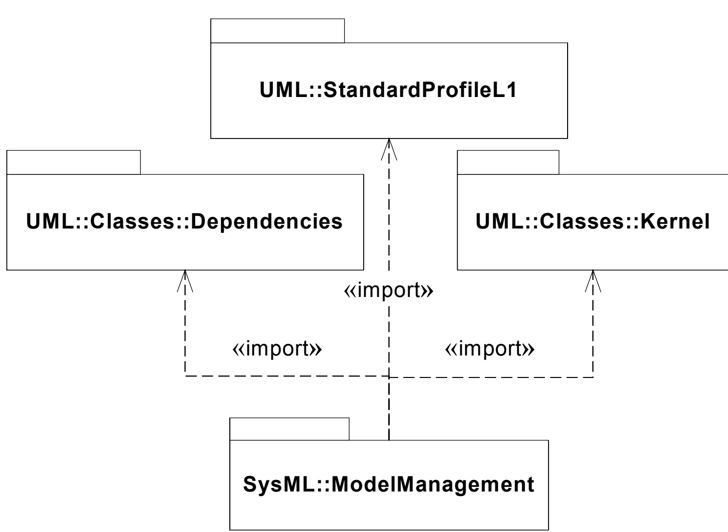


Figure 16-1. Package structure for Model Management.

16.4 UML extensions

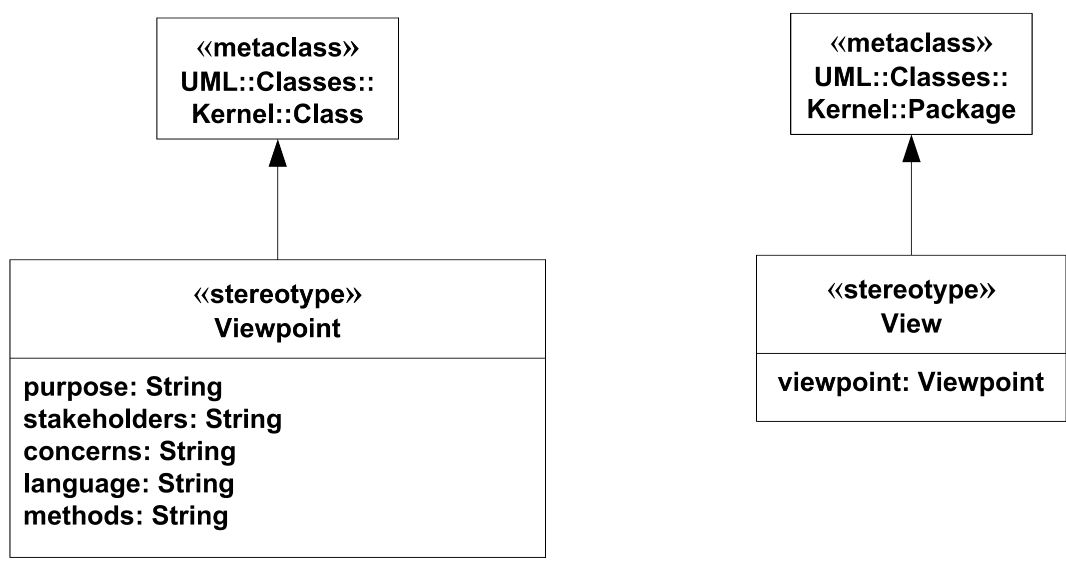


Figure 16-2. Abstract syntax for Model Management.

16.4.1 Stereotypes

16.4.1.1 Conform

Definition

A Conform relationship is dependency between a supplier viewpoint and a client view that fulfills the requirement.

Description

Conform is a specialization of the UML trace dependency. As with other dependencies, the arrow direction points from the (client/source) view to the (supplier/target) viewpoint to which it conforms.

Constraints

[1] The target must be an element stereotyped by «viewpoint».

[2] The source must be an element that is stereotyped by «view».

16.4.1.2 View

Definition

A View is an abstraction of a whole system that addresses one or more concerns of the system stakeholders. A view has only one viewpoint.

Description

View is a stereotype of package.

Attributes

viewpoint : Viewpoint [1] The viewpoint that is expressed by the view.

Constraints

N/A

16.4.1.3 Viewpoint

Definition

A Viewpoint specifies the purpose, stakeholders, stakeholder concerns, language selections and method selections related to a view.

Description

View is a stereotype of Class.

Attributes

purpose	: String	The intention for defining the viewpoint.
stakeholders	: String	The individuals or organizations that have concerns about the system.
concerns	: String	The interest of the stakeholders.
languages	: String	The methods used to specify the viewpoint.

methods : String The methods used to specify the viewpoint.

Constraints

N/A

16.4.2 Diagram extensions

N/A

16.5 Usage examples

The following diagrams illustrate how Model Management diagrams are used.

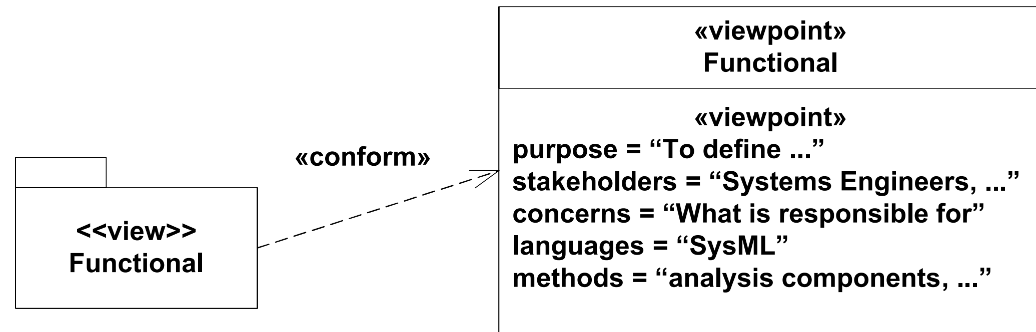


Figure 16-3. Functional view conforming to its viewpoint.

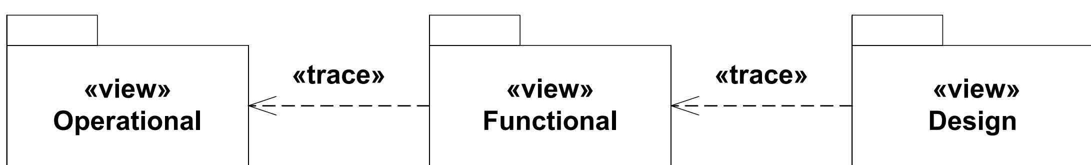


Figure 16-4. Traceability across views: black box perspective

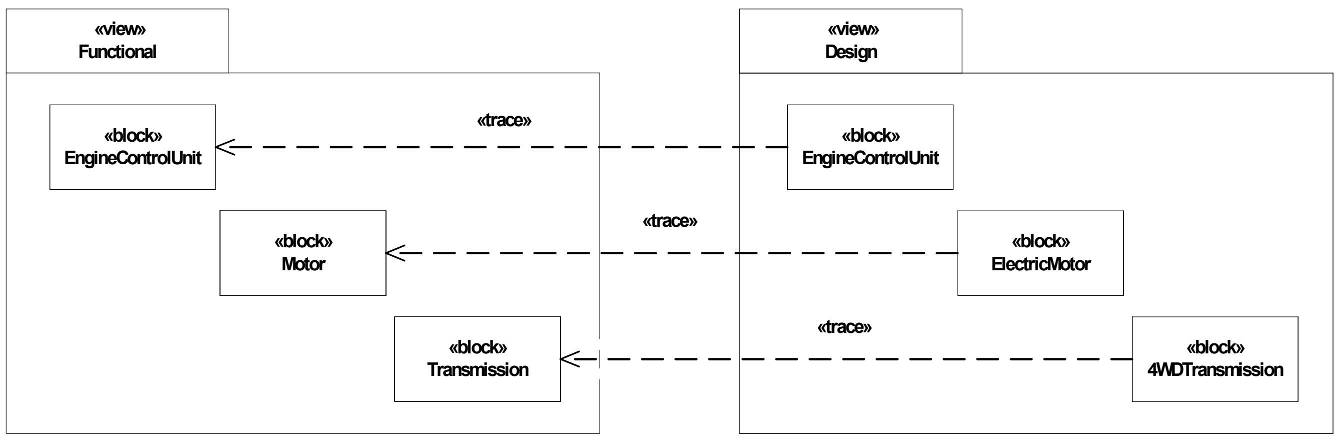


Figure 16-5. Traceability between Functional and Design views: white box perspective

17 Types

17.1 Overview

This chapter defines a number of enumerations used by the SysML profile. Specifically, the enumerations RequirementKind, RiskKind, VerifyMethodKind and Verdict used by the Requirements package and the enumeration ControlValue used by the Activities packages are specified. These enumerations are defined in this package, with no enumeration literals defined and specialized in the non-normative model library to provide a default set of enumeration literals.

This architecture has been used to provide a flexible mechanism for users to tailor the literals to their specific needs.The Real and Complex numeric types are also defined in this chapter.

17.2 Diagram elements

This section describes the concrete syntax and abstract syntax reference for the enumerations and literals defined..

Table 1. *Graphical nodes for Requirements.*

NODE NAME	*CONCRETE SYNTAX*	*ABSTRACT SYNTAX REFERENCE*	*COMPLIANCE*
Complex	«datatype» Complex realpart:Real imaginarypart:Real	**SysML::Types**	**Basic**
ControlValue	«enumeration» ControlValue	**SysML::Types**	**Basic**
Real	«primitive» Real	**SysML::Types**	**Basic**
RequirementKind	«enumeration» RequirementKind	**SysML::Types**	**Basic**
RiskKind	«enumeration» RiskKind	**SysML::Types**	**Basic**
Verdict	«enumeration» Verdict	**SysML::Types**	**Basic**
VerifyMethodKind	«enumeration» VerifyMethodKind	**SysML::Types**	**Basic**

17.3 Package structure

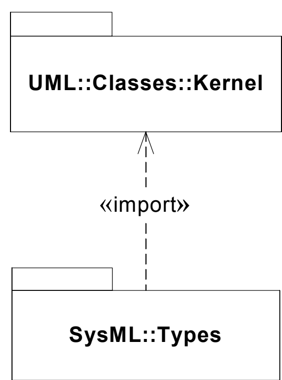


Figure 17-1. Package structure for Requirements.

17.4 UML extensions

There are no metaclass extensions defined in the ModelLibrary package. The package contains pre-defined enumerations and datatypes (M1, or user model level constructs).

17.4.1 Enumerations

17.4.1.1 ControlValue

Definition

ControlValue is an enumeration that provides a means to specify the nature of a control flow. This enumeration is specialized in the non-normative library to provide a default set of enumeration literals that may be customized by the user. See Appendix D, "Non-Normative Model Library".

Description

The ControlValue enumeration is a type available for modelers to apply when control is to be treated as data and for UML control pins. It can be used for behavior and operation parameters, object nodes, and attributes, and so on. The possible runtime values are given as enumeration literals. Modelers can extend the enumeration with additional literals, such as suspend, resume, with their own semantics.

Constraints

[1] UML::ObjectNode::isControlType is true for object nodes with type ControlValue.

17.4.1.2 RequirementKind

Definition

RequirementKind is a enumeration that provides classification categories for requirements. This enumeration is specialized in in Appendix D, "Non-Normative Model Library" to provide a default set of enumeration literals that may be customized by the user.

Description

Requirements may be classified using various taxonomies and each organization will have their own specific set of requirement classifications. In order to accomodate this wide variation in taxonomies the RequirementKind enumeration is defined here, but the specific literals are specified via subclassing (specializing) this enumeration.

17.4.1.3 RiskKind

Definition

RiskKind is a enumeration that provides classification categories for risk. This enumeration is specialized in the non-normative library (see Appendix D, "Non-Normative Model Library") to provide a default set of enumeration literals that may be customized by the user.

Description

Risk may be classified using various taxonomies and each organization will have their own specific set of risk classifications. In order to accomodate this wide variation in taxonomies the RiskKind enumeration is defined here, but the specific literals are specified via subclassing (specializing) this enumeration.

17.4.1.4 Verdict

Definition

Verdict is a enumeration whose literal describe the outcome of a test case. This enumeration is specialized in the non-normative library (see Appendix D, "Non-Normative Model Library") to provide a default set of enumeration literals that may be customized by the user..

Description

The outcome of a verification activity may be specified in various ways by different organizations. In order to accomodate this wide variation in taxonomies the Verdit enumeration is defined here, but the specific literals are specified via subclassing (specializing) this enumeration.

17.4.1.5 VerifyMethodKind

Definition

VerifyMethodKind is a enumeration whose literal describe the method of verification to be applied for a requirement. This enumeration is specialized in the non-normative library (see Appendix D, "Non-Normative Model Library") to provide a default set of enumeration literals that may be customized by the user..

Description

Verification may be accomplished by a wide variety of methods by different organizations. In order to accomodate this wide variation in taxonomies the VerifyMethodKind enumeration is defined here, but the specific literals are specified via subclassing (specializing) this enumeration.

17.4.2 DataTypes

17.4.2.1 Complex

Definition

Complex is a pre-defined datatype in SysML that represents complex values.

Description

A complex value has a real and imaginary parts.

17.4.2.2 Real

Definition

Real is a primitive datatype that represents a real value.

Description

Real is a pre-defined datatype in SysML that represents a real value.

17.5 Usage examples

See Requirements Chapter, Activities Chapter and Appendix B for useage examples.

18 Auxiliary Constructs

18.1 Overview

This chapter defines auxiliary constructs that support modeling Systems.

The Auxiliary Constructs package defines a set of general purpose constructs that can be used to enhance other diagrams. These constructs include Comment, ConstraintComment, Dependency, Enumeration, Problem, and Rationale. Of these seven constructs, the first four are reused from UML; the latter three are unique to SysML.

This chapter also defines ValueType and DistributedValue stereotypes.

A ValueType is a primitive numerical data type used to type properties that represent physical quantities, such as mass, length, and current. ValueTypes have additional properties, in addition to a value, specifying the associated quantity, unit and dimension. A property typed by a ValueType is refered to as a ValueProperty.

A DistributedValue is a stereotype that may be applied to value properties to specify the probability distribution associated with the property. The properties of a DistributedValue specify a probability distribution (ex. Uniform, Gaussian, etc.), and associated mean value and variance. DistributedProperty is defined to support modeling of properties that have associated probability distributions, such as manufacturing variances.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Auxiliary Constructs.

18.2 Diagram elements

Table 2. *Graphical nodes for Auxiliary Constructs.*

Node Name	*Concrete Syntax*	*Abstract Syntax Reference*	*Compliance*
Comment	This is a comment ...	**UML::Kernel::Comment**	**Basic**
Constraint Comment	{Vehicle.force = Vehicle.mass * Vehicle.acceleration}	**UML::Kernel::Constraint**	**Basic**
DistributedValue	«distributedValue» torque:NewtonMeters «distributedValue» distribution=”Gaussian” mean=100.0 variance=2.0 «distributeValue» distribution=”Gaussian” mean=100.0 variance=2.0 «distributedValue» torque:NewtonMeters	**SysML::AuxiliaryConstructs::«DistributedValue»**	**Basic**
Enumeration	**<<enumeration>>** EnumerationName literalName1 literalNameN	**UML::Kernel::Enumeration**	**Basic**

Table 2. *Graphical nodes for Auxiliary Constructs.*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Problem	**<<problem>>** The problem here is that …	**SysML::AuxiliaryConstructs::«Problem»**	**Basic**
Rationale	**<<rationale>>** The reason for this follows …	**SysML::AuxiliaryConstructs::«Rationale»**	**Basic**
ValueProperty	«part,value» torque:NewtonMeters «reference,value» torque:NewtonMeters	**SysML::AuxiliaryConstructs::ValueProperty**	**Basic**
ValueType	«valueType» **NewtonMeters** «valueType» quantity=”Torque” unit=”N*m” dimension=”ML^2/T^2"	**SysML::AuxiliaryConstructs::«ValueType»**	**Basic**

Table 3. *Graphical paths for Auxiliary Constructs.*

PATH NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Dependency	- - - - - - - - - ->	**UML::Kernel::Dependency**	**Basic**

18.3 Package structure

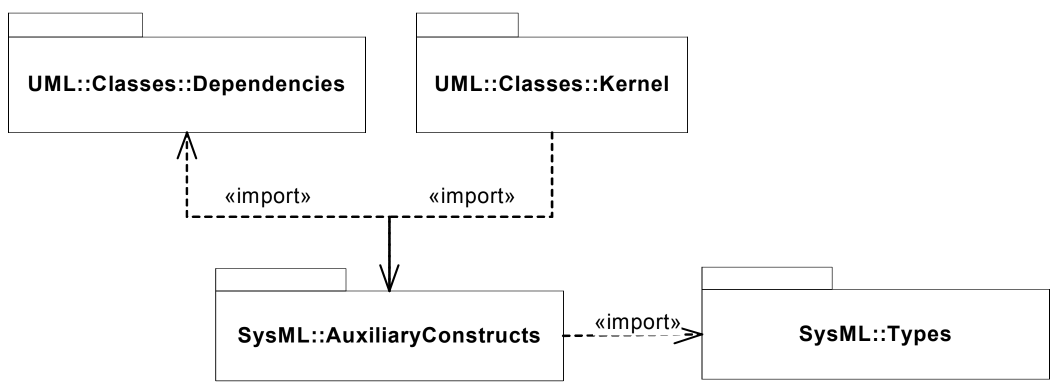


Figure 18-1. Package structure for Auxiliary Constructs.

18.4 UML extensions

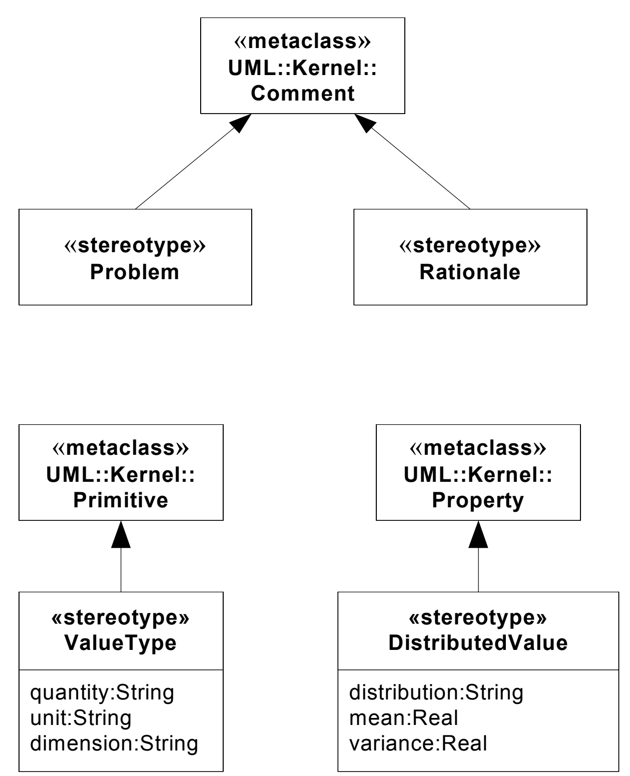


Figure 18-2. Abstract syntax for Auxiliary Constructs.

18.4.1 Stereotypes

18.4.1.1 DistributedValue

Definition

A DistributedValue is a ValueProperty with an associated probability distribution.

Description

A DistributedValue is a ValueProperty with an associated probability distribution. The type of probability distribution, for example Gaussian, and associated mean and variance may be specified.

Attributes

distribution	: String	distribution defines the type of probability distribution associated with the property, such as Uniform or Gaussian.
mean	: Real	mean defines the mean value of the distribution.
variance	: Real	variance defines the value of the variance of the distribution.

Constraints

[1] The DistributedValue stereotype may only be applied to a property typed by a ValueType.

18.4.1.2 Problem

Definition

A Problem documents a deficiency, limitation, or failure of one or more model elements to satisfy a requirement or need, or other undesired outcome.

Description

A Problem is a stereotype of Comment.

Attributes

N/A

Constraints

N/A

18.4.1.3 Rationale

Definition

A Rationale documents a reason or principle for the basis of a model element.

Description

A Rationale is a stereotype of Comment.

Attributes

N/A

Constraints

N/A

18.4.1.4 ValueProperty

Definition

A valueProperty is a property that is typed by a ValueType.

Description

A ValueProperty are property that is typed by a ValueType. ValueProperties may be owned by the containing block by composition (i.e. «part») or reference (i.e. «reference»).

Attributes

N/A

Constraints

N/A

Notation

[2] The notation for a ValueProperty is the same as the notation for property, except that the keyword «value» is shown before the name.

18.4.1.5 ValueType

Definition

A ValueType is a primitive numerical data type used to classify properties that represent physical quantities, such as mass, length, and current. ValueTypes have additional properties, in addition to a value, specifying the associated quantity, unit and dimension.

Description

A ValueType is an extension of UML DataType used to classify (or type) properties that represent physical quantities. Properties typed by a ValueType do not have identity but rather are simple values that can be manipulated according to the rules of real mathematics (addition, subtraction, test for equality, etc.) ValueTypes also have properties that specify the associated quantity, unit and dimension. SysML provides a library of pre-defined ValueTypes (See Chapter XXX Types).

Attributes

- quantity: String — Specifies the quantity associated with the ValueType. For example, “Acceleration”, “Mass”, “Length”, etc.

- unit: String — Specifies the unit of the ValueType. For example, m/s^2, kg, m.
- dimension: String — Specifies the dimention of the ValueType. For example, L (length), M (for mass), T (for time), Q (for Charge), K (for Kelvin) or combinations of these such as L/M^2 which is the dimension of Acceleration.

Notation

[1] The notation for a ValueType is the same as the notation for Block, except that the keyword «valueType» is shown before the name.

18.4.2 Diagram extensions

N/A

18.5 Usage examples

Figure 18-3 shows the Block Definition diagram for a Transmission. Also shown is the definition of a ValueType "Torque" with quantity, unit and dimension specified. The emTorque, iceTorque and outTorque are ValueProperties typed by the ValueType Torque. The Transmission also has ValueProperties mass and speed. These properties are typed by Kilograms and KPH, respectively. Kilometers and KPH are two pre-defined ValueTypes available in the SysML non-normative library. See See Appendix D for a complete list of predefined ValueTypes. Note that the primitive Real is also pre-defined in SysML. See Chapter 17, "Types".

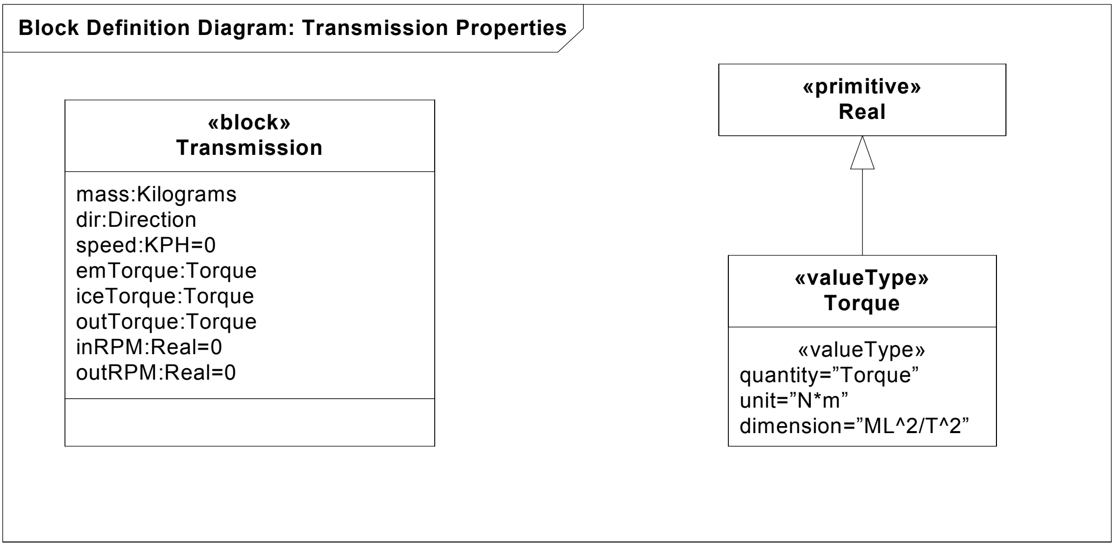


Figure 18-3. Block Definition diagram: Transmission properties

Figure 18-4 shows the Internal Block diagram for the Transmission. This diagram states that the speed, mass and outTorque properties are owned by the Transmission by composition (i.e. they are parts of the transmission). In addition, the mass property of the transmission is a DistributedValue with a Gaussion probability distribution and mean of 100 kg and variance of 2 kg.

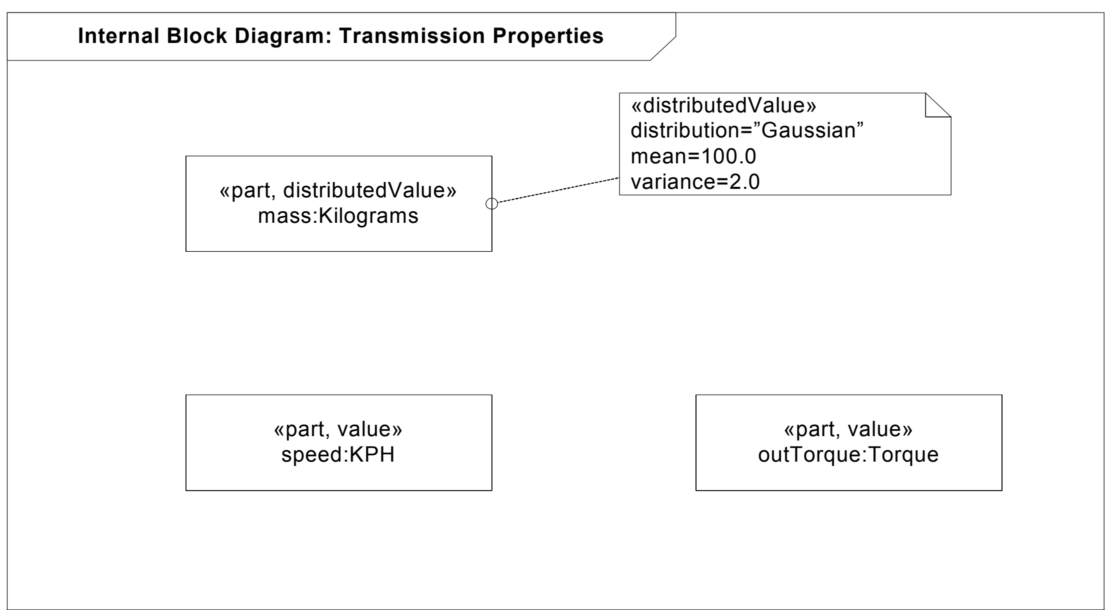


Figure 18-4. Internal Block diagram: Transmission properties

19 Profiles & Model Libraries

19.1 Overview

The Profiles package specifies mechanisms that allow metaclasses from existing metamodels to be extended so that they can be adapted for different purposes, such as customizing SysML for different platforms or domains. The Profiles mechanism is intended to be architecturally compatible with the OMG UML 2.0 and Meta Object Facility (MOF) specifications.

Profiles cannot only be used to extend SysML, they can also can be used to restrict the language by selecting the subset of the base metamodel that is required for the specific domain. For example, SysML does not require all of the UML metamodel. The Language Architecture chapter describes the subset of UML that is included in SysML.

The following sections describe the abstract syntax, package structure, UML extensions, compliance levels and usage examples for Profiles and Model Libraries.

19.2 Diagram elements

Table 4. *Graphical nodes for Profiles*[a]

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Stereotype	«stereotype» **StereotypeName**	**UML::Profiles::Stereotype**	**Basic**
Metaclass	«metaclass» **MetaClassName**	**UML::Profiles::Class**	**Basic**
Profile	«profile» **ProfileName**	**UML::Profiles::Profile**	**Basic**
Model Library	«modelLibrary» **LibraryName**	**UML::StandardProfileL1**	**Basic**

a. In the above table, boolean properties can alternatively be displayed as BooleanPropertyName=[True|False].

Table 5. *Graphical paths for Profiles*

PATH NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE

Table 5. *Graphical paths for Profiles*

Extension	«metaclass» **MetaClassName** {required} «stereotype» **StereotypeName**	**UML::Profiles::Extension**	**Basic**
Generalization	«stereotype» **StereotypeName** «stereotype» **StereotypeName**	**UML::Classes::Kernel::Generalization**	**Basic**
ProfileApplication	«apply»	**UML::Profiles::ProfileApplication**	**Basic**
Unidirectional Association	propertyName	**UML::Classes::Kernel::Association**	**Basic**

Table 6. *Graphical nodes for Profiles*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Model Element	«stereotypeName» PropertyName=ValueString MultiPropertyName=ValueString, ValueString BooleanPropertyName Element Name PathName Element Name	**UML::Classes::Kernel::Element**	**Basic**
Model Element	«stereotypeName» PropertyName=ValueString MultiPropertyName=ValueString, ValueString BooleanPropertyName Element Name	**UML::Classes::Kernel::Element**	**Basic**
Model Element	«stereotypeName» {PropertyName=ValueString; BooleanPropertyName} NodeName	**UML::Classes::Kernel::Element**	**Basic**
Model Element	NodeName «stereotypeName»{PropertyName=ValueString}ElementName «stereotypeName»{PropertyName=ValueString; BooleanPropertyName} ElementName	**UML::Classes::Kernel::Element**	**Basic**
Model Element	Element Name «stereotypeName» {PropertyName=ValueString; BooleanPropertyName}PathName Element Name	**UML::Classes::Kernel::Element**	**Basic**

Table 6. *Graphical nodes for Profiles*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Model Element	**«stereotypeName»** **NodeName** «stereotypeName» PropertyName=ValueString MultiPropertyName=ValueString, ValueString BooleanPropertyName	**UML::Classes::Kernel::Element**	**Basic**

19.3 Package Structure

SysML does not add any new abstract syntax and so does not require an additional package.

19.4 UML extensions

19.4.1 Metaclass extensions

N/A

19.4.2 Diagram extensions

19.4.2.1 Stereotype

The values of a stereotype that has been applied to a model element can be shown in one of three ways:

- ·As part of a comment symbol tied to the symbol representing the model element
- ·In compartments of a graphic node representing the model element.
- ·Above the name string within a graphic node or before the name string otherwise. Note that a restricted form of this notational option is simply just to show the stereotype name in guillemets.

In the case where a compartment or comment symbol is used, the user may elect to show the stereotype name in guillemets before the name string in addition to in the compartment or comment.

The values of the stereotype properties are displayed as name/value pairs, thus:

<namestring>'='<valuestring>

If a stereotype property is multi-valued then the valuestring is displayed as a comma-separated list:

<valuestring>::=<value>{','<value>}

Certain values have special display rules:

- ·As an alternative to a name/value pair, when displaying the values of boolean properties diagrams may use the convention that if the *namestring* is displayed then the value is True, otherwise the value is False;

- ·If the value is the name of a NamedElement then optionally its qualifiedName can be used.

If compartments are used to display stereotype values then an additional compartment is required for each applied stereotype whose values are to be displayed. Each such compartment is optionally headed by the name of the applied stereotype in guillemets. Any graphic node that is not displayed as an icon may have these compartments.Graphic nodes corresponding to the following SysML/UML concepts may have these compartments: any Classifier shown as a Class Node; Property (and subclasses); a CallBehaviorAction, CallOperation Action, CentalBufferNode (or subclasses) or ActivityParameterNode.

