MAENAD EAST-ADL
Domain Model Specification

Version M2.1.11

Revision History

Version	Date	Reason
1.02	2004-06-30	EAST-ADL developed in the ITEA EAST-EEA project.
2.0	2008-03-20	EAST-ADL2 developed in the EC FP6 project ATESST. http://www.atesst.org/home/liblocal/docs/EAST-ADL-2.0-Specification_2008-02-29.pdf
2.1	2010-06-30	Updated version from the EC FP7 project ATESST2 with Timing Concepts from ITEA TIMMO.
M2.1.9	2011-01-30	Preliminary version from the EC FP7 project MAENAD
M2.1.9.20110830	2011-08-30	Intermediate version from the EC FP7 project MAENAD
M2.1.10	2012-03-30	Preliminary version from the EC FP7 project MAENAD. Minor layout changes as compared to 2012-03-30. Added description of default multiplicities in Language Formalism.
M2.1.11	2013-04-02	Release from the EC FP7 project MAENAD

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

This document describes a language specification developed by an informal partnership of automotive 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.

This specification is provided by the copyright holders and contributors "as is" and any expressed 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.

Copyright © 2000-2004, AUDI AG

Copyright © 2000-2004, BMW AG

Copyright © 2000-2004, 2008-2010, Centro Ricerche Fiat

Copyright © 2007-2010, Continental Automotive

Copyright © 2000-2008, DaimlerChrysler AG

Copyright © 2006-2010, Delphi/Mecel

Copyright © 2000-2008, ETAS GmbH

Copyright © 2006-2010, Mentor Graphics Hungary

Copyright © 2000-2004, OPEL GmbH

Copyright © 2000-2004, PSA

Copyright © 2000-2004, Renault

Copyright © 2000-2004, Robert Bosch GmbH

Copyright © 2000-2007, Siemens VDO Automotive SAS

Copyright © 2000-2004, Valeo

Copyright © 2000-2004, Vector

Copyright © 2006-2008, Volvo Car Corporation

Copyright © 2000-2010, Volvo Technology AB

Copyright © 2006-2010, VW/Carmeq

Copyright © 2000-2004, ZF

Copyright © 2000-2010, CEA-LIST

Copyright © 2000-2004, INRIA

Copyright © 2006-2010, Kungliga Tekniska Högskolan

Copyright © 2000-2004, LORIA

Copyright © 2000-2004, Paderborn Univerisity-C-LAB

Copyright © 2000-2004, Technical University of Darmstadt

Copyright © 2000-2010, Technische Universität Berlin
Copyright © 2008-2010, University of Hull

Table of Contents – Overview

Revision History 2
USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES 3
Part I Introduction 18
1 Language Formalism 20
2 Abbreviations 22
Part II Structural Constructs 23
3 SystemModeling 24
4 FeatureModeling 29
5 VehicleFeatureModeling 39
6 FunctionModeling 44
7 HardwareModeling 60
8 Environment 69
Part III Behavioral Constructs 71
9 Behavior 72
Part IV Variability 80
10 Variability 81
Part V Requirements 95
11 Requirements 96
12 UseCases 106
13 VerificationValidation 111
Part VI Timing 119
14 Timing 120
15 TimingConstraints 125
16 Events 143
Part VII Dependability 149
17 Dependability 150
18 ErrorModel 159
19 SafetyConstraints 169
20 SafetyRequirement 172
21 SafetyCase 175
Part VIII Generic Constraints 180
22 GenericConstraints 181
23 Part IX Infrastructure 185
24 Datatypes 186
25 Values 194

26 Elements 198
27 UserAttributes 207
Part X Annexes 212
28 Annex A: Notation 213
29 Annex B: Needs 218
30 Needs 219
31 Annex C: BehaviorDescription 226
32 BehaviorDescription 227
33 AttributeQuantificationConstraint 235
34 ComputationConstraint 239
35 TemporalConstraint 244

Table of Contents - Complete

Revision History 2
USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES 3
Part I Introduction 18
1 Language Formalism 20
1.1 Levels of Formalism 20
1.2 Specification Structure 20
1.2.1 Overview 20
1.2.2 Element Descriptions 20
2 Abbreviations 22
Part II Structural Constructs 23
3 SystemModeling 24
3.1 Overview 24
3.2 Element Descriptions 24
3.2.1 AnalysisLevel (from SystemModeling) «atpStructureElement» 24
3.2.2 DesignLevel (from SystemModeling) «atpStructureElement» 25
3.2.3 ImplementationLevel (from SystemModeling) «atpStructureElement» 26
3.2.4 SystemModel (from SystemModeling) «atpStructureElement» 27
3.2.5 VehicleLevel (from SystemModeling) «atpStructureElement» 27
4 FeatureModeling 29
4.1 Overview 29
4.2 Element Descriptions 29
4.2.1 BindingTime (from FeatureModeling) 30
4.2.2 BindingTimeKind (from FeatureModeling) «enumeration» 31
4.2.3 Feature (from FeatureModeling) «atpStructureElement» 32
4.2.4 FeatureConstraint (from FeatureModeling) 33
4.2.5 FeatureGroup (from FeatureModeling) 34
4.2.6 FeatureLink (from FeatureModeling) 34
4.2.7 FeatureModel (from FeatureModeling) «atpStructureElement» 36
4.2.8 FeatureTreeNode (from FeatureModeling) {abstract} 36
4.2.9 VariabilityDependencyKind (from FeatureModeling) «enumeration» 37
5 VehicleFeatureModeling 39
5.1 Overview 39
5.2 Element Descriptions 39
5.2.1 DeviationAttributeSet (from VehicleFeatureModeling) 40
5.2.2 DeviationPermissionKind (from VehicleFeatureModeling) «enumeration» 41

5.2.3 VehicleFeature (from VehicleFeatureModeling) 41
6 FunctionModeling 44
6.1 Overview 44
6.2 Element Descriptions 45
6.2.1 AllocateableElement (from FunctionModeling) {abstract} 45
6.2.2 Allocation (from FunctionModeling) 46
6.2.3 AnalysisFunctionPrototype (from FunctionModeling) 46
6.2.4 AnalysisFunctionType (from FunctionModeling) 47
6.2.5 BasicSoftwareFunctionType (from FunctionModeling) 47
6.2.6 ClientServerKind (from FunctionModeling) «enumeration» 48
6.2.7 DesignFunctionPrototype (from FunctionModeling) 48
6.2.8 DesignFunctionType (from FunctionModeling) 49
6.2.9 EADirectionKind (from FunctionModeling) «enumeration» 49
6.2.10 FunctionalDevice (from FunctionModeling) 50
6.2.11 FunctionAllocation (from FunctionModeling) 50
6.2.12 FunctionClientServerInterface (from FunctionModeling) «atpType» 51
6.2.13 FunctionClientServerPort (from FunctionModeling) 51
6.2.14 FunctionConnector (from FunctionModeling) «atpStructureElement» 52
6.2.15 FunctionFlowPort (from FunctionModeling) 53
6.2.16 FunctionPort (from FunctionModeling) {abstract} «atpPrototype» 54
6.2.17 FunctionPowerPort (from FunctionModeling) 55
6.2.18 FunctionPrototype (from FunctionModeling) {abstract} «atpPrototype» 55
6.2.19 FunctionType (from FunctionModeling) {abstract} «atpType» 56
6.2.20 HardwareFunctionType (from FunctionModeling) 57
6.2.21 LocalDeviceManager (from FunctionModeling) 58
6.2.22 Operation (from FunctionModeling) 58
6.2.23 PortGroup (from FunctionModeling) 59
7 HardwareModeling 60
7.1 Overview 60
7.2 Element Descriptions 60
7.2.1 Actuator (from HardwareModeling) 60
7.2.2 AllocationTarget (from HardwareModeling) {abstract} 61
7.2.3 CommunicationHardwarePin (from HardwareModeling) 61
7.2.4 ElectricalComponent (from HardwareModeling) «atpType» 62
7.2.5 HardwareBusKind (from HardwareModeling) «enumeration» 62
7.2.6 HardwareComponentPrototype (from HardwareModeling) «atpPrototype» 63
7.2.7 HardwareComponentType (from HardwareModeling) «atpType» 63
7.2.8 HardwareConnector (from HardwareModeling) «atpStructureElement» 64

7.2.9 HardwarePin (from HardwareModeling) {abstract} «atpStructureElement» ... *64*
7.2.10 HardwarePort (from HardwareModeling) «atpStructureElement» ... *65*
7.2.11 HardwarePortConnector (from HardwareModeling) «atpStructureElement» ... *65*
7.2.12 IOHardwarePin (from HardwareModeling) ... *66*
7.2.13 IOHardwarePinKind (from HardwareModeling) «enumeration» ... *66*
7.2.14 Node (from HardwareModeling) ... *67*
7.2.15 PowerHardwarePin (from HardwareModeling) ... *67*
7.2.16 Sensor (from HardwareModeling) ... *68*
8 Environment ... 69
8.1 Overview ... 69
8.2 Element Descriptions ... 69
8.2.1 ClampConnector (from Environment) «atpStructureElement» ... *69*
8.2.2 Environment (from Environment) ... *70*
Part III Behavioral Constructs ... 71
9 Behavior ... 72
9.1 Overview ... 72
9.2 Element Descriptions ... 73
9.2.1 Behavior (from Behavior) ... *73*
9.2.2 FunctionBehavior (from Behavior) ... *74*
9.2.3 FunctionBehaviorKind (from Behavior) «enumeration» ... *75*
9.2.4 FunctionTrigger (from Behavior) ... *76*
9.2.5 Mode (from Behavior) ... *77*
9.2.6 ModeGroup (from Behavior) ... *78*
9.2.7 TriggerPolicyKind (from Behavior) «enumeration» ... *78*
Part IV Variability ... 80
10 Variability ... 81
10.1 Overview ... 81
10.2 Element Descriptions ... 83
10.2.1 ConfigurableContainer (from Variability) ... *83*
10.2.2 ConfigurationDecision (from Variability) ... *84*
10.2.3 ConfigurationDecisionFolder (from Variability) ... *86*
10.2.4 ConfigurationDecisionModel (from Variability) {abstract} ... *87*
10.2.5 ConfigurationDecisionModelEntry (from Variability) {abstract} ... *87*
10.2.6 ContainerConfiguration (from Variability) ... *88*
10.2.7 FeatureConfiguration (from Variability) ... *89*
10.2.8 InternalBinding (from Variability) ... *89*
10.2.9 PrivateContent (from Variability) ... *90*
10.2.10 ReuseMetaInformation (from Variability) ... *90*

10.2.11 SelectionCriterion (from Variability) ... *91*
10.2.12 Variability (from Variability) ... *91*
10.2.13 VariableElement (from Variability) ... *92*
10.2.14 VariationGroup (from Variability) ... *93*
10.2.15 VehicleLevelBinding (from Variability) ... *94*
Part V Requirements ... 95
11 Requirements ... 96
11.1 Overview ... 96
11.2 Element Descriptions ... 97
11.2.1 DeriveRequirement (from Requirements) ... *98*
11.2.2 OperationalSituation (from Requirements) ... *98*
11.2.3 QualityRequirement (from Requirements) ... *99*
11.2.4 QualityRequirementKind (from Requirements) «enumeration» ... *99*
11.2.5 Refine (from Requirements) ... *100*
11.2.6 Requirement (from Requirements) ... *101*
11.2.7 RequirementsHierarchy (from Requirements) ... *101*
11.2.8 RequirementsLink (from Requirements) ... *102*
11.2.9 RequirementsModel (from Requirements) ... *103*
11.2.10 RequirementsRelationship (from Requirements) {abstract} ... *103*
11.2.11 RequirementsRelationshipGroup (from Requirements) ... *104*
11.2.12 Satisfy (from Requirements) ... *104*
12 UseCases ... 106
12.1 Overview ... 106
12.2 Element Descriptions ... 106
12.2.1 Actor (from UseCases) ... *107*
12.2.2 Extend (from UseCases) ... *107*
12.2.3 ExtensionPoint (from UseCases) ... *108*
12.2.4 Include (from UseCases) ... *108*
12.2.5 RedefinableElement (from UseCases) {abstract} ... *109*
12.2.6 UseCase (from UseCases) ... *109*
13 VerificationValidation ... 111
13.1 Overview ... 111
13.2 Element Descriptions ... 113
13.2.1 VerificationValidation (from VerificationValidation) ... *113*
13.2.2 Verify (from VerificationValidation) ... *113*
13.2.3 VVActualOutcome (from VerificationValidation) ... *114*
13.2.4 VVCase (from VerificationValidation) ... *114*
13.2.5 VVIntendedOutcome (from VerificationValidation) ... *115*

13.2.6 VVLog (from VerificationValidation) ... *115*
13.2.7 VVProcedure (from VerificationValidation) ... *116*
13.2.8 VVStimuli (from VerificationValidation) ... *117*
13.2.9 VVTarget (from VerificationValidation) ... *117*
Part VI Timing ... 119
14 Timing ... 120
14.1 Overview ... 120
14.2 Element Descriptions ... 120
14.2.1 Event (from Timing) {abstract} ... *120*
14.2.2 EventChain (from Timing) ... *121*
14.2.3 PrecedenceConstraint (from Timing) ... *122*
14.2.4 Timing (from Timing) ... *122*
14.2.5 TimingConstraint (from Timing) {abstract} ... *123*
14.2.6 TimingDescription (from Timing) {abstract} ... *123*
14.2.7 TimingExpression (from Timing) ... *124*
15 TimingConstraints ... 125
15.1 Overview ... 125
15.2 Element Descriptions ... 128
15.2.1 AgeConstraint (from TimingConstraints) ... *128*
15.2.2 ArbitraryConstraint (from TimingConstraints) ... *129*
15.2.3 BurstConstraint (from TimingConstraints) ... *130*
15.2.4 ComparisonConstraint (from TimingConstraints) ... *131*
15.2.5 ComparisonKind (from TimingConstraints) «enumeration» ... *131*
15.2.6 DelayConstraint (from TimingConstraints) ... *132*
15.2.7 ExecutionTimeConstraint (from TimingConstraints) ... *132*
15.2.8 InputSynchronizationConstraint (from TimingConstraints) ... *133*
15.2.9 OrderConstraint (from TimingConstraints) ... *134*
15.2.10 OutputSynchronizationConstraint (from TimingConstraints) ... *134*
15.2.11 PatternConstraint (from TimingConstraints) ... *135*
15.2.12 PeriodicConstraint (from TimingConstraints) ... *136*
15.2.13 ReactionConstraint (from TimingConstraints) ... *137*
15.2.14 RepetitionConstraint (from TimingConstraints) ... *138*
15.2.15 SporadicConstraint (from TimingConstraints) ... *139*
15.2.16 StrongDelayConstraint (from TimingConstraints) ... *139*
15.2.17 StrongSynchronizationConstraint (from TimingConstraints) ... *140*
15.2.18 SynchronizationConstraint (from TimingConstraints) ... *141*
16 Events ... 143
16.1 Overview ... 143

16.2 Element Descriptions 144
16.2.1 AUTOSAREvent (from Events) 144
16.2.2 EventFaultFailure (from Events) 144
16.2.3 EventFeatureFlaw (from Events) 144
16.2.4 EventFunction (from Events) 145
16.2.5 EventFunctionClientServerPort (from Events) 145
16.2.6 EventFunctionClientServerPortKind (from Events) «enumeration» 146
16.2.7 EventFunctionFlowPort (from Events) 146
16.2.8 ExternalEvent (from Events) 147
16.2.9 ModeEvent (from Events) 147
16.2.10 StateEvent (from Events) 148
Part VII Dependability 149
17 Dependability 150
17.1 Overview 150
17.2 Element Descriptions 152
17.2.1 ControllabilityClassKind (from Dependability) «enumeration» 152
17.2.2 Dependability (from Dependability) 153
17.2.3 DevelopmentCategoryKind (from Dependability) «enumeration» 154
17.2.4 ExposureClassKind (from Dependability) «enumeration» 154
17.2.5 FeatureFlaw (from Dependability) 155
17.2.6 Hazard (from Dependability) 155
17.2.7 HazardousEvent (from Dependability) 156
17.2.8 Item (from Dependability) 157
17.2.9 SeverityClassKind (from Dependability) «enumeration» 157
18 ErrorModel 159
18.1 Overview 159
18.2 Element Descriptions 161
18.2.1 Anomaly (from ErrorModel) {abstract} «atpPrototype» 161
18.2.2 ErrorBehavior (from ErrorModel) 161
18.2.3 ErrorBehaviorKind (from ErrorModel) «enumeration» 162
18.2.4 ErrorModelPrototype (from ErrorModel) «atpPrototype» 163
18.2.5 ErrorModelType (from ErrorModel) «atpType» 164
18.2.6 FailureOutPort (from ErrorModel) 165
18.2.7 FaultFailurePort (from ErrorModel) {abstract} «atpPrototype» 166
18.2.8 FaultFailurePropagationLink (from ErrorModel) 166
18.2.9 FaultInPort (from ErrorModel) 167
18.2.10 InternalFaultPrototype (from ErrorModel) 167
18.2.11 ProcessFaultPrototype (from ErrorModel) 168

19 SafetyConstraints 169
19.1 Overview 169
19.2 Element Descriptions 169
19.2.1 ASILKind (from SafetyConstraints) «enumeration» *169*
19.2.2 FaultFailure (from SafetyConstraints) *170*
19.2.3 QuantitativeSafetyConstraint (from SafetyConstraints) *170*
19.2.4 SafetyConstraint (from SafetyConstraints) *171*
20 SafetyRequirement 172
20.1 Overview 172
20.2 Element Descriptions 172
20.2.1 FunctionalSafetyConcept (from SafetyRequirement) *172*
20.2.2 SafetyGoal (from SafetyRequirement) *173*
20.2.3 TechnicalSafetyConcept (from SafetyRequirement) *174*
21 SafetyCase 175
21.1 Overview 175
21.2 Element Descriptions 175
21.2.1 Claim (from SafetyCase) *175*
21.2.2 Ground (from SafetyCase) *176*
21.2.3 LifecycleStageKind (from SafetyCase) «enumeration» *177*
21.2.4 SafetyCase (from SafetyCase) *178*
21.2.5 Warrant (from SafetyCase) *178*
Part VIII Generic Constraints 180
22 GenericConstraints 181
22.1 Overview 181
22.2 Element Descriptions 181
22.2.1 GenericConstraint (from GenericConstraints) *181*
22.2.2 GenericConstraintKind (from GenericConstraints) «enumeration» *182*
22.2.3 GenericConstraintSet (from GenericConstraints) *183*
22.2.4 TakeRateConstraint (from GenericConstraints) *184*
23 Part IX Infrastructure 185
24 Datatypes 186
24.1 Overview 186
24.2 Element Descriptions 186
24.2.1 ArrayDatatype (from Datatypes) *186*
24.2.2 CompositeDatatype (from Datatypes) *187*
24.2.3 EABoolean (from Datatypes) *187*
24.2.4 EADatatype (from Datatypes) {abstract} «atpType» *188*
24.2.5 EADatatypePrototype (from Datatypes) «atpPrototype» *188*

24.2.6 EANumerical (from Datatypes) ... 189
24.2.7 EAString (from Datatypes) ... 189
24.2.8 Enumeration (from Datatypes) ... 190
24.2.9 EnumerationLiteral (from Datatypes) ... 190
24.2.10 Quantity (from Datatypes) ... 191
24.2.11 RangeableValueType (from Datatypes) ... 191
24.2.12 Unit (from Datatypes) ... 192
25 Values ... 194
25.1 Overview ... 194
25.2 Element Descriptions ... 194
25.2.1 EAArrayValue (from Values) ... 194
25.2.2 EABooleanValue (from Values) ... 194
25.2.3 EACompositeValue (from Values) ... 195
25.2.4 EAEnumerationValue (from Values) ... 195
25.2.5 EAExpression (from Values) «atpMixedString» ... 196
25.2.6 EANumericalValue (from Values) ... 196
25.2.7 EAStringValue (from Values) ... 197
25.2.8 EAValue (from Values) {abstract} «atpPrototype» ... 197
26 Elements ... 198
26.1 Overview ... 198
26.2 Element Descriptions ... 199
26.2.1 Comment (from Elements) ... 199
26.2.2 Context (from Elements) {abstract} ... 200
26.2.3 EAConnector (from Elements) {abstract} ... 200
26.2.4 EAElement (from Elements) {abstract} ... 201
26.2.5 EAPackage (from Elements) ... 201
26.2.6 EAPackageableElement (from Elements) {abstract} ... 202
26.2.7 EAPort (from Elements) {abstract} ... 202
26.2.8 EAPrototype (from Elements) {abstract} ... 202
26.2.9 EAType (from Elements) {abstract} ... 203
26.2.10 EAXML (from Elements) ... 203
26.2.11 Rationale (from Elements) ... 204
26.2.12 Realization (from Elements) ... 204
26.2.13 Relationship (from Elements) {abstract} ... 205
26.2.14 TraceableSpecification (from Elements) {abstract} ... 205
27 UserAttributes ... 207
27.1 Overview ... 207
27.2 Element Descriptions ... 208

27.2.1 UserAttributeDefinition (from UserAttributes) 208
27.2.2 UserAttributedElement (from UserAttributes) 209
27.2.3 UserElementType (from UserAttributes) 210
Part X Annexes 212
28 Annex A: Notation 213
28.1.1 Actuator (from HardwareModeling) 213
28.1.2 AnalysisLevel (from SystemModeling) 213
28.1.3 ArrayDatatype (from Datatypes) 213
28.1.4 CommunicationHardwarePin (from HardwareModeling) 213
28.1.5 CompositeDatatype (from Datatypes) 213
28.1.6 DeriveRequirement (from Requirements) 213
28.1.7 DesignLevel (from SystemModeling) 213
28.1.8 EABoolean (from Datatypes) 214
28.1.9 EADatatype (from Datatypes) 214
28.1.10 EANumerical (from Datatypes) 214
28.1.11 EAString (from Datatypes) 214
28.1.12 ElectricalComponent (from HardwareModeling) 214
28.1.13 Enumeration (from Datatypes) 214
28.1.14 EnumerationLiteral (from Datatypes) 214
28.1.15 FunctionAllocation (from FunctionModeling) 214
28.1.16 FunctionBehavior (from Behavior) 214
28.1.17 FunctionConnector (from FunctionModeling) 214
28.1.18 FunctionPrototype (from FunctionModeling) 215
28.1.19 FunctionType (from FunctionModeling) 215
28.1.20 HardwareComponentPrototype (from HardwareModeling) 215
28.1.21 Hazard (from Dependability) 215
28.1.22 HazardousEvent (from Dependability) 215
28.1.23 ImplementationLevel (from SystemModeling) 215
28.1.24 IOHardwarePin (from HardwareModeling) 215
28.1.25 Node (from HardwareModeling) 215
28.1.26 PortGroup (from FunctionModeling) 215
28.1.27 PowerHardwarePin (from HardwareModeling) 216
28.1.28 PrecedenceConstraint (from Timing) 216
28.1.29 RangeableValueType (from Datatypes) 216
28.1.30 Realization (from Elements) 216
28.1.31 Refine (from Requirements) 216
28.1.32 Requirement (from Requirements) 216
28.1.33 RequirementsHierarchy (from Requirements) 216

28.1.34 SafetyGoal (from SafetyRequirement) ... *216*
28.1.35 Satisfy (from Requirements) ... *216*
28.1.36 Sensor (from HardwareModeling) ... *217*
28.1.37 SystemModel (from SystemModeling) ... *217*
28.1.38 VehicleLevel (from SystemModeling) ... *217*
28.1.39 Verify (from VerificationValidation) ... *217*
29 Annex B: Needs ... 218
30 Needs ... 219
30.1 Overview ... 219
30.2 Element Descriptions ... 219
30.2.1 ArchitecturalDescription (from Needs) ... *219*
30.2.2 ArchitecturalModel (from Needs) ... *220*
30.2.3 Architecture (from Needs) ... *220*
30.2.4 BusinessOpportunity (from Needs) ... *221*
30.2.5 Concept (from Needs) {abstract} ... *221*
30.2.6 Mission (from Needs) ... *222*
30.2.7 ProblemStatement (from Needs) ... *222*
30.2.8 ProductPositioning (from Needs) ... *223*
30.2.9 Stakeholder (from Needs) ... *223*
30.2.10 StakeholderNeed (from Needs) ... *224*
30.2.11 VehicleSystem (from Needs) ... *224*
31 Annex C: BehaviorDescription ... 226
32 BehaviorDescription ... 227
32.1 Overview ... 227
32.2 Element Descriptions ... 229
32.2.1 BehaviorConstraintBindingAttribute (from BehaviorDescription) ... *229*
32.2.2 BehaviorConstraintBindingEvent (from BehaviorDescription) ... *230*
32.2.3 BehaviorConstraintInternalBinding (from BehaviorDescription) {abstract} ... *230*
32.2.4 BehaviorConstraintParameter (from BehaviorDescription) {abstract} ... *231*
32.2.5 BehaviorConstraintPrototype (from BehaviorDescription) «atpPrototype» ... *232*
32.2.6 BehaviorConstraintTargetBinding (from BehaviorDescription) ... *233*
32.2.7 BehaviorConstraintType (from BehaviorDescription) «atpType» ... *233*
33 AttributeQuantificationConstraint ... 235
33.1 Overview ... 235
33.2 Element Descriptions ... 235
33.2.1 Attribute (from AttributeQuantificationConstraint) «atpPrototype» ... *235*
33.2.2 AttributeQuantificationConstraint (from AttributeQuantificationConstraint) ... *236*
33.2.3 BehaviorAttributeBinding (from AttributeQuantificationConstraint) ... *236*

33.2.4 LogicalEvent (from AttributeQuantificationConstraint) ... *237*
33.2.5 Quantification (from AttributeQuantificationConstraint) ... *237*
34 ComputationConstraint ... 239
34.1 Overview ... 239
34.2 Element Descriptions ... 239
34.2.1 ComputationConstraint (from ComputationConstraint) ... *239*
34.2.2 LogicalPath (from ComputationConstraint) ... *240*
34.2.3 LogicalTransformation (from ComputationConstraint) ... *241*
34.2.4 TransformationOccurrence (from ComputationConstraint) ... *242*
35 TemporalConstraint ... 244
35.1 Overview ... 245
35.2 Element Descriptions ... 246
35.2.1 LogicalTimeCondition (from TemporalConstraint) ... *247*
35.2.2 State (from TemporalConstraint) ... *247*
35.2.3 SynchronousTransition (from TemporalConstraint) ... *248*
35.2.4 TemporalConstraint (from TemporalConstraint) ... *249*
35.2.5 Transition (from TemporalConstraint) ... *249*
35.2.6 TransitionEvent (from TemporalConstraint) ... *250*

Part I Introduction

The purpose of the EAST-ADL language is to capture automotive electrical and electronic systems with sufficient detail to allow modeling for documentation, design, analysis, and synthesis. These activities require system descriptions on several abstraction levels, from top level features down to tasks and communication frames. Moreover, the activities also involve the expression of non-structural aspects of the electrical/electronic system under development, e.g., requirements, behavior, and verification and validation.