Within a comment symbol, or if displayed before/above the symbol's namestring, the values from a specific stereotype are optionally preceded with the name of the applied stereotype within a pair of guillemets, which is useful if values of more than one applied stereotype should be shown.

When displayed in compartments or comment symbol at most one name/value pair can appear on a single line. When displayed above/before a namestring the name/value pairs are separated by semicolons and all pairs for a given stereotype are enclosed in braces.

In Figure 19-5, a simple stereotype *Clock* is defined to be applicable at will (dynamically) to instances of the metaclass Class and describes a clock software component for an embedded software system. It has description of the operating system version supported, an indication of whether it is compliant to the POSIX operating system standard and a reference to the operation that starts the clock.

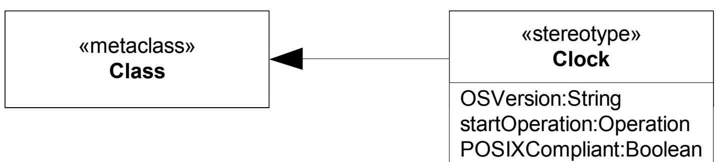


Figure 19-5. Defining a stereotype

Figure 19-6 shows how the stereotype *Clock* is applied to a class called *StopWatch*.

Figure 19-6. Using a stereotype

When, two stereotypes, *Clock* and *Creator*, are applied to the same model element, as is shown in Figure 19-7., the attribute values of each of the applied stereotypes can be shown in a comment symbol attached to the model element.

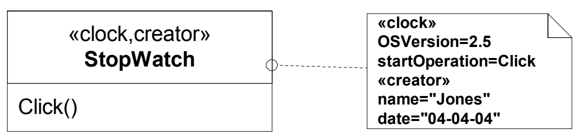


Figure 19-7. Using stereotypes and showing values

Finally, the two alternate notational forms are shown..

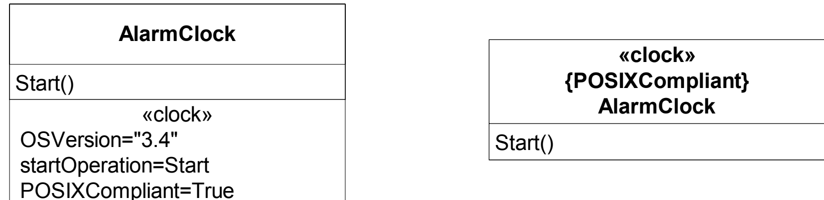


Figure 19-8. Other notational forms for showing values

In this case, AlarmClock is valid for OS version 3.4, is POSIX-compliant and has a starting operation called Start. The compartment form of notation is shown on the left and the in-symbol form on the right (note that not all properties of AlarmClock are shown on the right. Note that multiple stereotypes can be shown using these alternates also, either as multiple compartments, or by grouping the values for a given stereotype after its stereotype indication.

19.5 Usage examples

19.5.1 Defining a Profile

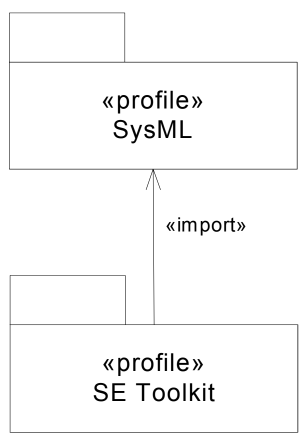


Figure 19-9. Definition of a profile

In this example, the modeler has created a new profile called SE Toolkit, which imports the SysML profile, so that it can build upon the stereotypes it contains. The SE Toolkit can extend those metaclasses from UML that the SysML profile imports.

19.5.2 Adding Stereotypes to a Profile

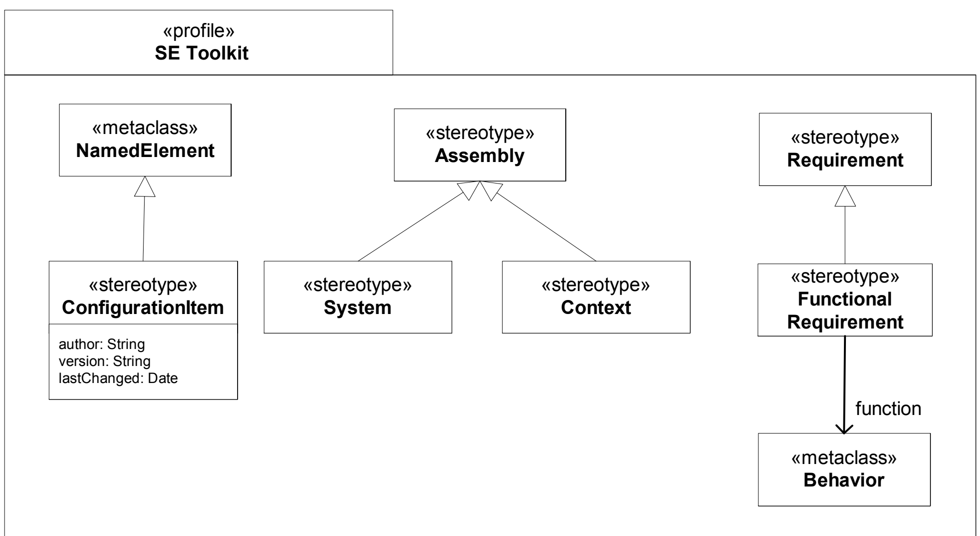


Figure 19-10. Profile Contents

I

In the SE Toolkit, both the mechanisms for adding new stereotypes are used. The first, exemplified by configurationItem, is called an extension, shown by a line with a filled triangle; this relates a stereotype to a reference (or base) class, in this case NamedElement from UML and adds new properties that every NamedElement stereotyped by configurationItem must have. NamedElement is an abstract class in UML so it is its subclasses that can have the stereotype applied. The second mechanism is demonstrated by the system and context stereotypes which are sub-stereotypes of an existing SysML stereotype, assembly; sub-stereotypes inherit any properties of their super-stereotype (in this case none) and also extend the same base class or classes. Note that TypedElements whose type is extended by «system» do not display the «system» stereotype; this also applies to InstanceSpecifications. Any notational conventions of this have to be explicitly specified in a diagram extension.

T

There is also an example of how stereotypes (i.e., FunctionalRequirement) can have unidirectional associations to metaclasses in the reference metamodel (in this case Behavior).

19.5.3 Defining a Model Library that uses a Profile

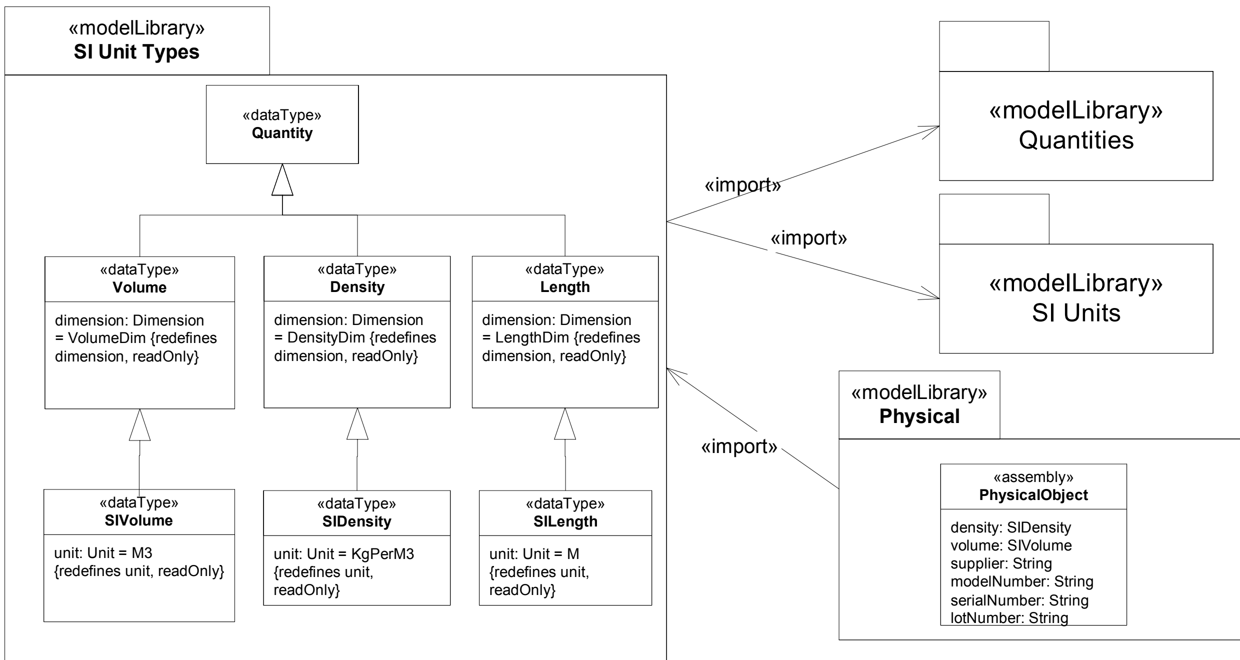


Figure 19-11. Model libraries example

The model library SI Unit Types imports the model libraries Quantities and SI Units from SysML, so that it can use model elements from them in its own definition. It defines subtypes of Quantity (as subset of which are shown here) that can be used when property values are measuresd in SI Units. A further model library, Physical, imports the SI Unit types so that it can define properties that have those types. One model element, PhysicalObject is shown, an assembly that can be used as a supertype for an physical object.

19.5.4 Guidance on whether to use a Stereotype or Class

This section provides guidance on when to use stereotypes. Stereotypes can be applied to any model element. Stereotyping a model element allows the model element to be identified with the «guillemet» notation. In addition, the sterotyped model element can have stereotype properties, and the stereotype can specify constraints on the model element.

The modeler must decide when to create a stereotype of a class versus when to specialize (subclass) the class. One reason is to be able to identify the class with the «guillemet» notation. In addition, the stereotype properties are different from properties of classes. Stereotype properties represent properties of the class that are not instantiated and therefore do not have a unique value for each instance of the class.

SE Toolkit::functionalRequirement, which extends Class through its superstereotype, Requirement, is an example where a stereotype is appropriate because every modeling element stereotyped by SE Toolkit::functionalRequirement has a reference to another modeling element. In another example, SE Toolkit::configurationItem defined above, which applies to classes amongst other concepts, is a stereotype because its properties characterise the author, version and last changed date of the modeling element themselves. One test of this is whether the new properties are inheritable; in this case author, version and last-changed date are not, because it is only those classes under configuration control that need the properties. To summarise, in the following circumstances a stereotype is appropriate:

- Where the model concept to be extended is not a class or class-based;
- Where the extensions include properties that reference other model elements;
- Where the extensions include properties that describe modeling data, not system data;

An example where a class is more appropriate is PhysicalObject from Figure 19-11. In this case, the properties density and volume, and the component numbers, have distinct values for each system element described by the class, and are inherited by every subclass of PhysicalObject.

19.5.5 Using a Profile

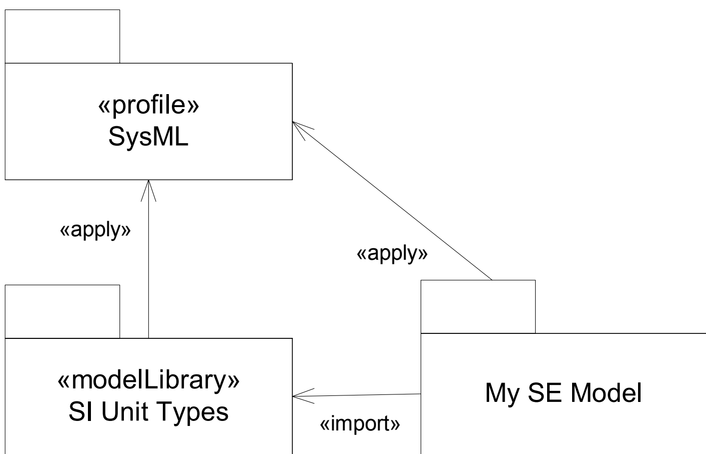


Figure 19-12. A model with applied profile and imported model library

My SE Model is a system engineering model that needs to use stereotypes from SysML. It therefore needs to have the SysML profile applied to it. In order to use the predefined SI units, it also needs to import the SI Unit Types model library. Having done this, elements in My SE Model can be extended by SysML stereotypes and types like SIVolume can be used to type properties.

19.5.6 Using a Stereotype

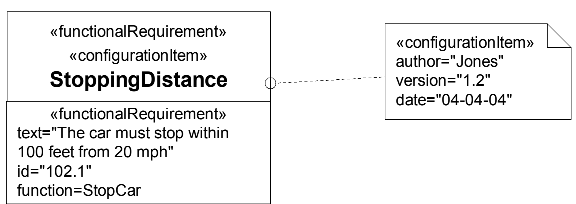


Figure 19-13. Using two stereotypes on one model element

StoppingDistance has two stereotypes applied, functionalRequirement, that identifies it as a requirement that is satisfied by a function, and configurationItem, which allows it to have configuration management properties. The modeler has provided values for all the newly available properties; those for criticalRequirement are shown in a compartment in the node symbol for StoppingDistance; those for configurationItem are shown in a separate note.

19.5.7 Using a Model Library Element

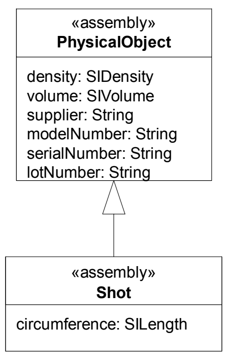


Figure 19-14. Using model library elements

Part V - Appendices

This section contains non-normative appendices for this specification.

Appendix A.Diagrams

A.1 Overview.

A SysML diagram taxonomy is shown in Figure A-1. SysML reuses, extends and adds to UML diagram types as follows:

- UML diagrams that are reused, but are not extended: Use Case diagram, Sequence diagram, and State Machine diagram.
- UML diagrams that are reused and extended: Activity diagram (extends UML Activity diagram), Block Definition diagram (extends UML Class diagram), Internal Block diagram (extends UML Composite Structure diagram), and Package diagram (extends UML Package diagram).
- New diagram types[1]: Parametric Constraint diagram, Allocation diagram/Allocation Traceability Table[2], and Requirements diagram.

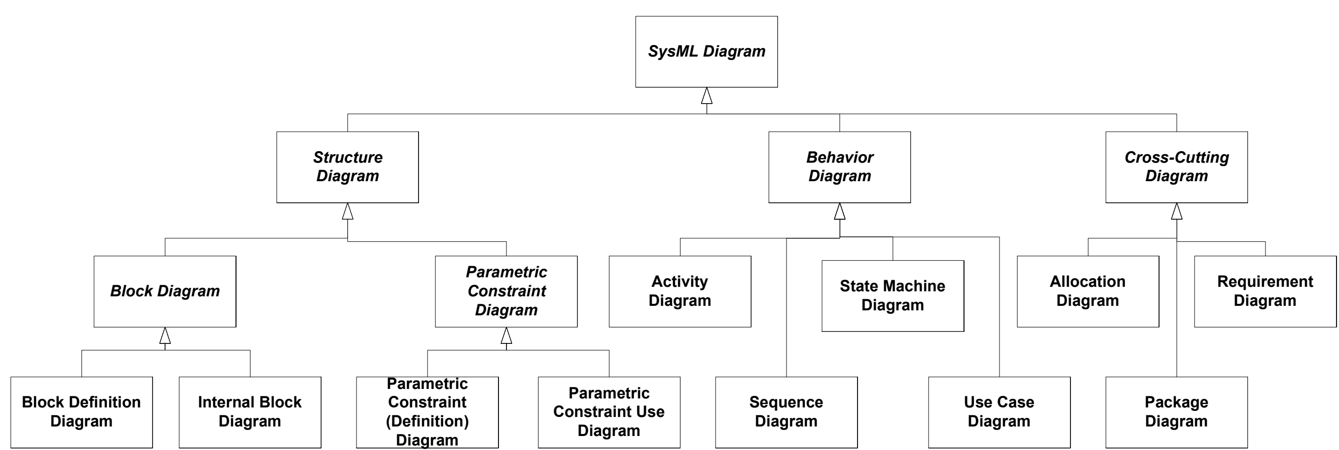


Figure A-1. SysML Diagram Taxonomy

1. The diagrams in this category could also be categorized as *UML diagrams that are reused and extended*, since these diagrams are constructed using a profile extension (cf. defining entirely new diagram semantics using a metamodel extension). However, after extensive experience with SysML prototypes that implement these diagrams, we believe that their innovations are sufficiently significant that they should be considered new diagram types.
2. Allocations can be represented in both graphic and tabular format, where the latter is referred to as an Allocation Traceability Table. See “Table extensions” on page 111 in the “Allocations” on page 107.

These diagrams are described in detail in the following chapters of this specification:

- Structure diagrams[3]
  - Block diagram
    - Block Definition diagram: See Chapter 8, "Blocks"
    - Internal Block diagram: See Chapter 8, "Blocks"
  - Parametric Constraint diagram
    - Parametric Constraint (Definition) diagram: See Chapter 9, "Parametric Constraints"
    - Parametric Constraint Use diagram: See Chapter 9, "Parametric Constraints"
- Behavior diagrams
  - Activity diagram: See Chapter 10, "Activities"
  - Sequence diagram: See Chapter 11, "Sequences"
  - State Machine diagram: See Chapter 12, "State Machines"
  - Use Case diagram: See Chapter 13, "Use Cases"
- Cross-Cutting diagrams
  - Allocation diagram: See Chapter 15, "Allocations"
  - Requirement diagram: See Chapter 14, "Requirements"
  - Package diagram: See Chapter 16, "Model Management"
  - Sequence diagram: See Chapter 11, "Sequences"

3. The subtypes of Block and Parametric Constraint diagrams reflect the definition/usage dichotomy of the structural constructs. See Part II - "Structural Constructs".

Appendix B. Sample Problem

B.1 Overview

The purpose of this appendix is to provide a comprehensive example that illustrates how SysML supports the specification, analysis, design and verification of a system using some of the basic and advanced features of the language.

The intent of this sample problem is to provide a sufficent number of related sample diagrams (at least one for every SysML diagram type) to demonstrate how the SysML diagram types complement each other, and to highlight how model elements in different diagrams are subject to architectural integrity rules (cf. language well-formedness rules). These diagrams that comprise this sample problem were extracted from a working model that was developed during the prototyping phase of SysML development, and is fully executable. If you are interested in obtaining an executable version of the sample problem, see Section 5.1, “Support Documents,” on page 8.

The sample problem does not highlight all of the features of the language. The reader should refer to the individual chapters for more detailed descriptions of all features of the language. The diagrams shown were selected for representing a particular aspect of the model, and the ordering of the diagrams is meant to be representative of a typical systems engineering process, but we are not recommending any particular method or approach here.

B.2 Problem Summary

The sample problem describes the the development of a Hybrid Sports Utility Vehicle (SUV) system. The problem is derived from a marketing analysis which indicated the need to increase the fuel economy and “eco-friendliness” of the vehicle from its current capability without sacrificing performance. Only a small subset of the functionality and associated vehicle system requirements and design are addressed to highlight this application.

B.3 Diagrams

B.3.1 Requirements Diagram for the “Hybrid SUV”

A portion of the high level user requirements for the Hybrid SUV are shown in the Requirement diagram of Figure B-1. Three presentation options for requirements are shown in the diagram: the Hybrid SUV requirement is shown as a Requirement symbol with the contents of the compartments elided (not shown); the Load, Eco-Friendliness, Emissions, Performance and Fuel Economy requirements are shown as a Requirement symbol displaying the «requirement» properties in a compartment; and the remaining elements are shown as Requirement symbols with all compartments elided.

The requirement properties shown (for example in the Performance requirement) include the standard SysML requirement properties “id”, “source”, “text”, “kind”, “verifyMethod” and “risk”. The following default, user defined literals for the enumeration types have been defined in the Appendix D, “Non-Normative Model Library”:

- RequirementKind = {Functional, Performance, Interface}
- VerifyMethodKind={Analysis, Demonstration, Inspection, Test}
- RiskKind={High, Medium, Low}

Additional user defined properties and/or enumeration literals could be added. See the Chapter 19, “Profiles & Model Libraries” chapter for information on customizing model elements in order to add user defined properties or enumeration literals.

It is important to recognize that not all information from the model must be shown on a diagram. The intent is to highlight points of interest. In this particular example, the Fuel Economy, Emissions, Performance and Range requirements are considered key requirements and will be the focus of further discussion.

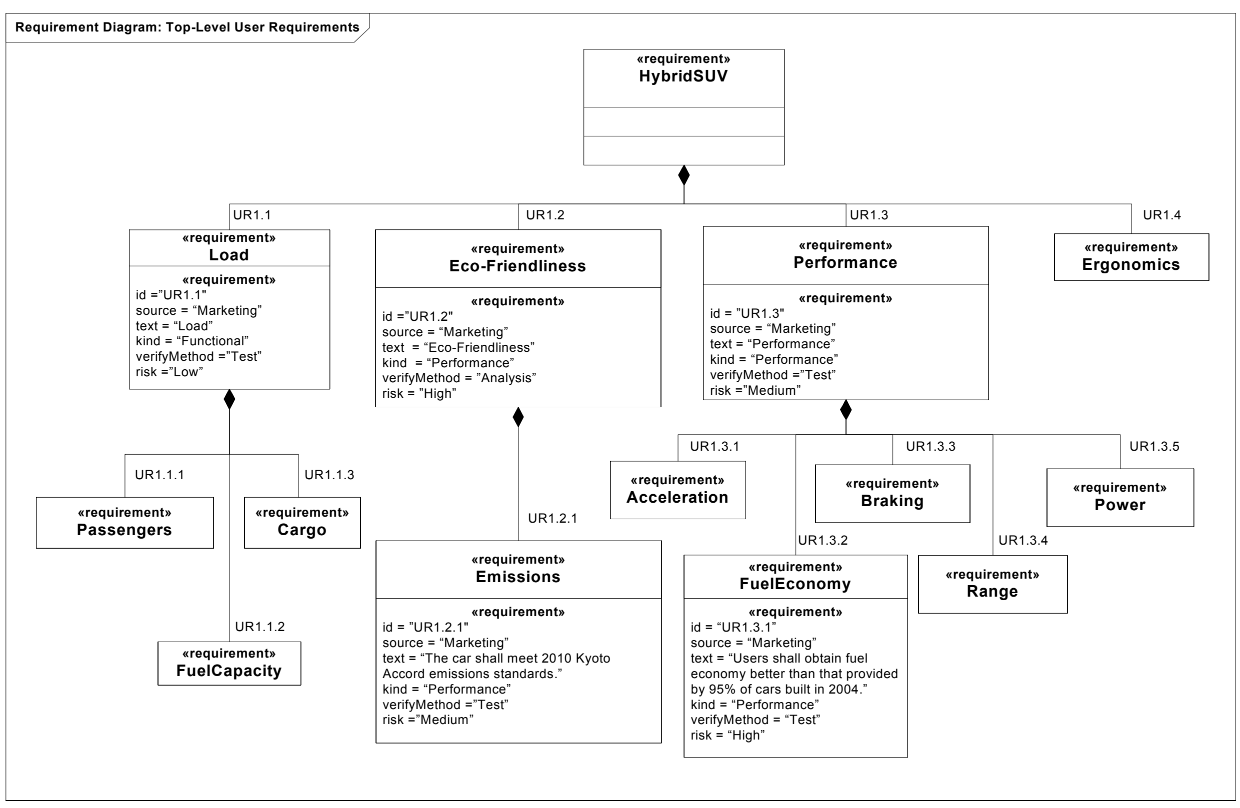


Figure B-1. Requirement Diagram:Top-Level User Requirements

B.3.2 Trade Study and Measures of Effectiveness

Trade studies are a basic activity for many system engineers. Given a set of requirements, it is frequently problematic to satisfy all requirements. Consequently, trade-offs must be made to arrive at a design that is optimal in a global sense. For example, maximizing the horsepower of the internal combustion engine will contribute to enhanced performance at the cost of fuel efficiency, emissions and range.

In order to determine this "global optimum" one typically uses a set of "Measures of Effectiveness" (MoE), which represent the criteria by which alternative designs or architectures are assessed. These MoEs are derived from a small subset of the requirements that are considered critical and are associated with a similarly small set of Key Performance Parameters (KPP) that determine the resulting effectiveness.

Figure B-2 shows the set of MoEs for the sample problem using the «effectiveness» model element. The MoEs form a hierarchy with the overall Measure of Effectiveness at the root of the tree. Each MoE has attributes to capture the normalized score for the design alternative being investigated (score:Real) as well as a relative importance of the MoE (weight:Real). The sum of the weights of all MoEs at any given level in the hierarchy must sum to 100. The normalization of the raw scores for a given design alternative is done using utility curves (sometimes called value functions, since they yield a resulting value for a

given raw input). These utility curves map the raw values of a MoE, for example the time to accelerate from 0 to 60 mph, to a value between 0 and 100 which may then be weighted according to the relative importance and aggregated, or rolled-up to yield the overall score. Our goal is to maximize the Overall MoE.

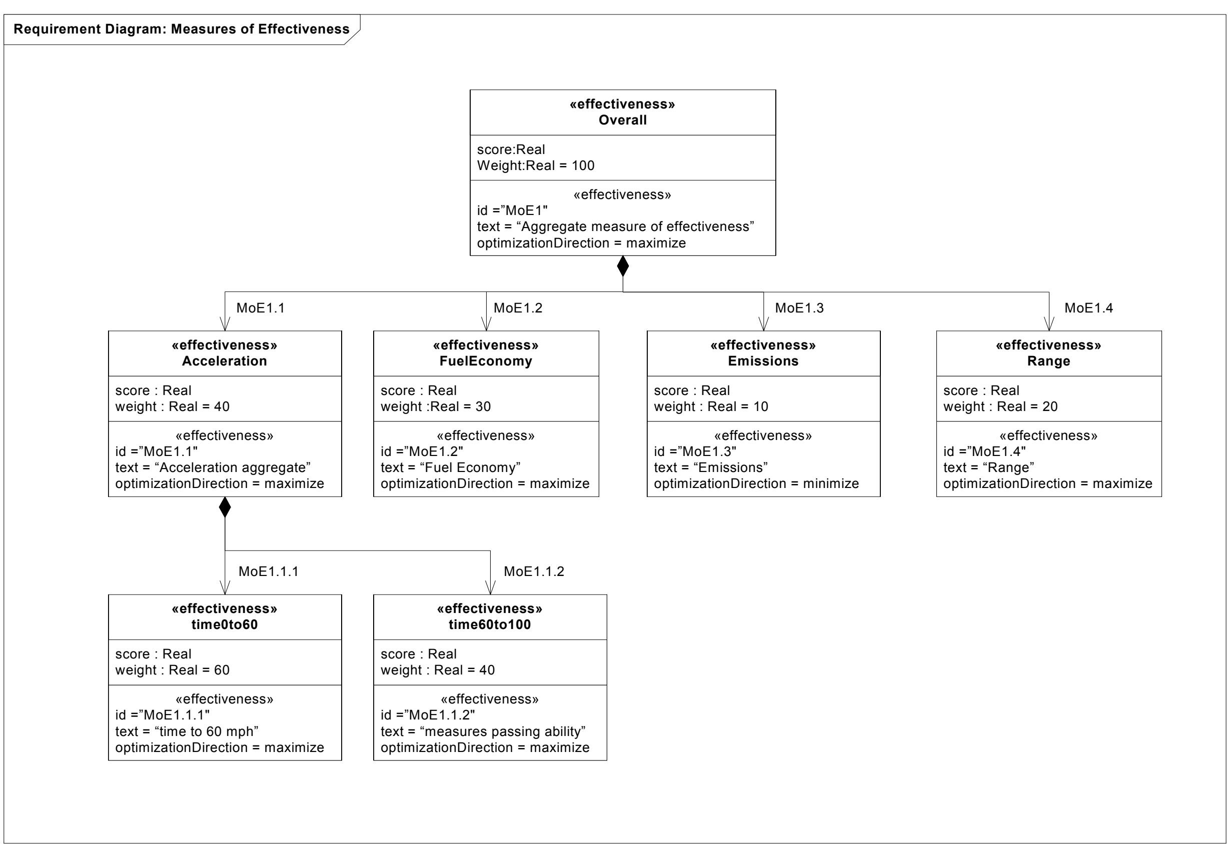


Figure B-2. Measures of Effectiveness for the Hybrid SUV

Figure B-3 shows the KPPs for the sample problem. Given the set of MoEs one can determine which parameters will influence the outcome. In this case the KPPs are the total mass of the vehicle, the displacement of the internal combustion engine, the horsepower of the electric motor, the capacity of the fuel tank, the energy density of the battery pack, and the coefficient of drag of the chassis/body design.

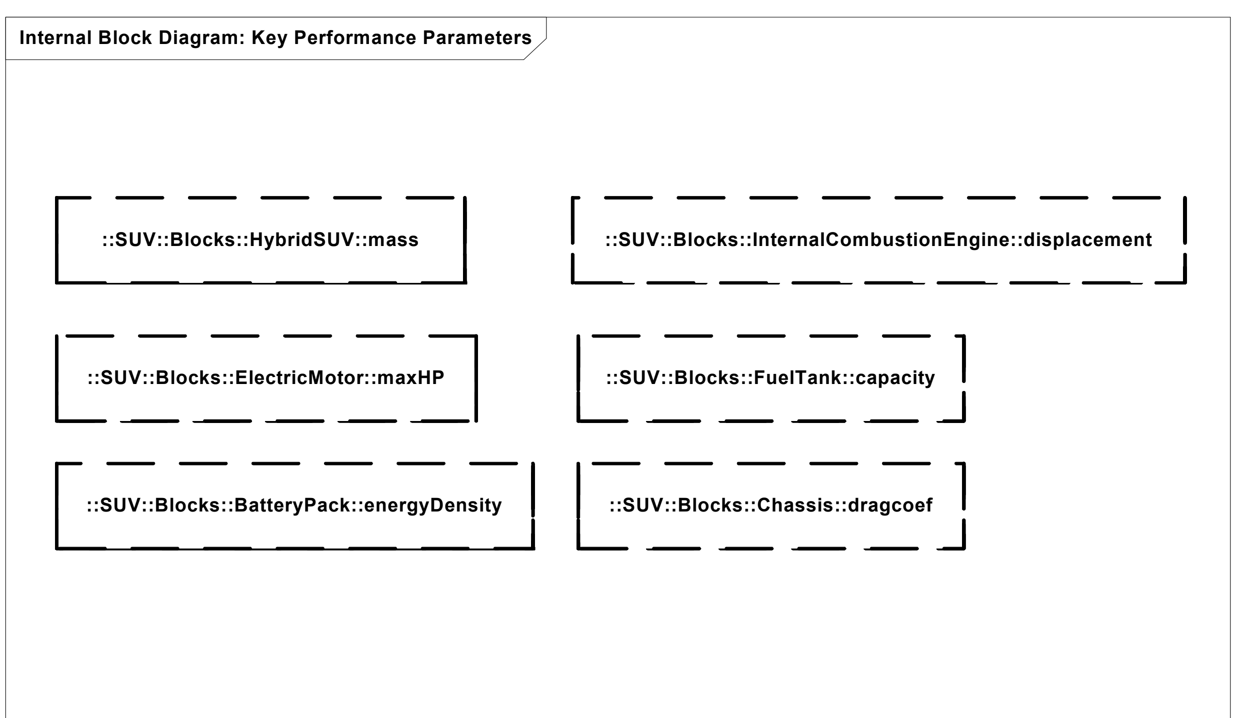


Figure B-3. Key Performance Parameters for the Hybrid SUV

Figure B-4 shows the relationship between the KPPs and the Acceleration MoE in the form of a parametric diagram. The parametric constraint usages (for example disp2hp:Disp2HP) specify how the various KPPs are combined to yield the normalized scores for the Acceleration MoE. The actual constraint equations, shown in this diagram in the constraint note, are part of the definition of the parametric constraints (see Figure B-5). By separating the constraint definition from it's usage in a particular context, one can develop a library of re-usable constraints. For example, Newton's Law (as specified by the constraint Newton) would likely be re-used many times.

Similar parametric diagrams would be created for each of the remaining MoEs to specify the relationship between the KPPs and the normalized score of the associated MoE.

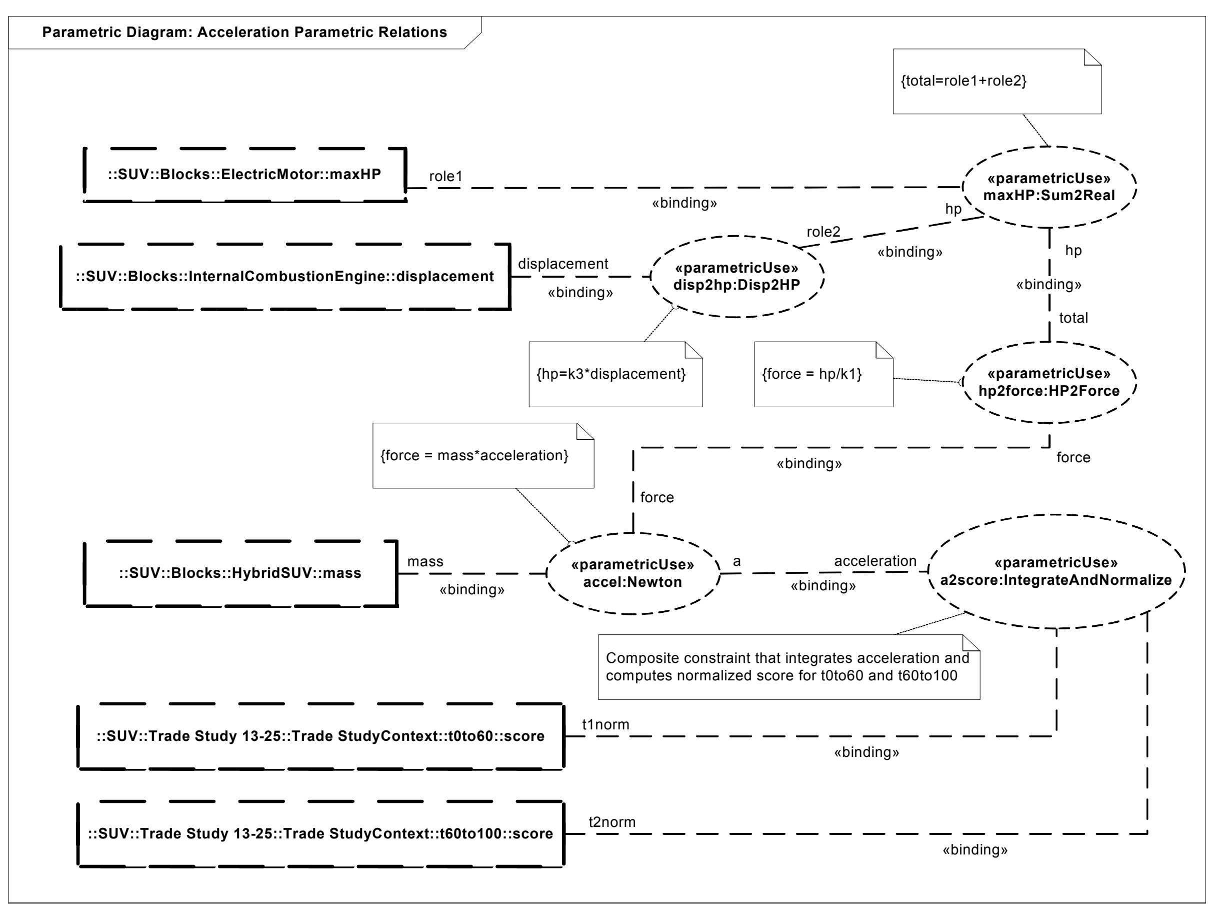


Figure B-4. Usage of constraints to map KPPs to MoE scores for Acceleration MoE

Figure B-5 shows the definition of the parametric constraint for Newtons Law used in Figure B-4. Parametric constraints are defined using Internal Block Diagrams to describe the parameters (roles) and the constraints that apply to them. The definitions of the other simple constraints in Figure B-4 (Sum2Real, Disp2HP, HP2Force) are similar and are not shown.

These definitions may be placed in a model library and reused.

The types Newtons, Kilograms, and MetersPerSec2 used in Figure B-5 are pre-defined value types which are type compatible with the pre-defined data type Real. See Appendix D, “Non-Normative Model Library” for pre-defined value types.

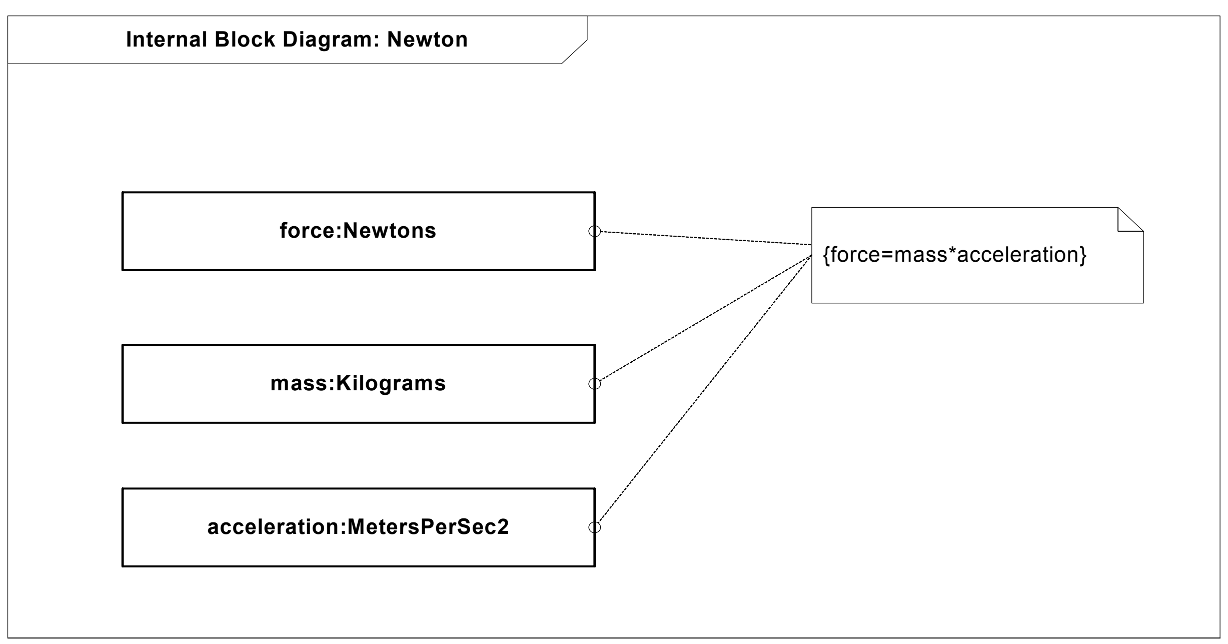


Figure B-5. Constraint definitions for Newton's Law

The composite constraint IntegrateAndNormalize shown in Figure B-4 re-uses two other constraints and has been included to illustrate the process of defining more complex constraints and the scaleability of the approach.

Figure B-6 is a Block Definition diagram that states that the IntegrateAndAccelerate constraint is composed of two parametric constraints: Integrate, and Normalize.

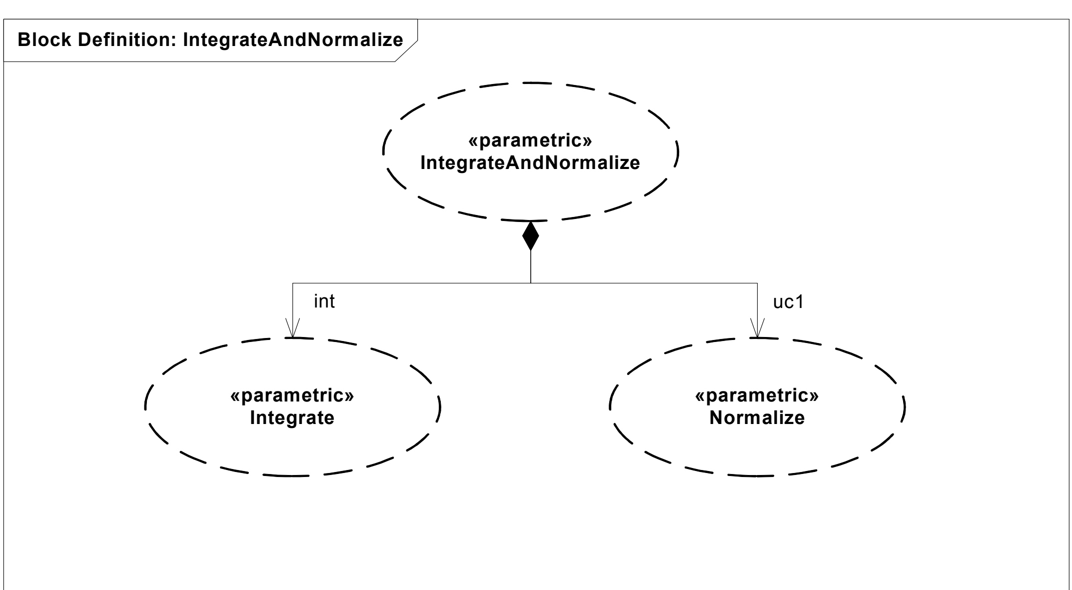


Figure B-6. Block Definition diagram of composite parametric constraint IntergrateAndNormalize

Figure B-7 shows the corresponding Internal Block Diagram for the IntegrateAndNormalize composite constraint showing the internal relationships between the parameters (roles) and component constraints. Note that not all parameters are exposed at the composite level. For example, only the parameters "a" (which represents the variable to be integrated) and t1norm and t2norm (which represent the normalized times) are bound to properties in Figure B-4. The other roles internal to the component parametric constraints are hidden at the top level.

The types MetersPerSec2, and Seconds used to type the parameters are pre-defined value types which are type compatible with the data type Real. See Appendix D, "Non-Normative Model Library" for a list of pre-defined value types.

Figure B-7 also highlights the fact that a parametric constraint block may contain more than one constraint. Each of the component constraints in this figure has two constraint equations. The definition of these simple component constraints would be done as in Figure B-5 and is left as an exercise for the reader.

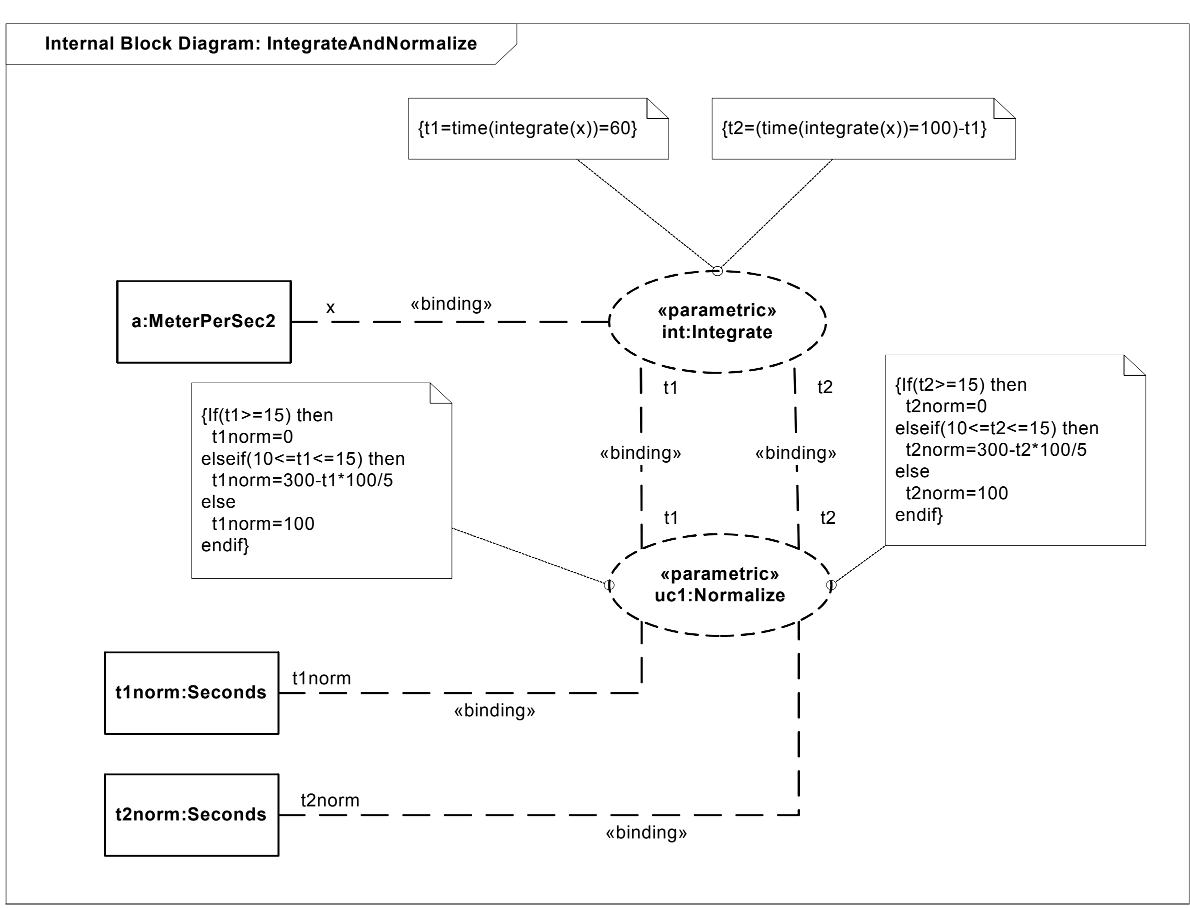


Figure B-7. Internal Block diagram of composite parametric constraint IntergrateAndNormalize

In summary, each MoE would have a set of constraints which specify how one would compute the normalized score from the relevant KPPs (similar to Figure B-4). These normalized scores would then be used, in conjunction with the weight defined for the MoE, to arrive at an overall measure of effectiveness for the given values of the KPPs in accordance with Figure B-2. Following this process, one can view the KPPs as the "nobs" that one can "adjust" in order to maximize the overall MoE.

B.3.3 Requirements Derivation

Following engineering analysis of the user requirements a set of system requirements are derived. Figure B-8 shows a portion of the derived requirements focusing on the key user requirements concerning Fuel Economy and Emissions.

In particular, the system requirement PowerSourceSelection is derived from the Fuel Economy, Emissions, Acceleration, and Range user requirements. "Trade study 13-25" is listed as the source of this requirement. The Regenerative Braking system requirement is derived from the Fuel Economy, Braking and Range user requirements. The same trade study is listed as the source of the Regenerative Braking requirement.

Figure B-8 also shows an alternative presentation option for requirement properties, the comment symbol in the upper right hand corner anchored to the FuelEconomy requirement. Only the id and text properties are shown as it is possible to elide (not show) properties in a comment symbol as well.

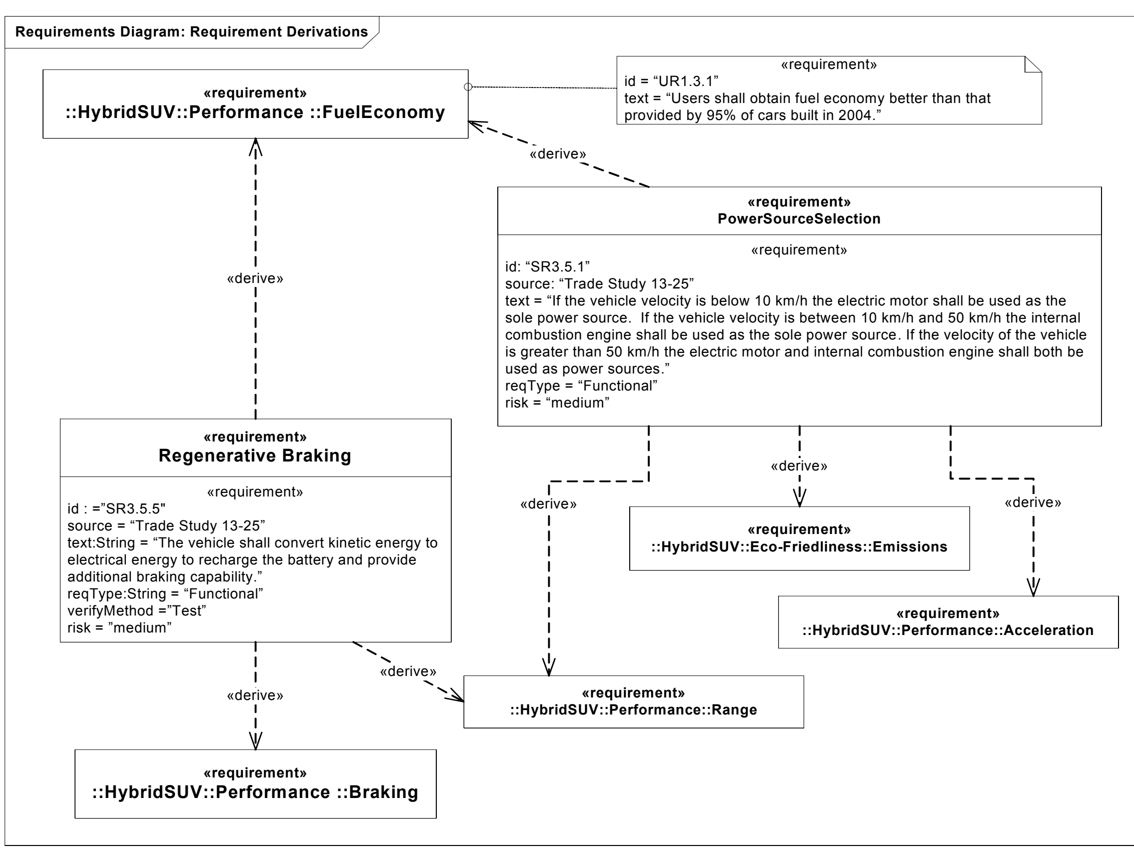


Figure B-8. Requirement Diagram:Requirements Derivation

B.3.4 Requirements Verification

Figure B-9 shows a requirements diagram capturing the verification planning for the key system requirement to provide regenerative braking.

This diagram indicates that the verification procedure for Regenerative Braking is specified by the ConverKineticToElectrical interaction (sequence diagram) and the ConvertKinetic activity. Both these model elements are also stereotyped as a «testCase» which implies they will return a verdict.

Requirements may be verified by behaviors (interactions, statemachines or activities) or other procedural specifications.

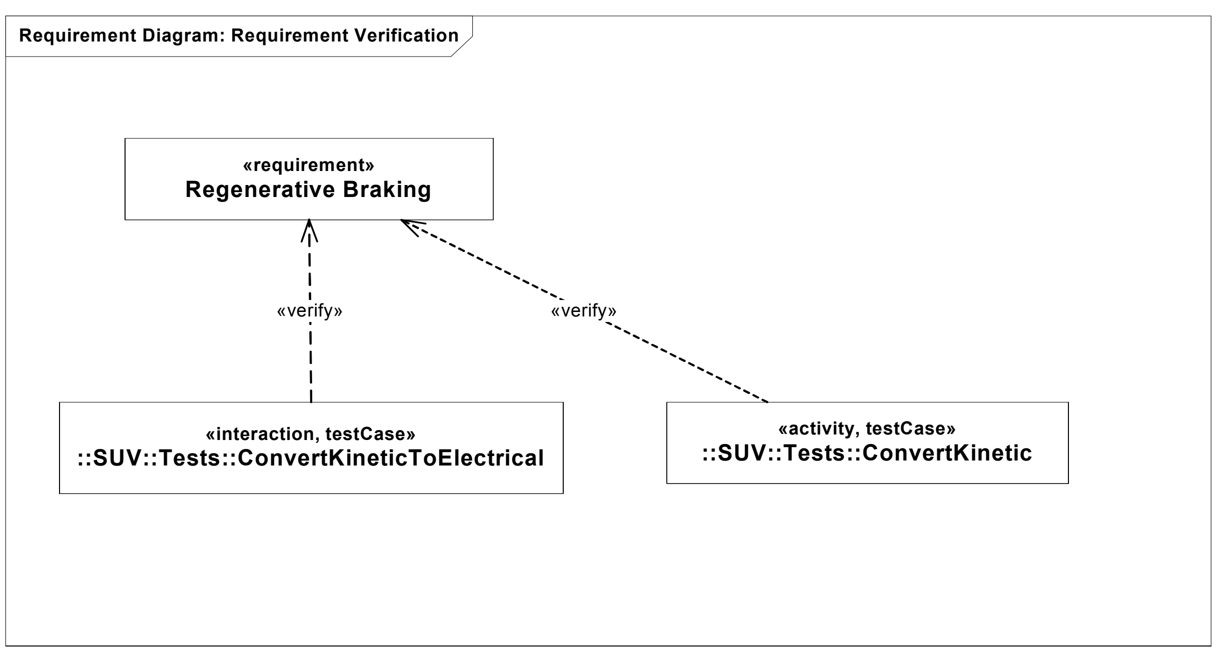


Figure B-9. Requirement Diagram:Requirements Verification

B.3.5 Use Case Diagram

Figure B-10 shows the use case diagram for Hybrid SUV. The top level "Drive" use case is decomposed using «include» use cases. The subject (the HybridSUV) and the actors (Driver, Maintenance, IsuranceService and DMV) interact with the system to realize the use case.

This diagram aids in analysis by establishing the scope and context of the system under development (:HybridSUV), identifying key external entities (people, external systems, etc.) that interact with the system along with the associated external interfaces, and providing the initial high level decomposition of behavior according to key system threads or scenarios.

Like any other model element, traceability to the driving requirements can be established using «satisfies» relationships (see Section B.3.15).

One example of re-use, the fact that the Brake use case is included in the Park and Drive use cases, is shown.

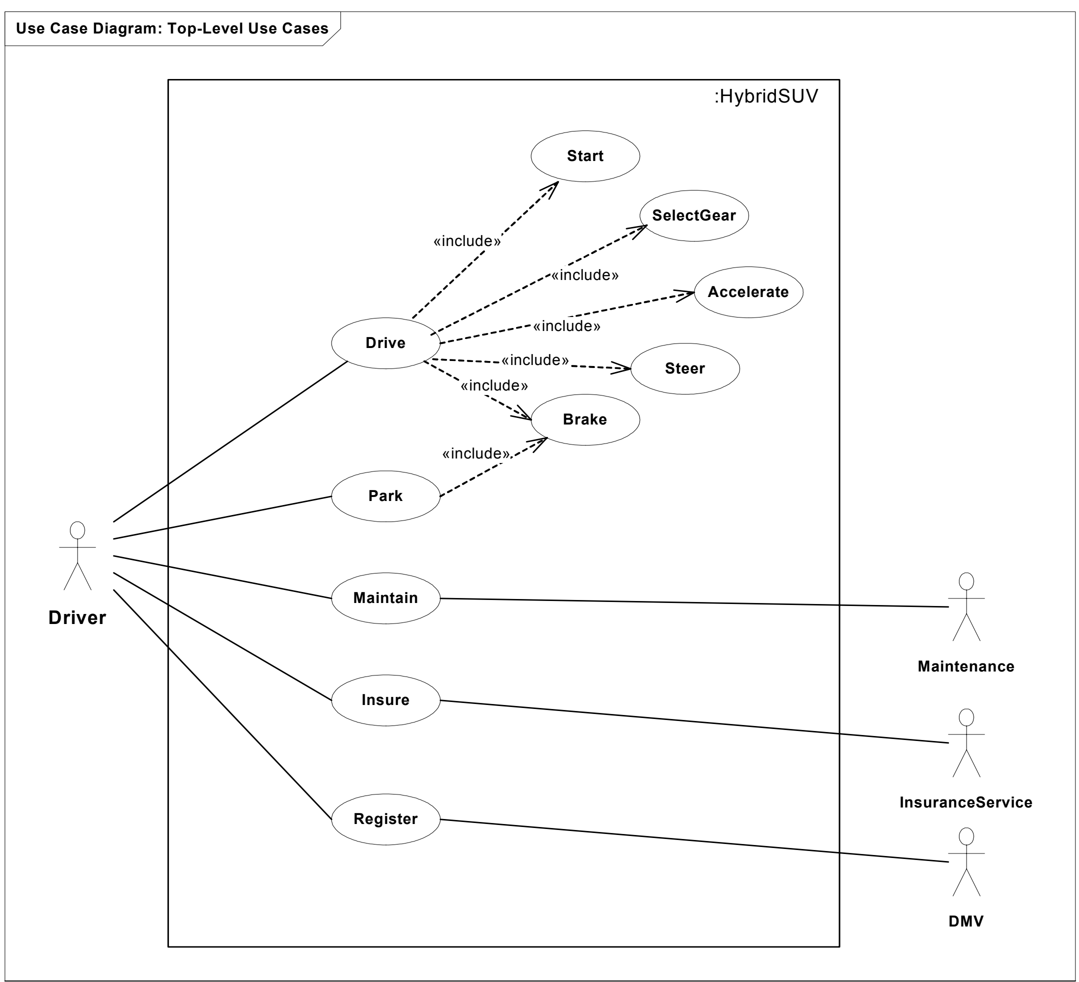


Figure B-10. Use Case Diagram

B.3.6 Sequence Diagrams

Figure B-11 shows a “black-box” sequence diagram describing the “Drive Car” use case. The diagram is considered a “black-box” diagram as it shows a single lifeline for the Hybrid SUV and does not show any internal parts. The diagram has interaction occurrences for each of the «included» use cases of Figure B-10 and shows a time ordering of executions (other time orderings are possible, i.e. this is a partial specification of behavior).

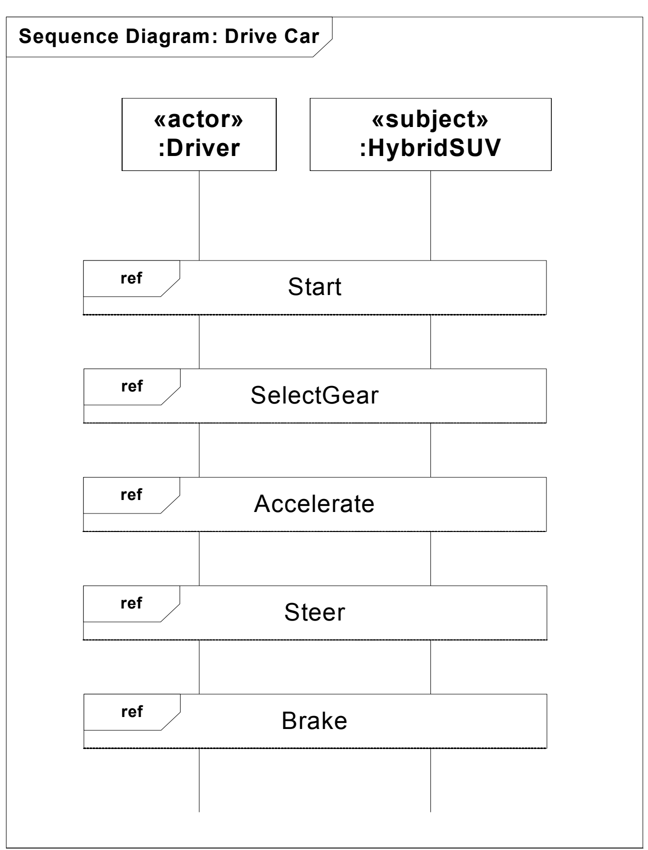


Figure B-11. Sequence Diagram: Top Level "black-box"

Figure B-12 shows the details of the Accelerate interaction occurrence. When the :Driver presses the accelerator the :Hybrid-SUV receives the ApplyAccelerator message with a parameter: "angle". In response to this event, the :HybridSUV performs its ControlPower activity.

This is still a "black-box" view of the system, however the "ref: Accelerate Allocate" text on the :HybridSUV lifeline, known as a part decomposition, indicates that a white-box (or allocated) view exists that shows the internal interactions of the :HybridSUV for this scenario.

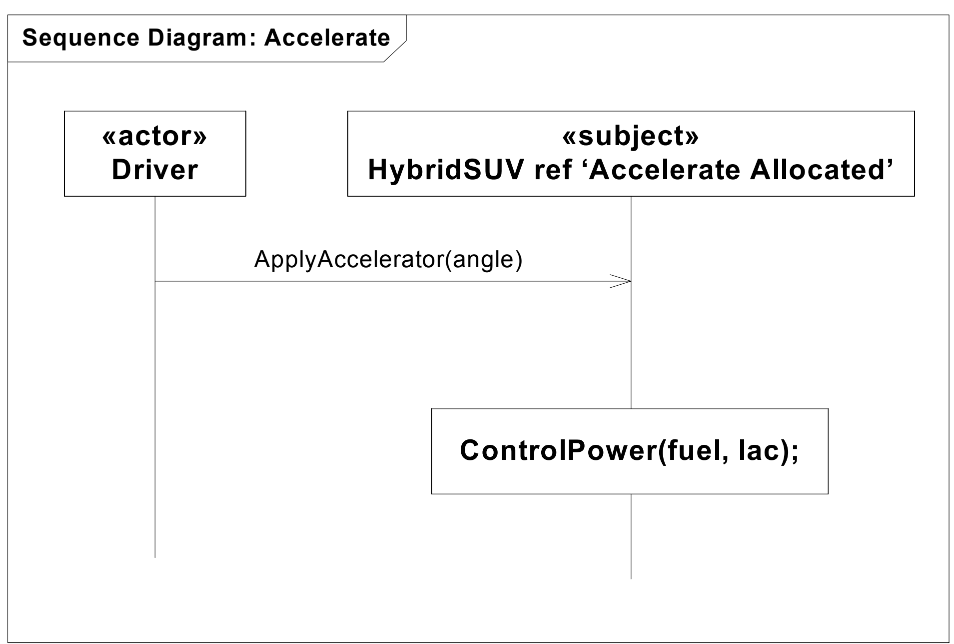


Figure B-12. Sequence Diagram: Accelerate Scenario

Figure B-13 shows the “white-box” sequence diagram for the Accelerate scenario. The choice of Activity Diagrams (see next section), Sequence Diagrams, or a combination of the two for defining scenarios is a matter of personal taste. Of course a diagram with this level of detail cannot be created until candidate parts (units, subsystems, etc. of the equipment breakdown structure) have been identified. See Section B.3.8 for the Block Definition diagram describing the blocks referenced in Figure B-13.