By hosting all aspects of the automotive electrical/electronic system with this domain model, the relations between them can be managed more efficiently. The different abstraction levels give a modeling context and a view of systems, functions, and features on different levels of detail, and with a clear separation of concerns.

This language specification describes how information needed for relevant analysis and synthesis can be captured but does not define how the analysis or synthesis should be done. This approach was chosen in order to allow company-specific processes while harmonizing the design artifacts to allow information exchange between tools and organizations. In supplementary material we provide a methodology description, where the language concepts are used in the context of a generic process.

The purpose of the domain model is to specify the concepts of the domain. The domain model of EAST-ADL also acts as a metamodel, which uses concepts from the AUTOSAR metamodel. This means that the EAST-ADL metamodel (i.e., the EAST-ADL domain model) can be imported into the AUTOSAR metamodel, where the references from EAST-ADL to AUTOSAR are restored. The current version of the corresponding AUTOSAR metamodel is 3.1.

To import EAST-ADL into an AUTOSAR metamodel:

1) Open the AUTOSAR metamodel in Enterprise Architect.

2) Import the EAST-ADL metamodel as an XMI-file.

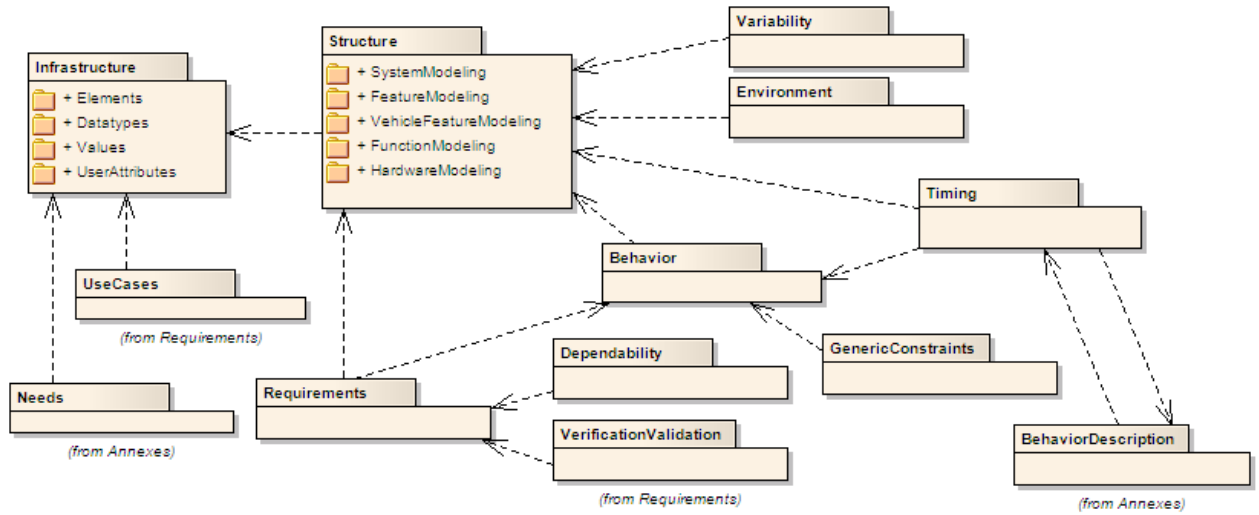

Figure 1 This diagram shows dependencies between packages in the domain model. All packages except the AUTOSAR package depend on the EAST-ADL Infrastructure package. The AUTOSAR package contains some concepts that EAST-ADL elements in the Infrastructure and Structure packages depend on.

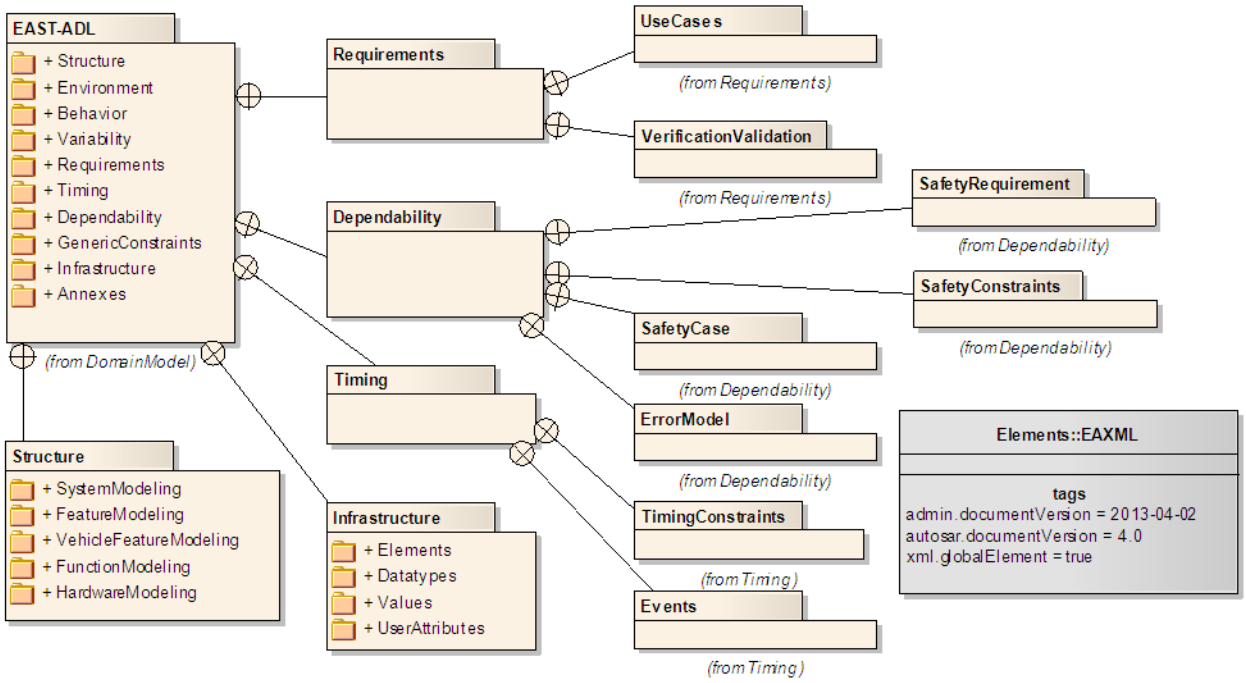

Figure 2. Packages in the EAST-ADL domain model.

1 Language Formalism

1.1 Levels of Formalism

The EAST-ADL domain model is specified using a combination of UML modeling techniques and precise natural language to balance rigor and understandability.

1.2 Specification Structure

The EAST-ADL domain model specification is organized into different parts:

Part I includes a general introduction to the specification.

Parts II–IX include chapters that are organized according to the EAST-ADL domain model subpackages.

Part X consists of annexes. This is where the notation for each element of the language is found.

Each part of the specification contains one or more chapters. Each chapter has the same structure: first an Overview section and then am Element Descriptions section.

The EAST-ADL specification has an Annex A proposing a possible notation for some of the metaclasses. Subsequent annexes contain preliminary extensions to the language that add modelling concepts that are not part of the basic content. It is likely that these extensions will be refined and subsequently integrated into the regular extensions in future releases of EAST-ADL.

1.2.1 Overview

This section of a chapter provides an overview of the EAST-ADL domain model constructs defined in each subpackage, which are usually described by one or more class diagrams that show the relationships between the elements of the package and, where applicable, relationships to other packages.

Elements from AUTOSAR are shown in the diagrams as classes with a pink background.

1.2.2 Element Descriptions

The Element Description specifies the individual elements within each EAST-ADL subpackage. All elements in the subpackage are ordered alphabetically and each element has the following specification information:

<Element (from subpackage)>

The element description starts with a header with the name of the element and the subpackage that it belongs to. If the element is abstract, “{abstract}” is shown in the header. If the element has a stereotype attached, this is shown within guillemets («...»).

Generalizations

This paragraph lists those domain model constructs that the current element specializes (inherits from).

Description

This paragraph provides a description of the current element and the direct context of this element (related domain model constructs).

Attributes

This paragraph specifies the element's attributes with names and types. The attribute has a unique name within the element. Each attribute has a type which is either a primitive or refers to an enumeration.

In addition, each attribute is supplied with a cardinality; EAST-ADL uses only cardinalities [0..1] for optional attributes and [1] for mandatory attributes.

Associations

This paragraph specifies the element's rolenames for related concepts, as referred to by this element by an association. The documentation of the rolename may include the stereotype «isOfType», which is used to specify that the related element types this element.

Dependencies

This paragraph specifies the element's rolenames for related concepts, as referred to by this element by a dependency. The dependencies are always stereotyped «instanceRef» which is the pattern used by AUTOSAR to identify that a more detailed model of associations rather than this dependency is necessary to identify the precise context of the target element.

Constraints

This paragraph specifies the element's constraints for verification of the correct use of the element. The constraints are given in natural language.

Semantics

This paragraph specifies the element's meaning in a concise form and defines how it may be used and specialized by other elements within the language. Definitions in this paragraph are not tailored to understandability (as in the "Description" paragraph) but precision and succinctness.

2 Abbreviations

AADL	Architecture Analysis and Design Language
ADL	Architecture Description Language
ATESST	Advancing Traffic Efficiency and Safety through Software Technology
AUTOSAR	AUTomotive Open System ARchitecture
EAST-EEA	Electronics Architecture and Software Technology - Embedded Electronic Architecture
ECU	Electronic Control Unit
FAA	Functional Analysis Architecture
FDA	Functional Design Architecture
HDA	Hardware Design Architecture
RIF	Requirement Interchange Format
SysML	System Modeling Language
TADL	Timing Augmented Description Language
TIMMO	Timing Model
UML	Unified Modeling Language
V&V	Verification & Validation
XMI	XML Metadata Interchange
XML	eXtensible Mark-up Language

Part II Structural Constructs

This part of the specification defines the structural constructs used in EAST-ADL. The structural view of a model focuses on the static structure of the instances of the system being modeled and their static relationships. This includes the internal structure of such instances and their external interfaces through which they can be connected to communicate with one another, by exchanging data or sending messages.

EAST-ADL abstraction layers are introduced to allow reasoning about the features on several levels of abstraction. Note, however, that the abstraction levels are only conceptual; the modeling elements are organized according to the artifacts, which may span more than one of these layers. Where applicable, entities on different abstraction levels are related with a realization association to allow traceability analysis. Traceability can also be deduced from the requirements structure.

The EAST-ADL abstraction layers with their corresponding artifacts are:

- Vehicle Level, with feature models describing decompositions of system characteristics organized as a software product line.

- Analysis Level, including the Functional Analysis Architecture (FAA). The FAA is built from an abstract functional definition of the system to capture analysis support of what the system shall do, ensuring relation with features from the Vehicle layer view. There is an n-to-m mapping between VehicleFeature and Feature entities and FAA entities (i.e., one or several functions may realize one or several features).

- Design Level, including the Functional Design Architecture (FDA). The FDA represents a decomposition of functionalities denoted in the FAA, including behavioral description but excluding software implementation constraints. The decomposition has the purpose of making it possible to meet constraints regarding non-functional properties such as allocation, efficiency, reuse, or supplier concerns. Again, there are n-to-m mappings by Realization relationships between entities in the FDA and entities in the FAA. Non-transparent infrastructure functionality of the AUTOSAR Basic SW Architecture, such as mode changes and error handling, are also represented at the Design Level in terms of BasicSoftwareFunctions.

- The Hardware Architecture models Electronic Control Units (ECUs), communication links, sensors and actuators and their connections. The Hardware Architecture is also considered at the Analysis Level as FunctionalDevices and at the Design Level as HardwareFunctions because models of sensors, actuators, and early assumptions of hardware may be needed either for the Functional Analysis Architecture or the Functional Design Architecture.

- Implementation Level refers to the System element in an AUTOSAR model.

The Environment contains Environment functions, which are encapsulations of plant models, i.e. models of the behavior of the vehicle and its non-electronic systems. Environment models are needed for validation and verification, from early analysis models to the implemented embedded system. Note that no specific EnvironmentFunction exists as such in the language, but DesignFunctions or AnalysisFunctions are used. However to connect such functions to the rest of the systems, special connectors are used, ClampConnectors which can traverse hierarchal containments.

3 SystemModeling

3.1 Overview

The SystemModel is the top-level container of an EAST-ADL model. It represents the electrical/electronic system in a vehicle and concepts related to the various abstraction levels.

For the design of electrical/electronic systems of arbitrary size and complexity, the possibility of hierarchical structuring of the instances is provided, so these models contain further elements in a hierarchy. Relations between these elements across the boundaries of the abstraction levels are contained in a SystemModel. This is possible because the SystemModel is a Context, and is thus able to contain relations.

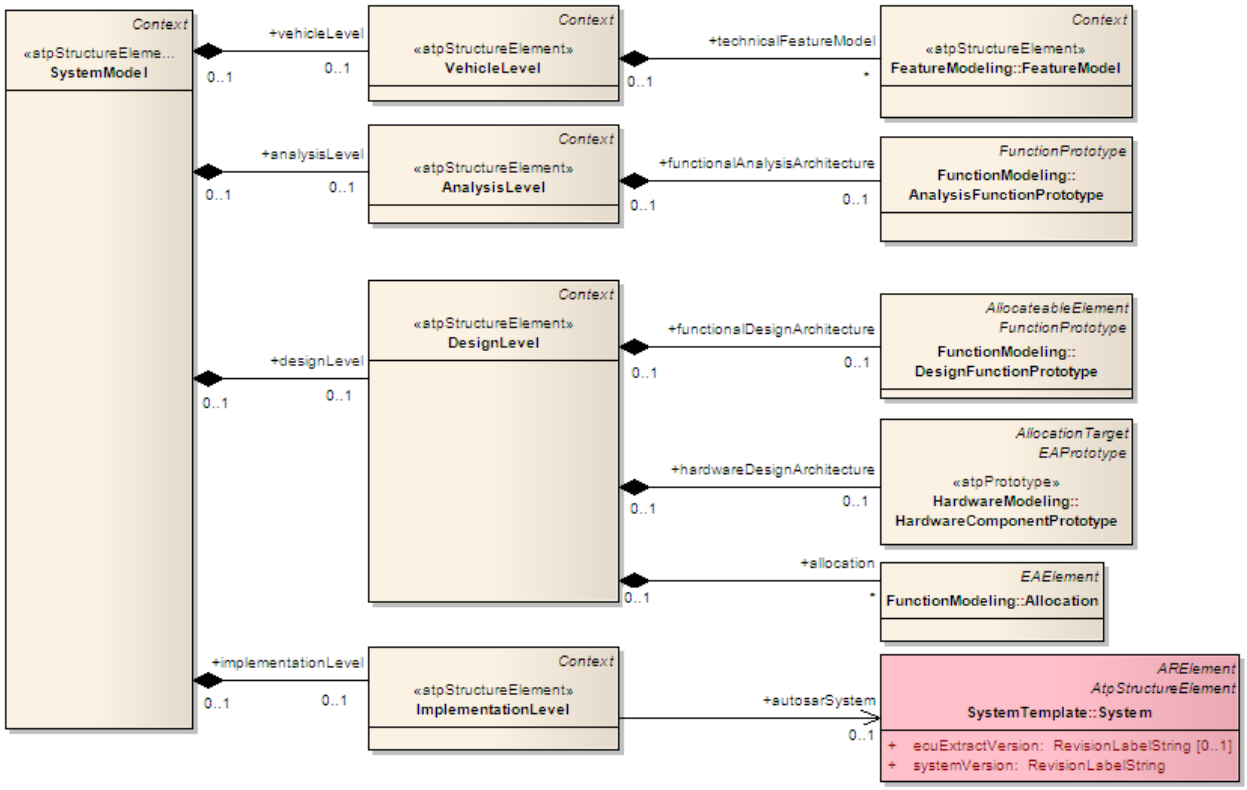

Figure 3. Diagram for SystemModel. Note how the ImplementationLevel refers to the System from the AUTOSAR SystemTemplate.

3.2 Element Descriptions

3.2.1 AnalysisLevel (from SystemModeling) «atpStructureElement»

Generalizations

- Context (from Elements)

Description

The AnalysisLevel represents the vehicle electrical/electronic system in terms of its abstract functional definition. It includes the functional analysis architecture (FAA), which represents the functional structure.

Attributes

No additional attributes

Associations

- functionalAnalysisArchitecture : AnalysisFunctionPrototype [0..1] {composite}
  The included functionalAnalysisArchitecture, this prototype shall be typed by an AnalysisFunctionType modeling the FunctionalAnalysisArchitecture. It is an abstract functional representation of the electrical/electronic system and realizes the VehicleFeatures.

Constraints

No additional constraints

Semantics

AnalysisLevel represents the vehicle electrical/electronic system in terms of its abstract functional definition. It defines the logical functionality and a logical decomposition of functionality down to the appropriate granularity.

3.2.2 DesignLevel (from SystemModeling) «atpStructureElement»

Generalizations

- Context (from Elements)

Description

The DesignLevel represents the vehicle electrical/electronic system on the design abstraction level. It includes primarily the Functional Design Architecture (FDA), and the HardwareDesignArchitecture (HDA).

FDA represents a top level Function. It is supposed to implement all the functionalities of a vehicle, as specified by a FAA or a Vehicle level (if no FAA has been defined during the process).

The design level in EAST-ADL includes the design architecture containing the functional specification and hardware architecture of the vehicle electrical/electronic system. The design architecture includes the FDA representing a decomposition of functionalities analyzed on the analysis level. The decomposition has the purpose of making it possible to meet constraints regarding non-functional properties such as allocation, efficiency, reuse, or supplier concerns. There is an n-to-m mapping between entities of the design level and the ones on the analysis level.

Non-transparent infrastructure functionality such as mode changes and error handling are also represented at the design level, such that their impact on applications' behaviors can be estimated.

The FDA parts are typed by DesignFunctionTypes and e.g. LocalDeviceManagers. The view of the HardwareArchitecture facilitates the realization of LocalDeviceManager as sensor/actuator HW elements.

The HDA is the hardware design from a system perspective. The HDA has two purposes:

1) It shows the physical entities and how they are connected.

2) It is an allocation target for the Functions of the FDA.

The HDA represents the hardware architecture of the embedded system. Its contained HW elements represent the physical aspects of the hardware entities and how they are connected. HardwareFunctionTypes associated to HW components represent the logical behavior of the contained HW elements.

Attributes

No additional attributes

Associations

- allocation : Allocation [*] {composite}
- functionalDesignArchitecture : DesignFunctionPrototype [0..1] {composite}
  The included FDA. This includes functional design, modeled by DesignFunctions; middleware functionality abstraction, to be modeled by BasicSoftwareFunctionTypes in the implementation level; and logical hardware, modeled by HardwareFunctionTypes.

  The FDA represents the elementary design function that is used to describe the leaves of the functional hierarchy. The composition of these leaves makes up the implementation behavior of the entire functional hierarchy.
- hardwareDesignArchitecture : HardwareComponentPrototype [0..1] {composite}
  The included HDA models the resources to which the functional design architecture parts may be allocated.

Constraints

No additional constraints

Semantics

The DesignLevel is the representation of the vehicle electrical/electronic system on the design abstraction level. It corresponds to the design of logical functions and boundaries extended in regards to resource commitment.

3.2.3 ImplementationLevel (from SystemModeling) «atpStructureElement»

Generalizations

- Context (from Elements)

Description

The ImplementationLevel represents the software architecture and the hardware architecture of the electrical/electronic system in the vehicle. The ImplementationLevel is defined by the AUTOSAR SystemArchitecture and SoftwareArchitecture. For example, functions of the FDA will be realized by AUTOSAR SW-Components in the ImplementationLevel. Traceability is supported from implementation level elements (AUTOSAR) to upper level elements by Realization relationships.

Attributes

No additional attributes

Associations

- autosarSystem : System [0..1]
  The AUTOSAR System from the SystemTemplate represents the AUTOSAR implementation of the SystemModel.

Constraints

No additional constraints

Semantics

The ImplementationLevel is the representation of the vehicle electrical/electronic system on the implementation abstraction level. It corresponds to the system implementation in Software and Hardware.

3.2.4 SystemModel (from SystemModeling) «atpStructureElement»

Generalizations

- Context (from Elements)

Description

The SystemModel is used to organize models/architectures according to their abstraction level; it can also hold with relationships between the different levels.

Attributes

No additional attributes

Associations

- vehicleLevel : VehicleLevel [0..1] {composite}
  The included vehicle abstraction level.
- designLevel : DesignLevel [0..1] {composite}
  The included design abstraction level.
- analysisLevel : AnalysisLevel [0..1] {composite}
  The included analysis abstraction level.
- implementationLevel : ImplementationLevel [0..1] {composite}
  The included implementation abstraction level.

Constraints

No additional constraints

Semantics

The SystemModel represents the electrical/electronic system of the vehicle, and concepts related to the various abstraction levels.

3.2.5 VehicleLevel (from SystemModeling) «atpStructureElement»

Generalizations

- Context (from Elements)

Description

The VehicleLevel represents the vehicle content from an external perspective through an arbitrary set of feature models. These contain VehicleFeatures that are organized to reflect the vehicle configuration and that have associated requirements, use cases, etc. for its definition.

Attributes

No additional attributes

Associations

- technicalFeatureModel : FeatureModel [*] {composite}
  This association identifies the core technical feature model of the complete system. This has a special role as it defines all the features of the complete system on vehicle level. In

addition to this feature model, there may be one or more so-called product feature models (cf. association productFeatureModel in meta-class Variability in the variability extension).

Usually there will be the core technical feature model and one or more so-called "product feature models" on vehicle level, which provide an orthogonal view on the core technical feature model tailored to a particular purpose, for example an end-customer feature model. However, there may be other use cases for feature models on vehicle level. More detailed treatment of this is beyond the scope of the language specification and can be found in the accompanying usage and methodology documentations.

Constraints

[1] All contained feature models are FeatureModels that only contain VehicleFeatures.

Semantics

The VehicleLevel represents the vehicle content through solution-independent features.

4 FeatureModeling

4.1 Overview

This package describes the basic feature modeling that is employed on the vehicle level as well as on the artifact levels, i.e., on AnalysisLevel and below. Details of feature modeling that are specific to the vehicle level are factored out and documented separately in the package VehicleFeatureModeling.

A feature in this sense is a characteristic or trait that individual variants of either the complete system (in case of feature models on VehicleLevel) or an individual Analysis- or DesignFunctionType (in case of public feature models of FunctionTypes) may or may not possess. By listing features that are common to all variants as well as those that apply only to some variants, a feature model defines the complete system's / FunctionType's commonality and variability. In addition to this use in the context of variability management, features can also be used to represent coarse-grained requirements, in order to define a high-level break-down of the system's main functionality. Therefore, feature modeling is not only useful for variability management but also when modeling completely invariant systems. More details are given below in the description of the meta-classes Feature and FeatureModel and in package VehicleFeatureModeling.