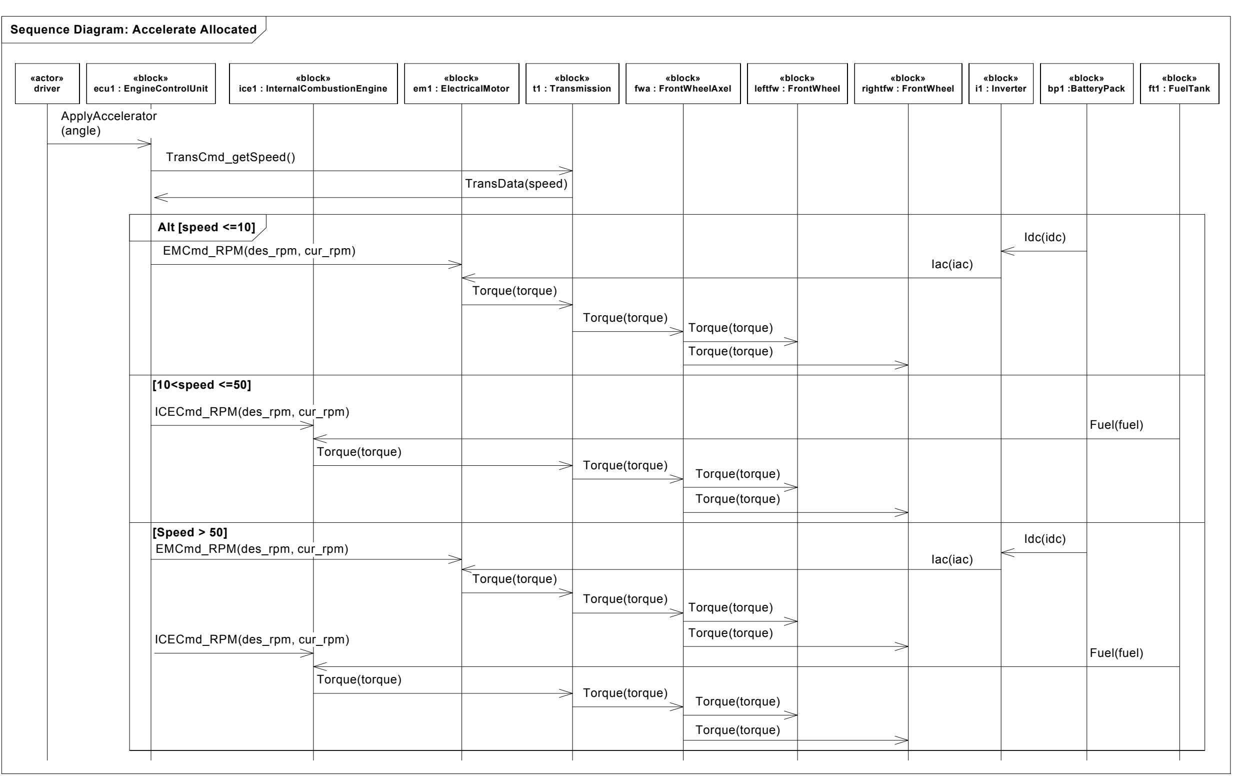


Figure B-13. Sequence Diagram: Accelerate Scenario allocated to components of Hybrid SUV

B.3.7 Activity Diagram for “Control Power”

Figure B-14 through Figure B-17 shows the Activity Diagram for the ControlPower Activity shown on Figure B-12. These diagrams are fully elaborated, including partitions, commonly called “swim-lanes” to illustrate the allocation of behavior to structure. The initial versions of these diagram may not have partitions, but would rather focus on behavior alone. Once the equipment breakdown structure is defined, partitions may be added or explicit allocation can be performed (see Section B.3.10 for more on explicit allocations).

Note that many of the elements displayed on these diagrams are representations of the same model element shown on other diagrams. for example the ApplyAccelerator SendSignalAction is the same underlying model element as the message ApplyAccelerator on Figure B-12 and Figure B-13.

The circles with letters inside represent off page connectors used to connect flows across pages of large diagrams.

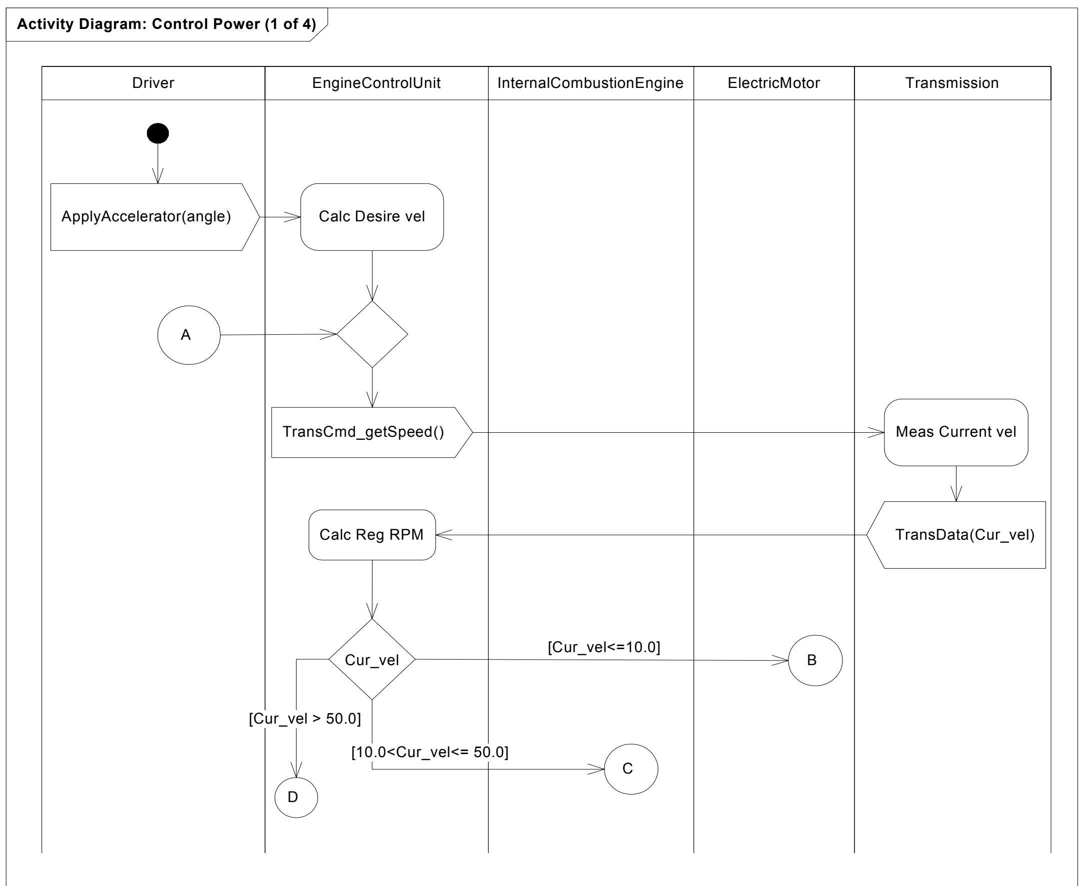


Figure B-14. Activity Diagram Example: Control Power (1 of 4)

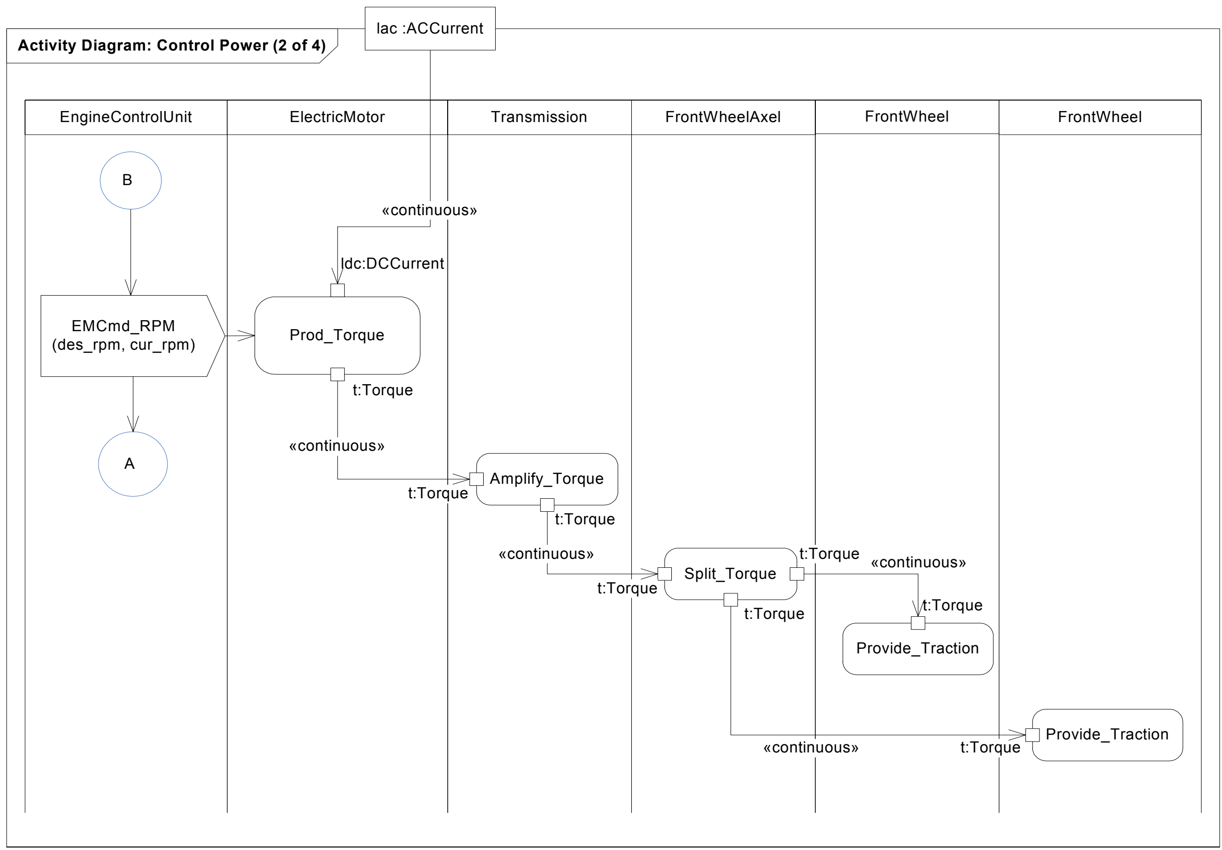


Figure B-15. Activity Diagram Example: Control Power (2 of 4)

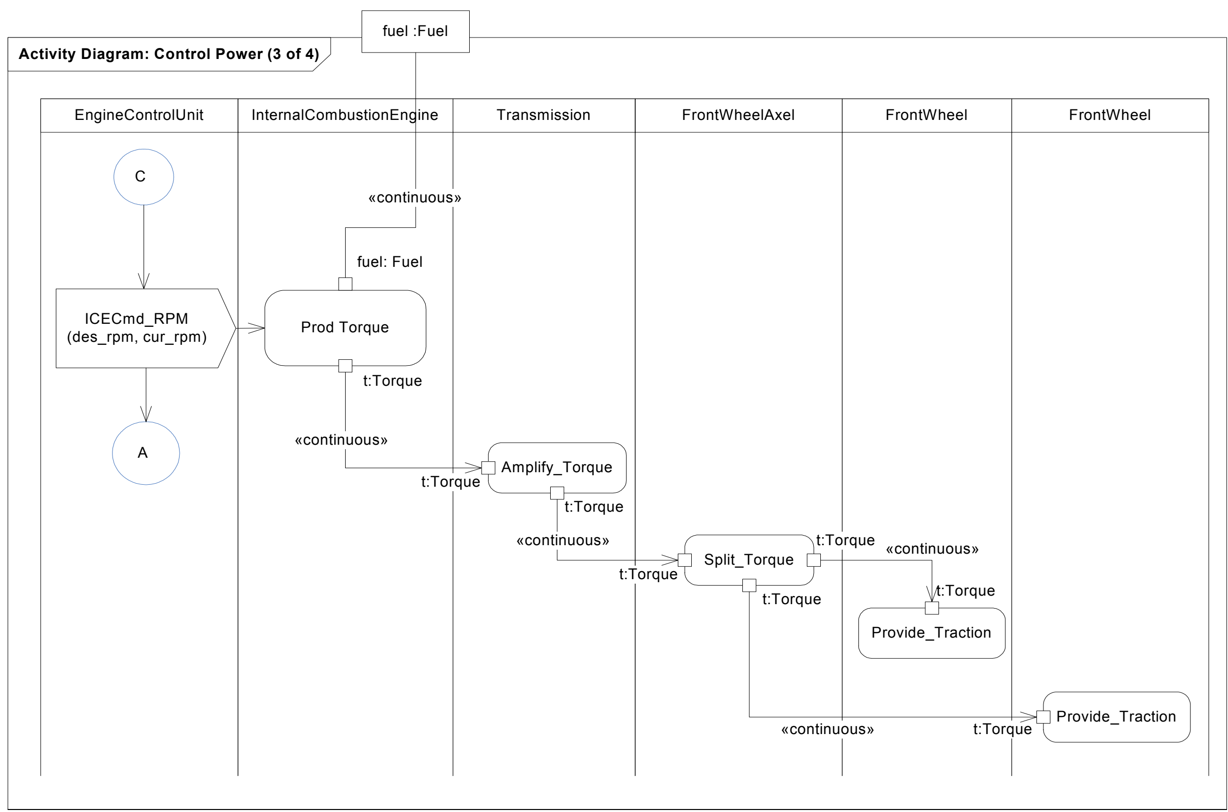


Figure B-16. Activity Diagram Example: Control Power (3 of 4)

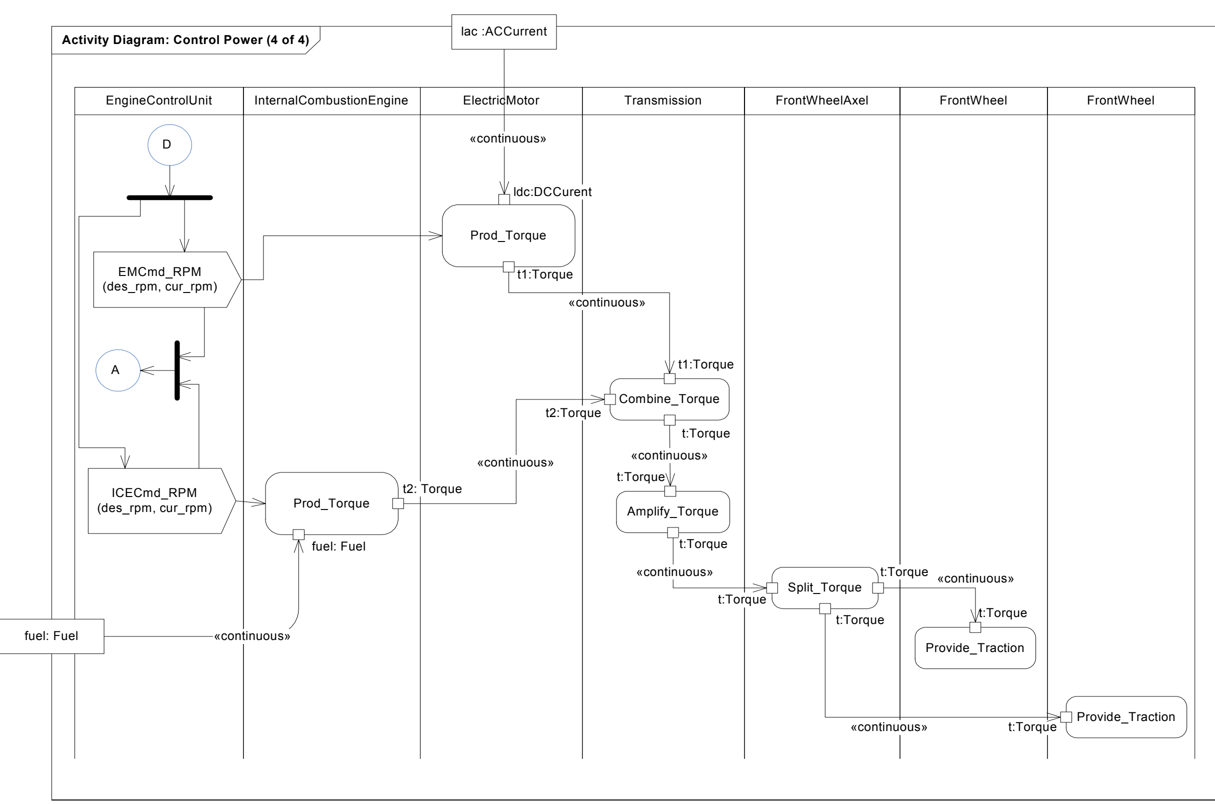


Figure B-17. Activity Diagram Example: Control Power (4 of 4)

B.3.8 External Block Diagram for the Hybrid SUV

Figure B-18 shows the external Block Diagram of the HybridSUV. This diagram defines the equipment breakdown structure of the vehicle.

All block properties are elided (not shown) as the intent of this diagram is to simply depict the part hierarchy.

Note that the Block FrontWheel plays two roles in the EBS, that of the left front wheel (leftfw) and that of the right from wheel (rightfw). Similarly, their are two RearWheel as indicated by the multiplicity of 2. If multiplicities are unspecified they default to 1, for example there is only one BrakingSubsystem in any given HybridSUV.

A Hybrid SUV has a Braking Subsystem, a Power Subsystem and a Chassis Subsystem. The units that compose the Braking Subsystem and Chassis Subsystem are not shown as the focus of this example is the Power Subsystem.

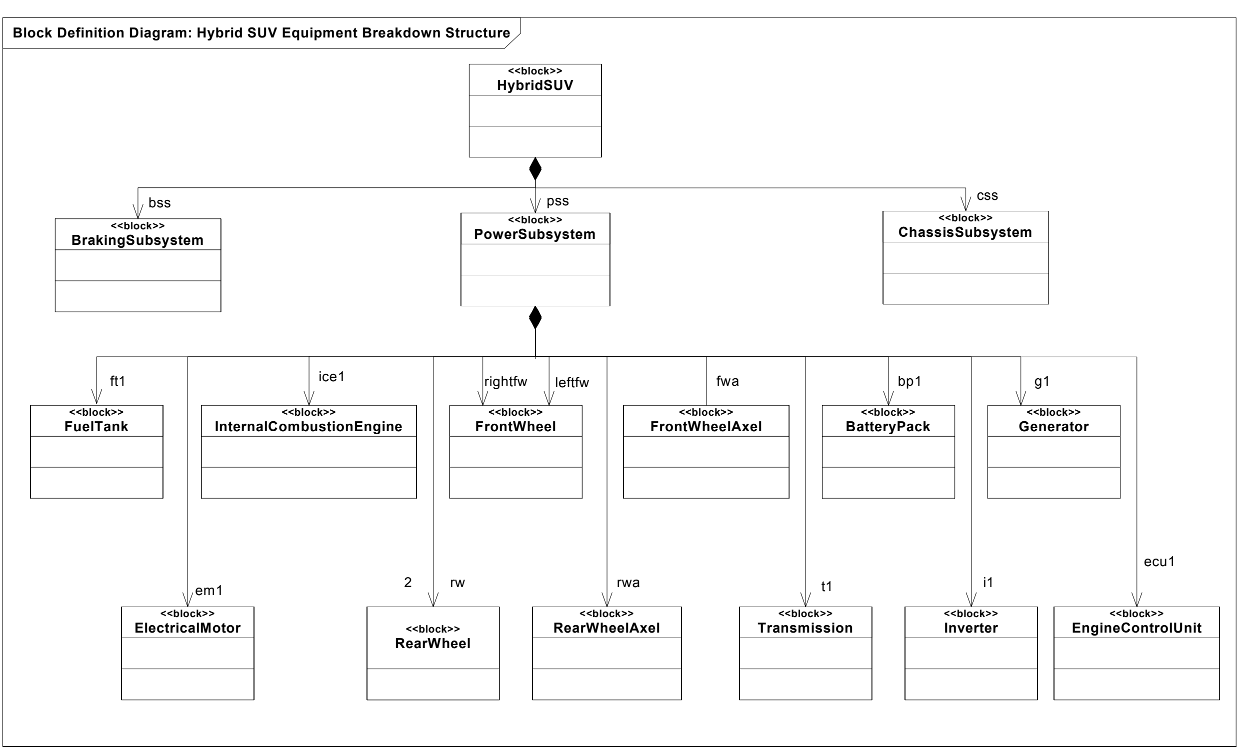


Figure B-18. Block Definition Diagram: Equipment Breakdown Structure

B.3.9 Transmission Properties

Figure B-19 shows the properties of the Transmission Block on a Block Definition diagram. A number of attributes, such as mass:Kilograms, dir:Direction, etc. capture information about the state of the transmission. The types are defined elsewhere, for example the ValueType Kilogram, which has associated dimension (=M), unit (=kg), and quantity (=Mass) is

defined in the SysML Model Library (see Appendix D). Similarly the ValueTypes Torque, KPH are defined in the SysML Model Library.

Also shown are the operations, such as shift(), which were determined from the activities allocated to the Transmission (see Figure B-20 for allocations). As a result of the allocations made in Figure B-20, this block has the stereotype «allocated» applied. The “AllocatedFrom” property displayed in the stereotype property symbol indicates which activities have been allocated to the transmission.

The ports (ServicePorts TJ1 and TJ2, and FlowPorts TS1, TS2 and TS3) and associated interfaces (fsTorque which is defined in Figure B-24) are also shown, resulting in a complete specification of the block that can be re-used in other systems.

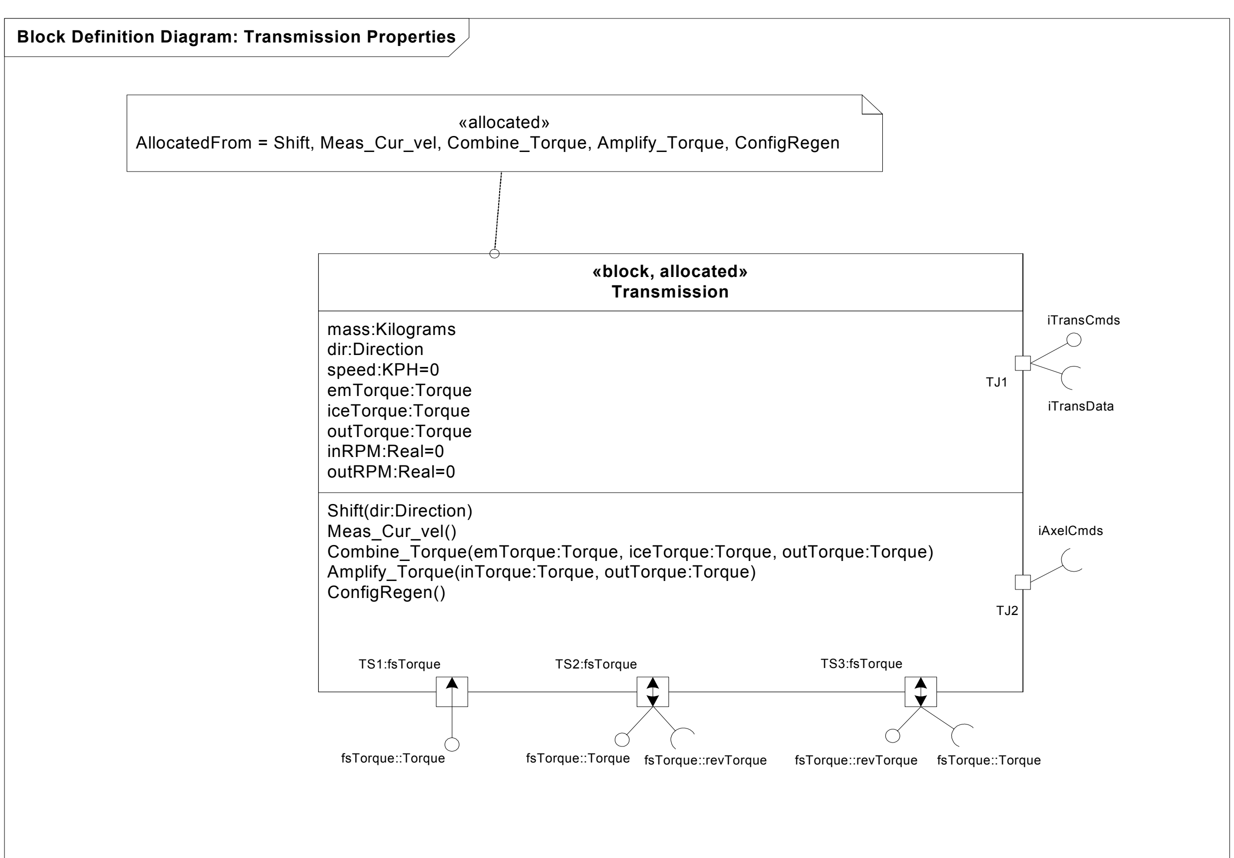


Figure B-19. Block Definition Diagram: Properties of Transmission

B.3.10 Allocations

Figure B-20 shows an example of the explicit allocation of behavior to structure, in particular allocations to the Transmission. The Activity diagrams of Figure B-14 through Figure B-17 showed implicit allocations of behavior to structure (function to form) via partitions.

Each leaf activity from the activity model is allocated to one Block.

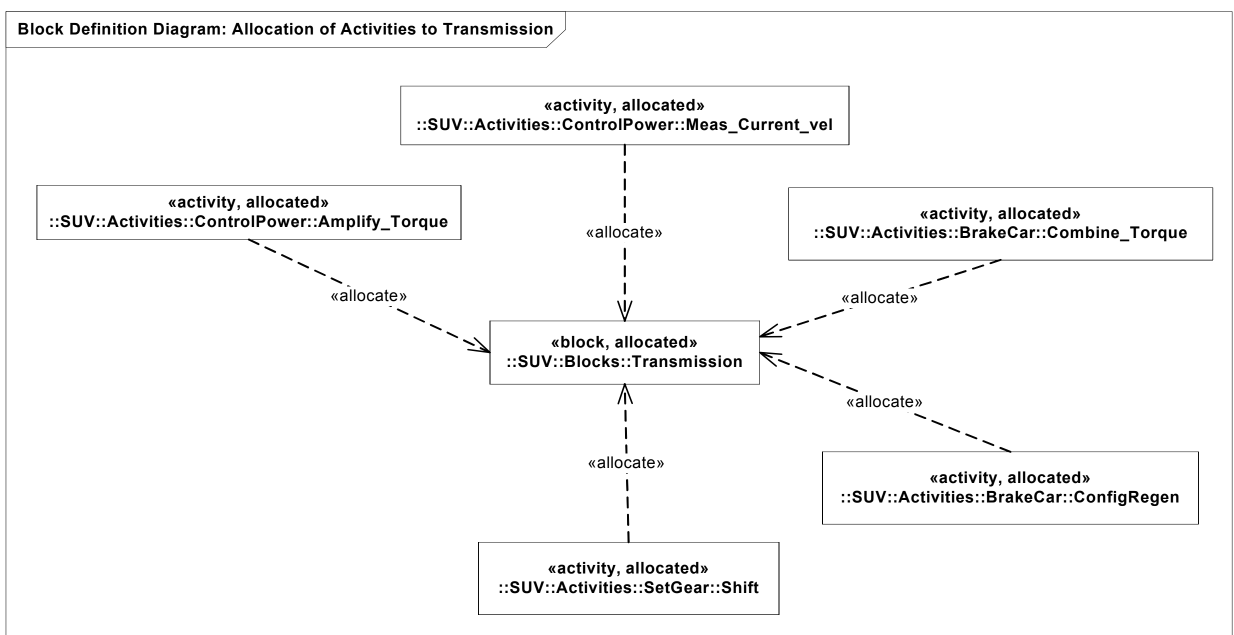


Figure B-20. External Block Diagram: Allocation of Behavior to the Transmission

Figure B-21 shows an alternate tabular presentation of the allocation traces in the model. This table could easily be generated by tools via queries on the model. Similar tables, for example verification matrices, could also be generated for other trace types.

Source Activity	Target Block								
	EngineControlUnit	InternalCombustionEngine	Transmission	FrontWheelAxel	FrontWheel	ElectricalMotor	BrakingSubsystem	Inverter	BatteryPack
InitializeICE	allocate								
StartICE		allocate							
SwitchGear	allocate								
Shift			allocate						
Calc_Desire_vel	allocate								
Meas_Current_vel			allocate						
Calc_Req_RPM	allocate								
Amplify_Torque			allocate						
Split_Torque				allocate					
Provide_Traction					allocate				
Prod_Torque		allocate							
Prod_Torque						allocate			
Brake							allocate		
SetRegenBrake	allocate								
ConfigRegen			allocate						
Combine_Torque			allocate	allocate					
GenerateAC						allocate			
RectifyCurrent								allocate	
StoreElectricalEnergy									allocate
ConvertDCtoAC								allocate	

Figure B-21. Example tabular format of allocation traces

B.3.11 Block Diagram

Figure B-22 shows the Internal Block diagram of the Power Subsystem

The diagram shows the following external interfaces of the PowerSubsystem connected to the frame of the diagram: Service Ports PSJ2 and PSJ1 and Flow Port GasIn.

Each of the parts of the PowerSubsystem, defined in Figure B-18, with the exception of the generator which is purposely elided to save space, is shown in this diagram along with the interconnections (eg. CANBus) between parts.

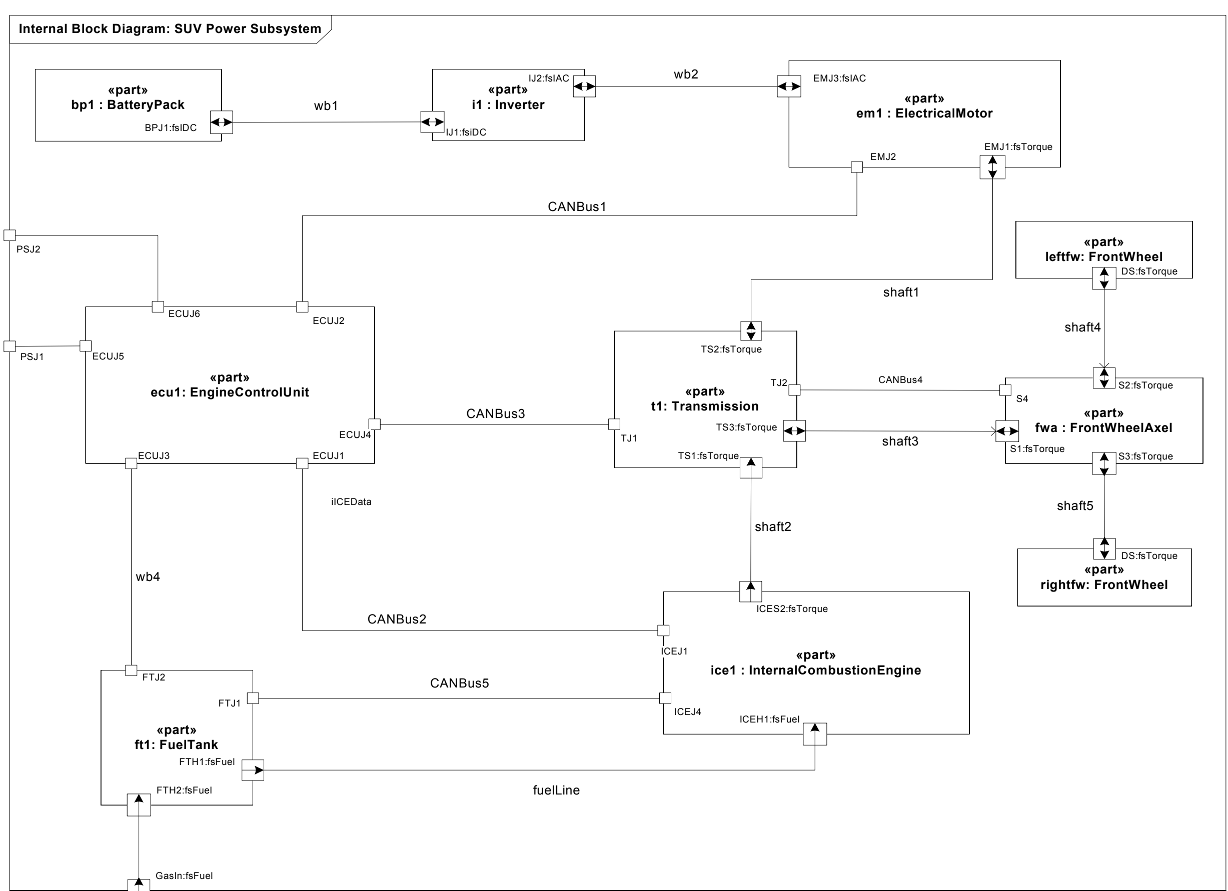


Figure B-22. Block Diagram: Internal structure of the Power Subsystem

B.3.12 Interfaces

Figure B-23 shows the interface definitions for the command and telemetry interfaces of the power subsystem.