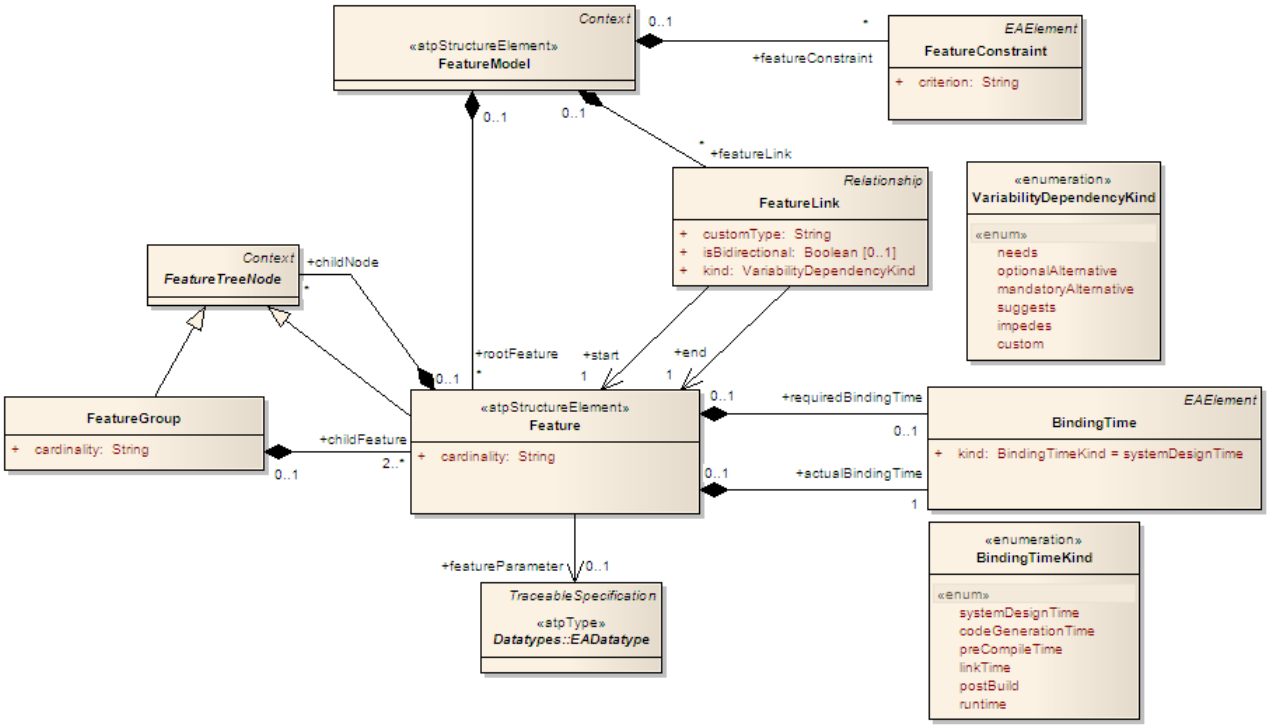

Figure 4. Diagram for FeatureModeling.

4.2 Element Descriptions

4.2.1 BindingTime (from FeatureModeling)

Generalizations

- EAElement (from Elements)

Description

The motivation for attributing features and variable elements with binding times is that binding times encapsulate important information about the variability under view.

Variability that must be bound (determined, decided) very early in the system development may not be visible in one single feature model but only in comparison with different feature models in the context of multi-level feature trees; late bound variability is variability providing the driver with choices for current equipment configuration.

Binding times are important because they describe if the variability must be decided during system development or if the variability is determined by a customer or if the variability itself is part of the product features that are sold to the customer. Possible binding times are:

- SystemDesignTime
- CodeGenerationTime
- PreCompileTime
- LinkTime
- PostBuild
- Runtime

Note that a binding time is never a particular point in time such as April 2nd, 2011, but always a certain stage in the overall development, production and shipment process as represented by the above values.

Each feature must have a binding time (association requiredBindingTime) and may have one further binding time (association actualBindingTime).

The required binding time describes the binding time that the feature is intended to have. But due to technical conditions it may occur that the actually realized binding time of the feature differs from the originally intended binding time. In this case one has to provide information about the actual binding time. In the rationale it must be described by what the required binding time is motivated by and what the reasons are for a (different) actual binding time.

Attributes

- kind : BindingTimeKind = systemDesignTime [1]
  The kind of the binding time, see enumeration BindingTimeKind for specification of binding times.

Associations

No additional associations

Constraints

No additional constraints

Semantics

See description.

4.2.2 BindingTimeKind (from FeatureModeling) «enumeration»

Generalizations

None

Description

BindingTimeKind represents the set of possible binding times.

Enumeration Literals

- codeGenerationTime
  Variability will be bound during code generation.

  From AUTOSAR:

  * Coding by hand, based on requirements document.

  * Tool based code generation, e.g. from a model.

  * The model may contain variants.

  * Only code for the selected variant(s) is actually generated.
- linkTime
  Variability will be bound during linking.

  From AUTOSAR:

  Configure what is included in object code, and what is omitted

  Based on which variant(s) are selected

  E.g. for modules that are delivered as object code (as opposed to those that are delivered as source code)
- postBuild
  Variability will be bound at certain occasions after shipment, for example when the vehicle is in a workshop.
- preCompileTime
  Variability will be bound during or immediately prior to code compilation.

  From AUTOSAR:

  This is typically the C-Preprocessor. Exclude parts of the code from the compilation process, e.g., because they are not required for the selected variant, because they are incompatible with the selected variant, because they require resources that are not present in the selected variant. Object code is only generated for the selected variant(s). The code that is excluded at this stage will not be available at later stages.
- runtime
  Variability will be bound by the customer after shipment by way of vehicle configuration.

  Variability with such a late binding time can also be seen as a special functionality of the system which is not documented as variability at all. However, it is sometimes advantageous to represent such cases as variability in order to be able to seamlessly include them in the overall variability management activities.
- systemDesignTime
  Variability will be bound during development of the electrical/electronic system.

  From AUTOSAR:

  * Designing the VFB.

  * Software Component types (portinterfaces).

* SWC Prototypes and the Connections between SWCprototypes.

* Designing the Topology

* ECUs and interconnecting Networks

* Designing the Communication Matrix and Data Mapping

Associations

No additional associations

Constraints

No additional constraints

Semantics

See description.

4.2.3 Feature (from FeatureModeling) «atpStructureElement»

Generalizations

- FeatureTreeNode (from FeatureModeling)

Description

A Feature represents a characteristic or trait of some object of consideration. The actual object of consideration depends on the particular purpose of the feature's containing feature model.

Example 1: The core technical feature model on vehicle level defines the technical properties of the complete system, i.e., vehicle. So its object of consideration is the vehicle as a whole and therefore its features represent characteristics or traits of the vehicle as a whole.

Example 2: The public feature model of some function F in the FDA defines the features of this particular software function. So its object of consideration is function F and therefore its features represent characteristics or traits of this function F.

Attributes

- cardinality : String [1]

  Specifies the Feature's cardinality stating how often this feature may be selected during configuration.

  Typical cardinalities include:

  - A cardinality of 0..1 means that this Feature is optional, i.e. it can be selected or deselected during configuration.

  - A cardinality of 1 means that this Feature is mandatory, i.e. it cannot be deselected but is always present in a configuration if its parent feature is present; mandatory root features are present in all configurations.

  - A cardinality of 0 means that this Feature is abstract, i.e. it cannot be selected and is never present in any configuration. This can be used to completely disable a feature and, in the case of non-leaf features, the whole subtree below it, for example to tentatively remove a subtree without (yet) deleting it completely from the model.

  - A cardinality with an upper bound greater than 1 or * (infinite), such as [0..2], [1..*], or [2..8], means that this Feature is cloned, i.e. it may be selected more than once during configuration. If such a feature is actually selected more than once in a particular configuration, then its entire subtree may be configured differently for each selection. Cloned features are in fact instantiated during configuration and each instance is provided with a name.

Note that using cloned features, i.e. features with cardinality having an upper bound greater than 1, has far-reaching consequences for how Features are applied. If this is not desired/needed in a certain project, cardinalities >1 can be prohibited by specifying an appropriate complianceLevel in the FeatureModel. As a general guideline, cloned features should be avoided as far as possible. In some situations, however, they can prove extremely useful and elegant. For example, consider the feature model of a wiper system; in order to allow for an extremely flexible configuration of the interval modes, a single parameterized cloned feature can be used: "IntervalMode[2..*] : Float". With this single cloned feature, any number of intervals can be created (but at least 2) and for each interval a precise duration in sec can be configured; without cloned features, this degree of flexibility could not easily be achieved.

Associations

- actualBindingTime : BindingTime [1] {composite}
  The actual binding time, independent of the required binding time.

  Due to technical conditions it may occur that the actually realized binding time of the feature/variation point differs from the originally intended binding time. In this case one has to provide information about the actual binding time.

  In the rationales it must be described what the reasons are for a (different) actual binding time.
- requiredBindingTime : BindingTime [0..1] {composite}
  The required binding time could possibly deviate from the actual binding time.

  The attribute reflects the intended binding time, and actual binding time can be later adapted to this required binding time, if surrounding constraints allow a change.

  Each feature/variation point must have a required binding time attribute.
- featureParameter : EADatatype [0..1]
  For parameterized features, this specifies the type of the feature's parameter.
- childNode : FeatureTreeNode [*] {composite}
  Features may have any number of Features or FeatureGroups as their children or none at all.

Constraints

No additional constraints

Semantics

Feature is a (non)functional characteristic, constraint or property that can be present or not in a (vehicle) product line.

4.2.4 FeatureConstraint (from FeatureModeling)

Generalizations

- EAElement (from Elements)

Description

Captures a constraint on the containing feature model's configuration which is too complex to be expressed by way of a FeatureLink. In general, all constraints that can be expressed by a FeatureLink can also be expressed by a FeatureConstraint, but not vice versa.

Attributes

- criterion : String [1]

The actual constraint. This is a logic expression in VSL like the criterion of a ConfigurationDecision. For the constraint to be met this expression always has to evaluate to true.

For example, to express a mutual exclusion of two features, use the expression "! (Radar & RainSensor)". However, note that this particular constraint could also be formulated as a FeatureLink with type "excludes".

Associations

No additional associations

Constraints

No additional constraints

Semantics

See description.

4.2.5 FeatureGroup (from FeatureModeling)

Generalizations

- FeatureTreeNode (from FeatureModeling)

Description

FeatureGroup is a specialization of the FeatureTreeNode, enabling grouping of several Features.

Attributes

- cardinality : String [1]
  The cardinality of the FeatureGroup, specifies how the grouped features, in featureGroup, can be combined. For example, a FeatureGroup owning the two Features A and B, and with a cardinality of [1], means that A and B are alternatives, but only one of them can be chosen. Mandatory features among the child features count as 1 and for cloned features all instances created in the configuration count.

Associations

- childFeature : Feature [2..*] {composite}
  FeatureGroups may only have Features as their children and must always have at least two children.

  It is perfectly legal to have child features in a feature group that are mandatory or cloned. However, except for special use cases, this is discouraged and therefore all child features of a FeatureGroup should usually be optional, i.e. have cardinality [0..1].

Constraints

No additional constraints

Semantics

FeatureGroup is a grouping entity for sibling Features to reflect variability for a set of Features.

4.2.6 FeatureLink (from FeatureModeling)

Generalizations

- Relationship (from Elements)

Description

A FeatureLink resembles a Relationship between two Features referred to as 'start' and 'end' feature (such as "feature S requires feature E" or "S excludes E").

The type of the FeatureLink specifies the precise semantics of the relationship. There are several predefined types, for example "needs" states that S requires E. In addition, user-defined types are allowed as well. For user-defined types, attribute 'customType' provides a unique identifier of the custom link type and attribute 'isBidirectional' states whether the link is uni- or bidirectional.

FeatureLinks are similar to FeatureConstraints but much more restricted. The rationale for having FeatureLinks in addition to FeatureConstraints is that in many cases FeatureLinks are sufficient and tools can deal with them more easily and appropriately (e.g. they can easily be presented visually as arrows in a diagram).

Attributes

- customType : String [1]
  The custom type of this FeatureLink identified by a String value. This attribute's value is ignored if attribute 'kind' is set to some other value than 'custom'.

  Each company or project can decide to use additional link types by defining unique key-words for them. In cases where FeatureModels are shared with third parties (other departments, companies, etc.) a globally unique type string must be used. Follow the instructions for finding globally unique keys for user attributes (cf. documentation of metaclass UserAttributeValue).
- isBidirectional : Boolean [0..1]
  Tells whether the FeatureLink is bidirectional or unidirectional. For predefined kinds, such as "needs", "mandatoryAlternative", etc., this attribute will be ignored and the kind determines whether the link is bidirectional or not (as defined in the documentation of attribute 'type', below). For custom kinds, this attribute may be provided to explicitly state the link's direction. If this attribute is not provided in case of a custom link type, then the link is assumed to be unidirectional.
- kind : VariabilityDependencyKind [1]
  The kind determines the precise semantics of the relation between the FeatureLink's start and end feature. There are 5 predefined kinds as defined by enumeration VariabilityDependencyKind and in the case of kind 'custom' the attribute customType can be used to define a custom feature link type.

Associations

- start : Feature [1]
  The source [supplier] Feature of the relationship.
- end : Feature [1]
  The target [client] Feature of the dependency.

Constraints

[1] The start and end Features of a FeatureLink must be contained in the FeatureModel that contains the FeatureLink.

Semantics

The FeatureLink is a relationship between Features that may constrain the selection of Features involved in the relationship.

4.2.7 FeatureModel (from FeatureModeling) «atpStructureElement»

Generalizations

- Context (from Elements)

Description

FeatureModel denotes a model owning Features. The FeatureModel can be used to describe variability and commonality of a specified electrical/electronic system at any abstraction level in the SystemModel.

The FeatureModel can be used either to describe the variability within a particular Function or to describe the overall variability of a vehicle (cf. VehicleLevel). The FeatureModel describing internal variability of a FunctionType refers to the VehicleLevel by a «realizes» link (informative).

Note, however, that a FeatureModel per definition does not always have to define variability. If a feature model contains only mandatory features, then its purpose is completely unrelated to variability. The features in such a FeatureModel could serve, for example, as invariant "coarse-grained requirements". The most important example is the core technical feature model on vehicle level which is also used for SystemModels that do not contain any variability at all. However, most uses of feature models in EAST-ADL are primarily motivated by variability definition and management.

A public, local FeatureModel of an artifact element realizes a VehicleFeature of the VehicleLevel.

Attributes

No additional attributes

Associations

- rootFeature : Feature [*] {composite}
  The root Features owned by the FeatureModel. Note that only root Features are directly contained in the model; non-root Features are contained in their parent Feature or parent FeatureGroup.
- featureLink : FeatureLink [*] {composite}
  The FeatureLinks owned by the FeatureModel.
- featureConstraint : FeatureConstraint [*] {composite}
  FeatureConstraints owned by the FeatureModel.

Constraints

No additional constraints

Semantics

The FeatureModel has no specific semantics. Further subclasses of FeatureModel will add semantics appropriate to the concept they represent.

4.2.8 FeatureTreeNode (from FeatureModeling) {abstract}

Generalizations

- Context (from Elements)

Description

The abstract base class for all nodes in a feature tree.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

FeatureTreeNode has no specific semantics. Further subclasses of FeatureTreeNode will add semantics appropriate to the concept they represent.

4.2.9 VariabilityDependencyKind (from FeatureModeling) «enumeration»

Generalizations

None

Description

This enumeration encapsulates the available types of constraints that can be applied to a FeatureLink or VariationGroup (the latter is applicable only if the variability extension is used).

Enumeration Literals

- custom
  When used in a FeatureLink: the attribute customType in the FeatureLink defines the custom feature link type as explained there.

  When used in a VariationGroup: this kind states that the dependency between the elements denoted by association variableElement of the VariationGroup will be defined by a logical expression in attribute 'constraint' of the VariationGroup.
- impedes
  Weak from of "excludes".

  When used in a FeatureLink: the FeatureLink's start feature S and its end feature E must usually(!) not be selected in a single configuration. You can select S together with E but you should have a good reason to do so. Always bidirectional.

  When used in a VariationGroup: accordingly as above.
- mandatoryAlternative
  When used in a FeatureLink: either the FeatureLink's start feature S or its end feature E must be selected in any configuration: S xor E. Always bidirectional.

  When used in a VariationGroup: this kind states that exactly(!) one element of the elements denoted by association variableElement of the VariationGroup must be selected in any valid final system configuration.
- needs
  When used in a FeatureLink: if the FeatureLink's start feature S is selected, then also its end feature E must be selected: not (S and not E). Always unidirectional.

  When used in a VariationGroup: assuming the ordered association variableElement in meta-class VariationGroup refers to elements VE1, VE2, ..., VEn, this kind states that VE1 requires (i.e. may not appear without) all other elements VE2, VE3, ..., VEn.
- optionalAlternative

When used in a FeatureLink: the FeatureLink's start feature S and end feature E are incompatible and must never be both selected in a single configuration: not (S and E). Always bidirectional.

When used in a VariationGroup: this kind states that at most(!) one element of the elements denoted by association variableElement of the VariationGroup must be selected in any valid final system configuration.

- suggests
  Weak form of "needs".

  When used in a FeatureLink: if the FeatureLink's start feature S is selected, then usually(!) also its end feature E must be selected. You can select S without E but you should have a good reason to do so. Always unidirectional.

  When used in a VariationGroup: accordingly as above.

Associations

No additional associations

Constraints

No additional constraints

Semantics

Predefined kinds of constraints that can be associated to a FeatureLink or VariationGroup.

5 VehicleFeatureModeling

5.1 Overview

At the highest abstraction level, i.e., the Vehicle Level, EAST-ADL provides support for classification and definition of product lines (the entire vehicle for a car maker or some of its sub-systems for suppliers). The different possible configurations of the embedded electronic architecture are captured on a high abstraction level in terms of features. A feature in this sense is a characteristic or trait that individual variants of the vehicle may or may not have.

The specification of the features themselves, together with their forms of realization, the dependencies between them, and the requirements to be respected for their realization is performed at the Vehicle Level and it should be done independently of any product line. This would be the basis for a consistent reuse of features in different product lines and projects. At this level, a feature represents particular high-level requirements to be realized in all product line members that respect some conditions, e.g., US cars with elegance trim and engine size higher than 2.4.

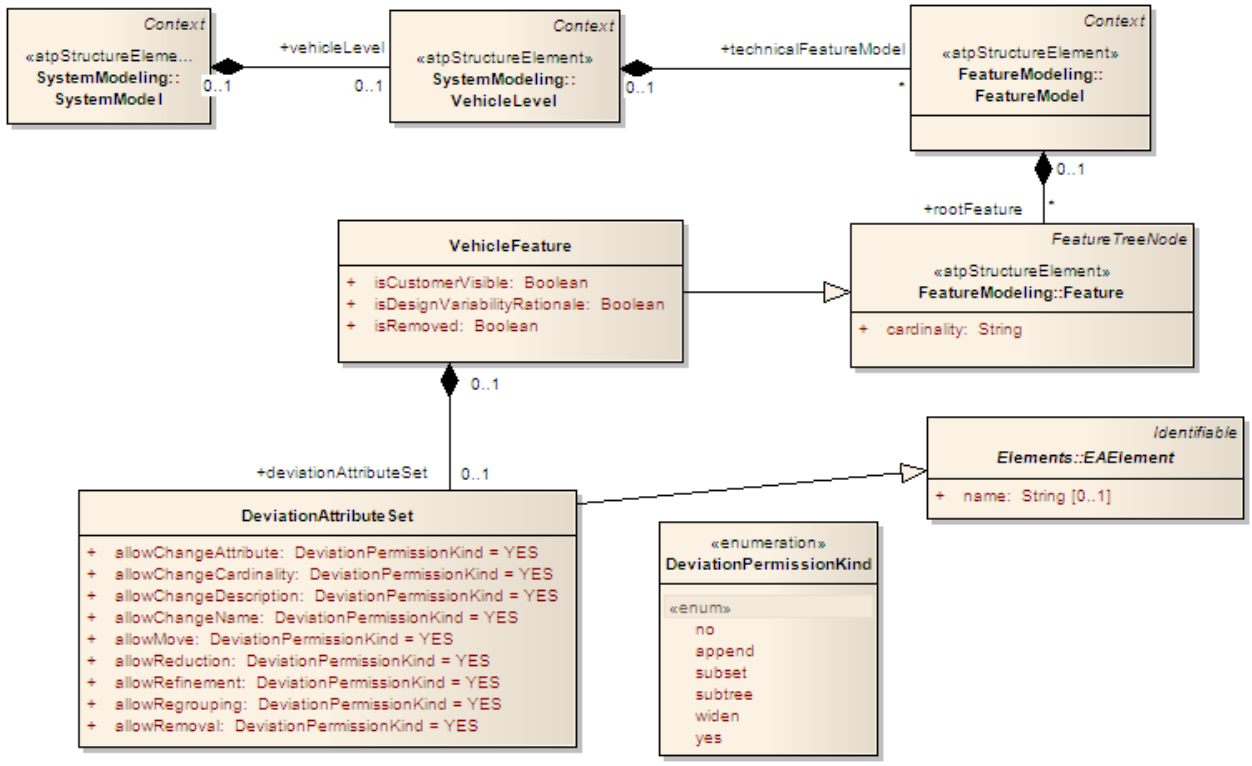

Figure 5. Diagram for VehicleFeatureModeling.

5.2 Element Descriptions

5.2.1 DeviationAttributeSet (from VehicleFeatureModeling)

Generalizations

- EAElement (from Elements)

Description

DeviationAttributeSet specifies the set of rules of allowed deviations from the reference model in a referring model. These rules are important, because they make sure that the different FeatureModels, referring to one reference model, follow specific rules for deviation, so a later integration into one FeatureModel may be possible.

Attributes

- allowChangeAttribute : DeviationPermissionKind = YES [1]
  This rule sets whether and how the VehicleFeature attributes may be changed. Allowed values: no, append, yes.
- allowChangeCardinality : DeviationPermissionKind = YES [1]
  This rule sets whether and how the VehicleFeature cardinality (i.e. variability of the VehicleFeature) may be changed. Allowed values: no, subset, yes.
- allowChangeDescription : DeviationPermissionKind = YES [1]
  This rule sets whether and how the VehicleFeature description may be changed. Allowed values: no, append, yes.
- allowChangeName : DeviationPermissionKind = YES [1]
  This rule sets whether and how the VehicleFeature name may be changed. Allowed values: no, append, yes.
- allowMove : DeviationPermissionKind = YES [1]
  This rule sets whether and how the VehicleFeature may be moved to another place in the feature diagram. Allowed values: no, subtree, yes.
- allowReduction : DeviationPermissionKind = YES [1]
  This rule sets if the reference feature may have a child without a corresponding referring feature among the children of the referring feature. Allowed values: no, subtree, yes.
- allowRefinement : DeviationPermissionKind = YES [1]
  This rule sets whether and how adding may be done of a child feature (without a corresponding feature in the reference model). Allowed values: no, yes.
- allowRegrouping : DeviationPermissionKind = YES [1]
  This rule sets whether and how the immediate child features of the VehicleFeature are allowed to be regrouped (i.e. creation or deletion of FeatureGroups below the respective VehicleFeature). Allowed values: no, widen, yes.
- allowRemoval : DeviationPermissionKind = YES [1]
  This rule sets if the feature in the referring model (compared to the reference model) may be deleted. Allowed values: no, yes.

Associations

No additional associations

Constraints

No additional constraints

Semantics

See description.

5.2.2 DeviationPermissionKind (from VehicleFeatureModeling) «enumeration»

Generalizations

None

Description

The DeviationPermissionKind is an enumeration with enumeration literals defining possible values for deviation attributes.

Enumeration Literals

- append
  The name, description or other attribute may only be changed by appending text without changing the original text. This kind is only applicable to deviation attributes "allowChangeName", "allowChangeDescription" and "allowChangeAttribute".
- no
  The deviation is not allowed.
- subset
  The cardinality may only be changed such that the new cardinality is a subset of the original cardinality. This kind is only applicable to deviation attribute "allowChangeCardinality".
- subtree
  In case of deviation attribute "allowMove": the parent of the VehicleFeature may be changed, but the original parent must remain a predecessor (i.e. moving the VehicleFeature itself is allowed but it may only be moved further down within the same subtree).

  In case of deviation attribute "allowReduction": the children of the VehicleFeature may be moved elsewhere, but they must remain successors of the VehicleFeature (i.e. moving them away is allowed but they may only be moved further down within the same subtree).

  This kind is only applicable to deviation attributes "allowMove" and "allowReduction".
- widen
  Feature groups may only be widened, i.e. it is only legal to add features into a feature group that were not grouped before, but not to ungroup features. This kind is only applicable to deviation attribute 'allowRegrouping'.
- yes
  The deviation is allowed.

Associations

No additional associations

Constraints

No additional constraints

Semantics

See description.

5.2.3 VehicleFeature (from VehicleFeatureModeling)

Generalizations

- Feature (from FeatureModeling)

Description

VehicleFeature represents a special kind of feature intended for use on Vehicle Level. The main difference to features in general is that they provide support for the multi-level concept (via their DeviationAttributeSet) and several additional attributes with meta-information specific to the vehicle level viewpoint.

Attributes

- isCustomerVisible : Boolean [1]
  This attribute states whether the VehicleFeature is customer visible (in contrast to a VehicleFeature that is e.g. technically driven).

  VehicleFeatures describe the system's characteristics on the level of the complete system and on a high abstraction level but they can still have a strong technical viewpoint. Therefore, they are usually not suitable for being directly presented to the end-customer. There are two approaches to deal with this situation.

  (1) The simple approach uses this attribute to denote those VehicleFeatures that are suitable for immediate end-customer configuration: if this attribute is set to true, then the feature will be directly presented to the end-customer for selection or deselection; if set to false, then the feature will be hidden from the end-customer and is thus reserved for internal configuration.

  (2) The more sophisticated approach is to define a dedicated product feature model to capture the customer viewpoint (available in the variability extension) in addition to the technical feature model on Vehicle Level and to provide a configuration decision model that maps configurations of this end-customer-oriented product feature model to the core technical feature model on Vehicle Level. This approach is much more flexible because the customer-view on the product line's variability can be structured freely and independently from the core technical feature model; furthermore, this approach can cope much better with evolution because the end-customer-oriented feature model can be evolved independently of the core technical feature model (and vice versa). When applying this second approach, this attribute isCustomerVisible will no longer be used, i.e., its value will be ignored.

  The simple approach #1 is suitable for simple product line scenarios. Approach #2 should be used for complex scenarios with large core technical feature models and/or longer evolution periods of the overall product line infrastructure.

- isDesignVariabilityRationale : Boolean [1]
  A VehicleFeature marked as a design variability rationale captures a variant showing up on a concrete artifact level that needs to be modeled on the Vehicle Level as well, in order to be directly available for immediate configuration on Vehicle Level. It is, from the abstraction layer's point of view, not a true Vehicle Level feature.

  If true, then isCustomerVisible is usually false but there may be rare exceptions.

- isRemoved : Boolean [1]
  This attribute describes if the VehicleFeature is removed (but kept in the database for tracking of evolution, which is required by the multi-level concept).

Associations

- deviationAttributeSet : DeviationAttributeSet [0..1] {composite}
  Possible deviation attributes included in the VehicleFeature. If the VehicleFeature is part of a reference feature model in the context of multi-level feature models, the attribute can constrain the allowed deviations for the respective referring features.

Constraints

[1] VehicleFeatures can only be contained in FeatureModels on VehicleLevel.

Semantics

A VehicleFeature is a functional or non-functional characteristic, constraint or property that can be present or not in a vehicle product line on the level of the complete system, i.e. vehicle.

6 FunctionModeling

6.1 Overview

The function modeling is performed in the FunctionalAnalysisArchitecture (in the AnalysisLevel) and the FunctionalDesignArchitecture (in the DesignLevel). The root component of the function compositional hierarchy on AnalysisLevel is the FunctionalAnalysisArchitecture (FAA); the root component of the function compositional hierarchy on DesignLevel is the FunctionalDesignArchitecture (FDA), see the diagram for SystemModeling.

The main modeling concept applied here is functional component modeling: Functions interact with one another via ports that are connected by connectors owned by the composing function. Occurrences of functions are modeled by typed prototypes in the composing function. These occurrences are typed by types. This naming convention of the type-prototype pattern is from AUTOSAR, however the concept of types and typed elements is also available in e.g. UML2.

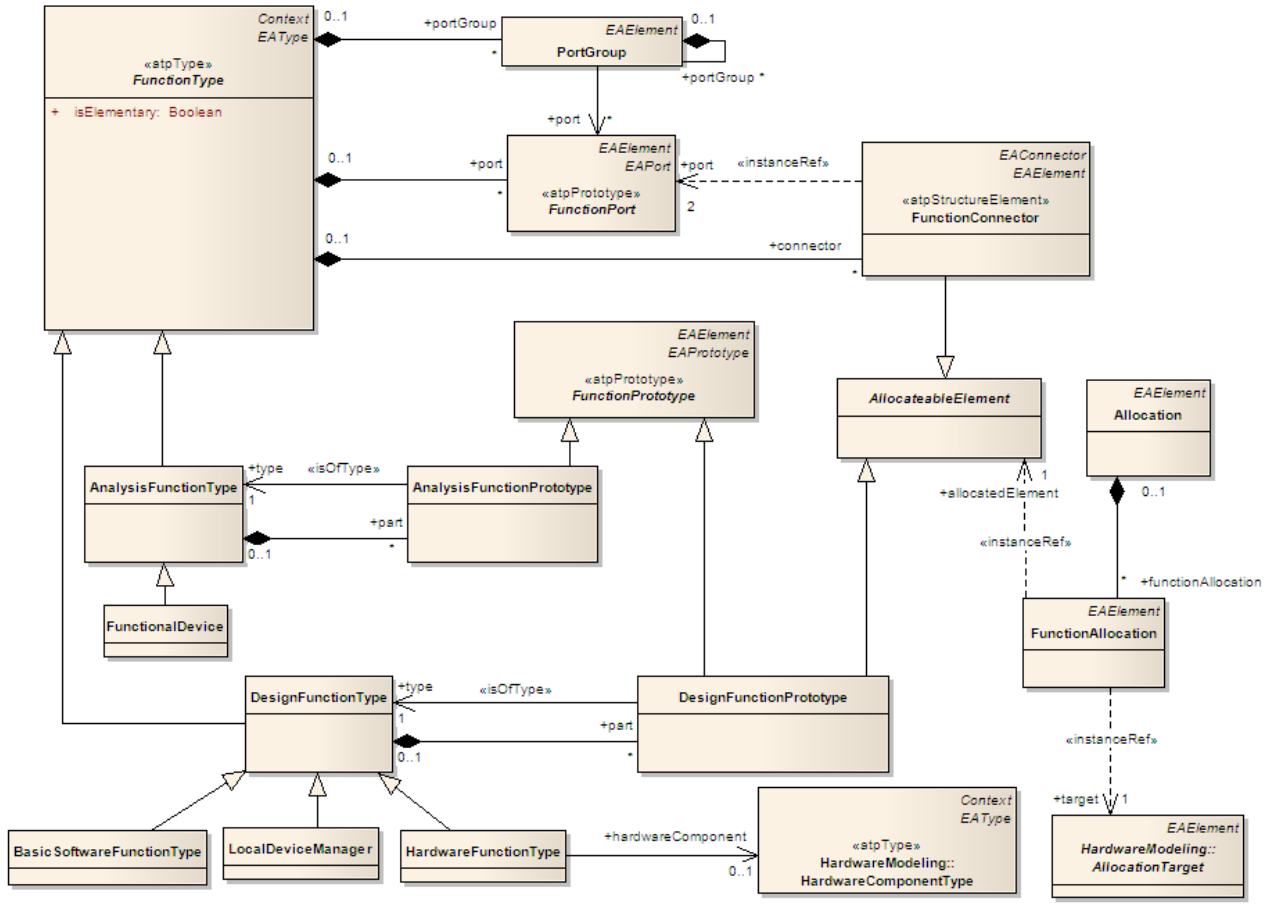

Figure 6. Diagram for FunctionModeling showing the concepts for function modeling at different abstraction levels, elements in the DesignLevel are allocateable on elements in the hardware design architecture.

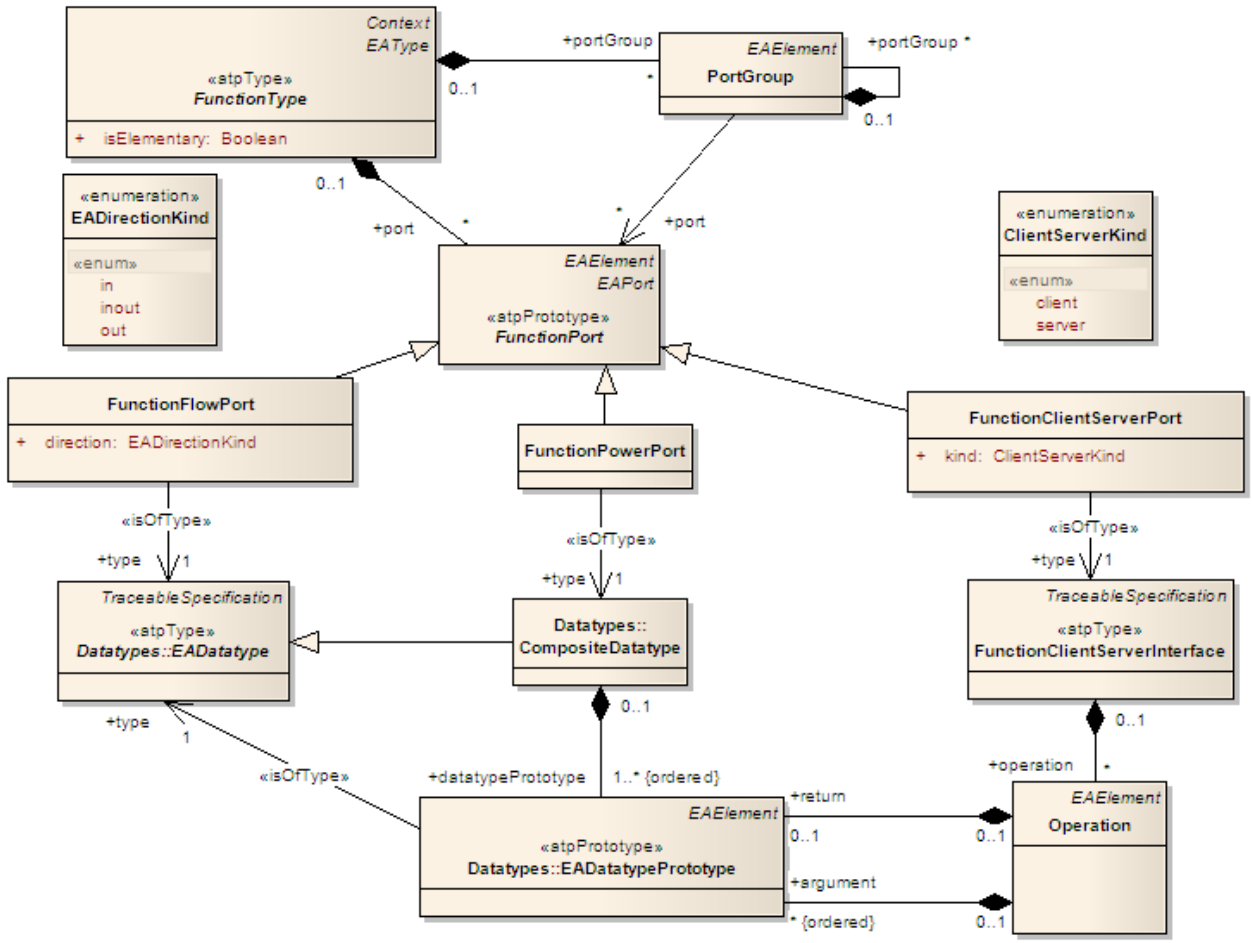

Figure 7. Diagram for FunctionPorts and their respective typing.

6.2 Element Descriptions

6.2.1 AllocateableElement (from FunctionModeling) {abstract}

Generalizations

None

Description

The AllocateableElement is an abstract superclass for elements that are allocateable.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The AllocateableElement abstracts all elements that are allocateable.

Subclasses of the abstract class AllocateableElement add their own semantics.

6.2.2 Allocation (from FunctionModeling)

Generalizations

- EAElement (from Elements)

Description

The Allocation element contains function allocations. It can bundle function allocations that belong together, e.g., all function allocations for a simulation.

Attributes

No additional attributes

Associations

- functionAllocation : FunctionAllocation [*] {composite}
  The owned FunctionAllocations.

Constraints

No additional constraints

Semantics

The Allocation element contains function allocations, i.e., it can bundle function allocations that belong together.

6.2.3 AnalysisFunctionPrototype (from FunctionModeling)

Generalizations

- FunctionPrototype (from FunctionModeling)

Description

The AnalysisFunctionPrototype represents references to the occurrence of the AnalysisFunctionType that types it when it acts as a part.

The AnalysisFunctionPrototype is typed by an AnalysisFunctionType.

Attributes

No additional attributes

Associations

- type : AnalysisFunctionType [1]
  «isOfType»
  The type that defines this AnalysisFunctionPrototype.

Constraints

No additional constraints

Semantics

The AnalysisFunctionPrototype represents an occurrence of the AnalysisFunctionType that types it.

6.2.4 AnalysisFunctionType (from FunctionModeling)

Generalizations

- FunctionType (from FunctionModeling)

Description

The AnalysisFunctionType is a concrete FunctionType and therefore inherits the elementary function properties from the abstract metaclass FunctionType. The AnalysisFunctionType is used to model the functional structure on AnalysisLevel. The syntax of AnalysisFunctionTypes is inspired from the type-prototype pattern used by AUTOSAR.

The AnalysisFunctions may interact with other AnalysisFunctions (i.e., also FunctionalDevices) through their FunctionPorts.

Furthermore, an AnalysisFunction may be decomposed into contained parts that are AnalysisFunctionPrototypes. This allows the functionalities provided by the parent AnalysisFunction to be broken up hierarchically into subfunctionalities.

A FunctionBehavior may be associated with each AnalysisFunction. In the case where the AnalysisFunction is decomposed, the behavior is a specification for the composed behavior of the parts.

Attributes

No additional attributes

Associations

- part : AnalysisFunctionPrototype [*] {composite}
  The parts contained in this AnalysisFunctionType.

Constraints

[1] AnalysisFunctionTypes may only be used on AnalysisLevel.

Semantics

The AnalysisFunctionType represents a node in a tree structure corresponding to the functional decomposition of a top level AnalysisFunction. The AnalysisFunction represents the analysis function used to describe the functionalities provided by a vehicle on the AnalysisLevel. At the AnalysisLevel, AnalysisFunctions are defined and structured according to the functional requirements, i.e., the functionalities provided to the user.

6.2.5 BasicSoftwareFunctionType (from FunctionModeling)

Generalizations

- DesignFunctionType (from FunctionModeling)

Description

The BasicSoftwareFunctionType allow for the representation of a layered architecture of functionality on the DesignLevel. A BasicSoftwareFunctionType then represents a function in the service layer.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The BasicSoftwareFunctionType is an abstraction of the middleware.

6.2.6 ClientServerKind (from FunctionModeling) «enumeration»

Generalizations

None

Description

This element is an enumeration for the kind of the FunctionClientServerPort, which can either be a "client" or a "server".

Enumeration Literals

- client
- server

Associations

No additional associations

Constraints

No additional constraints

Semantics

The ClientServerKind is an enumeration with literals that are used to distinguish between client and server.

6.2.7 DesignFunctionPrototype (from FunctionModeling)

Generalizations

- AllocateableElement (from FunctionModeling)
- FunctionPrototype (from FunctionModeling)

Description

The DesignFunctionPrototype represents references to the occurrence of the DesignFunctionType that types it when it acts as a part.

The DesignFunctionPrototype is typed by a DesignFunctionType .

Attributes

No additional attributes

Associations

- type : DesignFunctionType [1]
  «isOfType»
  The type that defines this DesignFunctionPrototype.

Constraints

No additional constraints

Semantics

The DesignFunctionPrototype represents an occurrence of the DesignFunctionType that types it.

6.2.8 DesignFunctionType (from FunctionModeling)

Generalizations

- FunctionType (from FunctionModeling)

Description

The DesignFunctionType is a concrete FunctionType and therefore inherits the elementary function properties from the abstract metaclass FunctionType. The DesignFunctionType is used to model the functional structure on DesignLevel. The syntax of DesignFunctionTypes is inspired by the type-prototype pattern used by AUTOSAR.

The DesignFunctions may interact with other DesignFunctions (i.e., also BasicSoftwareFunctions, HardwareFunctions, and LocalDeviceManagers) through their FunctionPorts.

Furthermore, a DesignFunction may be decomposed into the contained parts that are DesignFunctionPrototypes. This allows the functionalities provided by the parent DesignFunction to be broken up hierarchically into subfunctionalities.

Execution time constraints on the DesignFunctionType can be expressed by ExecutionTimeConstraints, see the Timing package.

If two or more occurrences of an elementary Function are allocated on the same ECU, the code will be placed on the ECU only once (so these occurrences will use the same code but separate memory areas for data).

Attributes

No additional attributes

Associations

- part : DesignFunctionPrototype [*] {composite}
  The parts contained in this DesignFunctionType.

Constraints

[1] DesignFunctionTypes may only be used on DesignLevel.

Semantics

The DesignFunctionType represents a node in a tree structure corresponding to the functional decomposition of a top level DesignFunction. The DesignFunction represents the design function used to describe the functionalities provided by a vehicle on the DesignLevel. At the DesignLevel, DesignFunctions are defined and structured according to the functional and hardware system design.

6.2.9 EADirectionKind (from FunctionModeling) «enumeration»

Generalizations

None

Description

This element is an enumeration for the direction of a Port, which can either be "in", "out", or "inout".

Enumeration Literals

- in
- inout
- out

Associations

No additional associations

Constraints

No additional constraints

Semantics

The EADirectionKind is an enumeration with literals describing the direction of ports.

6.2.10 FunctionalDevice (from FunctionModeling)

Generalizations

- AnalysisFunctionType (from FunctionModeling)

Description

The FunctionalDevice represents an abstract sensor or actuator that encapsulates sensor/actuator dynamics and the interfacing software. The FunctionalDevice is the interface between the electronic architecture and the environment (connected by ClampConnectors, see the Environment chapter). As such, it is a transfer function between the AnalysisFunction and the physical entity that it measures or actuates.

A Realization dependency can be used for traceability from LocalDeviceManagers in the DesignLevel and Sensors/Actuators in the hardware design architecture that are represented by the FunctionalDevice.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints.

Semantics

The behavior associated with the FunctionalDevice is the transfer function between the environment model representing the environment and an AnalysisFunction. The transfer function represents the sensor or actuator and its interfacing hardware and software (connectors, electronics, in/out interface, driver software, and application software).

6.2.11 FunctionAllocation (from FunctionModeling)

Generalizations

- EAElement (from Elements)

Description

FunctionAllocation represents an allocation constraint binding an AllocateableElement (computation functions or communication connectors) on an AllocationTarget (computation or communication resource).

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- allocatedElement : AllocateableElement [1]
  «instanceRef»
- target : AllocationTarget [1]
  «instanceRef»

Constraints

No additional constraints

Semantics

AllocationTarget is specialized by HardwareComponentPrototype in the HardwareModeling package and AllocateableElement is specialized by the concrete elements DesignFunctionPrototype and FunctionConnector in the FunctionModeling package.

6.2.12 FunctionClientServerInterface (from FunctionModeling) «atpType»

Generalizations

- TraceableSpecification (from Elements)

Description

The FunctionClientServerInterface is used to specify the operations in FunctionClientServerPorts.

Attributes

No additional attributes

Associations

- operation : Operation [*] {composite}
  The owned Operation.

Constraints

No additional constraints

Semantics

The operations of the FunctionClientServerInterface are required or provided through the FunctionClientServerPorts typed by the FunctionClientServerInterface.

6.2.13 FunctionClientServerPort (from FunctionModeling)

Generalizations

- FunctionPort (from FunctionModeling)

Description

The FunctionClientServerPort is a FunctionPort for client-server interaction. A number of FunctionClientServerPorts of clientServerType "client" can be connected to one FunctionClientServerPort of clientServerType "server", i.e. when connected the multiplicity for the connection is n to 1 for client and server.

Attributes

- kind : ClientServerKind [1]

Associations

- type : FunctionClientServerInterface [1]
  «isOfType»
  The interface of this FunctionClientServerPort.

Constraints

[1] A FunctionClientServerPort of clientServerType "client" can only be connected to one FunctionClientServerPort of clientServerType "server".

Semantics

The FunctionClientServerPort is a FunctionPort for client-server interaction.

FunctionClientServerPorts are single buffer overwrite and nonconsumable.

6.2.14 FunctionConnector (from FunctionModeling) «atpStructureElement»

Generalizations

- AllocateableElement (from FunctionModeling)
- EAConnector (from Elements)
- EAElement (from Elements)

Description

The FunctionConnector indicates that the connected FunctionPorts exchange signals or client-server requests/responses.

A FunctionConnector used to connect ports of parts within a FunctionType are called assembly connecors. A FunctionConnector between a port of a part and a port of the FunctionType itself is called a delegation connector.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- port : FunctionPort [2]
  «instanceRef»

Constraints

[1] Can connect two FunctionFlowPorts of different directions when this is an assembly FunctionConnector.

[2] Can connect two FunctionFlowPorts of the same direction when this is a delegation FunctionConnector.

[3] Can connect two ClientServerPorts of different kinds when this is an assembly FunctionConnector.

[4] Can connect two ClientServerPorts of the same kind when this is a delegation FunctionConnector.

[5] Can connect two FunctionFlowPorts with direction inout.

Semantics

The FunctionConnector connects a pair of FunctionFlowPorts or FunctionClientServerPorts. If two FunctionFlowPorts are connected, data elements of the type of the output FunctionFlowPort flow from the output FunctionFlowPort to the input FunctionFlowPort. If FunctionClientServerPorts are connected, the client calls the server according to the operations of the interfaces.

The FunctionPrototype with the connected port has to be identified by the FunctionConnector as well.

The FunctionConnector is normally routed according to the hardware topology and the allocation of source and destination. If there are redundant paths, a FunctionAllocation may be used to prescribe allocation.

6.2.15 FunctionFlowPort (from FunctionModeling)

Generalizations

- FunctionPort (from FunctionModeling)

Description

The FunctionFlowPort is a metaclass for flowports, inspired by the SysML FlowPort.

Attributes

- direction : EADirectionKind [1]

Associations

- type : EADatatype [1]
  «isOfType»
  The single EADatatype for this port.
- defaultValue : EAValue [0..1] {composite}

Constraints

No additional constraints

Semantics

FunctionFlowPorts are single buffer overwrite and nonconsumable.

FunctionFlowPorts can be connected if their FunctionPort signatures match; i.e.:

EADatatypes that are ValueTypes are compatible if

* They have the same "dimension".

* They have the same "unit".

EADatatypes that are RangeableValueTypes are compatible if

* The source EADatatype has the same or better "accuracy".

* They have the same baseRangeable.

* The source EADatatype has the same or smaller "maxValue".

* The source EADatatype has the same or higher "minValue".

* The source EADatatype has the same or higher "resolution".

* They have the same "significantDigits".

EADatatypes that are EnumerationValueTypes are compatible if

* They have the same baseEnumeration.

A FunctionFlowPort with direction=in is called an input FunctionFlowPort:

The input FunctionFlowPort indicates that the containing Function requires input data. The EADatatype of this data is defined by the associated EADatatype. The data is sampled at the invocation of the containing entity for discrete Functions. For continuous Functions, the input FunctionFlowPort represents a continuous input connection point.

The input FunctionFlowPort declares a reception point of data. It represents a single element buffer, which is overridden with the latest data. The type of the data is defined by the associated EADatatype.

A FunctionFlowPort with direction=out is called an output FunctionFlowPort:

The output FunctionFlowPort indicates that the containing Function provides output data. The EADatatype of this data is defined by the associated EADatatype. The data is sent at the completion of the containing entity for discrete Functions. For continuous Functions, the output FunctionFlowPort represents a (time-)continuous output connection point.

The output FunctionFlowPort declares a transmission point of data. The type of the data is defined by the associated EADatatype.

6.2.16 FunctionPort (from FunctionModeling) {abstract} «atpPrototype»

Generalizations

- EAPort (from Elements)
- EAElement (from Elements)

Description

The ports conserve variables for component interaction.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Subclasses of the abstract class FunctionPort add their own semantics.

6.2.17 FunctionPowerPort (from FunctionModeling)

Generalizations

- FunctionPort (from FunctionModeling)

Description

The FunctionPowerPort is a FunctionPort for denoting the physical interactions between environment and sensing/actuation functions.

Attributes

No additional attributes

Associations

- type : CompositeDatatype [1]
  «isOfType»
  The Datatype for the flow physical variables of this FunctionPowerPort, specifying the Across and Through variables with two separate datatypePrototypes.

Constraints

[1] The owner of a FunctionPowerPort is either a FunctionalDevice, a HardwareFunctionType, or a FunctionType for environment

[2] Two connected FunctionPowerPort must have the same Datatype.

[3] The typing Datatype shall have two datatypePrototypes called Across and Through, with Datatypes that are consistent and representing the variables of the PowerPort.

Semantics

The FunctionPowerPort conserves physical variables in a dynamic process.

The typing CompositeDatatype owns two EADatatypePrototypes called Across and Through, representing the exchanged physical variables of the FunctionPowerPort. In two or more directly connected function power ports, the Across variables always get the same value and the Through variables always sum up to zero.

6.2.18 FunctionPrototype (from FunctionModeling) {abstract} «atpPrototype»

Generalizations

- EAElement (from Elements)
- EAPrototype (from Elements)

Description

FunctionPrototype represents a reference to the occurrence of a FunctionType when it acts as a part.

A concrete specialization of the FunctionPrototype is typed by a concrete specialization of FunctionType.

FunctionTrigger in the Behavior package is associated with a FunctionPrototype.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The FunctionPrototype is an abstract concept with concrete specializations for the use on the AnalysisLevel and DesignLevel.

6.2.19 FunctionType (from FunctionModeling) {abstract} «atpType»

Generalizations

- EAType (from Elements)
- Context (from Elements)

Description

The abstract metaclass FunctionType abstracts the function component types that are used to model the functional structure, which is distinguished from the implementation of component types using AUTOSAR. The syntax of FunctionTypes is inspired from the concept of Block from SysML.

FunctionBehavior and FunctionTrigger in the Behavior package are associated to a FunctionType.

Attributes

- isElementary : Boolean [1]
  True, when this type must not have any parts.

Associations

- port : FunctionPort [*] {composite}
  Owned ports.
- connector : FunctionConnector [*] {composite}
  The connectors that connect ports of parts as assembly connectors or ports of this type and ports of parts as delegation connectors.
- portGroup : PortGroup [*] {composite}
  Grouping of ports owned by this element.