If the name, or composition (i.e. the signals specified in each interface) of one of these interfaces is changed all usages of the interface and signals on all other diagrams will be updated as they represent different usages of the same model element. This greatly simplifies interface control by ensuring consistency for all usages of the interface across the model.

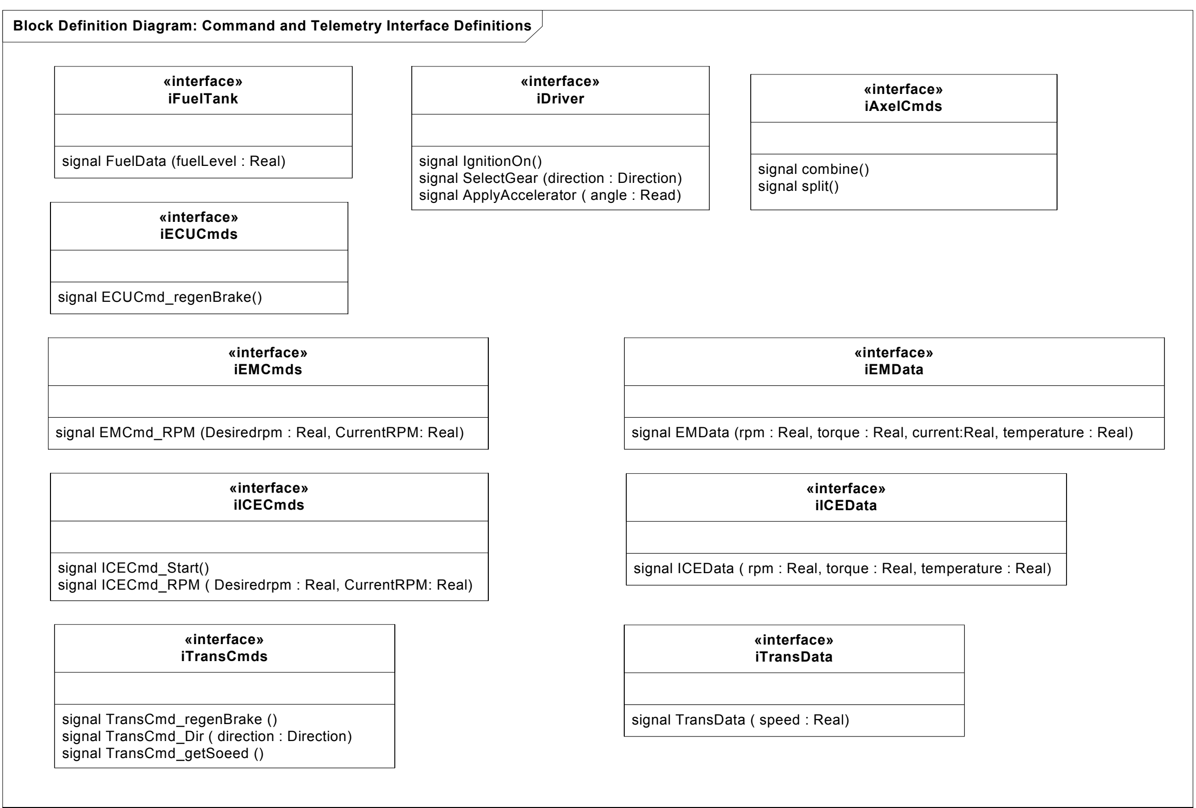


Figure B-23. External Block Diagram: Command and Telemetry Interface Definitions

Figure B-24 shows the Flow Specifications defining the interfaces for physical (matter, data, energy) flows. This Figure also shows the definitions of the blocks and value types which type the Flow Properties (which are the attributes of the Flow Specifications, for example t:Torque). Torque, ACCurrent and DCCurrent are user defined value types which are type compatible with the pre-defined data type Real.

Note that the signals defined in the Flow Specifications, and shown on the sequence diagram of Figure B-13, are only required if model execution based upon discrete event simulation is desired. Discrete event simulation is supported by several UML 2.0 tools today. These signals are included in this example as a useful design pattern in such cases. Operation calls (for example get/set operations) could have been used in lieu of signals to implement the communications.

The same comment regarding model concordance and interface control as above applies to Flow Specifications (which are a kind of Interface).

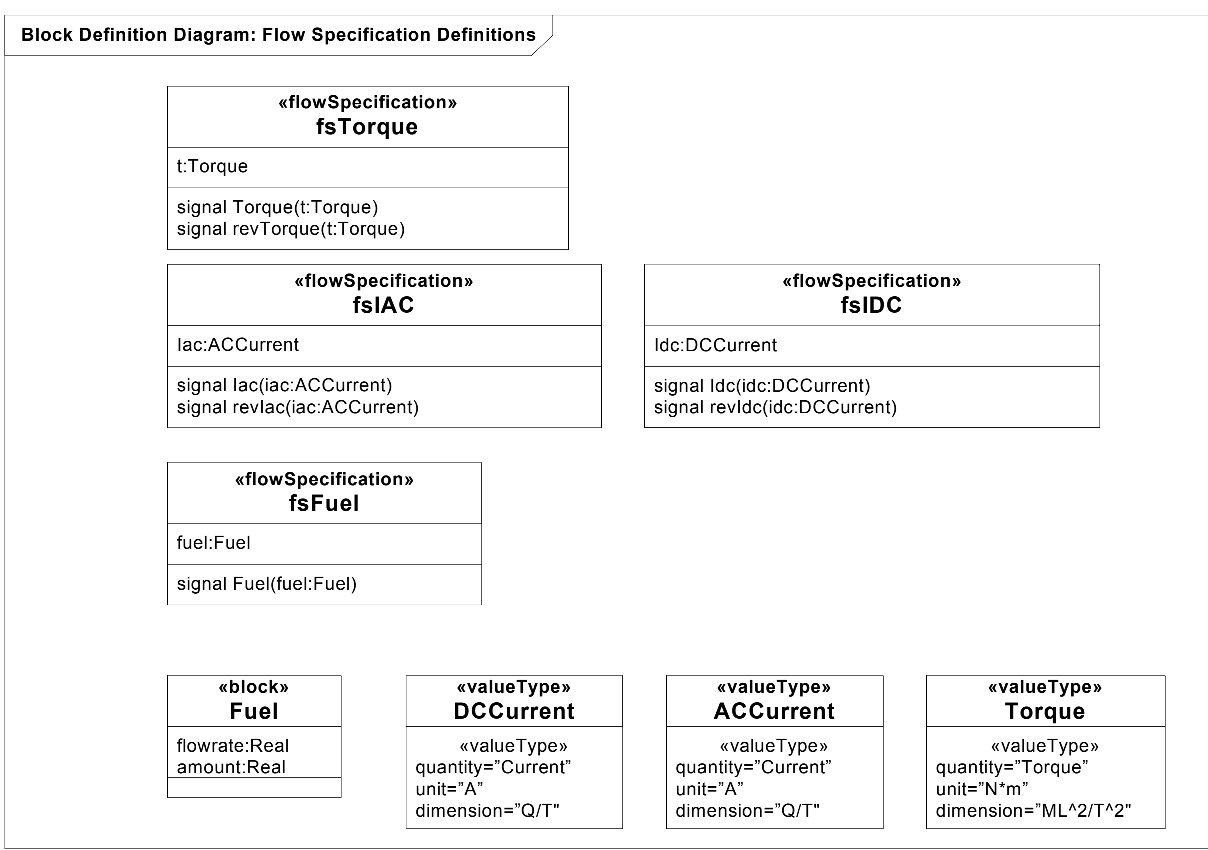


Figure B-24. External Block Diagram: Flow Specification Definitions

B.3.13 State Machine Diagram for the Transmission “Shift” behavior

Figure B-25 shows the state machine diagram for the Transmission. This diagram describes the dynamic behavior of the Transmission for the Activity “Shift”. This version of the state machine diagram uses the notation commonly refered to as “state centric”. In this notation the nodes represent the states and the trigger, guard, action associated with transitions are specified using text associated with the transition.

The inverted fork notation in the lower left hand corner of the Reverse and Forward states indicates that these states have sub-states defined.

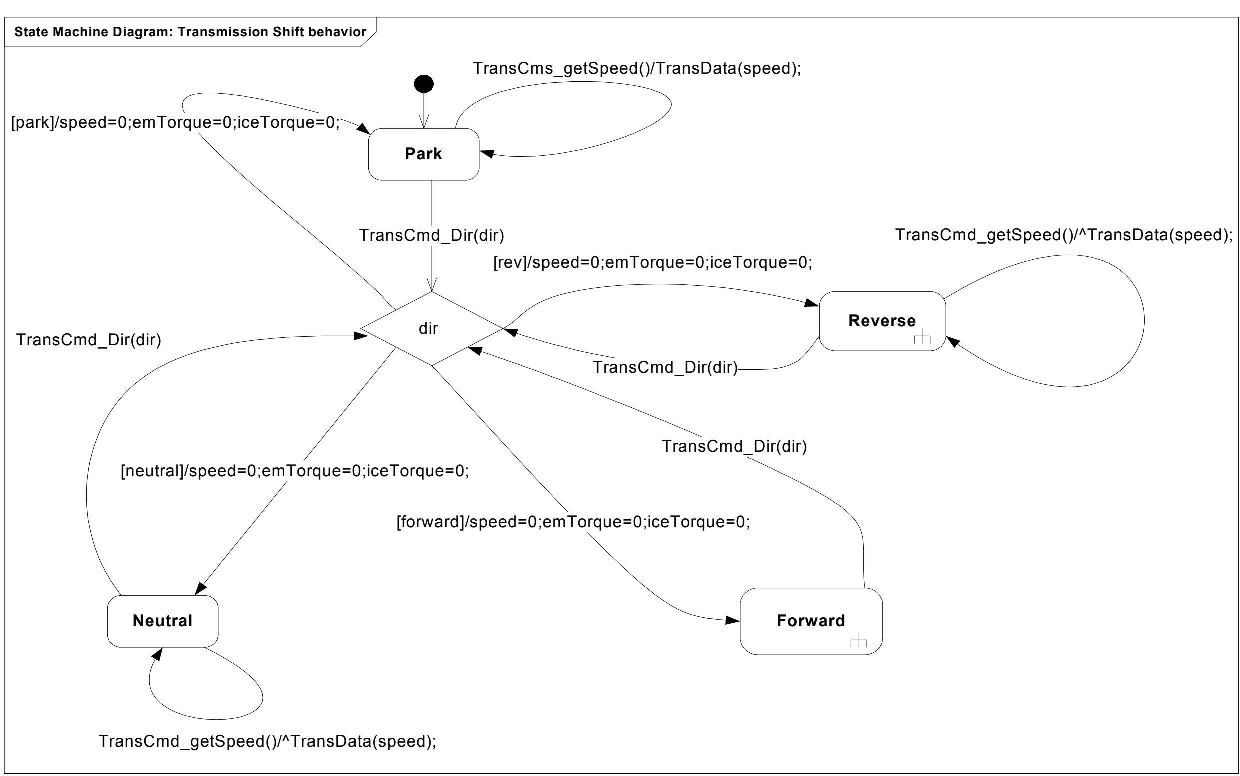


Figure B-25. State-Centric State Machine Diagram: Transmission "Switch Gear" Behavior

Figure B-26 shows the same statemachine diagram drawn using the "transition centric" notation. Using this notation, triggers, actions and guards are shown as nodes on the diagram. Both diagrams are semantically equivalent. The choice of state centric or transition centric statemachine notation is a matter of personal taste.

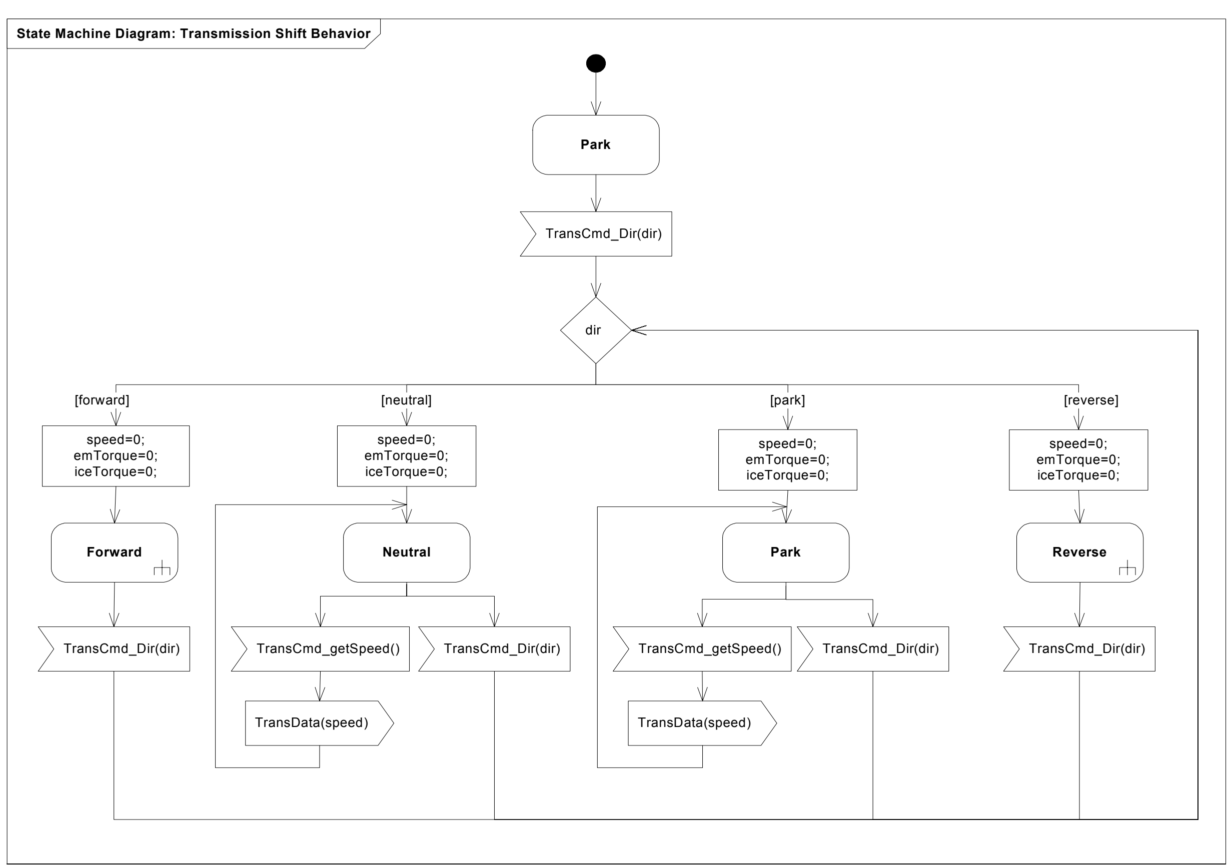


Figure B-26. Transition-Centric State Machine Diagram: Transmission "Switch Gear" Behavior

B.3.14 Parametric Block Diagram

Figure B-27 shows another application of parametric blocks in addition to the trade study application discussed in Section B.3.2, namely roll-up of technical performance measures. The first parametric constraint usage, TotalMass, rolls-up the mass of the level one parts in the equipment brakedown structure (the Braking Subsystem, Power Subsystem and Chassis Subsystem). This pattern could be used recursively to roll up the total mass of each subsystem.

The second parametric constraint usage places a constraint that the total mass be less than 1500 kg.

The definitions of these parametric constraints would be done on an Internal Block diagram as was done in Figure B-5 and is omitted in the interest of space.

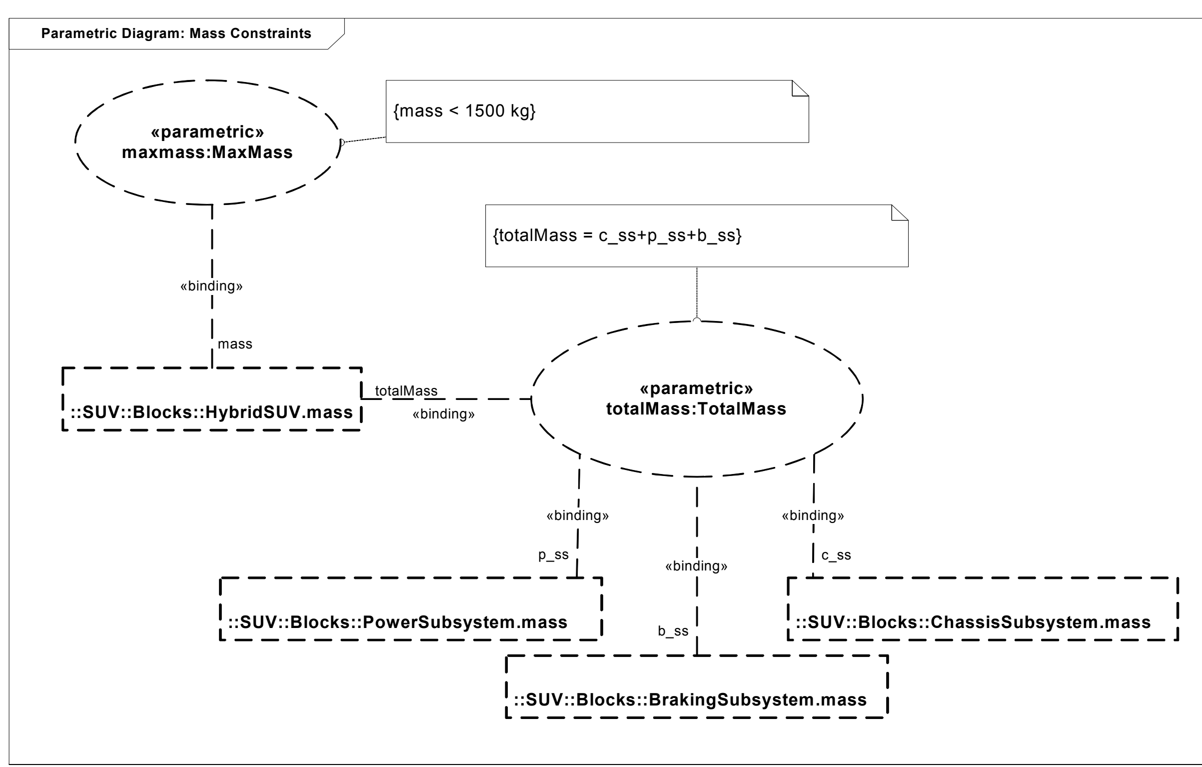


Figure B-27. Mass Constraints

B.3.15 Requirements Satisfaction

Figure B-28 shows an example of requirements satisfaction relationship between various model elements and the requirements they satisfy.

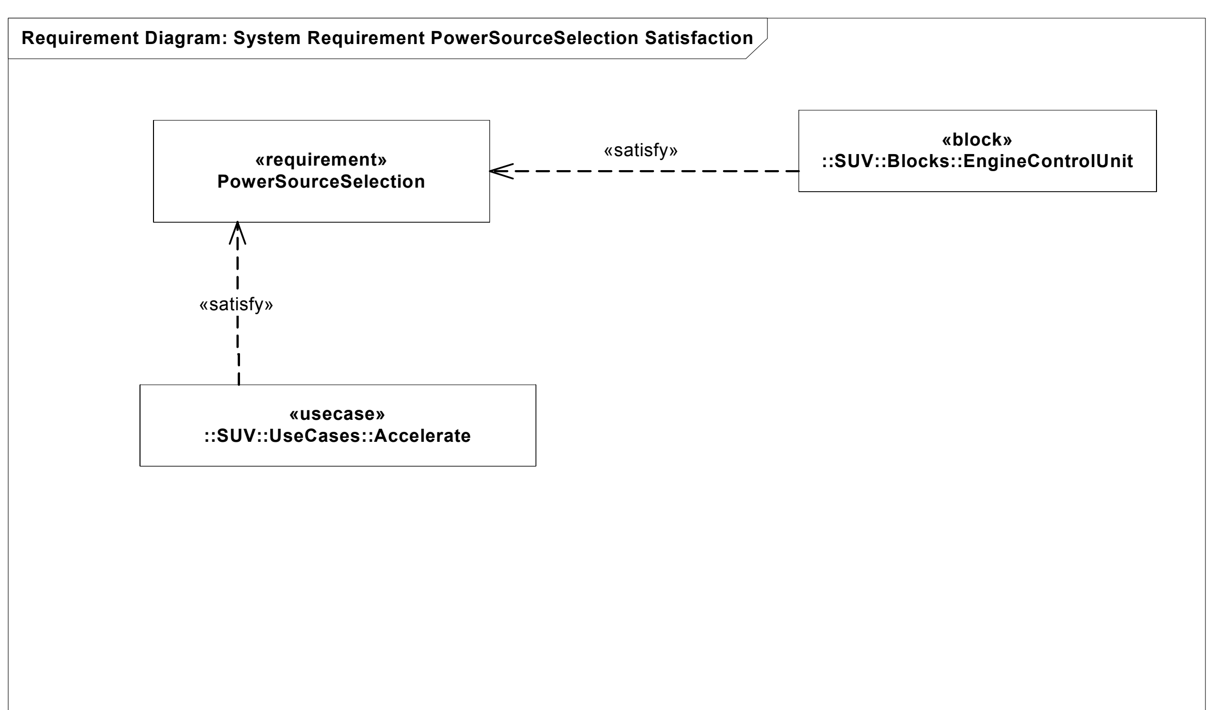


Figure B-28. Requirement Diagram: Requirement Satisfaction

B.3.16 Complete Traceability

Figure B-29 shows complete traceability from the user requirement, through derived system requirements to detailed behavior specifications for each subsystem/unit. Having established the traceability as we were working through the analysis and design, the production of this Requirement diagram should be a trivial exercise. Compare this with traditional analysis and design, where in many cases establishing traceability is a separate activity done after the fact.

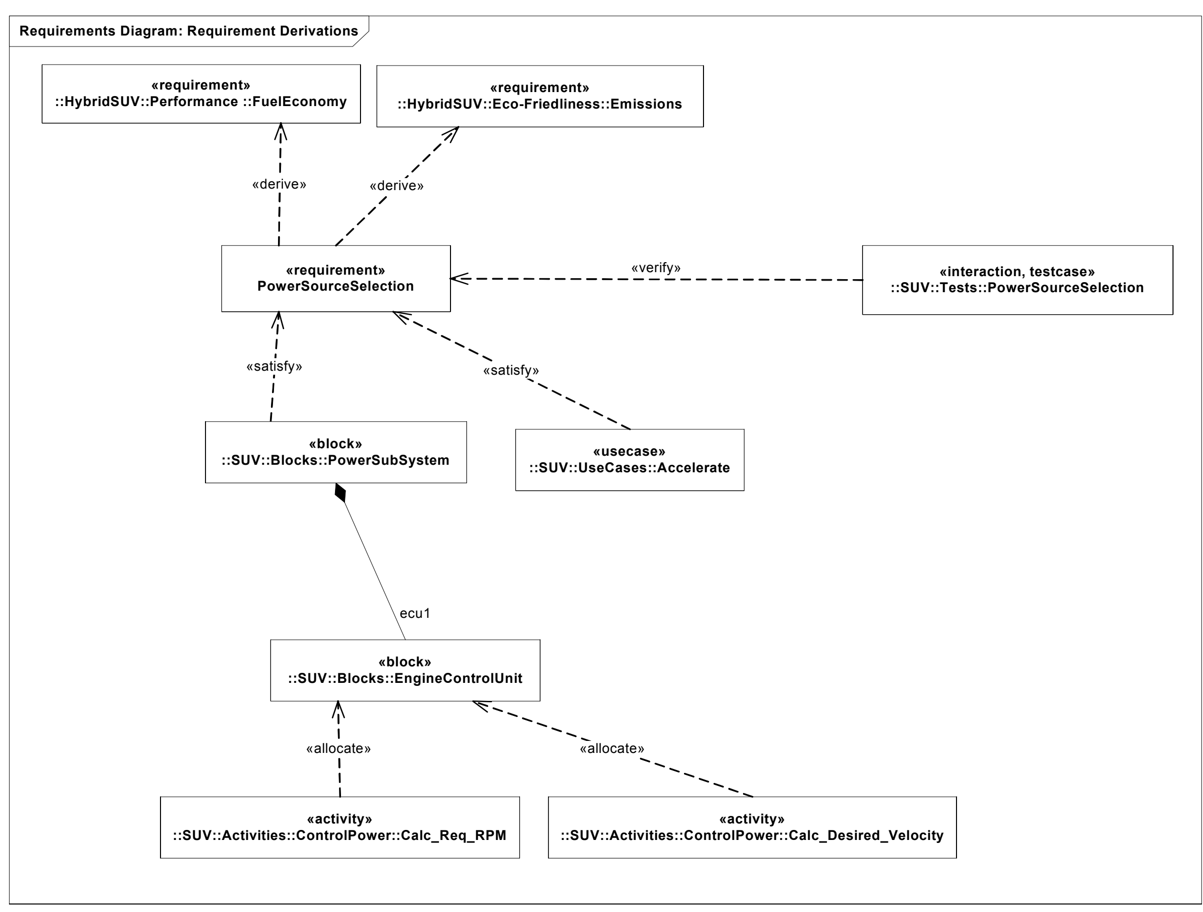


Figure B-29. Requirement Diagram: Requirement Traceability

Only a portion of the traceability is shown on Figure B-29. Tools that support SysML should be able to generate complete or custom traceability tables via queries on the model. Figure B-30 shows and example of such a table.

Dependency	TraceKind	SourceName	SourceKind	TargetName	TargetKind
of ::SUV::Blocks::EngineControlUnit	allocate	InitializeICE	Activity	EngineControlUnit	Block
of ::SUV::Blocks::InternalCombustionEngine	allocate	StartICE	Activity	InternalCombustionEngine	Block
of ::SUV::Blocks::EngineControlUnit	allocate	SwitchGear	Activity	EngineControlUnit	Block
of ::SUV::Blocks::Transmission	allocate	Shift	Activity	Transmission	Block
of ::SUV::Blocks::EngineControlUnit	allocate	Calc_Desire_vel	Activity	EngineControlUnit	Block
of ::SUV::Blocks::Transmission	allocate	Meas_Current_vel	Activity	Transmission	Block
of ::SUV::Blocks::EngineControlUnit	allocate	Calc_Req_RPM	Activity	EngineControlUnit	Block
of ::SUV::Blocks::Transmission	allocate	Amplify_Torque	Activity	Transmission	Block
of ::SUV::Blocks::FrontWheelAxel	allocate	Split_Torque	Activity	FrontWheelAxel	Block
of ::SUV::Blocks::FrontWheel	allocate	Provide_Traction	Activity	FrontWheel	Block
of ::SUV::Blocks::InternalCombustionEngine	allocate	Prod_Torque	Activity	InternalCombustionEngine	Block
of ::SUV::Blocks::ElectricalMotor	allocate	Prod_Torque	Activity	ElectricalMotor	Block
of ::SUV::Blocks::BrakingSubsystem	allocate	Brake	Activity	BrakingSubsystem	Block
of ::SUV::Blocks::EngineControlUnit	allocate	SetRegenBrake	Activity	EngineControlUnit	Block
of ::SUV::Blocks::Transmission	allocate	ConfigRegen	Activity	Transmission	Block
of ::SUV::Blocks::Transmission	allocate	Combine_Torque	Activity	Transmission	Block
of ::SUV::Blocks::FrontWheelAxel	allocate	Combine_Torque	Activity	FrontWheelAxel	Block
of ::SUV::Blocks::ElectricalMotor	allocate	GenerateAC	Activity	ElectricalMotor	Block
of ::SUV::Blocks::Inverter	allocate	RectifyCurrent	Activity	Inverter	Block
of Blocks::BatteryPack	allocate	StoreElectricalEnergy	Activity	BatteryPack	Block
of ::SUV::Blocks::Inverter	allocate	ConvertDCtoAC	Activity	Inverter	Block
of ::SUV::Requirements::HybridSUV::Performance::Range	derive	Regenerative Braking	Requirement	Range	Requirement
of FuelEconomy	derive	Regenerative Braking	Requirement	FuelEconomy	Requirement
of Braking	derive	Regenerative Braking	Requirement	Braking	Requirement
of ::SUV::Requirements::HybridSUV::Performance::FuelEconomy	derive	PowerSourceSelection	Requirement	FuelEconomy	Requirement
of HybridSUV::'Eco-Friendliness'::Emissions	derive	PowerSourceSelection	Requirement	Emissions	Requirement
of ::SUV::Requirements::HybridSUV::Performance::Acceleration	derive	PowerSourceSelection	Requirement	Acceleration	Requirement
of ::SUV::Requirements::HybridSUV::Performance::Range	derive	PowerSourceSelection	Requirement	Range	Requirement
of Braking	satisfy	BrakingSubsystem	Block	Braking	Requirement
of Braking	satisfy	Drive	Operation	Braking	Requirement
of ::SUV::Tests::PowerSourceSelection	verify	PowerSourceSelection	Interaction	PowerSourceSelection	Requirement
of Regenerative Braking	verify	EnergyConversion	Statemachine	Regenerative Braking	Requirement
of Regenerative Braking	verify	ConvertKineticToEnergy	Interaction	Regenerative Braking	Requirement

Figure B-30. Sample Traceability Table

Appendix C. Non-Normative Extensions

This appendix describes extensions to SysML that are being considered for standardization, but at this time are non-normative (i.e, they are not part of the official SysML standard). Users and tool vendors are encouraged to experiment with these extensions as they set fit, and provide feedback to the SysML specification team (mailto:feedback@SysML.org) regarding the usefulness of these extensions.

Non-normative extensions consist of stereotypes and diagram extensions and are organized by language unit, consistent with how the main body of this specification is organized.

C.1 Activities

C.1.1 Overview

Two nonnormative extensions to activities are described for:

- Enhanced Functional Flow Block Diagrams.
- Streaming activities that accept inputs and/or provide outputs while they are active.

More information on these extensions and the standard SysML extensions is available at [Bock. C., "SysML and UML 2 Support for Activity Modeling," to appear in the Journal of the International Council of Systems Engineering].

C.1.2 Diagram Elements

Table 1 describes the concrete syntax for the non-normative extensions defined in this section.