Constraints

[1] Elementary FunctionTypes shall not have parts.

Semantics

The FunctionType abstracts the function component types that are used to model the functional structure on AnalysisLevel and DesignLevel.

Leaf functions of an EAST-ADL function hierarchy are called elementary Functions.

Elementary Functions have synchronous execution semantics:

1. Read inputs

2. Execute (duration: Execution time)

3. Write outputs

Execution is defined by a behavior that acts as a transfer function.

Subclasses of the abstract class FunctionType add their own semantics.

If a behavior is attached to the FunctionType, the execution semantic for a discrete elementary FunctionType complies with the run-to-completion semantic. This has the following implications:

1. Input that arrives at the input FunctionPorts after execution begins will be ignored until the next execution cycle.

2. If more than one input value arrives per FunctionPort before execution begins, the last value will override all previous ones in the public part of the input FunctionPort (single element buffers for input).

3. The local part of a FunctionPort does not change its value during execution of the behavior.

4. During an execution cycle, only one output value can be sent per FunctionPort. If consecutive output values are produced on the same FunctionPort during a single execution cycle, the last value will override all previous ones on the output FunctionPort (single element buffers for output).

5. Output will not be available at an output FunctionPort before execution ends.

6. Elementary FunctionTypes may not produce any side effects (i.e., all data passes the FunctionPorts).

6.2.20 HardwareFunctionType (from FunctionModeling)

Generalizations

- DesignFunctionType (from FunctionModeling)

Description

The HardwareFunctionType is the transfer function for the identified HardwareComponentType or a specification of an intended transfer function. HardwareFunctionType types DesignFunctionPrototypes in the FunctionalDesignArchitecture. The ports of such DesignFunctionPrototypes are typically connected to a plant model with ClampConnectors.

DesignFunctionPrototypes typed by HardwareFunctionType may be allocated to HardwareComponents in which case the HardwareFunctionType must match the HardwareFunctionType of the target HardwareComponent. Typically, the same HardwareFunctionType types the prototype that is allocated to its target HardwareComponent.

HardwareFunctionTypes are typically transfer functions of sensors, actuators, amplifiers and other peripherals with a fixed transfer function. Thus, HardwareFunctionTypes are generally not defined for ECUNodes.

Attributes

No additional attributes

Associations

- hardwareComponent : HardwareComponentType [0..1]
  The HardwareComponentType with the specified HardwareFunction.

Constraints

[1] A DesignFunctionPrototype typed by a HardwareFunctionType shall be connected to the EnvironmentModel via ClampConnectors and to BSWFunctions via FunctionConnectors.

[2] A DesignFunctionPrototype typed by a HardwareFunctionType may only contain prototypes typed by HardwareFunctionType.

Semantics

The HardwareFunctionType is the transfer function for the associated hardware components such as sensors, actuators, amplifiers, etc or a specification of an intended transfer function.

A DesignFunctionPrototype typed by a HardwareFunctionType allocated to Sensors or Actuators is the interfacing element to the plant model.

6.2.21 LocalDeviceManager (from FunctionModeling)

Generalizations

- DesignFunctionType (from FunctionModeling)

Description

The LocalDeviceManager represents a DesignFunction that act as a manager or functional interface to Sensors, Actuators and other devices. It is responsible for translating between the electrical/logical interface of the device, as provided by a BasicSoftwareFunction, and the physical interface of the device. For example, consider a temperature sensor with voltage output. The HardwareFunctionType defines the transfer from temperature to voltage. A BasicSoftwareFunction relays the voltage from the microcontroller's I/O. The role of the LocalDeviceManager is now to translate from voltage to temperature value, taking into account the sensor's characteristics such as nonlinearities, calibration, etc. The resulting temperature is available to the other DesignFunctions. By separating the device specific part from the middleware and ECU specific parts, it is possible to systematically change interface function together with the device.

The role of the LocalDeviceManager is to act as an interface between the electrical output of Sensors or electrical input of Actuators and the physical quantity of those devices.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] A DesignFunctionPrototype typed by a LocalDeviceManager shall be allocated to the same ECU node as the device that it manages is connected to.

[2] A LocalDeviceManager shall interface BSWFunctions and DesignFunctions.

Semantics

The LocalDeviceManager encapsulates the device-specific or functional parts of a Sensor or Actuator, device, etc. interface.

6.2.22 Operation (from FunctionModeling)

Generalizations

- EAElement (from Elements)

Description

The Operation is the provided/required operation of a FunctionClientServerInterface. It can specify its return values and arguments by EADatatypePrototypes.

Attributes

No additional attributes

Associations

- argument : EADatatypePrototype [*] {ordered} {composite}
  The argument value of the Operation.
- return : EADatatypePrototype [0..1] {composite}
  The return value of the Operation.

Constraints

No additional constraints

Semantics

The Operation is the provided/required operation of a FunctionClientServerInterface.

6.2.23 PortGroup (from FunctionModeling)

Generalizations

- EAElement (from Elements)

Description

The PortGroup represents several FunctionPorts grouped into one. All FunctionPorts that are part of a PortGroup are graphically represented as a single FunctionPort. The PortGroup has no semantic meaning except that it makes graphical representation of the connected FunctionPorts easier to read, and provides a means to logically organize several FunctionPorts into one group.

Connectors are still connected to the contained FunctionPorts, but tool support may simplify connections by allowing semiautomatic or automatic connection to all FunctionPorts of a PortGroup.

Note that the term "PortGroup" is also used by AADL.

Attributes

No additional attributes

Associations

- port : FunctionPort [*]
  The grouped FunctionPorts.
- portGroup : PortGroup [*] {composite}
  Grouping of ports owned by this element.

Constraints

[1] The FunctionPorts in a PortGroup must all be of the same component; all FunctionPorts in a PortGroup must be of the same kind (FunctionFlowPort with same EADirectionKind or FunctionClientServerPort with same ClientServerKind).

Semantics

The PortGroup provides the means to organize FunctionPorts and FunctionConnectors. It does not add semantics. In the model, the FunctionPorts contained in the PortGroup are connected as individual FunctionPorts.

7 HardwareModeling

7.1 Overview

The package HardwareModeling contains the elements to model physical entities of the embedded electrical/electronic system. These elements allow the hardware to be captured in sufficient detail to allow preliminary allocation decisions.

The allocation decisions are based on requirements on timing, storage, data throughput, processing power, etc. that are defined in the Functional Analysis Architecture and the Functional Design Architecture.

Conversely, the Functional Analysis Architecture and the Functional Design Architecture may be revised based on analysis using information from the Hardware Design Architecture. An example is control law design, where algorithms may be modified for expected computational and communication delays. Thus, the Hardware Design Architecture contains information about properties in order to support, e.g., timing analysis and performance in these respects.

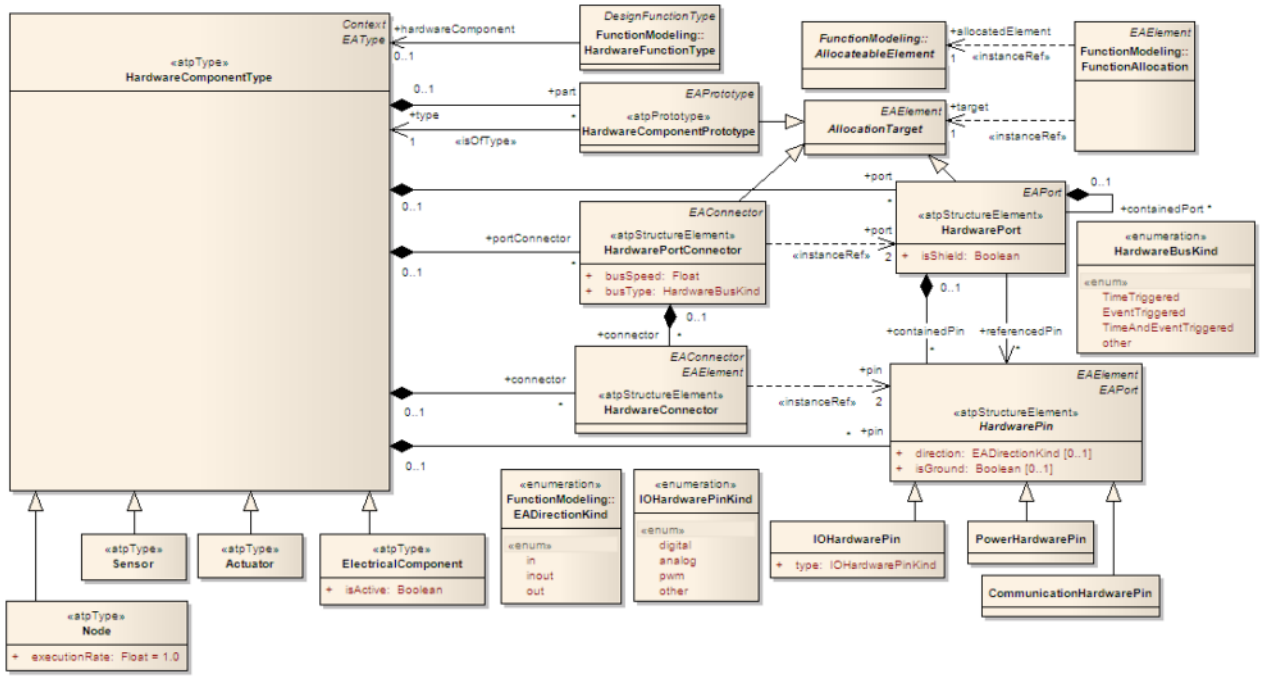

Figure 8. Diagram for HardwareModeling.

7.2 Element Descriptions

7.2.1 Actuator (from HardwareModeling)

Generalizations

- HardwareComponentType (from HardwareModeling)

Description

The Actuator is the element that represents electrical actuators, such as valves, motors, lamps, brake units, etc. Non-electrical actuators such as the engine, hydraulics, etc. are considered part of the plant model (environment). Plant models are not part of the Hardware Design Architecture.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The Actuator metaclass represents the physical and electrical aspects of actuator hardware. The logical aspect is represented by a HardwareFunctionType associated with the Actuator.

7.2.2 AllocationTarget (from HardwareModeling) {abstract}

Generalizations

- EAElement (from Elements)

Description

The AllocationTarget is a superclass for elements to which AllocateableElements can be allocated.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An AllocationTarget is a resource element in the Hardware Design Architecture which may host functional behaviors in the Functional Design Architecture.

7.2.3 CommunicationHardwarePin (from HardwareModeling)

Generalizations

- HardwarePin (from HardwareModeling)

Description

CommunicationHardwarePin represents an electrical connection point that can be used to define how the wire harness is logically defined.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The CommunicationHardwarePin represents the hardware connection point of a communication bus.

Depending on modeling style, one or two pins may be defined for a dual-wire bus.

7.2.4 ElectricalComponent (from HardwareModeling) «atpType»

Generalizations

- HardwareComponentType (from HardwareModeling)

Description

ElectricalComponent represents a hardware element as e.g. relays, batteries, capacitors and other non-computational, non-interactional (with plant) elements.

Attributes

- isActive : Boolean [1]
  Indicates if the PowerSupply is active or passive.

Associations

No additional associations

Constraints

No additional constraints

Semantics

ElectricalComponent may be active (e.g., a battery) or passive (main relay).

7.2.5 HardwareBusKind (from HardwareModeling) «enumeration»

Generalizations

None

Description

HardwareBusKind is an enumeration type representing different kinds of busses.

Enumeration Literals

- EventTriggered
  Bus is event-triggered
- other
  Another type of bus communication
- TimeAndEventTriggered
  Bus is both time and event-triggered
- TimeTriggered
  Bus is time-triggered

Associations

No additional associations

Constraints

No additional constraints

Semantics

HardwareBusKind represents the kind of a hardware connector as given by the definition of the respective Enumeration Literal.

7.2.6 HardwareComponentPrototype (from HardwareModeling) «atpPrototype»

Generalizations

- EAPrototype (from Elements)
- AllocationTarget (from HardwareModeling)

Description

Appears as part of a HardwareComponentType and is itself typed by a HardwareComponentType. This allows for a reference to the occurrence of a HardwareComponentType when it acts as a part. The purpose is to support the definition of hierarchical structures, and to reuse the same type of Hardware at several places. For example, a wheel speed sensor may occur at all four wheels, but it has a single definition.

Attributes

No additional attributes

Associations

- type : HardwareComponentType [1]
  «isOfType»

Constraints

No additional constraints

Semantics

The HardwareComponentPrototype represents an occurrence of a hardware element, according to the type of the HardwareComponentPrototype.

7.2.7 HardwareComponentType (from HardwareModeling) «atpType»

Generalizations

- EAType (from Elements)
- Context (from Elements)

Description

The HardwareComponentType represents a hardware element on an abstract level, allowing preliminary engineering activities related to hardware.

Attributes

No additional attributes

Associations

- portConnector : HardwarePortConnector [*] {composite}

- connector : HardwareConnector [*] {composite}
  Connectors owned by this element.
- part : HardwareComponentPrototype [*] {composite}
  Parts owned by this element.
- pin : HardwarePin [*] {composite}
  Hardware pins owned by this type.
- port : HardwarePort [*] {composite}

Constraints

No additional constraints

Semantics

The HardwareComponentType is a structural entity that defines a part of an electrical architecture. Through its ports it can be connected to electrical sources and sinks. Its logical behavior, the transfer function, may be defined in a HardwareFunctionType referencing the HardwareComponentType. This is typically connected through its ports to the environment model to participate in the end-to-end behavioral definition of a function.

7.2.8 HardwareConnector (from HardwareModeling) «atpStructureElement»

Generalizations

- EAConnector (from Elements)
- EAElement (from Elements)

Description

Hardware connectors represent wires that electrically connect the hardware components through its pins.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- pin : HardwarePin [2]
  «instanceRef»

Constraints

No additional constraints

Semantics

The connector joins the two referenced pins electrically.

7.2.9 HardwarePin (from HardwareModeling) {abstract} «atpStructureElement»

Generalizations

- EAPort (from Elements)
- EAElement (from Elements)

Description

HardwarePin represents electrical connection points in the hardware architecture. Depending on modeling style, the actual wire or a logical connection can be considered.

Attributes

- direction : EADirectionKind [0..1]
  The direction of current through the pin.
- isGround : Boolean [0..1]
  Indicates that the pin is connected to ground.

Associations

No additional associations

Constraints

No additional constraints

Semantics

Hardware pin represents an electrical connection point.

7.2.10 HardwarePort (from HardwareModeling) «atpStructureElement»

Generalizations

- AllocationTarget (from HardwareModeling)
- EAPort (from Elements)

Attributes

- isShield : Boolean [1]
  True if this port is representing the shield.

Associations

- containedPin : HardwarePin [*] {composite}
- referencedPin : HardwarePin [*]
- containedPort : HardwarePort [*] {composite}

Constraints

No additional constraints

Semantics

-

7.2.11 HardwarePortConnector (from HardwareModeling) «atpStructureElement»

Generalizations

- AllocationTarget (from HardwareModeling)
- EAConnector (from Elements)

Attributes

- busSpeed : Float [1]
- busType : HardwareBusKind [1]

Associations

- connector : HardwareConnector [*] {composite}

Dependencies

- port : HardwarePort [2]
  «instanceRef»

Constraints

No additional constraints

Semantics

-

7.2.12 IOHardwarePin (from HardwareModeling)

Generalizations

- HardwarePin (from HardwareModeling)

Description

IOHardwarePin represents an electrical connection point for digital or analog I/O.

Attributes

- type : IOHardwarePinKind [1]
  kind defines whether the IOHardwarePort is digital, analog or PWM (Pulse Width Modulated).

Associations

No additional associations

Constraints

No additional constraints

Semantics

The IOHardwarePin represents an electrical pin or connection point.

7.2.13 IOHardwarePinKind (from HardwareModeling) «enumeration»

Generalizations

None

Description

IOHardwarePinKind is an enumeration type representing different kinds of I/O Hardware Ports.

Enumeration Literals

- analog
  I/O with varying amplitude.
- digital
  I/O with fixed amplitude.
- other
  Another type of I/O port.
- pwm
  PWM (Pulse Width Modulated) modulated I/O, i.e. a signal with fixed frequency and amplitude but varying duty cycle.

Associations

No additional associations

Constraints

No additional constraints

Semantics

IOHardwarePinKind represents the kind of IOHardwarePin as given by the definition of the respective Enumeration Literal.

7.2.14 Node (from HardwareModeling)

Generalizations

- HardwareComponentType (from HardwareModeling)

Description

Node represents the computer nodes of the embedded electrical/electronic system. Nodes consist of processor(s) and may be connected to sensors, actuators and other ECUs via a BusConnector.

Node denotes an electronic control unit that acts as a computing element executing Functions. In case a single CPU ECU is represented, it is sufficient to have a single, non-hierarchical Node.

Attributes

- executionRate : Float = 1.0 [1]
  ExecutionRate is used to compute an approximate execution time. A nominal execution time divided by executionRate provides the actual execution time to be used e.g. for timing analysis in feasibility studies.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The Node element represents an ECU, i.e. an Electronic Control Unit, and an allocation target of FunctionPrototypes.

The Node executes its allocated FunctionPrototypes at the specified executionRate. The executionRate denotes how many execution seconds of an allocated functionPrototype´s execution time are processed in each real-time second. Actual execution time is thus found by dividing the parameters of the ExecutionTimeConstraint with executionRate.

Example: If an ECU is 25% faster than a standard ECU (e.g., in a certain context, execution times are given assuming a nominal speed of 100 MHz; our CPU is then 125 MHz), the executionRate is 1.25. An execution time of 5 ms would then become 4 ms on this ECU.

7.2.15 PowerHardwarePin (from HardwareModeling)

Generalizations

- HardwarePin (from HardwareModeling)

Description

PowerHardwarePin represents a pin that is primarily intended for power supply, either providing or consuming energy.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

A PowerHardwarePin is primarily intended to be a power supply. The direction attribute of the pin defines whether it is providing or consuming energy.

7.2.16 Sensor (from HardwareModeling)

Generalizations

- HardwareComponentType (from HardwareModeling)

Description

Sensor represents a hardware entity for digital or analog sensor elements. The Sensor is connected electrically to the electrical entities of the Hardware Design Architecture.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Sensor denotes an electrical sensor. The Sensor represents the physical and electrical aspects of sensor hardware. The logical aspect is represented by a HardwareFunctionType associated with the Sensor.

8 Environment

8.1 Overview

The Environment model is used to describe the environment of the vehicle electric and electronic architecture. It is modeled by continuous functions representing the system environment.

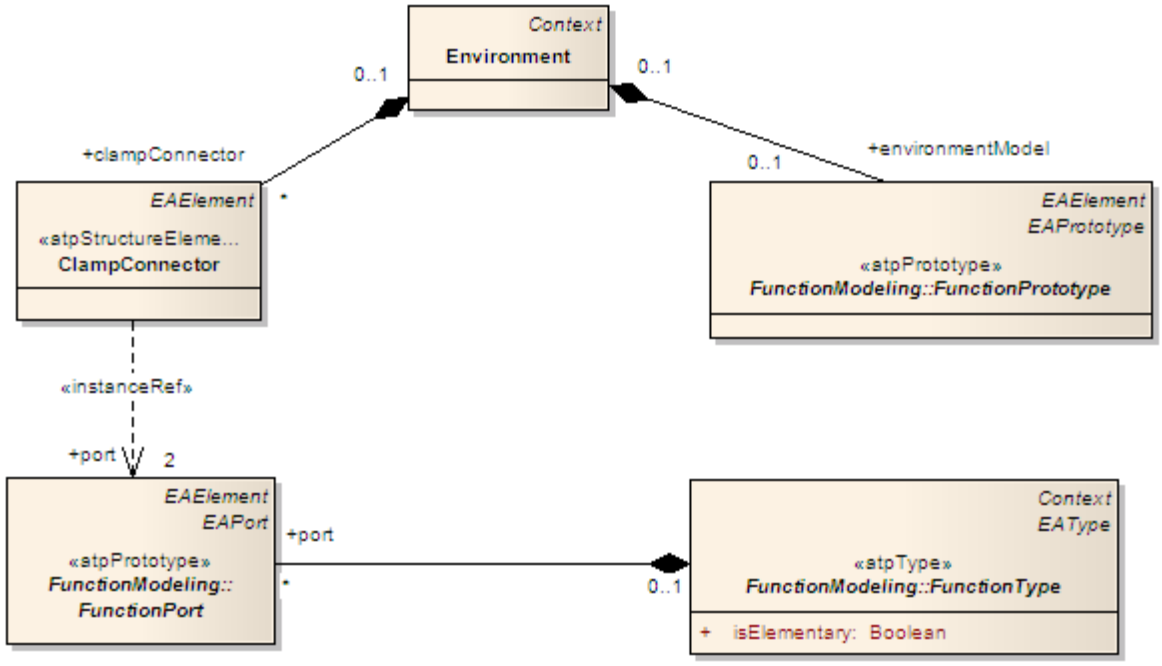

Figure 9. Diagram for Environment. The EnvironmentModel is a packageable element, but note that it is not a part of the SystemModel.

8.2 Element Descriptions

8.2.1 ClampConnector (from Environment) «atpStructureElement»

Generalizations

- EAElement (from Elements)

Description

The clamp connector connects ports across function boundaries and containment hierarchies. It is used to connect from an EnvironmentModel to the FunctionalAnalysisArchitecture, the FunctionalDesignArchitecture, the autosarSystem or another EnvironmentModel. Typically, the EnvironmentModel contains physical ports, which restrict the valid ports in the FunctionalAnalysisArchitecture to those on FunctionalDevices and in the FunctionalDesignArchitecture to those on HardwareFunctions. In case the connection concerns logical interaction, this restriction does not apply. The ClampConnector is always an assembly connector, never a delegation connector.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- port : FunctionPort [2]
  «instanceRef»

Constraints

[1] Can connect two FunctionFlowPorts of different direction.

[2] Can connect two FunctionClientServerPorts of different clientServerType.

[3] Can connect two FunctionFlowPorts with direction inout.

[4] Cannot connect ports in the same SystemModel.

Semantics

ClampConnectors represents the interaction link between a functional model of the EE Architecture and the functional model of the plant.

8.2.2 Environment (from Environment)

Generalizations

- Context (from Elements)

Description

The collection of the environment functional descriptions. This collection can be done across the EAST-ADL abstraction levels.

An environment model can contain functionPrototypes given by either AnalysisFunction or DesignFunction. The environment model does not have abstraction levels as in the system model (e.g., analysisLevel, designLevel).

A functionPrototype of the environment model can have interactions with FAA FunctionalDevice and an FDA HardwareFunction through the ClampConnector.

Attributes

No additional attributes

Associations

- environmentModel : FunctionPrototype [0..1] {composite}
- clampConnector : ClampConnector [*] {composite}

Constraints

No additional constraints

Semantics

Environment is a container element for the entities surrounding the EE architecture and their connections to the EE architecture. The function hierarchy of the Environment interacts with the EE System through Clamp Connectors connected to the SystemModel's functions.

Part III Behavioral Constructs

This part specifies the dynamic, behavioral constructs represented by metaclasses in EAST-ADL.

9 Behavior

9.1 Overview

This chapter describes the behavioral constructs of the EAST-ADL language. What we mean by behavior here is either a function performing some computation on provided data (FlowPort interaction) or the execution of a service called upon by another function (in a ClientServer interaction).

The execution of the behavior assumes a strict run-to-completion, single buffer-overwrite management of data. That is, each execution starts with the reading of data, which are not stored locally and are constantly replaced by fresh data arriving on ports. The function then performs some calculation and finally outputs some data on the output ports. The execution is non-concurrent within an elementary function: only one behavior is active at any point in time. Among a set of functions, behavior is fully concurrent, except for timing precedence constraints. This is to avoid making assumptions that are not met at Design and Implementation Levels. Design Level: All functions are as concurrent as hardware design allows. Timing precedence constraints may constrain this further.

A FunctionBehavior in EAST-ADL is mainly a reference point to some description provided elsewhere (outside the EAST-ADL model) in a tool-dependent format, as depicted in the diagram for the behavior of a function below. This enables reuse of current behavior descriptions contained in the tools currently used by automotive companies and suppliers. Given that, requirements and traceability information can be provided for behavior in relation to the rest of the EAST-ADL model. A list of typical tool formats is provided as an enumeration, FunctionBehaviorKind. Depending on the EAST-ADL language implementation, such a behavior description can be provided in the model itself; for instance, when using a UML implementation of the EAST-ADL, UML behavior modelin can be used. Yet, it should be noted that the behavior described shall be compliant with the execution semantics of an EAST-ADL function.

The rest of the behavioral constructs (see the following diagram of the behavior model organization) relates to the organization of the triggering of behavior attached to functions. At a high level one can define activation Modes, which may span across the whole architecture. Such Modes can be regrouped in exclusive sets. Whenever a FunctionTrigger or a FunctionBehavior refers to a Mode, this means its activation is dependent on the Mode being active or not. Thus, different execution configurations can be defined.

Triggering of the behavior itself, defined by the FunctionTrigger, can be either time- or event-based and be either type-wise or prototype-wise to allow further adjustments of functions in a particular context. Events and timing constraints are defined in the Timing, Events, and TimingConstraints sections.