Table 1. *Graphical nodes for non-normative extensions to Activities*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Activity	«effbd» ActivityName	**SysML::Activities::«EFFBD»**	**Non-normative**
	«streaming» ActivityName	**SysML::Activities::«Streaming»**	**Non-normative**

Table 1. *Graphical nodes for non-normative extensions to Activities*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
	«nonStreaming» ActivityName	**SysML::Activities::«Non-Streaming»**	**Non-normative**

C.1.3 Package Structure

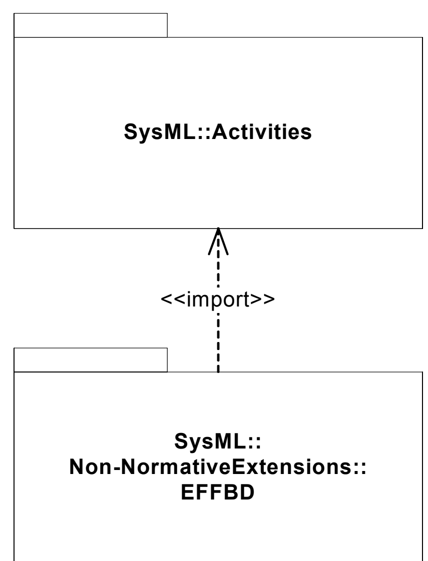


Figure C-1. Package Structure for SysML Activities

C.1.4 UML Extensions

Abstract Syntax

Package Activities

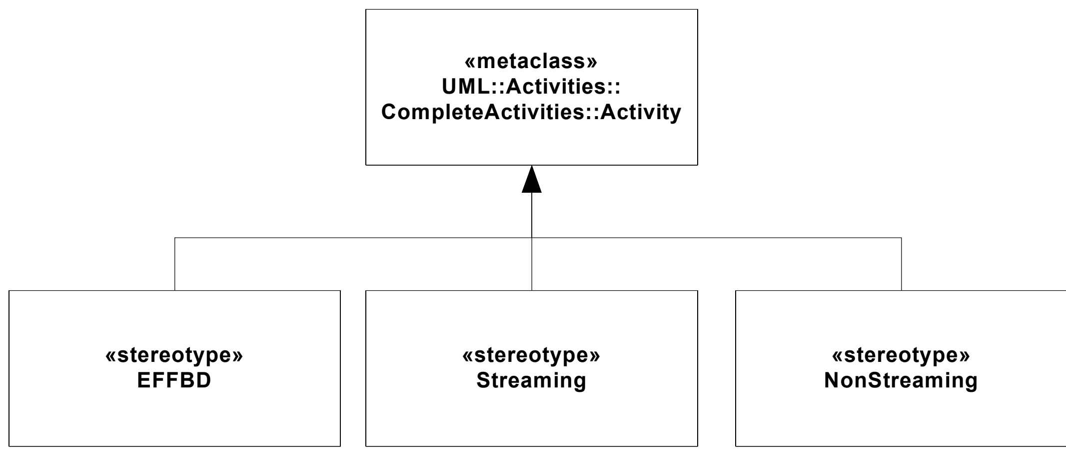


Figure C-2. Abstract Syntax for Activity Non-Normative extensions.

C.1.4.1 Stereotypes

EFFBD

Description

Enhanced Functional Flow Block Diagrams (EFFBD) are a widely-used systems engineering diagram, also called a behavior diagram. Most of its functionality is a constrained use of UML activities, as described below. This extension does not address replication, resources, or kill branches. Kill branches can be translated to actvities using interruptible regions and join specifications.

When this stereotype is applied to an activity it specifies that the activity conforms to the constraints necessary for EFFBD defined below.

Constraints

When the «EFFBD» stereotype is applied to an activity, its contents must conform to the following contraints:

[1] (On Activity) Activities do not have partitions.

[2] (On Activity) All decisions, merges, joins and forks are well-nested. In particular, each decision and merge are matched one-to-one, as are forks and joins, accounting for the output parameter sets acting as decisions, and input parameters and control acting as a join.

[3] (On Action) All actions require exactly one control edge coming into them, and exactly one control edge coming out, except when using parameter sets.

[4] (On ControlFlow) All control flows into an action target a pin on the action that has isControl = true.

[5] (On ObjectNode) Ordering is first-in first out, ordering = FIFO.

[6] (On ObjectNode) Object flow is never used for control, isControlType = false, except for pins of parameters in parameter sets.

[7] (On Parameter) Parameters take and produce no more than one item, multiplicity.upper =1.

[8] (On Parameter) Output parameters produce exactly one value, multiplicity.lower = 1. The «optional» stereotrype cannot be applied to parameters.

[9] (On Parameter) Parameters cannot be streaming or exception.

[10] (On ParameterSet) Parameter sets only apply to output parameters.

[11] (On ParameterSet) Parameter sets only apply to control. Parameters in parameter sets must have pins with isControlType = true.

[12] (On ParameterSet) Parameter sets have exactly one parameter, and it must not be shared with other parameter sets.

[13] (On ParameterSet) If one output parameter is in a parameter set, then all output parameters of the behavior or operation must be in parameter sets.

[14] (On ActivityEdge) Edges cannot have time constraints.

[15] The following SysML stereotypes cannot be applied: «rate», «controlOperator», «noBuffer», «overwrite».

Streaming

Description

When the «Streaming» stereotype is applied to an activity it specifies that the activity can accept inputs or provide outputs after they start and before they finish, respectively.

Constraints

[1] The activity has at least one streaming parameter.

NonStreaming

Description

When the «NonStreaming» stereotype is applied to an activity it specifies that the activity can accept inputs only when it starts, and provide outputs only when it finishes.

Constraints

[1] The activity has no streaming parameters

C.1.4.2 Diagram Extensions

N/A

C.1.5 Usage Examples

Figure C-3 shows an example activity diagram with the «EFFBD» stereotype applied, translated from [Long. J., "Relationships between common graphical representations in system engineering," 2002]. The stereotype applies the constraints specified in above, for example, that the data outputs on all functions are required and that queuing is FIFO.

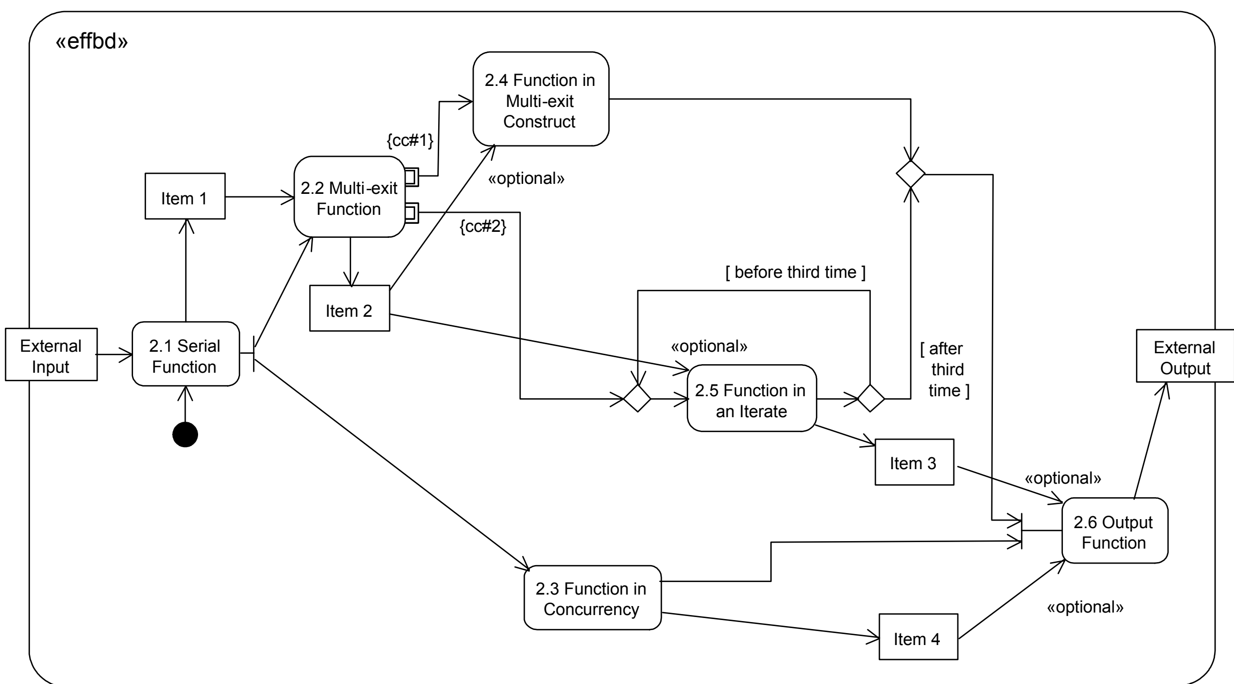


Figure C-3. Example activity with «EFFBD» stereotype applied

Figure C-4 shows an example activity diagram with the «Streaming» and «NonStreaming» stereotypes applied, adapted from [MathWorks, "Using Simulink," 2004]. It is a numerical solution for the differential equation $x'(t) = -2x(t) + u(t)$. Item types are omitted brevity. The «streaming» and «nonStreaming» stereotypes indicate which subactivities take inputs and produce outputs while they are executing. They are simpler to use than the {stream} notation on streaming inputs and outputs.

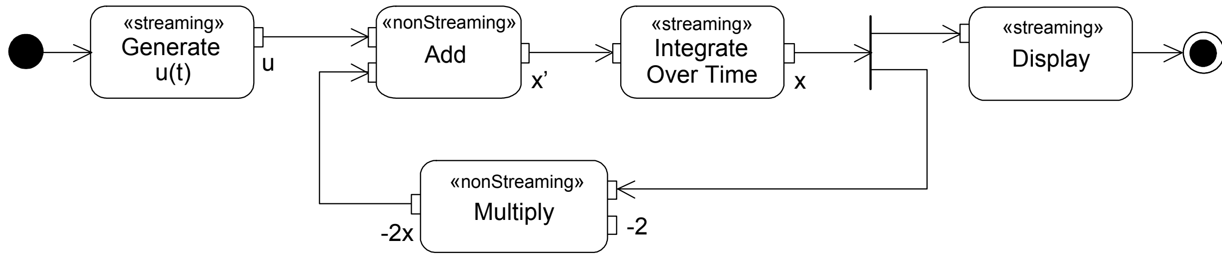


Figure C-4. Example activity with «streaming» and «nonStreaming» stereotypes applied to subactivities.

C.2 Requirements

C.2.1 Overview

This section describes a non-normative extension to support trade studies, which are an essential aspect of any systems engineering effort. In particular, a trade study is used to evaluate a set of alternatives based on a defined set of criteria. Each criteria may have a weighting to reflect its relative importance and a score based on the alternative under investigation. These criteria are known as Measures of Effectivenss.

A Measure of Effectiveness (MoE) states an optimization condition that a system must satisfy. Whereas the requirements for a system define the domain of the solution, the solution space, the Measures of Effectiveness drive the solution to a particular region in that space.

The measures of effectiveness are tightly related to stakeholder needs. For example: the requirements differences between a PC and a laptop are largely in the laptop optimization conditions for minimum weight, minimum thickness, and maximum battery life. These critiera are some of those that customers (one of the kinds of stakeholder) consider in deciding what to purchase.

A MoE is a stereotype of UML::Classes::Kernel::Class. Composite MoEs can be created by using the composition association. The interpretation of a composite MoE is that its score will be determined based on the aggregate of the product of the score and weight of its component MoEs. This rule is applied recursively at each level of decomposition to arrive at an overall value for the alternative being investigated.

Scores are only entered (vs computed according to the above rule) for the leaves, or tips of the MoE hierarchy, based on assessment of the associated critical performance parameter for the alternative under investigation. Parametric constraints are used to related the critical performance parameters to the MoEs. These scores are normlized using a value function, also called a utility curve, that maps a raw score, for example battery life in hours, to a value betwee 0 and 100 so that the aggregation can be performed. (It wouldn't make sense to add battery life in hours to laptop weight in kilograms, for example). Again, parametric constraints may be used to specify the utility curve used for normalization.

See Appendix B for a detailed example of performing trade studies using MoEs and parametric constraints.

There are also a number of predefined enumeration types in the model library SysML Types associated with Requirements. These are used to classify requirements, capture the risk associated with a requirement, and specify the verification method associated with the requirement. See Section C.4.7 for a description of the predefined enumerations that support Requirements.

C.2.2 Diagram elements

Table 2. *Graphical nodes included in effectiveness element*

NODE NAME	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Measure of Effectiveness	«effectiveness» MOEName score : Real weight : Real = 100 «effectiveness» id = "MyMOEIdentifier" text = "MOE description" optimizationDirection = maximize	**SysML::«effectiveness»**	**Non-normative**

Table 3. *Graphical paths for effectiveness.*

PATH TYPE	CONCRETE SYNTAX	ABSTRACT SYNTAX REFERENCE	COMPLIANCE
Derive	**«derive»** - - - - - - - - - - - ->	**SysML::Requirements**	**Basic**

C.2.3 Package Structure

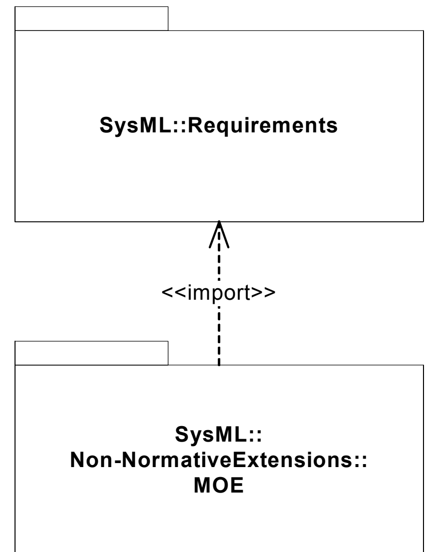


Figure C-5. Package Structure for SysML Requirements

C.2.4 UML Extensions

Abstract Syntax

Package Requirements

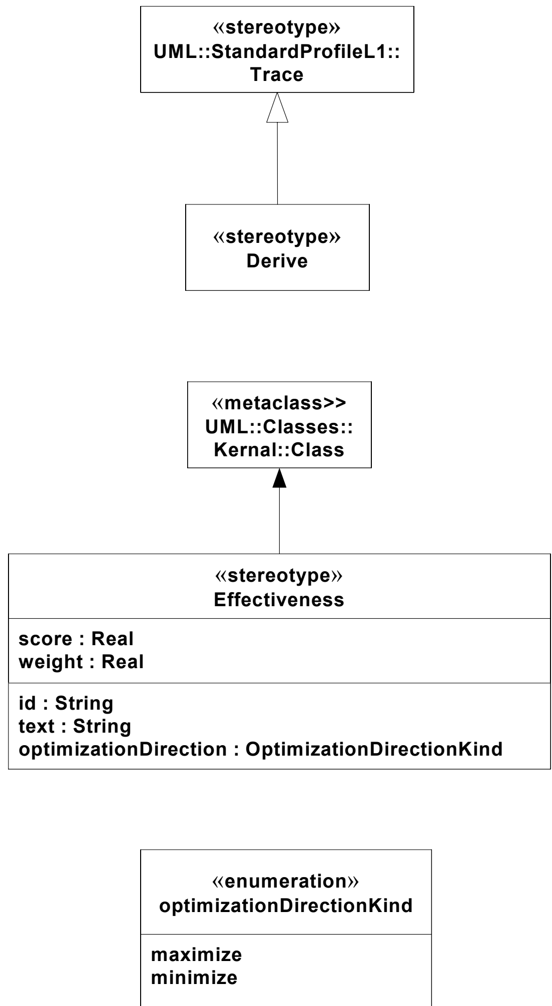


Figure C-6. Abstract Syntax for Effectiveness Metamodel.

C.2.4.1 Stereotypes

Derive

No change except for constraint 1 for requirement is relaxed to read:

[1] The source element must be an element stereotyped by «requirement» or «effectiveness».

Measure of Effectiveness

Description

A Measure of Effectiveness (MoE) states an optimization condition that a system must satisfy. Whereas the requirements for a system define the domain of the solution, the solution space, the Measures of Effectiveness drive the solution to a particular region in that space. Each MoE has a weight attribute to reflect its relative importance and a score attribute to capture its value based on the alternative under investigation.

A MoE is a stereotype of UML::Classes::Kernel::Class. Composite MoEs can be created by using the composition association. The interpretation of a composite MoE is that its score will be determined based on the aggregate of the product of the score and weight of its component MoEs. This rule is applied recursively at each level of decomposition to arrive at an overall value for the alternative being investigated.

Attributes

id	: String	The identifier of the effectiveness.
text	: String	The textual description or a reference to the textual description of the measure of effectiveness (MoE).
optimizatonDirection	: OptimizationDirectionKind	The indicator as to the direction of the optimization.

Instance Attributes

score	: Real	The normalized score for the design alternative being investigated. The normalization of the raw scores for a given design alternative is done using utility curves (sometimes called value functions, since they yield a resulting value for a given raw input).
weight	: Real	The relative importance of the MoE. The sum of the weights of all MoEs at any given level in the hierarchy must sum to 100.

Constraints

[1] The property *isAbstract* must be set to *false*.

[2] The property *ownedOperation* must be empty.

[3] Classes stereotyped by «effectiveness» may not participate in associations except for composite associations with other classes stereotyped by «effectiveness»

[4] A component (part) of a class stereotyped by «effectiveness» must also be a Measue of Effectiveness (MoE).

[5] The subtypes of a class stereotyped by «effectiveness» must also be stereotyped by «effectiveness».

[6] A class stereotyped by «effectiveness» can participate in a «trace» dependency only if the other end of the dependency is not stereotyped by «effectiveness».

[7] The sum of the values of the weight attribute for all component (parts) of a composite class stereotyped by «effectiness» must be 100.

[8] The value of the score attribute of a class stereotyped by «effectiness» must be in the range [0..100]

OptimizationDirectionKind (pre-defined enumeration)

Description

This pre-defined enumeration specifies whether the Measure of Effectiveness should be maximized or minimized.

Enumration Literals

maximize	maximize indicates that the objective is to maximize the MoE.
minimize	minimize indicates that the objective is to minimize the MoE.

C.2.4.2 Diagram Extensions

None

C.2.5 Compliance Level

Non-normative

C.2.6 Usage Example

Figure C-7 shows a hierarchy of measures of effectiveness defined to support a trade study for the Hybrid SUV. The Overall «effectiveness» is the aggregates the overall score for the alternative being investigated. The Acceleration, FuelEcononmy, Emissions and Range «effectiveness» are derived from the critical top level User Requirements of the same name.

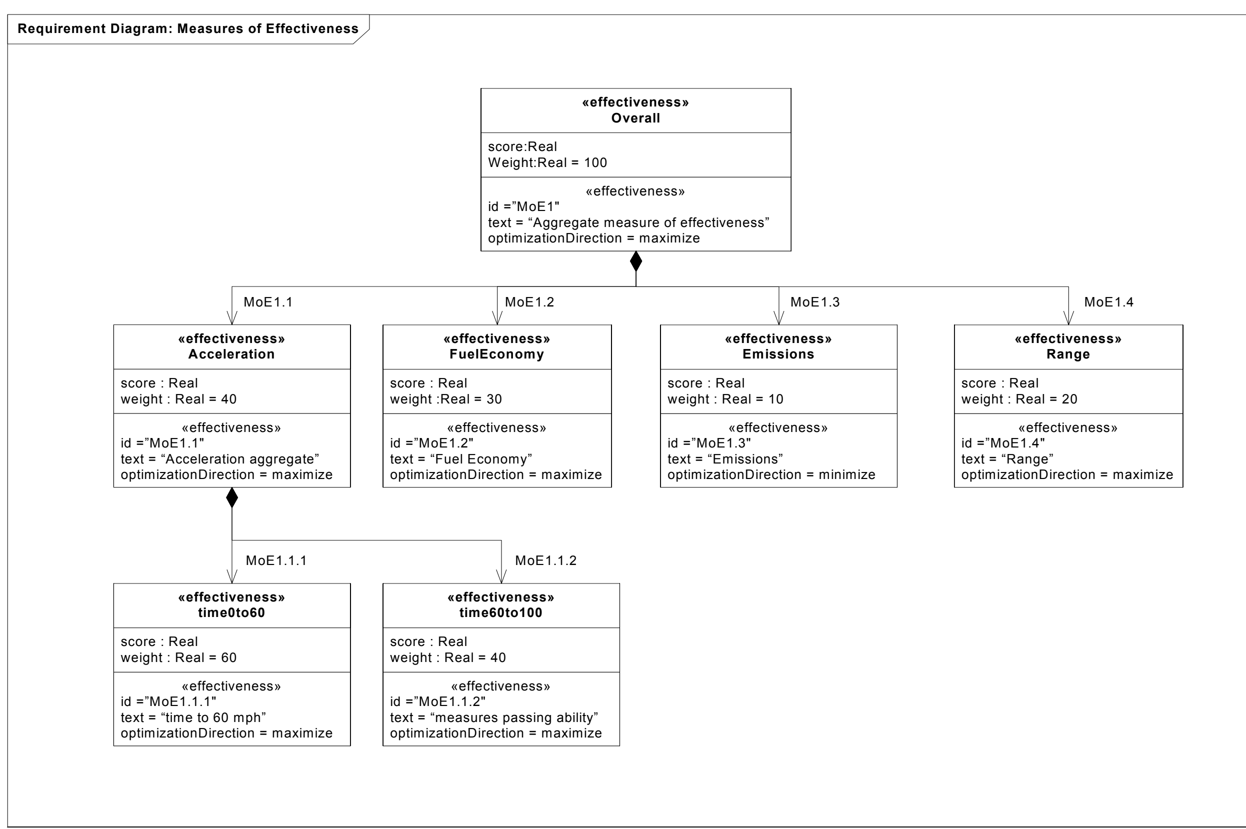


Figure C-7. Measures of Effectiveness.

Appendix D. Non-Normative Model Library

This appendix describes a non-normative model library for SysML that are being considered for standardization, but at this time are non-normative (i.e, they are not part of the official SysML standard). Users and tool vendors are encouraged to experiment with this model library as they set fit, and provide feedback to the SysML specification team (mailto:feedback@SysM-Lorg) regarding its usefulness.

These pre-defined ValueTypes and Default Enumerations are used in the Sample Problem thourghout the specification as well as in Appendix B.

D.1 Pre-defined ValueTypes

This package of the model library provides a number of predefined ValueTypes for use in modeling systems. All pre-defined ValueTypes are Real numbers (i.e. they specialize the SysML pre-defined primitive Real). Real is itself defined in in SysML (see Chapter XXX Types). A ValueType is an extension of UML::Classes::Kernel::PrimitiveType and had stereotype properties that specify the physical quantity represented, the unit of measure and the dimension. See the Auxiliary Constructs chapter for the definition of the ValueType extension.

Table 1 lists the predefined ValueTypes.

Table 1 Predefined ValueTypes

Name of ValueType	Quantity Represented	Unit	Dimension
MeterPerSec2	Acceleration	m/s^2	L/T^2
FeetPerSec2	Acceleration	ft/s^2	L/T^2
Radians	Angle	radian	NA
Degrees	Angle	degrees	NA
RadPerSec2	Angular Acceleration	radian/s^2	1/T^2
RadPerSec	Angular Frequency	radian/s	1/T
SquareMeters	Area	m^2	L^2
SquareFeet	Area	ft^2	L^2
Farads	Capacitance	F	Q^2T^2/ML^2
Coulombs	Charge	C	Q
Amperes	Current	A	Q/T
KgPerMeter3	Density	kg/m^3	M/L^3
Meters	Length	m	L
Feet	Length	ft	L
Joules	Energy, Work, Heat	J	ML^2/T^2
Calories	Energy, Work, Heat	cal	ML^2/T^2
BTU	Energy, Work, Heat	btu	ML^2/T^2
ElectronVolts	Energy, Work, Heat	eV	ML^2/T^2
Newtons	Force	N	ML/T^2
Pounds	Force	lb	ML/T^2
Hertz	Frequency	Hz	1/T
Henrys	Inductance	H	ML^2/Q^2

Name of ValueType	Quantity Represented	Unit	Dimension
Kilograms	Mass	kg	M
Slug	Mass	slug	M
Watts	Power	watt	ML^2/T^3
HP	Power	hp	ML^2/T^3
Bars	Pressure	bar	M/LT^2
Atmospheres	Pressure	atm	M/LT^2
Ohms	Resistance	ohm	ML^2/Q^2T
Kelvin	Temperature	K	K
DegC	Temperature	degC	K
DegF	Temperature	degF	K
Seconds	Time, Period	s	T
NewtonMeters	Torque	N*m	ML^2/T^2
FootPounds	Torque	ft*lb	ML^2/T^2
KPH	Speed	km/h	L/T
MPH	Speed	mi/h	L/T
CubicMeters	Volume	m^3	L^3
CubicFeet	Volume	ft^3	L^3

D.2 Pre-defined Enumeration Literals

This package of the model library specializes the enumerations defined in the package SysML::Types to provide a set of default enumerations literals. The base enumerations defined in the package SysML::Types and are used by SysML for classifying risk, requirement types, verification method, verification verdict and controlvalue. The definition of the base enumerations do not have any enumeration literals defined.

This approach of defining base enumerations and specializing them in this non-normative library is done so that users can add their own literals and thus customize SyML to meet their needs. See the Types chapter for the definition of the associated Enuneration Types (which are normative).

Table 2 lists the subtypes of the normative Enumeration literals for each of the enumerations defined in SysML::Types and the associated default literals.

Table 2 Pre-defined Enumeration Literals

Base Enumeration in SysML::Types	Specilized Enumeration	Default Enumeration Literal	Example Description
ControlValue	DefaultControlValue	Enable	The *Enable* literal means to start a new execution of a behavior
		Disable	The *Disable* literal means a termination of an executing behavior that can only be started again from the beginning

Base Enumeration in SysML::Types	Specilized Enumeration	Default Enumeration Literal	Example Description
RequirementKind	DefaultRequirementKind	Functional	A *Functional* requirement specifies what the item must *do*
		Performance	For a given function, a *Performance* requirement states *how well* that function is to be performed based on a specific metric of measure (including units), which can considered a Key Performance Paramter.
		Interface	An *Interface* requirement specifies that the required characteristics at a point or region of connection of the item to the outside world (i.e., location, geometry, inputs and outputs by name and specification, allocation of signals to pins, etc).
RiskKind	DefaultRiskKind	High	*High* indicates an unacceptable level of risk
		Medium	*Medium* indicates an acceptable level of risk
		Low	*Low* indicates a minimal level of risk or no risk
Verdict	DefaultVerdict	Pass	*Pass* indicates that the test behavior gives evidence for correctness of the SUT for that specific test case.
		Fail	*Fail* indicates that the purpose of the test case has been violated.
		Inconclusive	*Inconclusive* is used for cases where neither a Pass nor a Fail can be given.
		Error	*Error* indicates errors (exceptions) within the test system itself.
VerificationMethodKind	DefaultVerification-MethodKind	Analysis	*Analysis* indicates that verification will be performed by technical evaluation using mathematical representations, charts, graphs, circuit diagrams, data reduction, or representative data. Analysis also includes the verification of requirements under conditions, which are simulated or modeled; where the results are derived from the analysis of the results produced by the model.
		Demonstration	*Demonstration* indicates that verification will be performed by operation, movement or adjustment of the item under specific conditions to perform the design functions without recording of quantitative data. Demonstration is typically considered the least restrictive of the verification types.

Base Enumeration in SysML::Types	Specilized Enumeration	Default Enumeration Literal	Example Description
		Inspection	*Inspection* indicates that verification will be performed by examination of the item, reviewing descriptive documentation, and comparing the appropriate characteristics with a predetermined standard to determine conformance to requirements without the use of special laboratory equipment or procedures.
		Test	*Test* indicates that verification will be performed through systematic exercising of the applicable item under appropriate conditions with instrumentation to measure required parameters and the collection, analysis, and evaluation of quantitative data to show that measured parameters equal or exceed specified requirements.

Appendix E. OMG XMI Model Interchange

E.1 Overview

The XML Metadata Interchange (XMI) file for the SysML v. 1.0a Profile model, which is based on the UML v. 2.0 metamodel, is currently only available in XMI v. 1.1 format. Both the SysML v. 1.0a Profile model, and its accompanying XMI file are support documents for this specification. See Section 5.1, "Support Documents," on page 8 for support document availability.

We plan to provide an XMI file in XMI v. 2.1 format, which is based on the MOF 2.0 meta-metamodel and supports UML 2.0 model interchange, in a support document that will accompany a future revision of this specification.

Appendix F. ISO AP233 Model Interchange

F.1 Overview

Work is currently underway to provide ISO AP233 model interchange support for the SysML specification. We anticipate that the *Abstract Syntax for SysML v. 1.0a* model and XMI file, which are support documents for this specification, will assist the development of ISO AP233 model interchange support for SysML. See Section 5.1, "Support Documents," on page 8 for support document availability.

The following sections describe the technical approach for interchanging SysML models using the ISO AP233 data exchange protocol.

F.2 Background

AP233 is a data exchange protocol for systems engineering based on STEP (ISO 10303) Product Data Representation and Exchange standardization initiative. STEP is designed to provide a data interchange schema based on a tool-independent metamodel.[1] AP233 is intended to support the whole system development life cycle, ranging from requirements definition to system verification and validation.

Within projects the system engineering activities tie together the different domain engineering disciplines with one consistent system view. The same applies to the systems engineering data that forms the core of a systems description and has to be linked to the remaining domain engineering data. Figure F-1 is a UML diagram that shows how AP233 is related to other protocols. AP233 makes re-use of the STEP-PDM definitions, indicated by the «use» dependency to STEP-PDM. The remaining packages represent other STEP application protocols. The dependencies can be read as "AP233 depends on the definition of …", which means if a protocol will be changed, then this means that AP233 may also be affected.