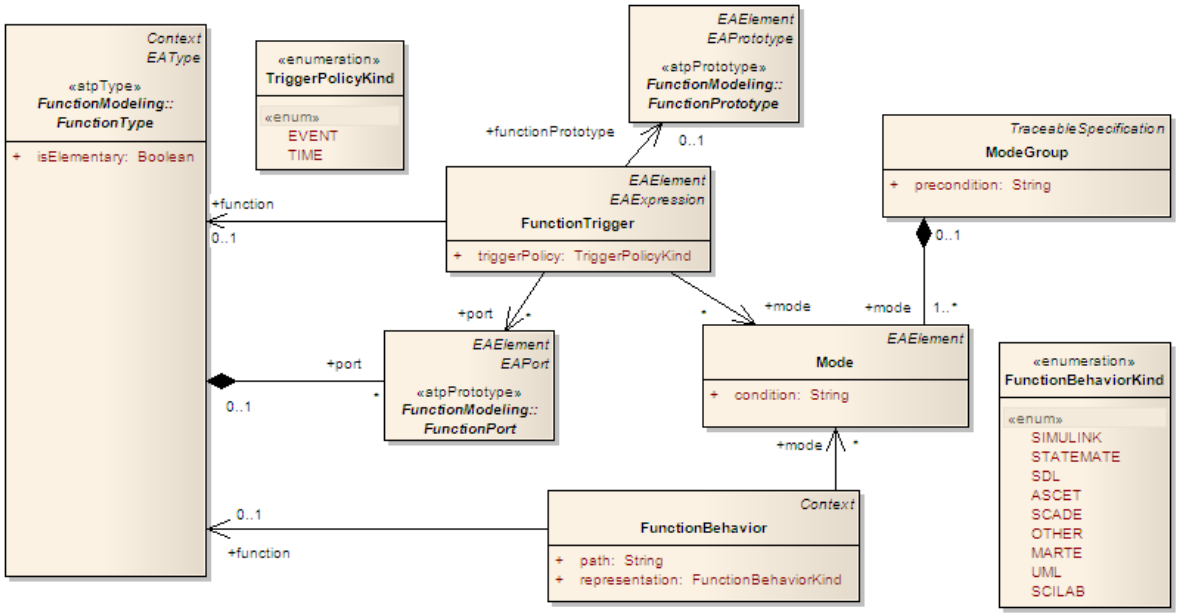

Figure 10. Diagram for the behavior of a function.

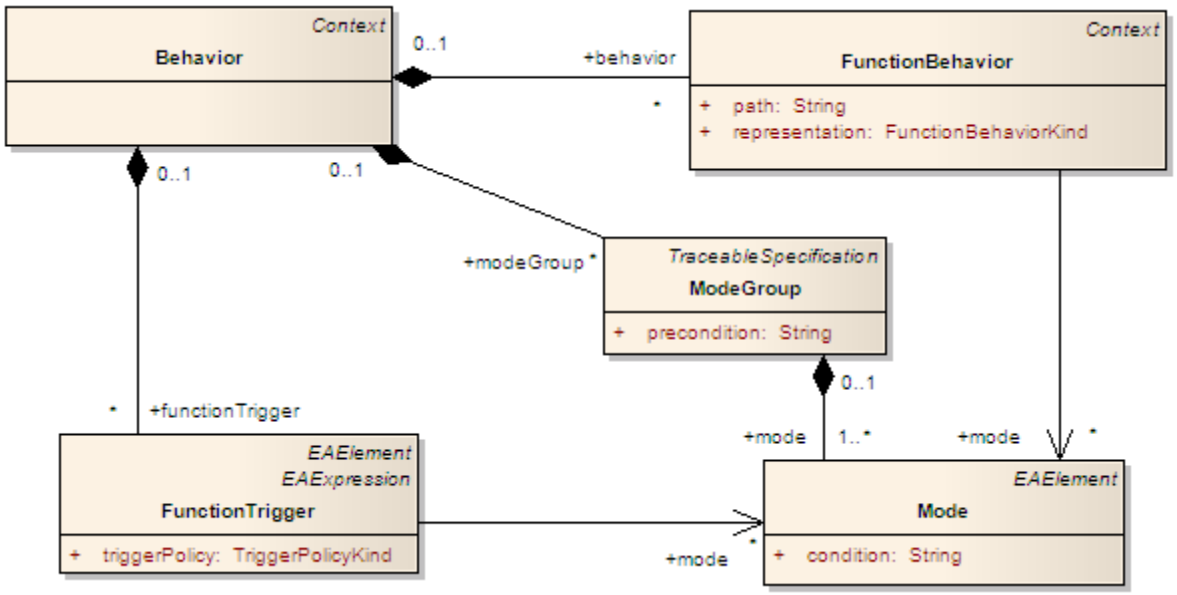

Figure 11. Diagram for behavior model organization.

9.2 Element Descriptions

9.2.1 Behavior (from Behavior)

Generalizations

- Context (from Elements)

Description

Behavior is a container of FunctionBehaviors. It enables grouping of the behaviors assigned to functions in a particular context on which TraceableSpecifications can be applied. This can take any appropriate form depending on the language implementation (for instance in a UML implementation it could be a Package).

The collection of functional behaviors can be performed across the EAST-ADL abstraction levels.

Attributes

No additional attributes

Associations

- behavior : FunctionBehavior [*] {composite}
  This is the set of FunctionBehaviors managed by the container.
- modeGroup : ModeGroup [*] {composite}
  The contained mode groups.
- functionTrigger : FunctionTrigger [*] {composite}

Constraints

No additional constraints

Semantics

This element has the same role and semantics as Context, but for behavioral aspects.

9.2.2 FunctionBehavior (from Behavior)

Generalizations

- Context (from Elements)

Description

FunctionBehavior represents the behavior of a particular FunctionType - referred to by the association to FunctionType. What is meant by behavior is a transfer function performing some data computation (in case of FlowPort interaction) or an operation that can be called by another function (in case of ClientServer interaction). The representation property indicates the kind of representation used to describe the behavior (see FunctionBehaviorKind). The representation itself (e.g., defined in an external model file) is identified by a URL String in the path property. If the representation is provided in the same model file as the system itself, the path property is not used. It is merely a placeholder for the purpose of containing information about and links to the external behavioral model.

FunctionBehavior may refer to execution modes by the association to the element Mode. This is not mandatory; however, when provided, the relation indicates the list of execution Modes in which the FunctionBehavior can potentially be executed (see element Mode).

The triggering of a FunctionBehavior is unknown to the behavior. It is defined by FunctionTriggers (see this element).

Note that the association between FunctionBehavior and FunctionType is specified as a one-way navigable link from FunctionBehavior to FunctionType: what this means is that the EAST-ADL language specification does not require a FunctionType be aware of the FunctionBehavior it is assigned to. Only the navigation from behavior to function is mandatory; the implementation of a reverse link might however be provided depending on the tool support.

Although each FunctionBehavior can refer to at most one FunctionType, note that several FunctionBehaviors can refer to the same FunctionType. In this case, when a FunctionType has several behaviors, only one behavior shall be active at any given time instant, i.e., no concurrent behaviors are allowed in EAST-ADL functions. For instance we cannot have one active behavior in Simulink and one in Modelica. Both can be referenced in the same function, but at any given time, only one is executable. Conditions such as modes and variability must prevent two behaviors being potentially active at the same time.

Note also that FunctionBehaviors are assigned to FunctionTypes and not to FunctionPrototypes. This means that among a set of FunctionPrototypes, which share the same type, behaviors are also shared. However when a FunctionBehavior refer to Modes, which are referred to by different FunctionTriggers, different triggering conditions can be provided among a set of FunctionPrototypes for the same set of behaviors - see FunctionTrigger.

In the case where the identified FunctionType is decomposed into parts, the behavior is a specification for the composed behavior of the FunctionType.

Attributes

- path : String [1]
  The path to the file or model entity containing the behavior.
- representation : FunctionBehaviorKind [1]
  The type of representation used to describe the behavior.

Associations

- function : FunctionType [0..1]
  The FunctionType to which the behavior is assigned.
- mode : Mode [*]
  The execution Modes in which the behavior can be potentially executed.

Constraints

No additional constraints

Semantics

The semantics of FunctionBehavior follows the semantics of the behavioral representation/tool used (for instance SIMULINK, ASCET, etc.). However, in relation to the EAST-ADL model, the FunctionBehavior has synchronous execution semantics:

1. Read inputs from input ports

2. Execute behavior with fixed inputs (run to completion)

3. Provide outputs to output ports

The data transfer between the EAST-ADL ports and the FunctionBehavior is representation/tool-specific and considered part of the execution of the FunctionBehavior.

9.2.3 FunctionBehaviorKind (from Behavior) «enumeration»

Generalizations

None

Description

FunctionBehaviorKind is an enumeration, which lists the various standards or tools used to describe a FunctionBehavior. It is used as a property of a FunctionBehavior. Several standards or tools are listed; however, one can always extend this list by using the literal OTHER.

Enumeration Literals

- ASCET
- MARTE
- OTHER
- SCADE
- SCILAB
- SDL
- SIMULINK
- STATEMATE
- UML

Associations

No additional associations

Constraints

No additional constraints

Semantics

Distinction between UML and MARTE comes from the slight differences in the behavioral definitions (namely concerning data-flow oriented behaviors).

It should be noted that though one can use several languages to provide a representation of a FunctionBehavior, the semantics shall remain compliant with the overall EAST-ADL execution semantics, see FunctionBehavior.

9.2.4 FunctionTrigger (from Behavior)

Generalizations

- EAExpression (from Values)
- EAElement (from Elements)

Description

FunctionTrigger represents the triggering parameters necessary to define the execution of an identified FunctionType or FunctionPrototype. When referring to a FunctionType, a FunctionTrigger applies to all FunctionPrototypes of the given type. When referring to a FunctionPrototype, the trigger is only valid for this particular FunctionPrototype.

Triggering is defined either as event-driven or time-driven - depending on the property triggerPolicy. If set to TIME, the timing constraint is defined with an event constraint associated with the Function's or FunctionPrototype's EventFunction. The function event refers to the activation of the function. If set to EVENT the referenced ports trigger the function using AND semantics, i.e., activate the function.

In addition, a FunctionTrigger may refer to a list of Modes in which the trigger will be considered as potentially active. As of FunctionBehaviors may also refer to Modes, it is possible to arrange various function configurations for which different sets of triggers and behaviors are active. And this, at various levels of granularity, either with a type-wise scope (by referring to a FunctionType) or specifically at prototype level (by referring to a FunctionPrototype).

Note that several FunctionTriggers may be assigned to the same Function (Type or Prototype), for instance to define alternative trigger conditions and/or timing constraints.

Attributes

- triggerPolicy : TriggerPolicyKind [1]
  Defines the triggering policy, either EVENT or TIME. The function event refers to the activation of the function. If set to EVENT, one or several ports of the Function triggers the function, i.e., activates the function.

Associations

- port : FunctionPort [*]
  The FunctionPorts that act as triggers individually or as specified in the triggerCondition.
- function : FunctionType [0..1]
  The FunctionType that the FunctionTrigger refers to.
- functionPrototype : FunctionPrototype [0..1]
  The FunctionPrototype that the FunctionTrigger refers to.
- mode : Mode [*]
  The execution Modes in which the FunctionTrigger is active.

Constraints

[1] The port association must not be empty when triggerPolicy is EVENT.

[2] The port association is empty when triggerPolicy is TIME.

[3] Function and functionPrototype are mutually exclusive associations. A FunctionTrigger either identifies a FunctionType or a FunctionPrototype as its target function, but not both.

[4] Only FunctionFlowPort of FlowDirection=in shall be referred to in the association port.

Semantics

Association Mode defines in which modes the trigger is active.

The FunctionBehavior referenced by the FunctionTrigger is invoked when the FunctionTrigger is active. If multiple ports are referenced, this implies an AND semantics.

It is possible to have multiple triggers on a function, e.g., a slow period complemented with an event trigger allows fast response when needed but a minimal execution rate.

9.2.5 Mode (from Behavior)

Generalizations

- EAElement (from Elements)

Description

Modes are a way to introduce various configurations in the system to account for different states of the system, or of a hardware entity, or an application. The use of modes can be used to filter different views of the model.

Modes are characterized by a Boolean condition provided as a String, which evaluates to true when the Mode is active.

As far as behavior is concerned, Modes enable the logical organisation of a set of triggers and behaviors over a set of functions. Modes are referred to by both FunctionTriggers and FunctionBehaviors (see FunctionTrigger and FunctionBehavior).

Modes can be further organized in mutually exclusive sets with ModeGroups (see that element).

Attributes

- condition : String [1]
  A Boolean expression that characterizes the Mode, it evaluates to true when the Mode is active. The syntax and grammar of this expression is unspecified.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The Mode is active if and only if the condition is true.

9.2.6 ModeGroup (from Behavior)

Generalizations

- TraceableSpecification (from Elements)

Description

ModeGroups serve as containers of Modes. The Modes in a ModeGroup are mutually exclusive. This means that only one Mode of a ModeGroup is active at any point in time. A precondition in the form of a Boolean expression is assigned to the ModeGroup so that ModeGroups can be switched on and off as a whole.

Attributes

- precondition : String [1]
  A Boolean expression that evaluates to true when the ModeGroup is active.

Associations

- mode : Mode [1..*] {composite}
  The modes in this group.

Constraints

No additional constraints

Semantics

The ModeGroup defines a set of modes of which exactly one is active if precondition is true and otherwise none is active.

9.2.7 TriggerPolicyKind (from Behavior) «enumeration»

Generalizations

None

Description

TriggerPolicyKind represents an enumeration for triggering policies.

Enumeration Literals

- EVENT

Triggering by event.

- TIME
  Triggering by time.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The TriggerPolicyKind contains EVENT and TIME as possible triggering policies.

Part IV Variability

This part covers variability extension to EAST-ADL.

10 Variability

10.1 Overview

This package contains elements to express variability in the analysis architecture, design architecture and implementation architecture. These abstraction levels in EAST-ADL will sometimes be called the artifact levels.

Variability management in EAST-ADL is heavily based on feature modeling. However, since feature modeling is used in EAST-ADL also for other purposes than variability management, the feature modeling concepts were not defined in this extension but as part of the EAST-ADL core in package FeatureModeling. For more details on this, please refer to packages FeatureModeling and VehicleFeatureModeling.

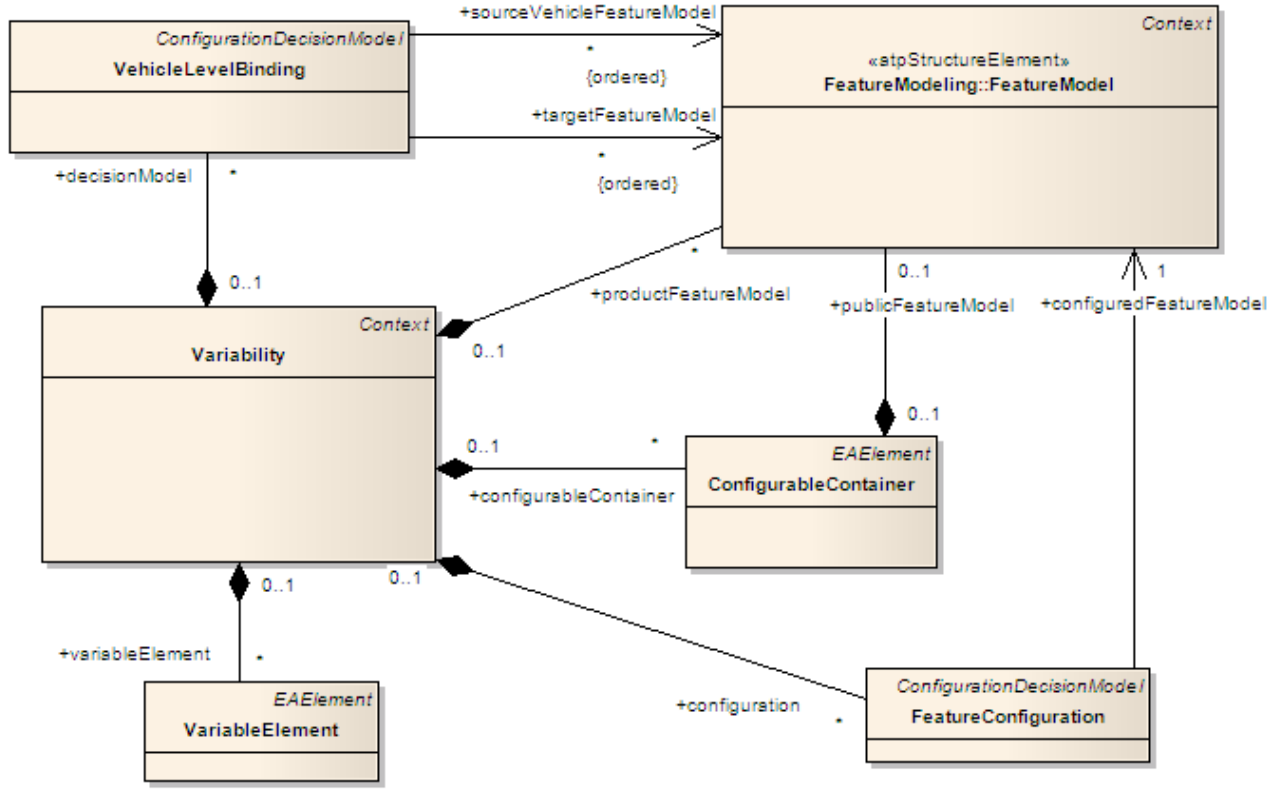

Figure 12. Diagram depicting the organization of variability modeling elements.

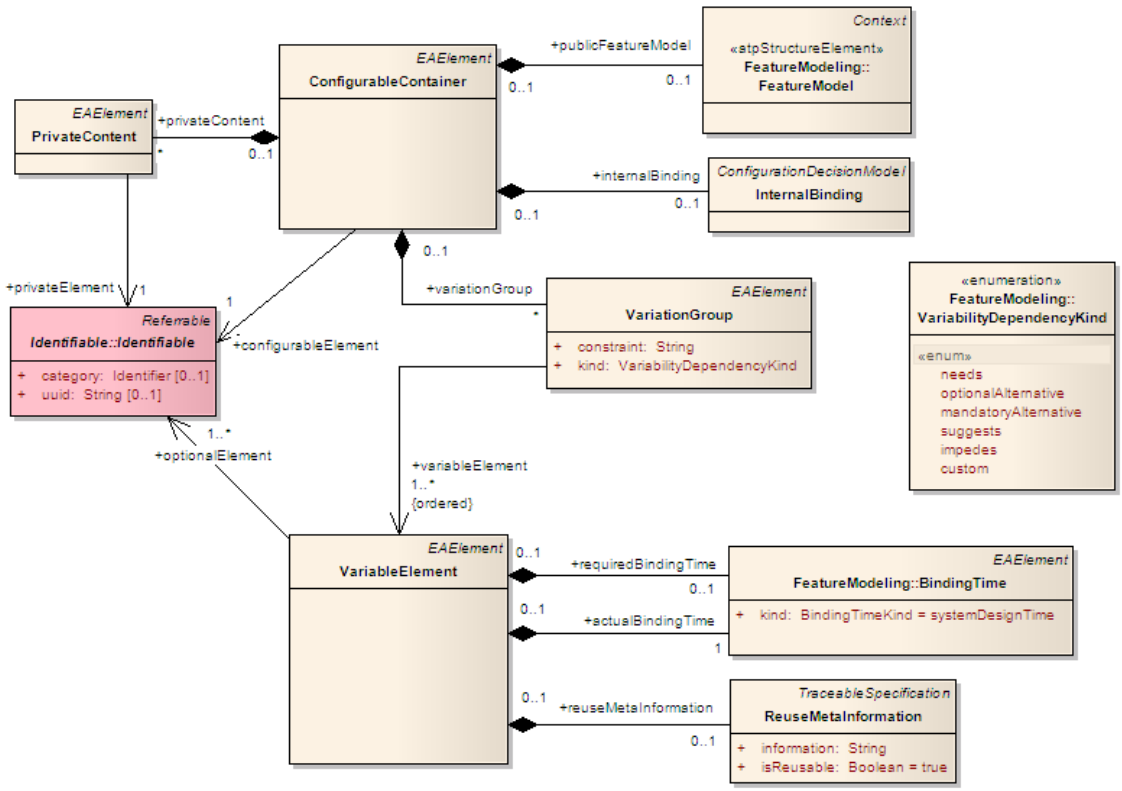

Figure 13. Diagram depicting the elements involved in artifact-level variation management.

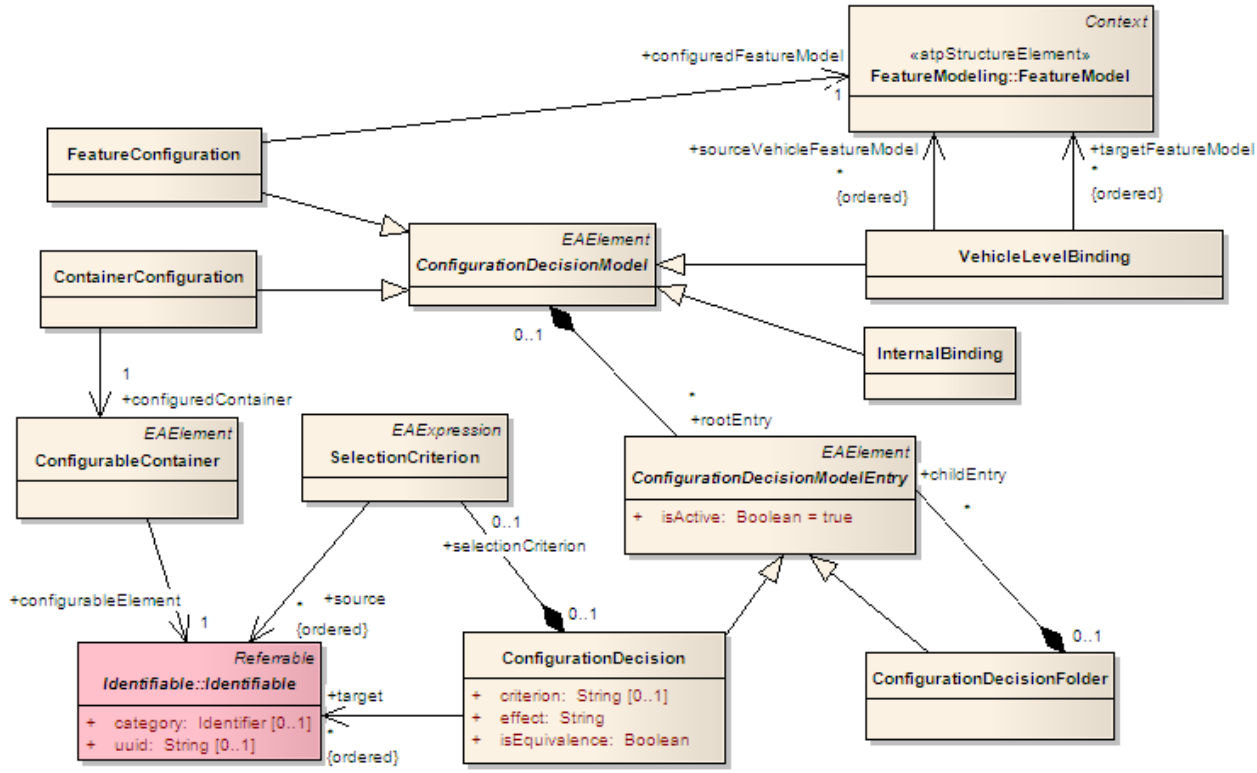

Figure 14. Diagram depicting the elements for configuration modeling.

10.2 Element Descriptions

10.2.1 ConfigurableContainer (from Variability)

Generalizations

- EAElement (from Elements)

Description

ConfigurableContainer is a marker class that marks an element identified by association configurableElement as a configurable container of some variable content, i.e. VariableElements and other, lower-level ConfigurableContainers. In order to describe the contained variability to the outside world and to allow configuration of it, the ConfigurableContainer can have a public feature model and an internal configuration decision model not visible from the outside, called "internal binding".

In addition, the ConfigurableContainer can be used to extend the EAST-ADL variability approach to other languages and standards by pointing from the ConfigurableContainer to the respective (non EAST-ADL) element with association configurableElement. This provides the public feature model and the ConfigurationDecisionModel to that non EAST-ADL element.

The variable content of a ConfigurableContainer is defined as all VariableElements and all other ConfigurableContainers that are directly or indirectly contained in the Identifiable denoted by association configurableElement. Instead of 'variable content' the term 'internal variability' may be used.

Note that, according to this rule, the containment between a ConfigurableContainer and its variable content, i.e. its contained VariableElements and lower-level ConfigurableContainers, is not directly defined between these meta-classes. Instead, the containment is defined by the Identifiable pointed to by association configurableElement. For example, consider a FunctionType "WiperSystem" containing two FunctionPrototypes "front" and "rear" both typed by FunctionType "WiperMotor"; to make the wiper system configurable and the rear wiper motor optional, a ConfigurableContainer is created that points to FunctionType "WiperSystem" (with association configurableElement) and a VariableElement is created that points to FunctionPrototype "rear" (with association optionalElement); the containment between the ConfigurableContainer and the VariableElement is therefore not explicitly defined between these classes but instead only between FunctionType "WiperSystem" and "FunctionPrototype" rear. In addition, the variability-related visibility of "rear" can be changed with PrivateContent: by default the variability of "rear" will be public and visible for direct configuration from the outside of its containing ConfigurableContainer, i.e. "WiperSystem"; by defining a PrivateContent marker object pointing to the FunctionPrototype "rear", this can be changed to private and this variability will not be visible from the outside of "WiperSystem".