The following areas are covered in the AP233 data model:

- Requirements
- Functional architecture
- Physical architecture
- Verification/Validation
- Management
- Supporting Modules as work, person, properties

1. For more information about STEP see http://www.tc184-sc4.org/SC4_Open/SC4_Work_Products_Documents/STEP_(10303).

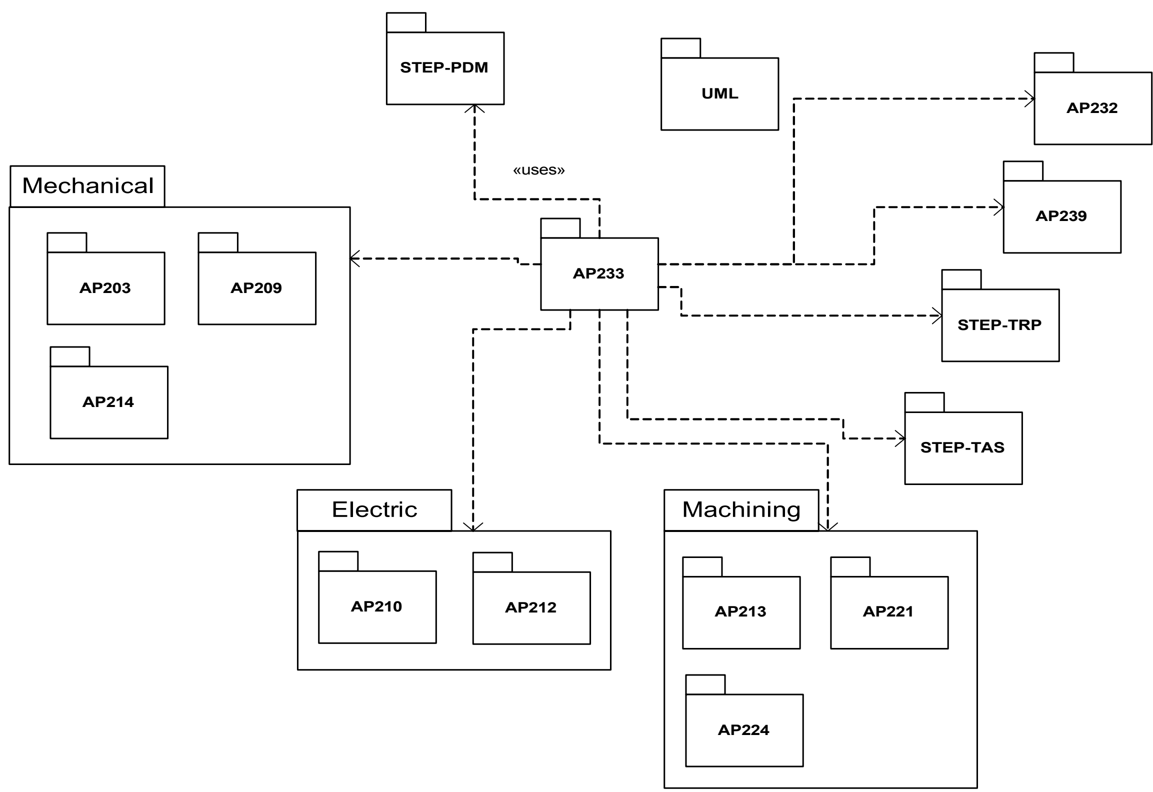


Figure F-1. AP233 and related protocols

Figure F-2 shows that the basic item in AP233 is a product. Each package makes re-use of the definition of a product (in the PDM sense). A requirement in the requirement package is derived from Product (in the package Product) and inherits the properties of Product.

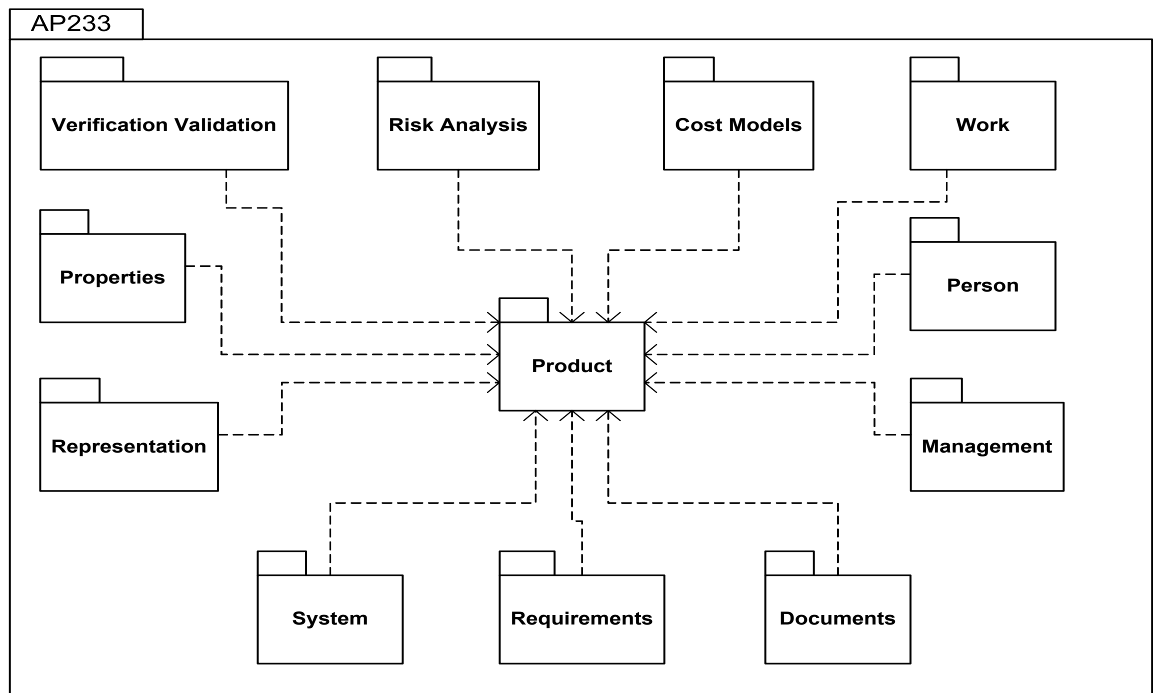


Figure F-2. AP233 Toplevel Architecture

AP233 is a follow-on activity of the European SEDRES (System Engineering and Exchange Standardisation) project which developed systems engineering data model based on the STEP technology. AP233 was launched based on the SEDRES results and started with a modularization of the data model to ease re-use of parts of other protocols.

The current status of AP233 is the following:

- Requirements module implemented
  - Text-based requirements
  - Property-based requirements
  - Basic structure module
  - Tracing between structure and requirements
- AP233 Demonstrator Tool: In order to facilitate understanding, demonstration and utilization of AP233 a demonstrator tool is being developed. It implements basic features for the definition of requirements (in different appearances like text, property and spread-sheet), a system break-down and traceability between requirements and systems. In addition, it has multiple interfaces to read and write the data, not only in STEP and XML but also interfaces to the Office world such as Work and Visio.
- Next Steps:
  - Structural Module
  - Behavioral Module
  - Risk Module

- Scheduling Module
- Rules Module
- Cost Module

The major stakeholder for the AP233 development is the Incose/MDSD (Model-Driven System Design).

F.3 Approach

From a systems engineering perspective, future SysML tools are just a subset of tools which are used throughout the life cycle for system development and maintainance. The challenge for any tool integration activity is to provide a mapping between the different meta models which are used to capture the tools data. As ISO10303 STEP and in particular AP233 (for systems engineering) provide a neutral data repository for tool integration, the SysML meta model has to be mapped to AP233.

In order to decouple the different meta-models from AP233 and SysML it has been decided to define a mapping model. This mapping model than is used to map the correspondent elements of AP233 and SysML to it. The mapping model is a high-level (independent) representation of the systems engineering concepts implemented in SysML and AP233. The mapping model is defined in UML.

The AP233 modeling is done using the STEP modeling language Express. Basically Express implements similar concepts as UML: classes(entities), attributes, associations and inheritance. In addition to that Express has some data modeling related modeling elemens currently not implemented in UML. But in order to have a common mapping platform it has been decided to perform the mapping in UML. For this it is necessary to convert the AP233 model to UML. In order to achieve common semantics for the AP233 model in UML a dedicated profile has been developed.

shows the relationship between the different models. At the bottom the mapping model can be seen which is used to bridge the AP233 and SysML meta models. On the right hand side the AP233 model in UML is an instance of a UML profile, to capture and preserve Express semantics. The left hand side shows SysML as an extension of UML2. Although UML1.x is sufficient to specify an Express profile, it will be eventually replaced with UML2.

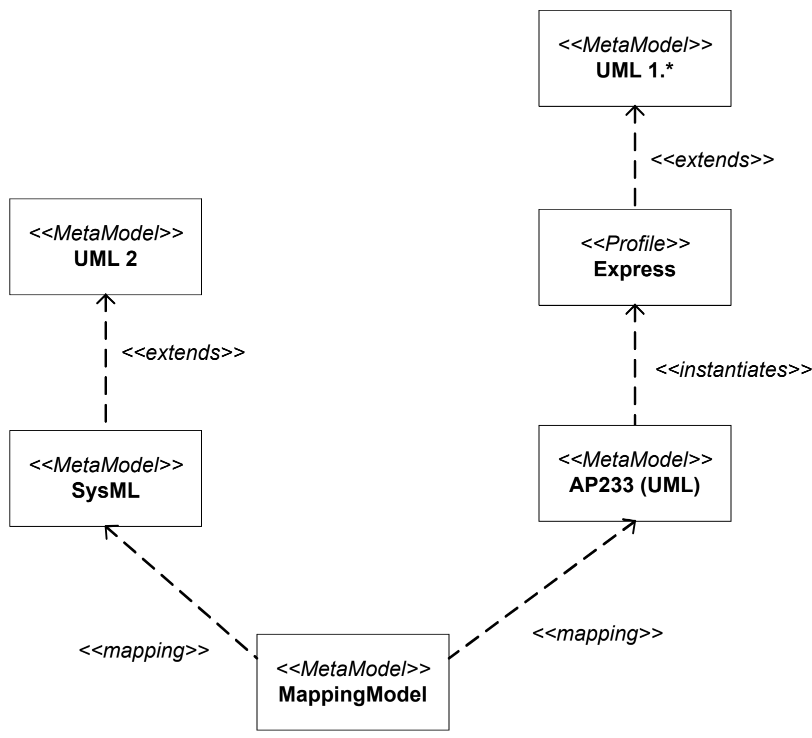


Figure F-3. Models in use for the SysML to AP233 alignment

F.3.1 Capturing Express models in UML

The development of the UML profile for Express is shown in Figure F-3 . In the diagram shows only the 'core' subset of the modeling language Express. The basic element is an entity which can be compared to a class in UML. It may have attributes, attributes and associations. The attribut definition is similar to UML. Attributes are described with a name, type and multiplicity. Attributes may be optional. The 'SubTypeOf' class defines the inheritance relationship in Express.

Express has the concept of a type which can be either an enumeration, a basic data type (real, number, string,..), a select statement or a container class type (set, list). The select statement can be seen as a 'one of' relationship (e.g. a person can drive either a car or a bike at a given time).

Derived from this conceptual model is the UML profile. This is a manual step. The relationship between can be seen as the problem-model, the UML profile is an implementation of this problem in a UML tool using the standardized UML extensibility mechanisms. The UML profile for the Express modeling language looks as the following:

Appendix F-5shows the Express profile for UML manually derived from the Express meta model.

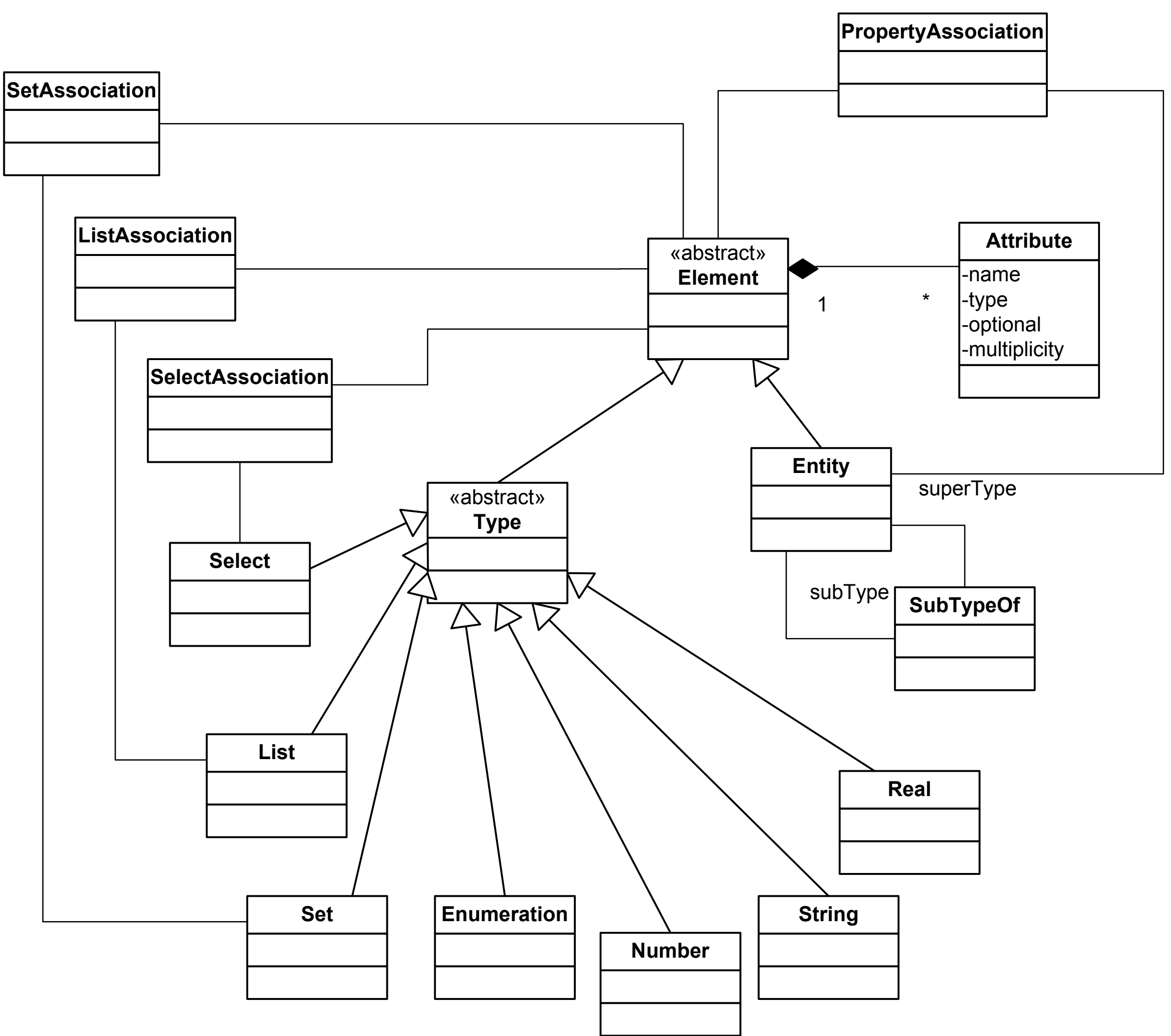


Figure F-4. Excerpt of the Express meta model

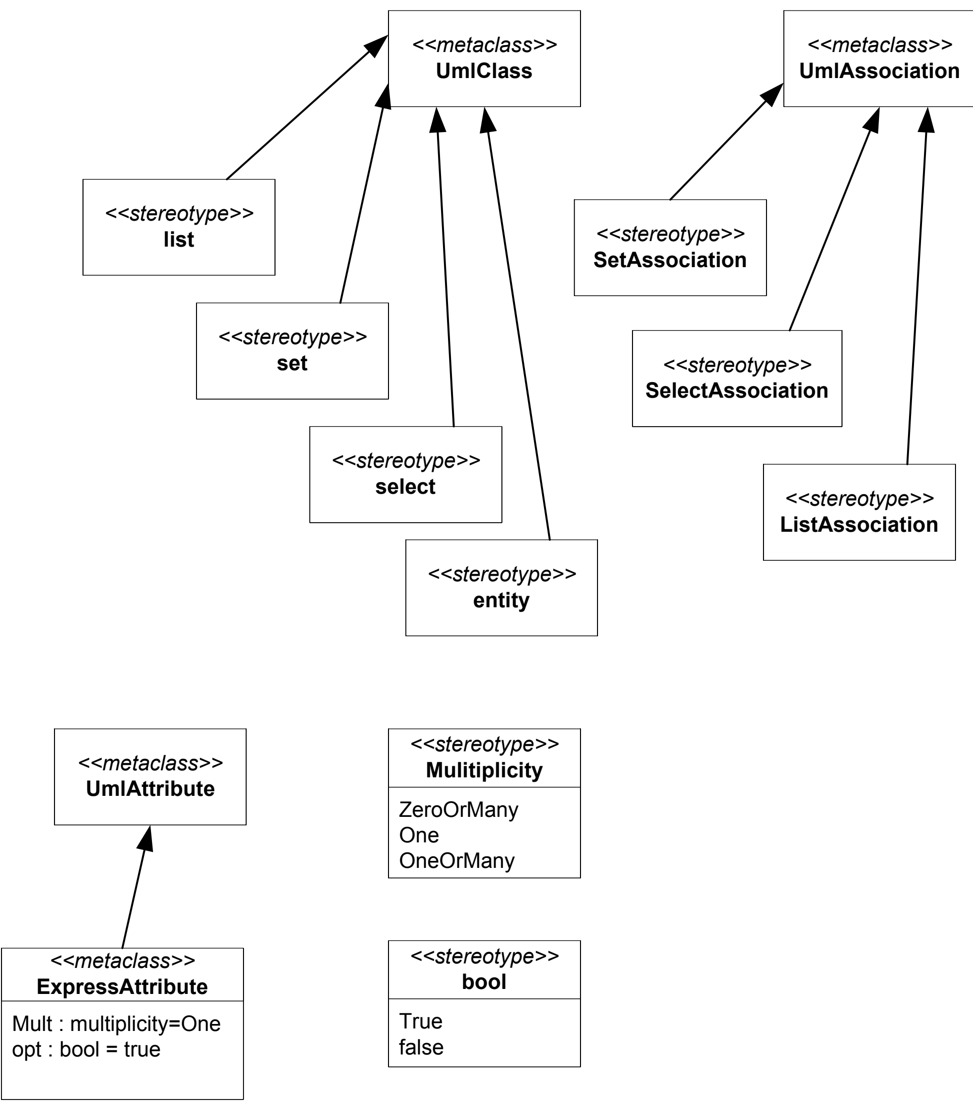


Figure F-5. UML Profile for Express

F.3.2 Converting Express models to UML

The AP233 UML model will be automatically derived from the AP233 Express model. The process is shown in Figure F-6. Input for the conversion process is the AP233 express model. This will be parsed be a dedicated express parser which takes the express model an produces an informal XML model capturing the express model. This can be used by a XSLT process which takes the XML file and generates a XMI file based on the Express profile for UML.

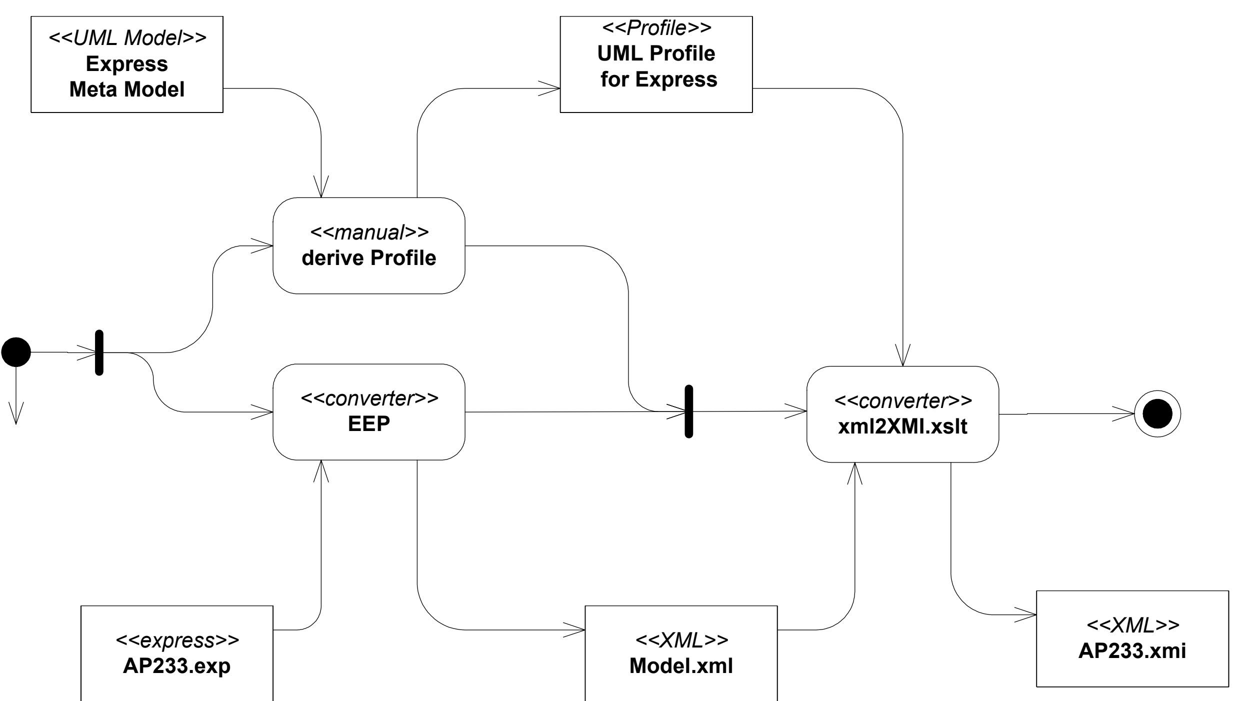


Figure F-6. Activities to derive the AP233 UML model from the Express model

F.4 Model Alignment

F.4.1 SysML Requirements Model

In order to demonstrate the model alignment the following shows how a requirement module is described in the another section of this specification. It shows the basic elements used to describe requirements and relate them to other modeling elements such as functions, components or test caes. The fundamental element is a requirement with the according parameters (*Requirement*) . A test case (*testCase*) is used to demonstrate the success of the implementation of a given requirement. To check the completeness of the design the requirement is linked to a UML element which satisfies the requirement (*RequirementSatisfaction*). As this model is to define the modeling language it makes reuse of the UML provided modeling elements. For example, the RequirementVerification is a UML dependency link.

The main purpose to recall the requirements module of SysML here is to explain the differences of the different models to justify the approach.

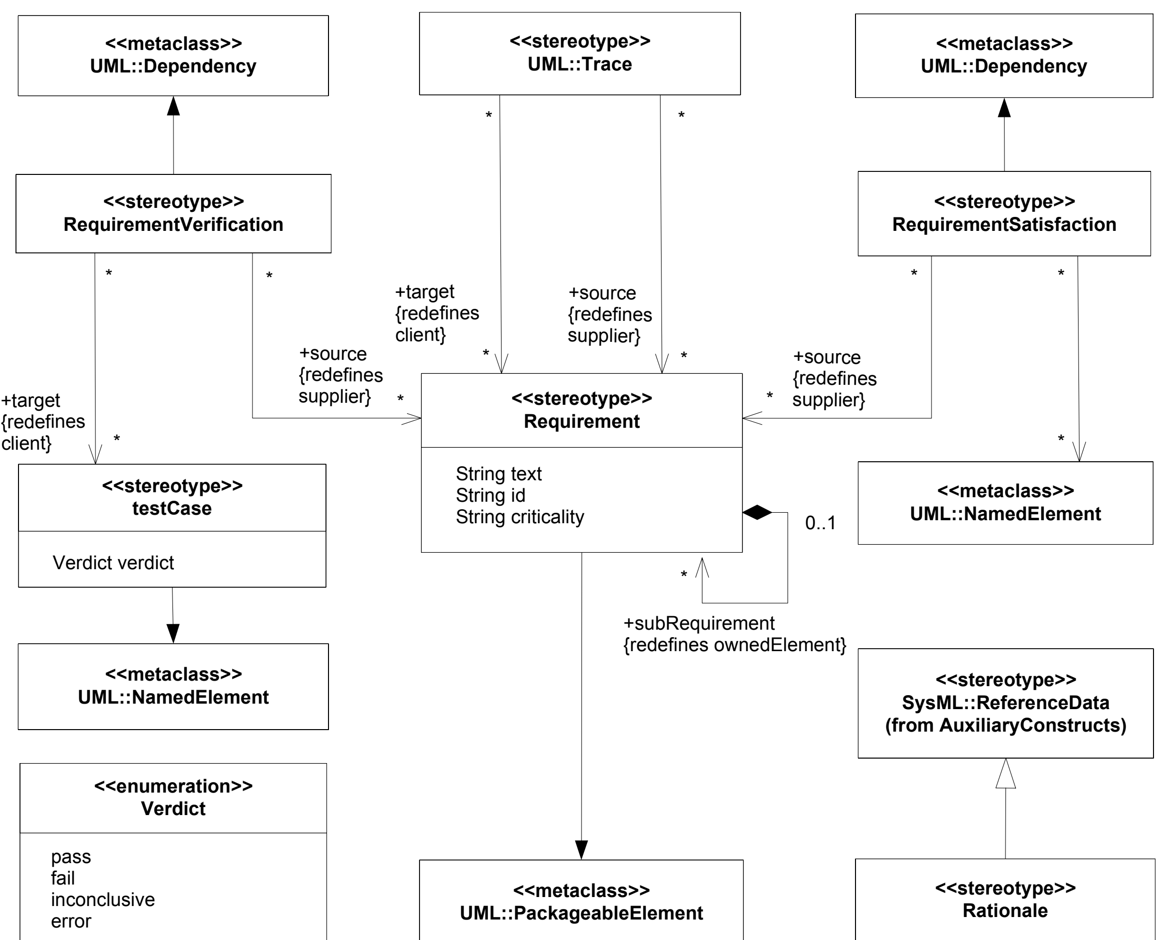


Figure F-7. SysML requirements model

F.4.2 AP233 Requirements Model

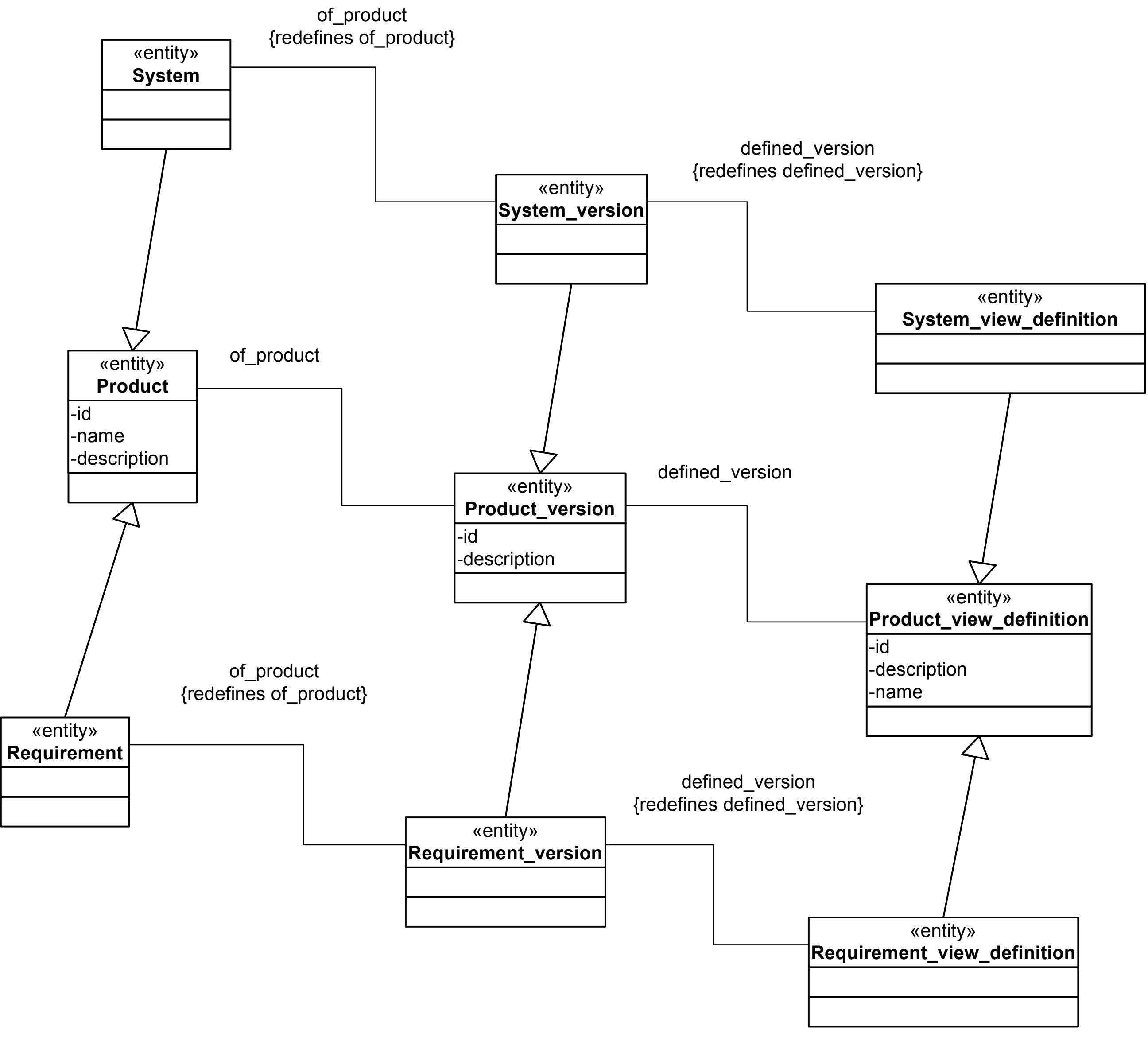


Figure F-8. Basic pattern for AP233

This chapter explains in brief the model of AP233 with a focus on the requirements module and the representation of requirements. The main purpose is to explain the AP233 model and to point out the differences between SysML and AP233 models. Appendix F-8 shows an excerpt of AP233, the basic PDM (Product Data Management) pattern, consisting of *Product*, *Product_version* and *Product_view_definition*. In AP233 the re-use has been done to enable the versioning of the data captured in an AP233 file and to ease the interfacing of PDM systems. In the definition of PDM everything which has to be produced is a product. Therfore, requirements, systems or documents are products.

In order to track the changes for each product along the life cycle configuration control has to be applied. The pattern *Product*, *Product_version* and *Product_view_definition* defines the following:

- *Product*: Defines the identity of a product with an unique identifier, a name and a description
- *Product_version*: Captures the different versions of a product, each version is described by an identifier and a name
- *Product_view_defintion*: Defines the different views in which the product appears, e.g. a diagram or a table

This means that every product is represented in a tree-like manner in an AP233 file.

For each product a tree-like information is given in the AP233 file: For each product an instance of *Product* is the root. Each product may appear in different versions (*Product_version*) and each version may appear and be reference from different views (*Product_view_definitoni*).

Each product which shall be captured in AP233 this pattern has to be re-used. The re-use of this pattern will be done in AP233 via inheritance. Therefore a requirement is represented by the entities *Requirement*, *Requirement_version* and *Requirement_view_definition*. All of them are derived from the according product items. This means a

- *Requirment* 'is a' *Product*
- *Requirment_version* 'is a' *Product_version*'
- *Requirement_view_definition* 'is a' *Produc_view_definition*

The definition of a system is done accordingly which is shown in the diagram. For other items such as connectors or ports this pattern has to be replicated too in the same way.

The assignment of properties to a product is shown in Appendix F-9. Proptiertes are additional descriptions or parameters which have to be attached to a product. Those properties can be for example descriptions, role or context definitions. For a SysML requirement the attributes as defined in Appendix F-7 (id, text and criticality) would be attached as property. As explained per above the *Product_view_definition* is used to attach the detailed information to the requirement.

The top of the diagram shows *Product_view_definition* and its derived classes *Requirement_view_definition* and *System_view_definition*. The *property_assigment_select* is used to attach the properties either to *Produc_view_definition* or to *Tracing_relationship*. *Element_property* describes the property being attached. The select statement *represented_item_select* contains just one choice, the *Element_property*. The class property representation describes the representation for the property attached. The class Representation gives detailed information on the representation occurrence of the property. It can be broken down hierarchically (*Represenation_relationship*) and has links to the different representation items (*Representation_item*).

The different representation items are further detailed in Appendix F-10. It shows the hierarchy of the different represenations: *Data_structure, Element_in_structure, String_representation_item*, *Binary_representation_item*, *Document_definition*, *Mathematical_representation_item*, *Property_value_with_unit*, *Descriptive_property_value* and *Plain_text_item*. It is possible to arrange the different representation items in data structures. A data structure consists of elements, an element of a data structure can be for example Binary_representation_item, Document_definition, Mathematical_representation_item, Property_value_with_unit or a Descriptive_property_value.

This represention module of AP233 described here is just a subset of the AP233 representation.

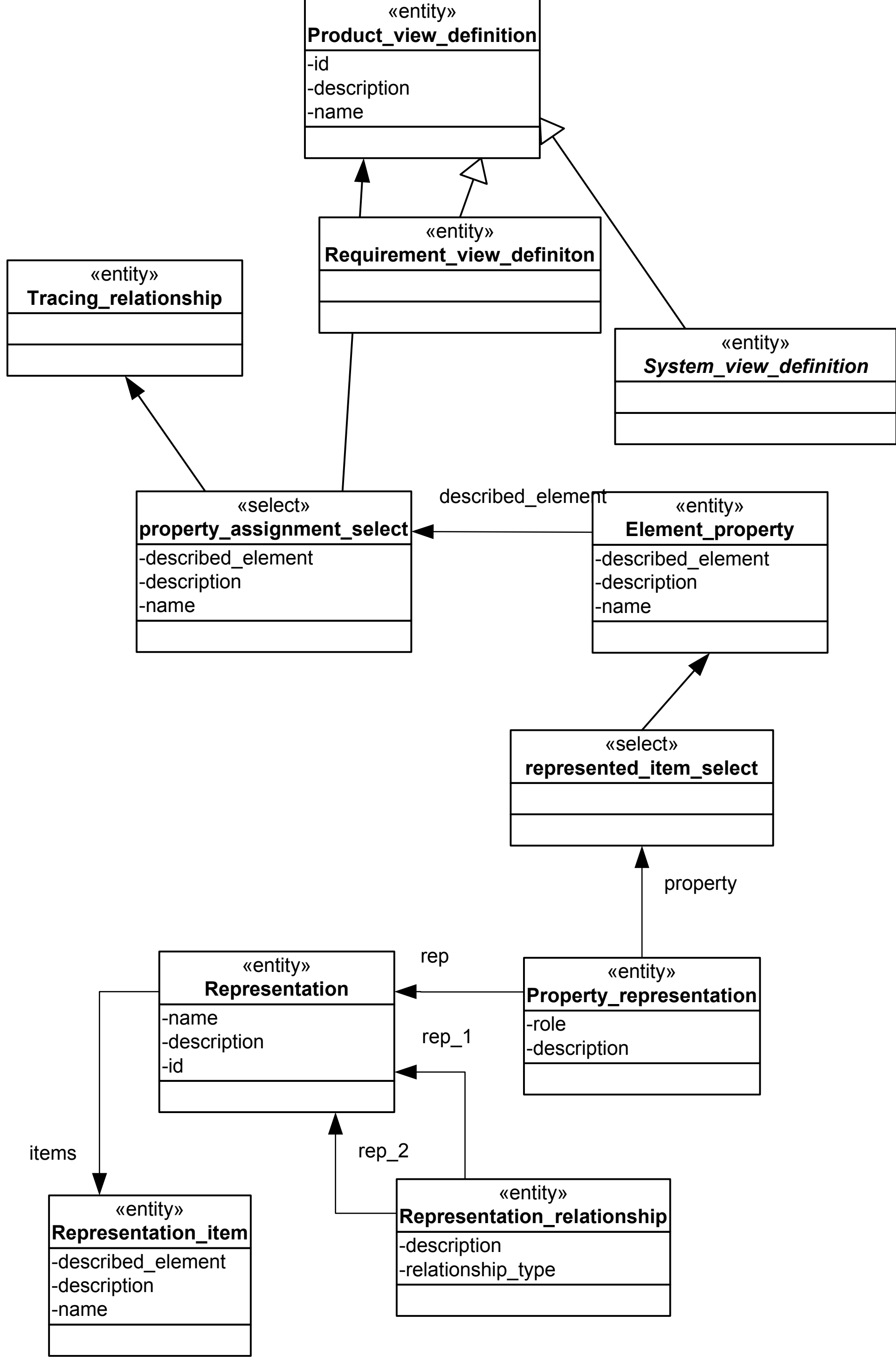


Figure F-9. Property assignment

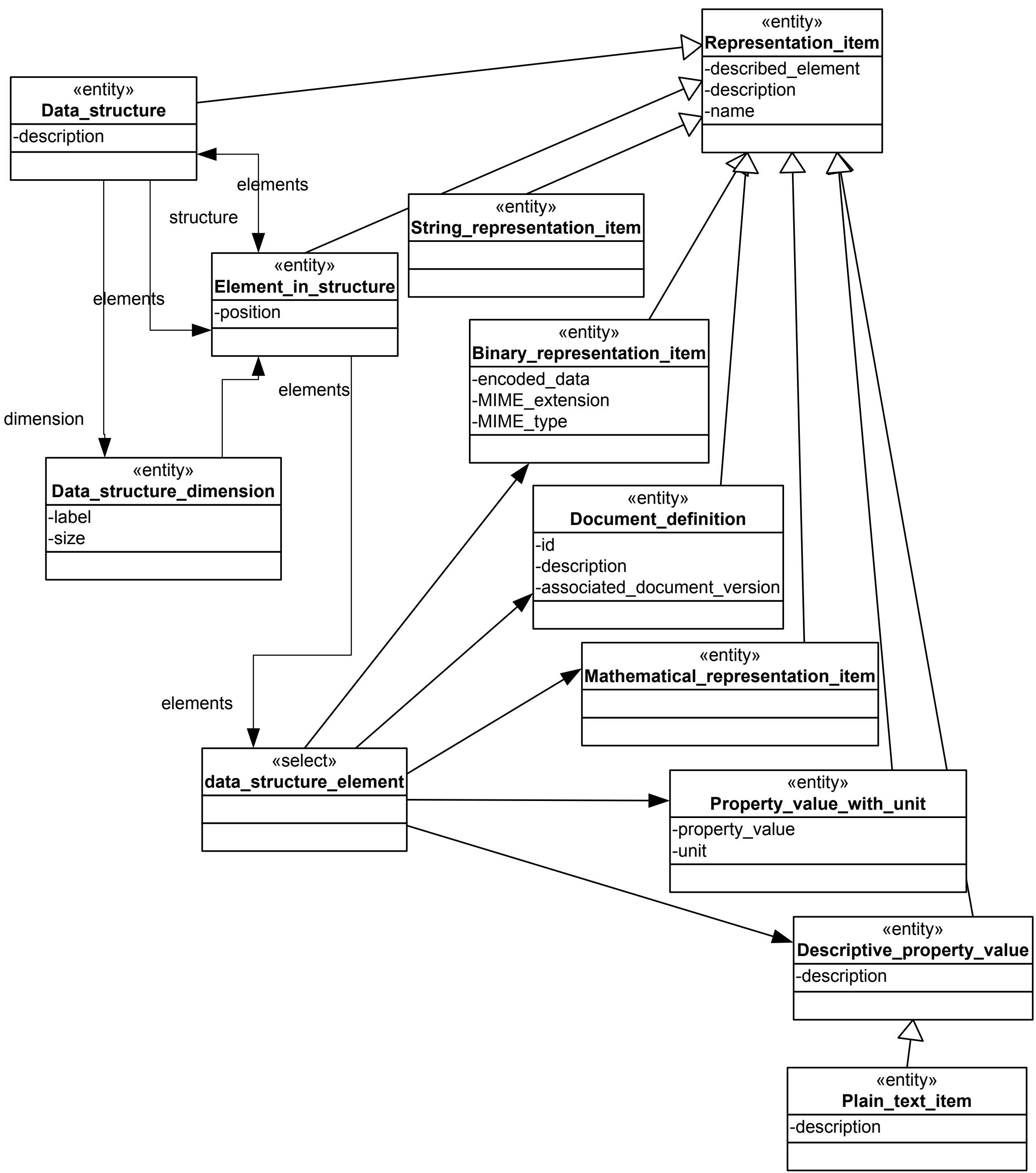


Figure F-10. Representation of properties in AP233

The hierachical breakdown structure of AP233 is shown in Appendix F-11. As explained per above the hierarchical breakdown structure will also be applied to the corresponding view definition items (e.g. *Requirement_view_definition*, *System_view_definition*, ...). The *View_definition_relationship* provides a hierarchial decomposition for *Product_view_definition* items. View_definition_relationship will be further refined for the items derived from *Product_view_definition*.

The class *Requirement_view_definition*, derived from *Product_view_definition* is used to represted the links to the requirements. As explained before it is used to attache the properties to the requirement. It is also used to define the hierarchical breakdown structure for requirements, called traceability links. The traceablility links are established by the item *Tracing_relationship*. It connects derived requirements on different hierarchical levels to each other.

System_view_definition represents items of the system hierarchy. *System_view_definitions* can be hierachically linked to each other (*System_view_definition_relationship*). Important to mention that *System_view_definition* itself is an abstract class and will be further detailled in the subsequent diagrams.

Allocations between requirements and system items are represented by *System_requirement_relationship*.

In diagram Appendix F-12 the different system breakdown hierarchies are shown. The class *System_design* represents system design items and is derived from *System_view_definition*. System design items can be seen as items on the specification (design) level. System design items can be hierarchically decomposed (*System_assembly_relationship*).

Concrete physical items are represented by System_occurrence (It can be seen as a class (*System_design*) and instance (*System_occurence*) releationship for object oriented systems). The items on design levels represent the 'shall-by' status of a system. The system occurence items do represent real existing items ('with serial numbers').

The physical assembly of a system follows the design items hierarchy, but is not necessarily the same, due to integration constraints. Therfore another breakdown is required to show how the system must be assembled, often called integration tree. Integration tree links are established by *System_occurrence_assembly* items. *System_occurrence* items have a link to *System_view_definition* items to show which part of the system they represent.

All of the items used to represent system hierarchy links (*System_occurrence_relationship*, *System_view_definition_relationship* and *System_assembly_relationship*) are derived from *View_definition_relationship*.

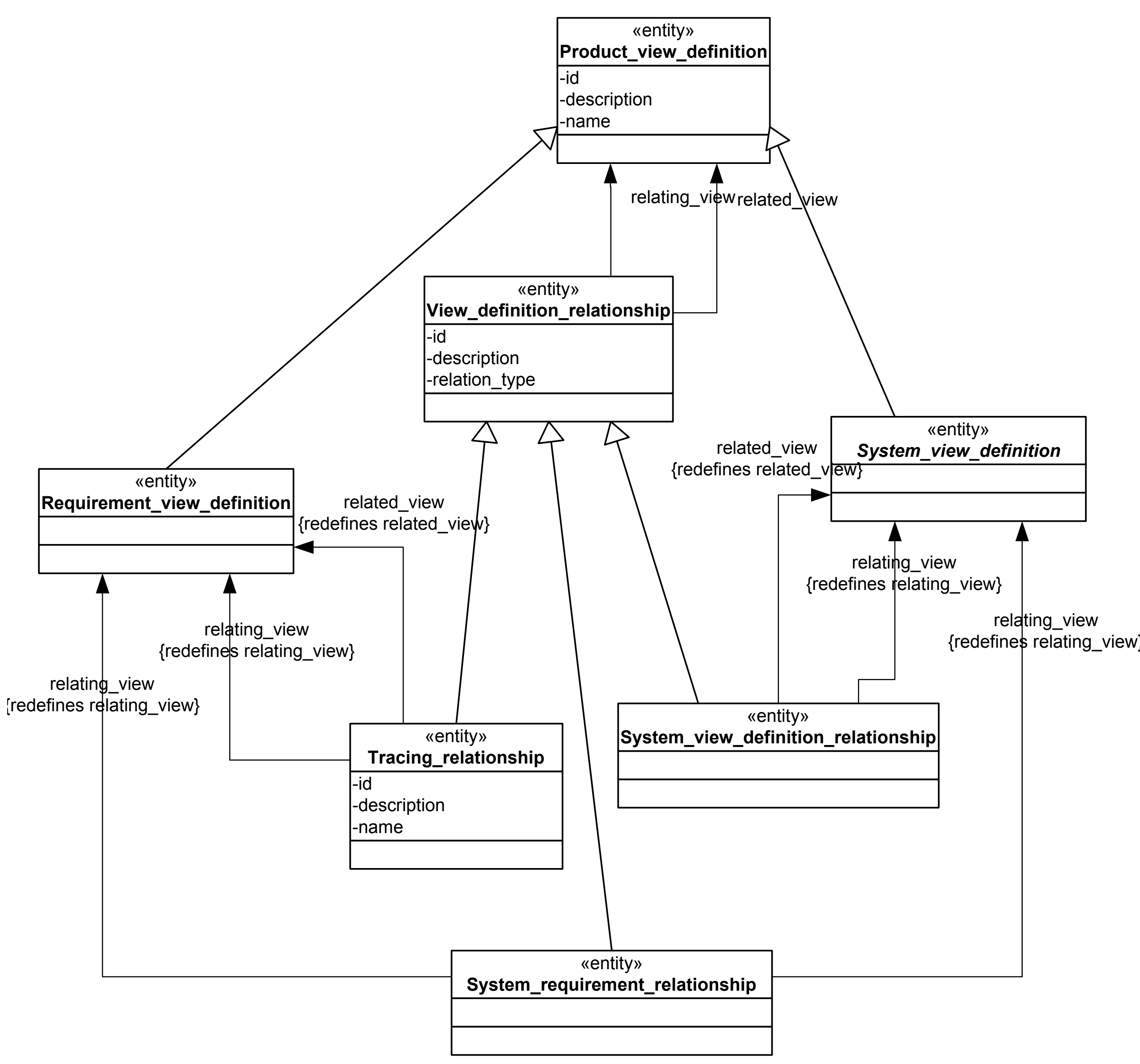


Figure F-11. Breakdown structure in AP233

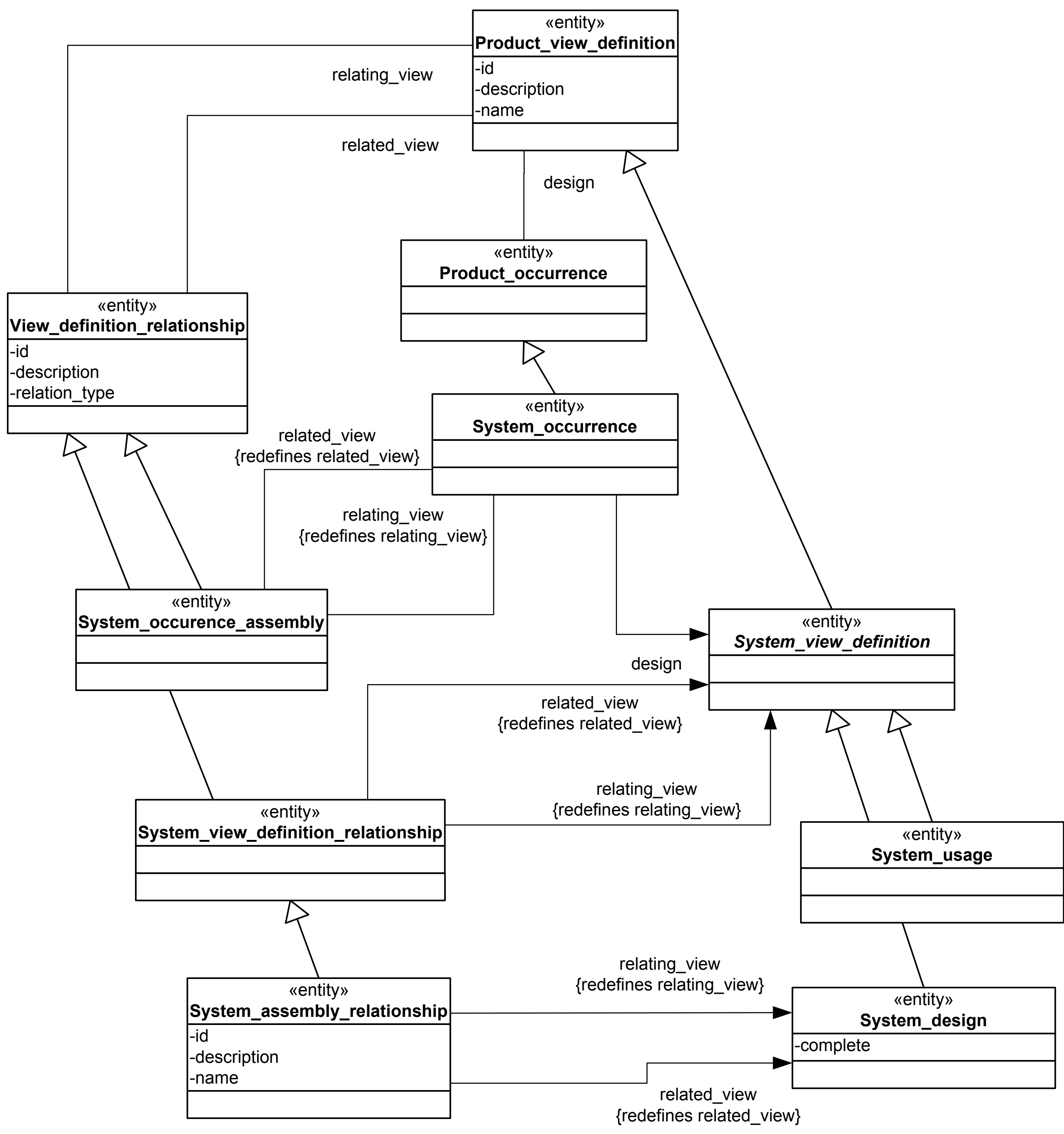


Figure F-12. System breakdown hierarchy

F.4.3 Mapping Module: Requirements

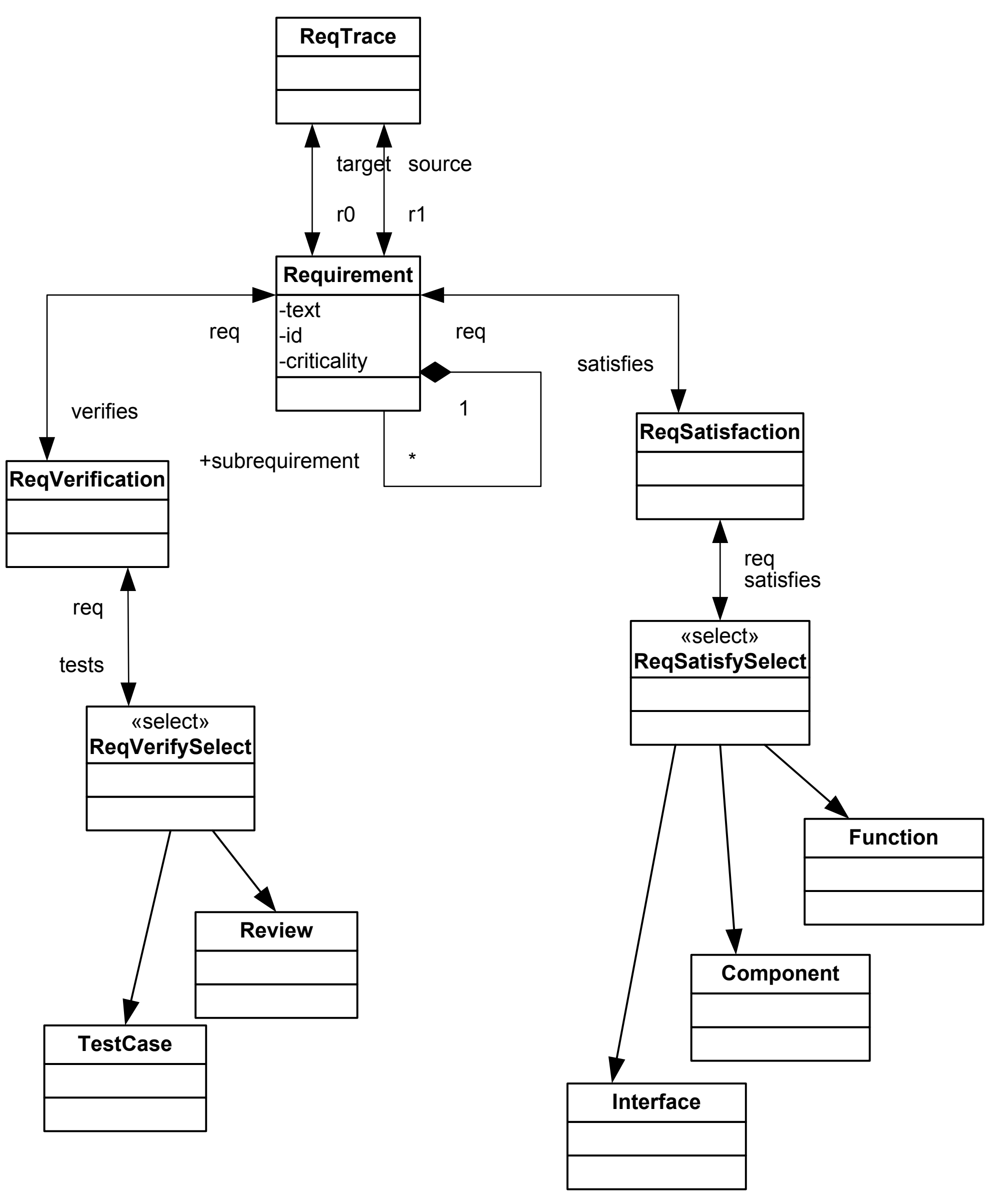


Figure F-13. Mapping Module Requirements

In Appendix F-13 the requirements part of the mapping model is shown. The main difference to the SysML and the AP233 model is the following: The mapping model doesn't address any specify data modeling items and makes not re-use of UML infra- or superstructure definition. Therfore the mapping model can be seen as an independent instance of the MOF. It serves as a kind of requirements model for system engineering conceptst.

Requirement hierarchies are indicated with *ReqTrace*, which connects requirements on different hierachical levels. A requirement can be linked to verification elements (*ReqVerification*) and system modeling elements (*ReqSatisfaction*). On the left hand side two examples for a requirement verifcation are shown (*TestCase*, *Review*). On the reight hand side different elements which can satisfy a requirements (*Function*, *Component*, *Interface*). Both, verification as well as system elements are just a deliberate selection and not exhaustive.

It is import to mention that, in order to ease the mapping, that the 'select' statement of Express has been reused. All of the different choices for example for the requirement satisfaction are listed explicitly, similiar to Express.

F.4.4 Mapping between AP233 and SysML

Table 1

#	SysML	Mapping Model	AP233
010	Requirement	Requirement	Requirement Requirement_version Requirement_view_definition
011	text	text	Plain_text_item
012	id	id	Requirement.id
013	criticality	criticality	Plain_text_item
020	Trace	ReqTrace	Tracing_relationship
021	target	r0	relating_view
022	source	r1	related_view
030	RequirementVerification	ReqVerification	
031	target	verifies	
032	source	req	
040	RequirementSatisfaction	ReqSatisfaction	System_requirement_relationship
041	target	satisfies	relating_view
042	source	req	related_view

F.5 Proof of Concept

ISO10303 STEP application protocols provide a neutral data representation which can be used to collect the data from different tools and capture them in an independent data repository. For this STEP protocols have to provide a generic superset of data items used by the tools. This and some addional concepts required for efficient data modeling STEP application protocols are considered to be 'complicated' and difficult to integration into an existing infrastructure. Therefore nowadays often short-handed XML solutions are preferred instead of integrating STEP.

In order to hide the complexity often a more abstract model (and a related API) is placed on top of STEP protocols. Those APIs do focus on user known concepts and are therefore easier to understand and integrated. But still using the power of STEP as the underlying data model. The mapping model, well mapped to AP233 is high-level model of system engineering concepts.

As both the AP233 and the mapping model are represented in UML they can be used to derive APIs applying the MDA approach to it. In order to obtain an open, robust framework two independent APIs will be generated. One on the AP233 native level and one on the mapping (conceptual) level. This approach also can be used to in cooperate other tools. In Figure F-14 the different models and APIs are shown.

The resulting API architecture is shown in the Figure F-15. It shows the different layer of abstraction from the bottom (in this case a STEP file in different representations) to the top different SE tools. The yellow layers show the two different APIs. As examples for tools we see on the top SysML tools, or UML tools which implements selected parts of SysML in terms of profiles and classic SE tools which have nothing in common (with respect to the meta-model) with UML based tools. The mapping model API provides an easy to integrate API based on the conceptual level. In order to integrate it an adaptor has to be written which converts the model (as instance of the meta model) into the representation of the mapping model. For SysML tools this converter can be written based on the mapping model. Based on the mapping between the mapping model and the AP233 model the adaptor can be automatically generated. Finally, the AP233 API supports the conversion between the model and file representations. The representation as instance of the model to the file representation.

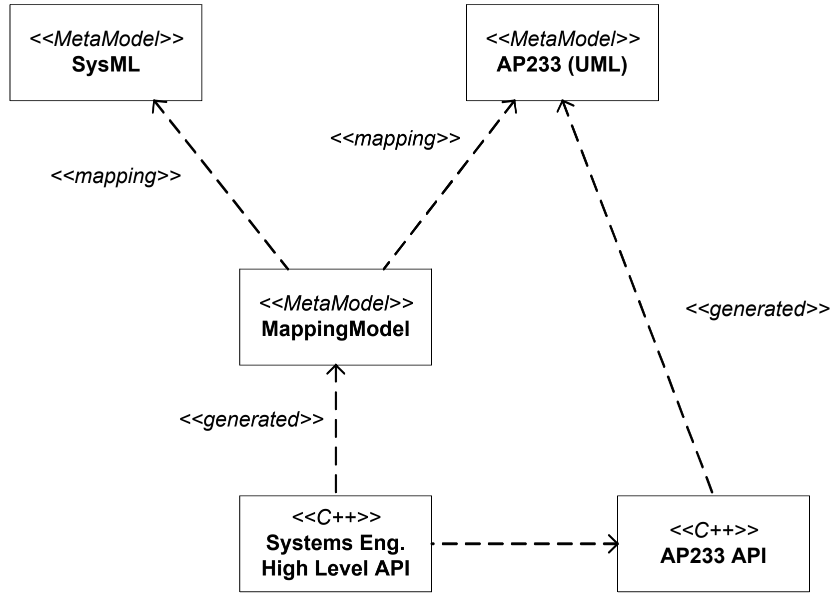


Figure F-14. Model-derived APIs

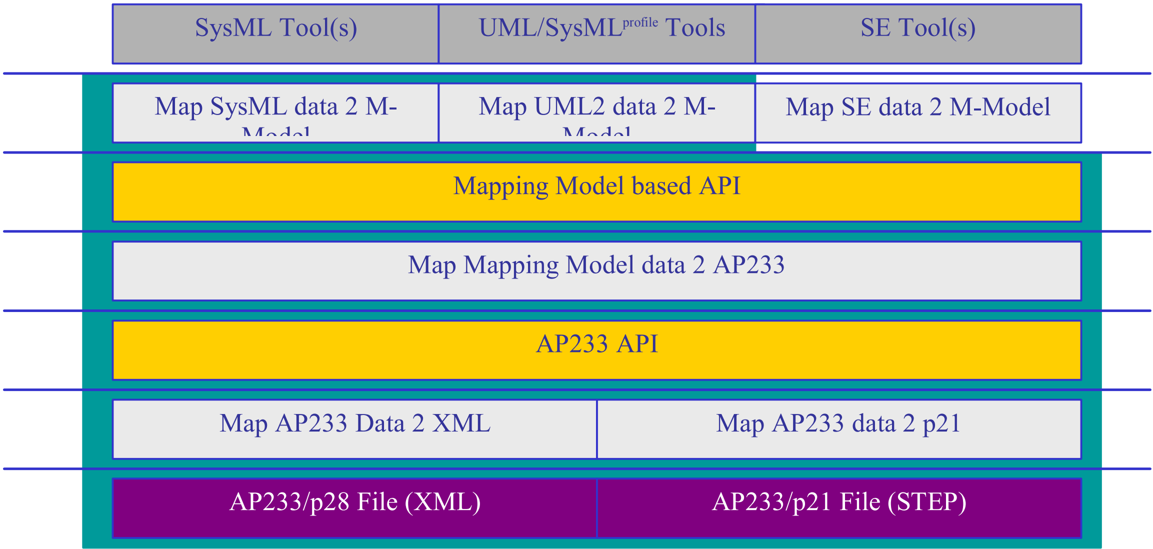


Figure F-15. API abstraction layers

Appendix G. Requirements Traceability Matrix

G.1 Overview

A Requirements Tracability Matrix (RTM) that shows how SysML satisfies the requirements of the UML for Systems Engineering RFP is available is a separate support document (*Compliance and Requirements Traceability for SysML v. 1.0a*). See Section 5.1, “Support Documents,” on page 8 for support document availability.

Index

A
Action 50, 82
Activity 54, 60
ActivityEdge 53
ActivityFinal 50
ActivityNode 50
ActivityPartition 54
Actor 88
Aggregation 31
Allocate 109
Allocated 110
Allocation 107, 108, 111
Association 31

B
Binding 40, 41
Block 29, 33, 35
BlockDefinition 29

C
Choice 82
CombinedFragment 73
Comment 128
Communication 88
Complex 122, 125
Composite 82
Composition 31, 94
Conform 116, 118
Connector 31
Constraint 128
Containment 31
Continuous 57
ControlFlow 53, 61
ControlNode 50
ControlOperator 50
ControlValue 58, 122, 123
Coregion 73
Crosscutting 91

D
DataType 29
DecisionNode 50
definition/usage dichotomy 39
defintion/usage dichotomy 25, 27
Dependency 129
Derive 94, 96, 189, 190
DestructionEvent 73
Duration 74

E
Entry 82
Enumeration 128
ExecutionSpecification 72
Exit 82
Extend 88
Extension 137

F
Final 82
FinalNode 50
FlowFinal 50
FlowPort 29, 34
FlowSpecification 30, 35
ForkNode 50
formalism 23
Found 75

G
Generalization 31, 137

H
History 82

I
Include 88
Initial 83
InitialNode 51
InteractionUse 72
Interface 30
Internal 36
InterruptibleActivityRegion 55
isControl 53
isStream 53

J
JoinNode 51
Junction 83

L
Lifeline 72
Lost 74

M
Measures of Effectiveness 152
MergeNode 51
Message 74
Metaclass 136
Model 136, 138, 139

N
natural language 24
NoBuffer 52
Non 30

O
ObejctNode 54
ObjectFlow 54
ObjectNode 51, 62
OptimizationDirectionKind 191
Optional 53, 59
OverWrite 52

P
Package 115
PackageAccess 116
PackageContainment 116
PackageDiagram 115
PackageImport 116
ParameterSet 55
Parametric 40, 43
ParametricConstraint 42
ParametricConstraintUse 43

Part 30
Probability 53, 59
Problem 129
Profile 136
ProfileApplication 137

R
Rate 52, 60
Rationale 129
Real 122
Realization 31
Receive 83
Region 83
Requirement 94, 97
RequirementKind 97, 122, 123
Requirements 99
Requirements Traceability Table 99
RiskKind 97, 122, 124

S
Satisfy 94, 98
Send 83
Sequence 72
ServicePort 31
Simple 83
State 83
StateInvariant 73
Stereotype 136
Subject 88
Submachine 84

T
Table 111
TestCase 94, 98
Time 74
Trace 116
Trade 152
Trade Study 152
Transition 84

U
Unidirectional 137
Use 88

V
ValueProperty 129
ValueType 129, 132
Verdict 98, 122, 124
Verify 94, 98
VerifyMethodKind 99, 122, 124
View 115, 118
Viewpoint 116, 118