Attributes

No additional attributes

Associations

- publicFeatureModel : FeatureModel [0..1] {composite}
  - The local feature model of the ConfigurableContainer.

  PublicFeatureModel represents internal variability of a ConfigurableContainer. Thus it can be seen as being part of the public interface of a ConfigurableContainer.
- privateContent : PrivateContent [*] {composite}
- internalBinding : InternalBinding [0..1] {composite}
  The ConfigurationDecisionModel of the ConfigurableContainer.

- variationGroup : VariationGroup [*] {composite}
  The variation groups that define certain dependencies and constraints between this ConfigurableContainer's variable elements.
- configurableElement : Identifiable [1]
  This association points to the actual element in the core model that is marked as a configurable container of some variable content by this ConfigurableContainer. The ConfigurableContainer in the variability extension can thus be seen as merely a marker element (this marker mechanism follows the global guideline for relating the EAST-ADL extensions to the core and is not specific to the variability extension).

Constraints

[1] Identifies one FunctionType or one HardwareComponentType.

[2] The publicFeatureModel is only allowed to contain Features (no VehicleFeatures).

Semantics

Marks the element identified by association configurableElement as a configurable container of variable content (i.e. it contains VariableElements and/or other, lower-level ConfigurableContainers) and optionally provides a public feature model and an internal configuration decision model for it, thus providing configurability support for them.

10.2.2 ConfigurationDecision (from Variability)

Generalizations

- ConfigurationDecisionModelEntry (from Variability)

Description

ConfigurationDecision represents a single, atomized rule on how to configure the target feature model(s) of the containing ConfigurationDecisionModel, depending on a given configuration of the source feature model(s). Two examples are: "all North American (USA+Canada) cars except A-Class have cruise control" (one ConfigurationDecision) or "all Canadian cars have adaptive cruise control" (another ConfigurationDecision). All ConfigurationDecisions within a single ConfigurationDecisionModel then specify how the target feature model(s) are to be configured depending on the configuration of the source feature model(s).

Example:

Lets assume we have two FeatureModels: FM1 and FM2. FM1 has possible end-customer decisions like USA, Canada, EU, Japan and A-Class, C-Class, etc. FM2 has another possible end-customer decision such as CruiseControl, AdaptiveCruiseControl, RearWiper, RainSensor. End-customer decisions in FM2 describe possible technical features of the delivered products. By way of a set of ConfigurationDecisions it is now possible to define the configuration of FM2 (i.e. if there is a RainSensor, etc.) dependent on a configuration of FM1. In other words, with a ConfigurationDecision we can express something like: "If USA is selected in FM1 AND A-Class is not selected in FM1, then CruiseControl will be selected in FM2".

The two most important constituents of a ConfigurationDecision are its 'criterion' and 'effect'. The effect is a list of things to select and deselect in the target configuration(s), whereas the criterion formulates a condition on the source configuration(s) under which this ConfigurationDecision's effect will actually be applied to the target configuration(s). In the first example above, the criterion would be "USA & not A-Class" and the effect would be "CruiseControl[+]".

Attributes

- criterion : String [0..1]

The criterion is a logical expression on the source configuration(s) that states under which condition the 'effect' will be applied to the target configuration(s). This attribute adheres to the syntax and semantics of the VSL language.

Note that association "selectionCriterion" provides an alternative means for defining such an expression in the form of an AUTOSAR mixed string expression. If both "criterion" and "selectionCriterion" are defined, they are assumed to be semantically equivalent and a tool may choose which one to use for variability and configuration management.

- effect : String [1]
  States which Features are included/selected by the ConfigurationDecision in case the logical expression in 'criterion' evaluates to true. Each of these Features needs to be defined in one of the target feature models of the containing ConfigurationDecisionModel. This attribute adheres to the syntax and semantics of the VSL language.

  The Features are documented as a comma-separated list of strings. Each string has the form <Name of FeatureModel>#<Name of Feature>. If a string is unique in all the source and target FeatureModels of the ConfigurationDecisionModel containing this ConfigurationDecision, then the first part (the FeatureModel name and the #-separator) can be omitted. If a Feature name is not unique in a single FeatureModel, then a dot-notation may be used to prepend the name(s) of predecessors in order to identify the Feature.

  Configuring a cloned feature does not mean selecting or deselecting it but instead instances of the cloned feature are created. Each such instance is provided with a name, which thus becomes a part of the configuration (not the feature model). If several instances are created for a single cloned feature, then the name is used to identify these instances. For example, consider a cloned feature Wiper with cardinality [*]. A first configuration decision might create an instance called "front" and a second might create another named "rear"; a third configuration decision creating or otherwise referring to an instance called "front" will denote the same instance as the first configuration decision. The name space for these instance names is a particular feature configuration.

  As an example for the syntax and semantics of the effect attribute, assume there are two FeatureModels called FMa and FMb and they both contain the Features Wiper and ClimateControl. In FMa (but not in FMb), Wiper and ClimateControl are both refined into the child features Simple and Advanced. In addition, the wiper in FMa has a RainSensor. To denote the RainSensor in FMa you can state:

  FMa#Wiper.RainSensor

  or simply write:

  RainSensor

  This is sufficient here, because the name of Feature RainSensor is unique within FMa and within all FeatureModels referenced by the ConfigurationDecisionModel. In contrast, to denote the advanced version of the climate control in FMa you can specify:

  FMa#ClimateControl.Advanced

  or simply:

  ClimateControl.Advanced

  but merely stating "Advanced" would not suffice because there are two features with that name. Finally, to denote the wiper of feature model FMb you write:

  FMb#Wiper

- isEquivalence : Boolean [1]
  Setting the attribute isEquivalence to true means that the features referred to in the ConfigurationDecision's effect are exclusively configured by this ConfigurationDecision

(i.e. no other ConfigurationDecision in the same ConfigurationDecisionModel may refer to these features). This means that this ConfigurationDecision is the ONLY way in which these features can be selected and therefore the usual logical implication that a ConfigurationDecision represents is turned into a logical equivalence, hence the name: the effect is applied to the target configurations if and only if the specified criterion holds.

When setting this attribute to true, the modeler can state that the target-side features in this ConfigurationDecision's effect are exclusively configured by this ConfigurationDecision, i.e. no other ConfigurationDecision may influence these target-side features.

Associations

- selectionCriterion : SelectionCriterion [0..1] {composite}
  The selectionCriterion is a logical expression on the source configuration(s) that states under which condition the 'effect' will be applied to the target configuration(s). It is defined as a mixed string expression.

  Note that attribute “criterion” provides an alternative means for defining such an expression in the form of a VSL expression. If both “criterion” and “selectionCriterion” are defined, they are assumed to be semantically equivalent and a tool may choose which one to use for variability and configuration management.
- target : Identifiable [*] {ordered}
  The target elements used in the mixed string expression.

Constraints

[1] Attribute "criterion" or association "selectionCriterion" (or both) must be defined.

Semantics

The ConfigurationDecision excludes or includes Features based on a given criterion.

The elements of the criterion and effect attributes may be identified through the target and the source in the selectionCriterion. The criterion and effect attributes can contain a VSL expression with qualified names of the identified elements.

10.2.3 ConfigurationDecisionFolder (from Variability)

Generalizations

- ConfigurationDecisionModelEntry (from Variability)

Description

ConfigurationDecisionFolder represents a grouping for ConfigurationDecisions.

Attributes

No additional attributes

Associations

- childEntry : ConfigurationDecisionModelEntry [*] {composite}
  The child entries of the ConfigurationDecisionFolder.

Constraints

No additional constraints

Semantics

ConfigurationDecisionFolder is a grouping entity for ConfigurationDecisions.

10.2.4 ConfigurationDecisionModel (from Variability) {abstract}

Generalizations

- EAElement (from Elements)

Description

A ConfigurationDecisionModel defines how to configure m target feature models, depending on a given configuration of n source feature models, thus establishing a configuration-related link from the n source feature models to the m target feature models (also called configuration link). With the information captured in a ConfigurationDecisionModel it is then possible to transform a given set of source configurations (one for every source feature model) into corresponding target configurations (one for every target feature model).

For example, a ConfigurationDecisionModel can capture information such as "if feature 'S-Class' is selected in the source feature model, then select feature 'RainSensor' in the target feature model" or "if feature 'USA' is selected in the source feature model, then select feature 'CupHolder' in the target feature model".

Note that in principle all ConfigurationDecisionModels have source / target feature models. However, they are only defined explicitly for those used on vehicle level; for ConfigurationDecisionModels used as an internal binding on FunctionTypes, the source and target feature models are defined implicitly (cf. metaclass InternalBinding). In addition, in the special case of FeatureConfiguration there is by definition no source and only a single target feature model, which is defined explicitly (cf. metaclass FeatureConfiguration).

The configuration information captured in a ConfigurationDecisionModel is represented by ConfigurationDecisions, each of which captures a single, atomized rule on how to configure the target feature model(s) depending on a given configuration of the source feature model(s).

Attributes

No additional attributes

Associations

- rootEntry : ConfigurationDecisionModelEntry [*] {composite}
  The root entries of the ConfigurationDecisionModel.

Constraints

No additional constraints

Semantics

See description.

10.2.5 ConfigurationDecisionModelEntry (from Variability) {abstract}

Generalizations

- EAElement (from Elements)

Description

ConfigurationDecisionModelEntry is the abstract base class for all content of a ConfigurationDecisionModel.

Attributes

- isActive : Boolean = true [1]

If active==TRUE then the ConfigurationDecisionModelEntry is actually applied when transforming source into target configurations; otherwise it will be ignored. With this attribute, configuration decisions can (temporarily) be disabled without having to delete them from the model.

If this is set to FALSE for a ConfigurationDecisionFolder, then the entire contents of this folder are deactivated, no matter to what value their isActive-attribute is set.

Associations

No additional associations

Constraints

No additional constraints

Semantics

See description.

10.2.6 ContainerConfiguration (from Variability)

Generalizations

- ConfigurationDecisionModel (from Variability)

Description

ContainerConfiguration defines an actual configuration of the variable content of a ConfigurableContainer, in particular the selection or deselection of contained VariableElements and the configuration of the public feature models of other contained ConfigurableContainers. For more details on the variable content of a ConfigurableContainer refer to the documentation of meta-class ConfigurableContainer.

The ContainerConfiguration inherits from ConfigurationDecisionModel even though it does not define a configuration link between feature models, similar to FeatureConfiguration. For more information on this, refer to the documentation of meta-class FeatureConfiguration.

The source and target feature models of a ContainerConfiguration are defined implicitly: it always has zero source feature models (as explained for FeatureConfiguration) and its target feature models can be deduced from the ConfigurableContainer being configured by applying the same rules as defined for InternalBinding.

Attributes

No additional attributes

Associations

- configuredContainer : ConfigurableContainer [1]
  The ConfiguredContainer being configured by this ContainerConfiguration.

Constraints

No additional constraints

Semantics

The ContainerConfiguration specifies a concrete configuration of the variable content of a ConfigurableContainer.

10.2.7 FeatureConfiguration (from Variability)

Generalizations

- ConfigurationDecisionModel (from Variability)

Description

FeatureConfiguration defines an actual configuration of a FeatureModel, in particular the selection or deselection of optional features, values for selected parameterized features, and instance creations for cloned features.

Note that configurations of feature models are realized as a specialization of metaclass ConfigurationDecisionModel. This is possible because a ConfigurationDecisionModel also captures the configuration, i.e., of its target feature model(s); while in the standard case of ConfigurationDecisionModel this target-side configuration depends on a given configuration of source feature model(s), here we simply define a "constant" target-side configuration without considering any source configurations. Therefore, the FeatureConfiguration meta-class has additional constraints compared to the super-class ConfigurationDecisionModel: the FeatureConfiguration has no source FeatureModel and only a single target FeatureModel, which serves as the FeatureModel being configured, explicitly defined through association 'configuredFeatureModel'. And since there is no source feature model to which the criterion can refer, all ConfigurationDecisions in a FeatureConfiguration must have "true" as their criterion.

Attributes

No additional attributes

Associations

- configuredFeatureModel : FeatureModel [1]

Constraints

No additional constraints

Semantics

The FeatureConfiguration specifies a concrete configuration of a feature model, in particular which Features of this FeatureModel are selected or deselected.

10.2.8 InternalBinding (from Variability)

Generalizations

- ConfigurationDecisionModel (from Variability)

Description

The InternalBinding is the private, internal ConfigurationDecisionModel of the ConfigurableContainer. It defines how the internal, lower-level variability of the ConfigurableContainer is bound, i.e. configured, depending on a given configuration of the ConfigurableContainer's public feature model. This way, the binding of this internal variability is encapsulated and hidden behind the public feature model, which serves as a variability-related interface.

Note that for this use case, the source and target feature models need not be defined explicitly because they are deduced implicitly: the ConfigurableContainer's public feature model serves as the (single) target feature model, and the source feature models are deduced from the ConfigurableContainer's internal variability (esp. other, lower-level ConfigurableContainers which are contained).

For a definition of the precise meaning of 'internal variability' (also called variable content) refer to the documentation of meta-class ConfigurableContainer.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

See description.

10.2.9 PrivateContent (from Variability)

Generalizations

- EAElement (from Elements)

Description

PrivateContent is a marker class that marks the artifact element denoted by association privateElement as private, i.e., it will not be presented to the outside of the containing ConfigurableContainer.

Refer to the documentation of meta-class ConfigurableContainer for a detailed explanation of how ConfigurableContainer and PrivateContent work together.

Attributes

No additional attributes

Associations

- privateElement : Identifiable [1]
  This association points to the actual element in the core model that is marked private by this PrivateContent object. Instances of the PrivateContent meta-class in the variability extension can thus be seen as merely a marker object (this marker mechanism follows the global guideline for relating the EAST-ADL extensions to the core and is not specific to the variability extension).

Constraints

[1] Identifies either one FunctionPrototype or one FunctionPort or one FunctionConnector or one HardwareComponentPrototype or one HardwarePort or one ClampConnector.

Semantics

Marks the element identified by association privateElement as private. Otherwise the elements visibility defaults to public.

10.2.10 ReuseMetaInformation (from Variability)

Generalizations

- TraceableSpecification (from Elements)

Description

ReuseMetaInformation represents the description information needed in the context of reuse. For example a specific entity is only a short-time solution that is not intended to be reused. Also a specific entity can only be reused for specific model ranges (that are not reflected in the product model).

Attributes

- information : String [1]
  The reuse information is stored in this attribute.
- isReusable : Boolean = true [1]
  This Boolean attributes just says whether the owning VariableElement itself can essentially be reused or not. Specific information or constraints on reuse are in the information attribute.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The ReuseMetaInformation represents information that explains if and how the respective entity can be reused.

10.2.11 SelectionCriterion (from Variability)

Generalizations

- EAExpression (from Values)

Description

A mixed string description, identifying the source elements. This means that the SelectionCriterion could evaluate to True or False if a optional identifiable (feature or artefact) is referenced as target. Or evaluate to a numerical if a FeatureParameter is referenced as target.

Attributes

No additional attributes

Associations

- source : Identifiable [*] {ordered}
  The elements used in the mixed string expression.

Constraints

No additional constraints

Semantics

See description.

10.2.12 Variability (from Variability)

Generalizations

- Context (from Elements)

Description

The collection of variability descriptions, related feature models, and decision models. This collection can be done across the EAST-ADL abstraction levels.

Attributes

No additional attributes

Associations

- productFeatureModel : FeatureModel [*] {composite}
  This association points to zero or more feature models intended to be used on the vehicle level in addition to the core technical feature model (cf. association technicalFeatureModel in meta-class VehicleLevel).

  Usually there will be the core technical feature model and one or more so-called "product feature models" on the vehicle level, which provide an orthogonal view on the core technical feature model tailored to a particular purpose, for example an end-customer feature model. However, there may be more and other use cases for feature models on vehicle level. More detailed treatment of this is beyond the scope of the language specification and can be found in the accompanying usage and methodology documentations.
- decisionModel : VehicleLevelBinding [*] {composite}
- configuration : FeatureConfiguration [*] {composite}
- variableElement : VariableElement [*] {composite}
- configurableContainer : ConfigurableContainer [*] {composite}

Constraints

No additional constraints

Semantics

See description.

10.2.13 VariableElement (from Variability)

Generalizations

- EAElement (from Elements)

Description

VariableElement is a marker class that marks an artifact element denoted by association optionalElement as being optional, i.e. it will not be present in all configurations of the complete system. A typical example is an optional FunctionPrototype.

In addition, the VariableElement can be used to extend the EAST-ADL variability approach to other languages and standards by pointing from the VariableElement to the respective (non EAST-ADL) element with association optionalElement, thus marking the non EAST-ADL element as optional and providing configuration support within its containing ConfigurableContainer.

Refer to the documentation of meta-class ConfigurableContainer for a detailed explanation of how ConfigurableContainer and VariableElement work together.

Attributes

No additional attributes

Associations

- actualBindingTime : BindingTime [1] {composite}

Actual binding time attribute. Due to technical conditions it may occur that the realized binding time of the feature/variation point differs from the originally intended binding time. In this case one has to provide information about the actual binding time. In the rationales it must be described what the reasons are for a (different) actual binding time.

- requiredBindingTime : BindingTime [0..1] {composite}
  Required binding time attribute. Each feature/variation point must have a required binding time attribute. The required binding time describes the binding time that the feature is intended to have.
- reuseMetaInformation : ReuseMetaInformation [0..1] {composite}
  Reuse-relevant meta-information for the element.
- optionalElement : Identifiable [1..*]
  This association points to the actual element in the core model that is marked optional by this VariableElement. The VariableElement in the variability extension can thus be seen as merely a marker element (this marker mechanism follows the global guideline for relating EAST-ADL extensions to the core and is not specific to the variability extension).

Constraints

[1] Identifies either one FunctionPrototype or one FunctionPort or one FunctionConnector or one HardwareComponentPrototype or one HardwarePin or one ClampConnector.

Semantics

Marks the element identified by association optionalElement as optional.

10.2.14 VariationGroup (from Variability)

Generalizations

- EAElement (from Elements)

Description

A VariationGroup defines a relation between an arbitrary number of VariableElements. It is primarily intended for defining how these VariableElements may be combined (e.g. one requires the other, alternative, etc.).

Attributes

- constraint : String [1]
  Only defined iff kind=="custom". A constraint specifying how the VariableElements in the variation group can be combined. This attribute adheres to the syntax and semantics of the VSL language.
- kind : VariabilityDependencyKind [1]
  The kind of the variation group (see enumeration VariationGroupKind).

Associations

- variableElement : VariableElement [1..*] {ordered}
  Associated variable elements.

Constraints

No additional constraints

Semantics

Defines a dependency or constraint between the variable elements denoted by association variableElement. The actual constraint is specified by attribute kind.

10.2.15 VehicleLevelBinding (from Variability)

Generalizations

- ConfigurationDecisionModel (from Variability)

Description

This class represents a binding on the vehicle level or coming from the vehicle level with explicitly defined source and target feature models. The source feature models must be on vehicle level, but the target feature models may be located on artifact level, e.g. the public feature model of the top-level FunctionType in the FDA. This way, a VehicleLevelBinding may be used to bridge the gap from vehicle level variability management to that on the artifact level.

Source feature models may be either the core technical feature model (as defined by association technicalFeatureModel of meta-class VehicleLevel) or one of the optional product feature models (as defined by association productFeatureModel of meta-class Variability in the variability extension).

Attributes

No additional attributes

Associations

- sourceVehicleFeatureModel : FeatureModel [*] {ordered}
- targetFeatureModel : FeatureModel [*] {ordered}

Constraints

[1] The sourceVehicleFeatureModels shall only contain VehicleFeatures.

[2] The sourceVehicleFeatureModels shall be different from the targetFeatureModels.

Semantics

See description.

Part V Requirements

This part covers the Requirements extension to EAST-ADL, which includes requirements, use cases and Verification and Validation.

11 Requirements

11.1 Overview

A requirement expresses a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification or other formally imposed properties.

Requirements can be introduced in different phases of the development process for different reasons. They could be introduced by marketing people, control engineers, system engineers, software engineers, Driver/OS developers, basic software developers or hardware engineers. This leads to the fact that requirements have many sources, and requirements are of many types (at different levels of detail) and have several relations between them. Under these conditions the number of requirements can become quickly unmanageable if appropriate management does not exist. Note that, requirements can change during the project development and the changes should be taken into account. Requirements are organized hierarchically through several kinds of refinement relations.

EAST-ADL has constructs that deal with these problems. Some of these constructs deals with general issues in software development and have been already addressed in the past by general processes. As done for the structure part of EAST-ADL, the requirements part will be compliant with UML2. The EAST-ADL adapts existing concepts whenever possible and develops new ones otherwise.

Elements inspired by SysML are Requirement, Satisfy, Refine, DeriveRequirement, and Verify.

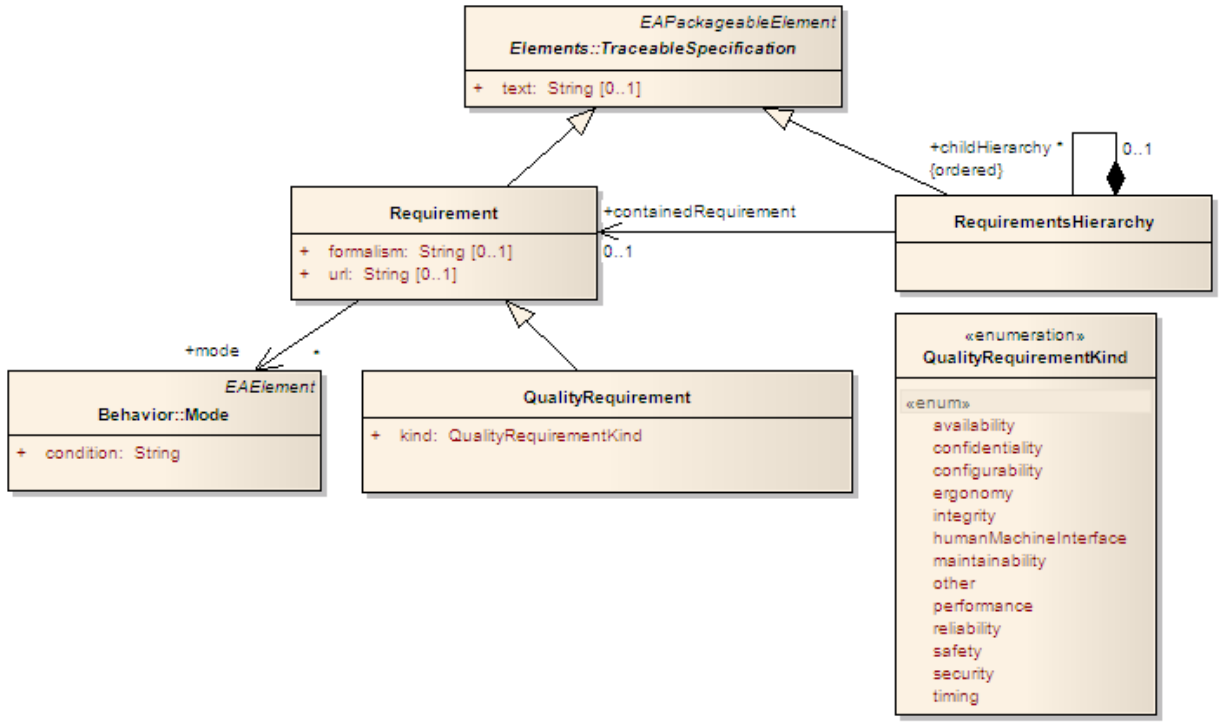

Figure 15. Diagram for Requirements overview.

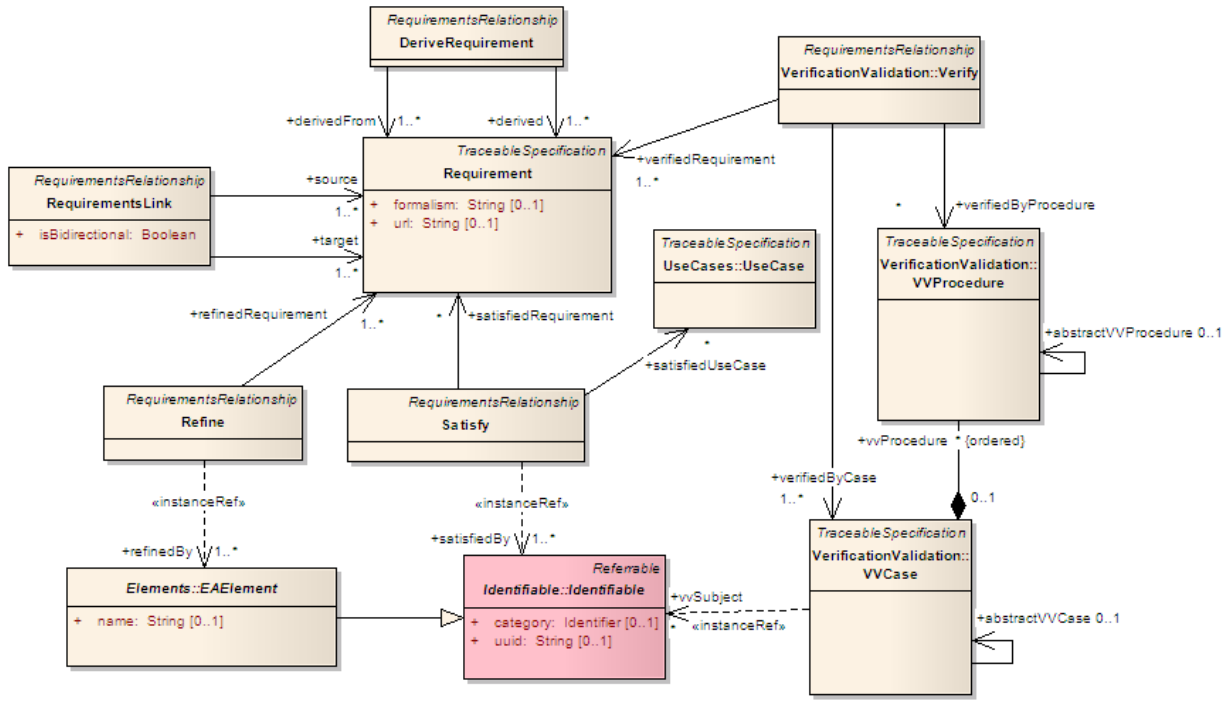

Figure 16. Diagram for Relationships including Requirement.

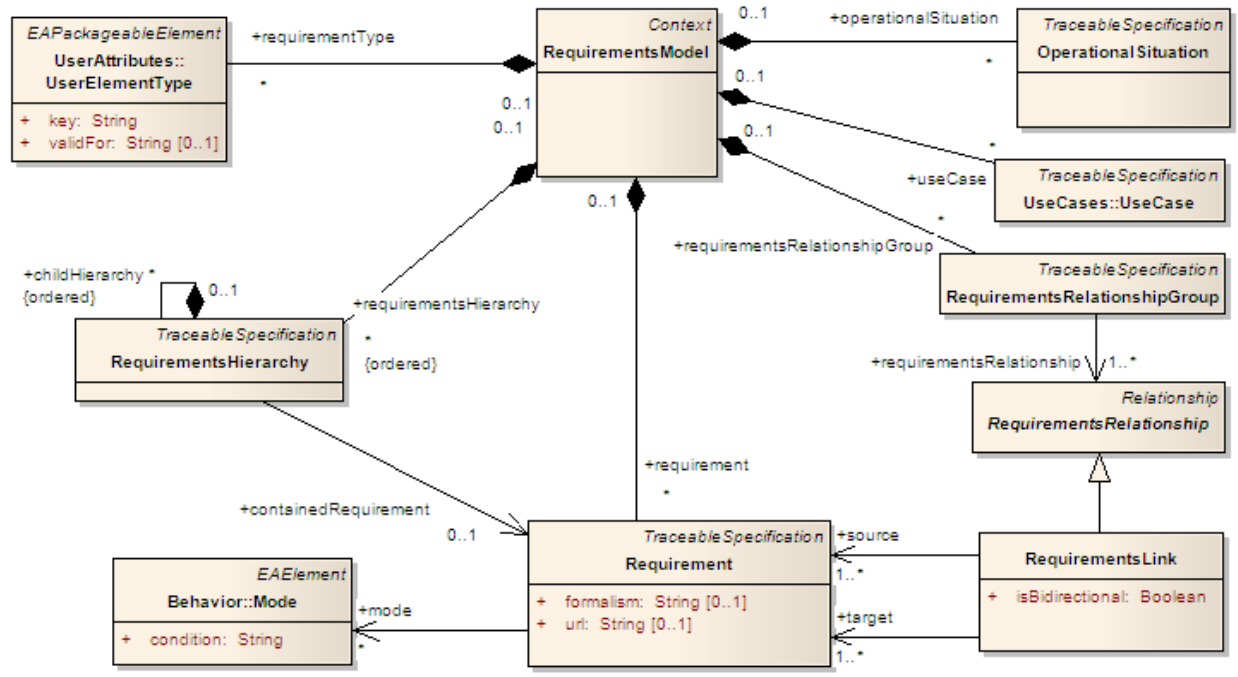

Figure 17. Diagram for Requirements organization.

11.2 Element Descriptions

11.2.1 DeriveRequirement (from Requirements)

Generalizations

- RequirementsRelationship (from Requirements)

Description

The DeriveRequirement is a relationship metaclass, which signifies a dependency relationship between two sets of Requirements, showing the relationship when a set of derived client Requirement (client requirement) is derived from a set of Requirements (supplier requirement).

Attributes

No additional attributes

Associations

- derivedFrom : Requirement [1..*]
  The set of requirements that the client requirement are derived from.
- derived : Requirement [1..*]
  The set of requirements that are derived from the supplier requirement.

Constraints

No additional constraints

Semantics

The DeriveRequirement metaclass signifies a derived/derived by relationship between Requirements, where the modification of the supplier Requirement may impact the derived client Requirement.

11.2.2 OperationalSituation (from Requirements)

Generalizations

- TraceableSpecification (from Elements)

Description

An operational situation is a state, condition or scenario in the environment that may influence the vehicle. The Operational Situation may be further detailed by a functional definition in the EnvironmentModel.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

OperationalSituation represent a state, condition or scenario that is external to the vehicle.

11.2.3 QualityRequirement (from Requirements)

Generalizations

- Requirement (from Requirements)

Description

QualityRequirements or non-functional requirements are used to introduce externally visible properties of the system considered as a whole. They specify criteria that can be used to judge the operation of a system. As opposed to a functional requirement specifying what a system is supposed to do, the non-functional requirements define how a system is supposed to be.

The attribute qualityRequirementType allows a more specific classification.

Attributes

- kind : QualityRequirementKind [1]

Associations

No additional associations

Constraints

No additional constraints

Semantics

QualityRequirement element represent a requirement which is non-functional.

11.2.4 QualityRequirementKind (from Requirements) «enumeration»

Generalizations

None

Description

QualityRequirementKind represents an enumeration with enumeration literals describing various types of quality requirements.

Enumeration Literals

- availability
  The requirement is related to availability, the readiness for correct service.
- confidentiality
  The requirement is related to confidentiality.
- configurability
  The requirement is related to the ability to configure the functionality.
- ergonomy
  The requirement is related to the ergonomy of the functionality.
- humanMachineInterface
  The requirement is related to the human-machine interface.
- integrity
  The requirement is related to integrity, absence of improper system alteration.
- maintainability
  The requirement is related to maintainability, the ability to undergo modifications and repairs.

- other
  The requirement is a quality requirement with a general classification.
- performance
  The requirement is related to performance in general.
- reliability
  The requirement is related to reliability, the continuity of correct service.
- safety
  The requirement is related to safety, the absence of catastrophic consequences on the user(s) and the environment.
- security
  The requirement is related to security.
- timing
  The requirement is related to timing.

Associations

No additional associations

Constraints

No additional constraints

Semantics

QualityRequirementKind represents the kind of QualityRequirement given by the definition of the respective Enumeration Literal.

11.2.5 Refine (from Requirements)

Generalizations

- RequirementsRelationship (from Requirements)

Description

The Refine is a relationship metaclass, which signifies a dependency relationship between Requirements and EAElements, showing the relationship when a client EAElement refines the supplier Requirement.

Attributes

No additional attributes

Associations

- refinedRequirement : Requirement [1..*]
  List of refined Requirements.

Dependencies

- refinedBy : EAElement [1..*]
  «instanceRef»

Constraints

[1] The property refinedBy must not have the types Requirement or RequirementContainer.

Semantics

The Refine metaclass signifies a refined requirement/refined by relationship between a Requirement and an EAElement, where the modification of the supplier Requirement may impact the refining client

EAElement. The Refine metaclass implies the semantics that the refining client EAElement is not complete, without the supplier Requirement.

11.2.6 Requirement (from Requirements)

Generalizations

- TraceableSpecification (from Elements)

Description

The Requirement represents a capability or condition that must (or should) be satisfied. A Requirement can also specify an informal constraint, e.g. "The development of the component X must be according to the standard Y", or "The realization of this function as a software component must adhere to the scope and external interface as specified by this function". It will be used to unite the common properties of specific requirement types. A Requirement may either be directly associated with a Context (by inheriting from TraceableSpecification) or it may be included in a RequirementsHierarchy, which represents a larger unit or module of specification information.

The traceability between Requirement entities and other specification or design entities will be ensured by the relationship dependencies described in the Infrastructure part of this specification.

Attributes

- formalism : String [0..1]
  Specifies the language used for the requirement statement.
- url : String [0..1]
  Reference to possible external file containing the requirement statement.

Associations

- mode : Mode [*]
  The mode where this requirement is valid.

Constraints

No additional constraints

Semantics

The string in the text attribute inherited from TraceableSpecification is the capability or condition that applies to the Identifiable that is associated to the Requirement through the Satisfy relation.

11.2.7 RequirementsHierarchy (from Requirements)

Generalizations

- TraceableSpecification (from Elements)

Description

RequirementsHierarchy represents a larger unit or module of specification information. It is used to bundle several Requirements which are semantically related to each other. Thus, to preserve the ordering of requirement specification objects, the order of child hierarchies is very important here.

The RequirementsHierarchy with its reference to Requirement is the basic element for structuring requirement information into a forest structure.

RequirementsHierarchy correponds to ReqIF SpecHierarchy.

Attributes

No additional attributes

Associations

- containedRequirement : Requirement [0..1]
  Requirement referenced by the virtual RequirementsHierarchy.
- childHierarchy : RequirementsHierarchy [*] {ordered} {composite}
  Sub hierarchies of a requirements hierarchy. Sub hierarchies may have references (each time max. one) to requirement specification objects. To preserve the original ordering of requirement specification objects, the ordering of sub hierarchies is very important here.

Constraints

[1] Only non-root RequirementsHierarchy which is contained in another RequirementHierarchy are allowed to reference a Requirement.

Semantics

RequirementsHierarchy organizes Requirements in groups. The semantics of the group is user-defined.

11.2.8 RequirementsLink (from Requirements)

Generalizations

- RequirementsRelationship (from Requirements)

Description

RequirementsLink represents a relation between two or more Requirements. Source and target Requirements of the relation are distinguished, which means that the relation is directed (from source to target). If such a distinction does not make sense, then use a RequirementsRelationGroup instead.

The standard case will be a relation with one source and one target Requirement. However, it is possible to have several source and/or several target Requirements so that general relations can be expressed with instances of this metaclass.

The semantic of a concrete Requirement relation can be provided by the modeler. In particular, three ways are conceivable:

(1) The user attributes of the relation can be used to specify its meaning, for example with a user attribute called "relationType" which is set to values such as "needs" or "excludes".

(2) The UserAttributeElementType can be used. Certain types will be used for certain relation semantics.

(3) RequirementsRelationGroups can be used, i.e. all relations with an "excludes" meaning are put in one relation group and all with a "needs" meaning are put in another.

Attributes

- isBidirectional : Boolean [1]
  When set to true, the semantic relation represented by this instance of RequirementRelation does not only apply to the direction from source to target (as always) but also in the opposite direction.

  Note that this means that the relation becomes directed in both directions but NOT undirected. To express an undirected association use a RequirementsRelationGroup.

Associations

- target : Requirement [1..*]
  The requirement(s) at which this relation ends.
- source : Requirement [1..*]
  The requirement(s) at which this relation starts.

Constraints

No additional constraints

Semantics

The RequirementsLink defines a relation from a set of source and target requirements. The isBidirectional attribute defines whether the relation is bidirectional. The semantics of the relation is user-defined.

11.2.9 RequirementsModel (from Requirements)

Generalizations

- Context (from Elements)

Description

The collection of requirements, their relationships, and use cases. This collection can be done across the EAST-ADL abstraction levels.

Attributes

No additional attributes

Associations

- requirementsRelationshipGroup : RequirementsRelationshipGroup [*] {composite}
- requirement : Requirement [*] {composite}
- requirementsHierarchy : RequirementsHierarchy [*] {ordered} {composite}
  Root elements of requirement hierarchies.
- operationalSituation : OperationalSituation [*] {composite}
- useCase : UseCase [*] {composite}
- requirementType : UserElementType [*] {composite}
  User element types contained in this RequirementModel. This allows for the introduction of additional user element types to be used within this RequirementsModel only. These are additional in that they are used in addition to the user attribute definitions which are provided globally for the entire EAST-ADL repository.

  These user element types given by this association correspond to ReqIF's SpecType.

Constraints

[1] The validFor attribute of the UserElementType shall be "Requirement".

Semantics

The RequirementsModel is a container element for requirement-related elements.

11.2.10 RequirementsRelationship (from Requirements) {abstract}

Generalizations

- Relationship (from Elements)

Description

Semantics:

RequirementsRelationship is an abstract association. The semantics is defined by its specializations.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

RequirementsRelationship is an abstract association. The semantics is defined by its specializations.

11.2.11 RequirementsRelationshipGroup (from Requirements)

Generalizations

- TraceableSpecification (from Elements)

Description

RequirementsRelationGroup represents a group of relations between Requirements.

RequirementsRelationGroup correponds to ReqIF RelationGroup.

Attributes

No additional attributes

Associations

- requirementsRelationship : RequirementsRelationship [1..*]
  The relations that are grouped by this relation group. Note that this is not a containment association, i.e., a single relation may be grouped by several RequirementRelationGroups.

Constraints

No additional constraints

Semantics

RequirementsRelationGroup represents a group of RequirementsRelations. The semantics of this grouping is defined by the user.

11.2.12 Satisfy (from Requirements)

Generalizations

- RequirementsRelationship (from Requirements)

Description

The Satisfy is a relationship metaclass, which signifies the relationship between a Requirement and an element intended to satisfy the Requirement.

Attributes

No additional attributes

Associations

- satisfiedRequirement : Requirement [*]
  List of Requirements that are satisfied by the client ADLElement or satisfied by the client AUTOSAR element.
- satisfiedUseCase : UseCase [*]
  List of satisfied UseCases that are satisfied by the client EAElements or satisfied by the client AUTOSAR elements.

Dependencies

- satisfiedBy : Identifiable [1..*]
  «instanceRef»

Constraints

[1] The EAElement in the association satisfiedBy may not be a Requirement or RequirementContainer.

[2] An element of type Satisfy is only allowed to have associations to either elements of type UseCase (see satisfiedUseCase) or elements of type Requirement (see satisfiedRequirement). Not both at the same time!

Semantics

The Satisfy metaclass signifies a satisfied requirement/satisfied by relationship between a set of Requirements and a set of satisfying entities, where the modification of the supplier Requirements may impact the satisfying client entities. The Satisfy metaclass implies the semantics that the satisfying client entities are not complete without the supplier Requirement.

12 UseCases

12.1 Overview

The use case package contains elements for defining the required usage of a system. Typically, UseCases are used to capture the functional requirements of a system, that is, what a system is supposed to do. In order to organize use cases in an EAST-ADL requirements hierarchy, a Refine relation can be used to link the UseCase to a requirement.

To enable a rich and logical organization of UseCases, specific relationships are introduced to enable the extension, inclusion or redefinition of existing UseCases.

The UseCase concept is explicitly linked to two main elements in the rest of the language: 1) the Satisfy relationship from Requirements, which links system entities, and the Requirement or the UseCase they satisfy; 2) the HazardousEvent concept from Dependability, which links a particular Hazard to a specific situation, depicted as a UseCase.

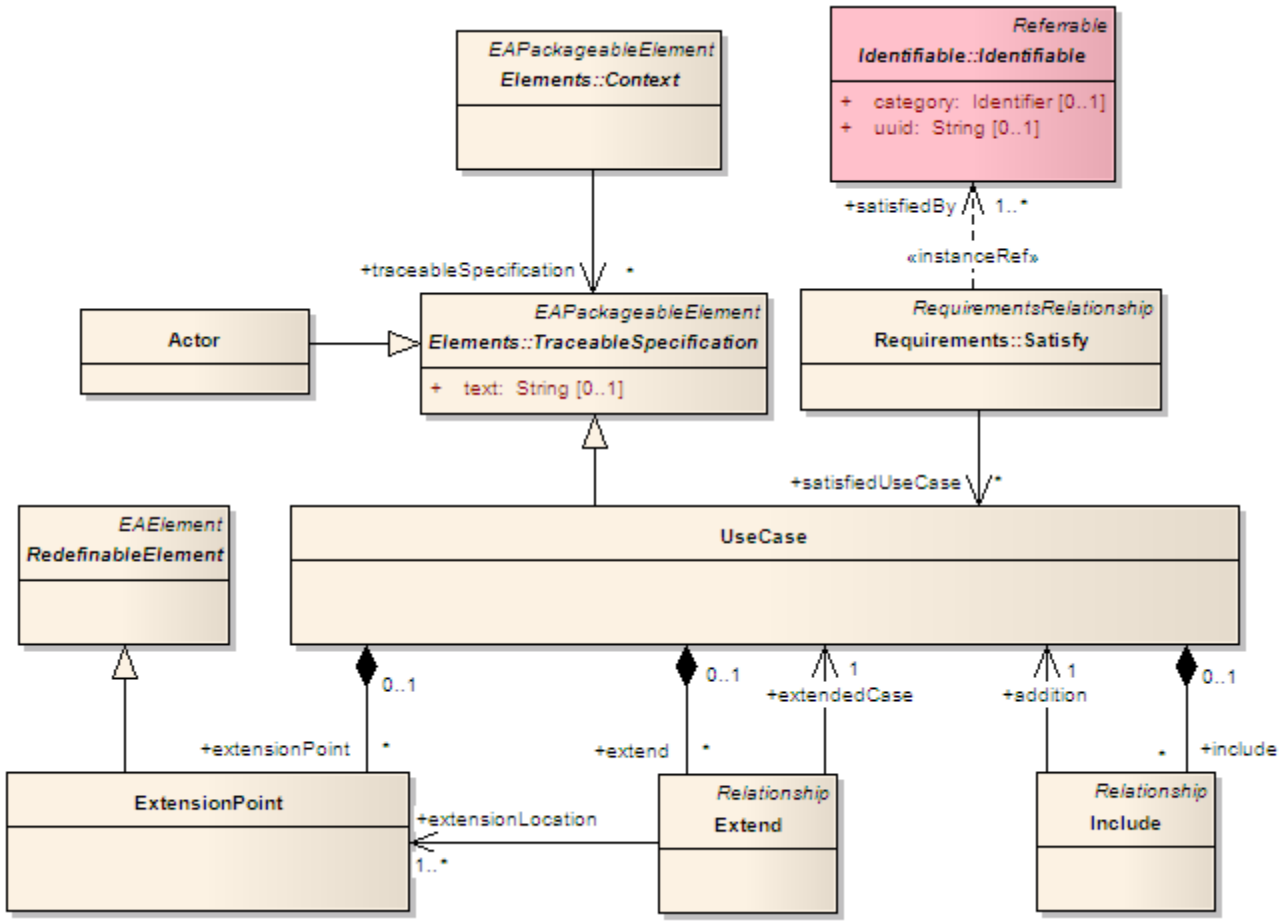

Figure 18. Diagram for UseCase.

12.2 Element Descriptions

12.2.1 Actor (from UseCases)

Generalizations

- TraceableSpecification (from Elements)

Description

Actor represents a type of role played by an entity that interacts with the UseCase, e.g. by exchanging signals and data, but which is external to the subject, i.e., in the sense that an instance of an Actor is not a part of the instance of its corresponding subject. Actors may represent roles played by human users, external hardware, or other subjects. Note that an Actor does not necessarily represent a specific physical entity but merely a particular facet (i.e., "role") of some entity that is relevant to the specification of its associated UseCases. Thus, a single physical instance may play the role of several different Actors and, conversely, a given Actor may be played by multiple different instances. Since an Actor is external to the subject, it is typically defined in the same classifier or package that incorporates the subject classifier.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The Actor element represents entities that interacts with a UseCase.

12.2.2 Extend (from UseCases)

Generalizations

- Relationship (from Elements)

Description

Extend represents the specification that the behavior of a UseCase may be extended by the behavior of another (usually supplementary) UseCase. The extension takes place at one or more specific ExtensionPoints defined in the extended UseCase. Note, however, that the extended UseCase is defined independently of the extending UseCase and is meaningful independently of the extending UseCase. On the other hand, the extending UseCase typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending UseCase defines a set of modular behavior increments that augment an execution of the extended UseCase under specific conditions. Note that the same extending UseCase can extend more than one UseCases. Furthermore, an extending UseCase may itself be extended.

Attributes

No additional attributes

Associations

- extensionLocation : ExtensionPoint [1..*]
  Identifies a point where the behavior of a UseCase can be augmented with elements of another (extending) UseCase.

- extendedCase : UseCase [1]
  The UseCase that is extended.

Constraints

No additional constraints

Semantics

An Extension relation identifies an extension UseCase which extends an extendedCase UseCase.

12.2.3 ExtensionPoint (from UseCases)

Generalizations

- RedefinableElement (from UseCases)

Description

ExtensionPoint represents a feature of a UseCase that identifies a point where the behavior of a UseCase can be augmented with elements of another (extending) UseCase.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

ExtensionPoint identifies the point where the useCase UseCase ca be extended.

12.2.4 Include (from UseCases)

Generalizations

- Relationship (from Elements)

Description

Include is a specialization of the Relationship and represents a relationship between two UseCases, implying that the behavior of the included UseCase is inserted into the behavior of the including UseCase. The including UseCase may only depend on the result (value) of the included UseCase. This value is obtained as a result of the execution of the included UseCase. Note that the included UseCase is not optional and is always required for the including UseCase to execute correctly.

Attributes

No additional attributes

Associations

- addition : UseCase [1]
  UseCase providing behavior to include.

Constraints

No additional constraints

Semantics

The Include relationship identifies an addition UseCase, which is inserted in the including UseCase.

12.2.5 RedefinableElement (from UseCases) {abstract}

Generalizations

- EAElement (from Elements)

Description

RedefinableElement represents an element that, when defined in the context of a classifier, can be redefined more specifically or differently in the context of another classifier that specializes (directly or indirectly) the context classifier

A redefinable element is a named element that can be redefined in the context of a generalization.

The RedefinableElement is an abstract metaclass.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

RedefinableElementrepresents an element that can be redefined in the context of another classifier. Semantics is given by its specializations.

12.2.6 UseCase (from UseCases)

Generalizations

- TraceableSpecification (from Elements)

Description

A UseCase specifies a usage of a system. Typically, they are used to capture the functionality of a system, that is, what a system is supposed to do.

Attributes

No additional attributes

Associations

- extensionPoint : ExtensionPoint [*] {composite}
  An ExtensionPoint identifies a point where the behavior of a UseCase can be augmented with elements of another (extending) UseCase.
- include : Include [*] {composite}
  Include is a Relationship between two UseCases; the behavior of the included UseCase is inserted into the behavior of the including UseCase.
- extend : Extend [*] {composite}
  This Relationship specifies that the behavior of a UseCase may be extended by the behavior of another (usually supplementary) UseCase.

Constraints

No additional constraints

Semantics

A UseCase identifies a usage of its corresponding system. extensionPoint identifies where the use case can be extended with extend UseCases and include identifies UseCases inserted in the including UseCase.

13 VerificationValidation

13.1 Overview

Many different verification and validation (V&V) techniques, methods, and tools are applied during the development of electrical/electronic systems. Different techniques are applicable at different abstraction levels. Also, choosing a technique depends on the properties being validated and/or verified. Furthermore, each partner in a project may develop and employ his own V&V processes and activities. Hence it is impossible for EAST-ADL to model all the objects that can be required by all the possible V&V techniques. As a consequence, EAST-ADL provides the means for planning, organizing and describing V&V activities on a fairly abstract level, and defines the links between those V&V activities, the satisfied and verified requirements, and the objects modeling the system (Functional Analysis Architecture, Functional components, Logical Tasks, etc.). EAST-ADL describes the common parts of all V&V techniques, including: the results expected from the V&V activities, the actual results which were obtained when applying the V&V techniques, and how the V&V activities are constrained. Information that is specific to an individual V&V technique is not described in EAST-ADL, but a place for storing this information is provided.

Individual V&V techniques may be used once or at several stages during an overall V&V effort. Some of them are specific to one modeling design stage; others can be applied at various design stages.

A set of V&V techniques and activities is necessary in order to completely verify and validate a given system. Often these techniques and activities are employed and performed by many different teams and departments, even by different companies. This situation demands the planning and organization of all V&V related information.

A very important aspect of V&V support in EAST-ADL is the distinction between abstract and concrete V&V information:

(1) At an abstract level, verification and validation information is defined without referring to a concrete testing environment and without specifying stimuli or the expected outcome of a particular VVProcedure on a detailed technical level.

(2) On the concrete level, verification and validation information specifies a concrete testing environment and provides all necessary details for testing, e.g. stimuli and expected outcomes, on a concrete technical level applicable to that testing environment.

Using a "what vs. how" definition of requirements one could say: the abstract level defines what needs to be done to verify and validate a certain system, but not precisely how this is done. Conversely, the concrete level defines the precise technical details of particular testing environments. The abstract VVCases and VVProcedures for a particular system form a "to-do"-list, which describes what needs to be done when actually testing the system with a concrete testing environment, but in a form applicable to all conceivable testing environments.

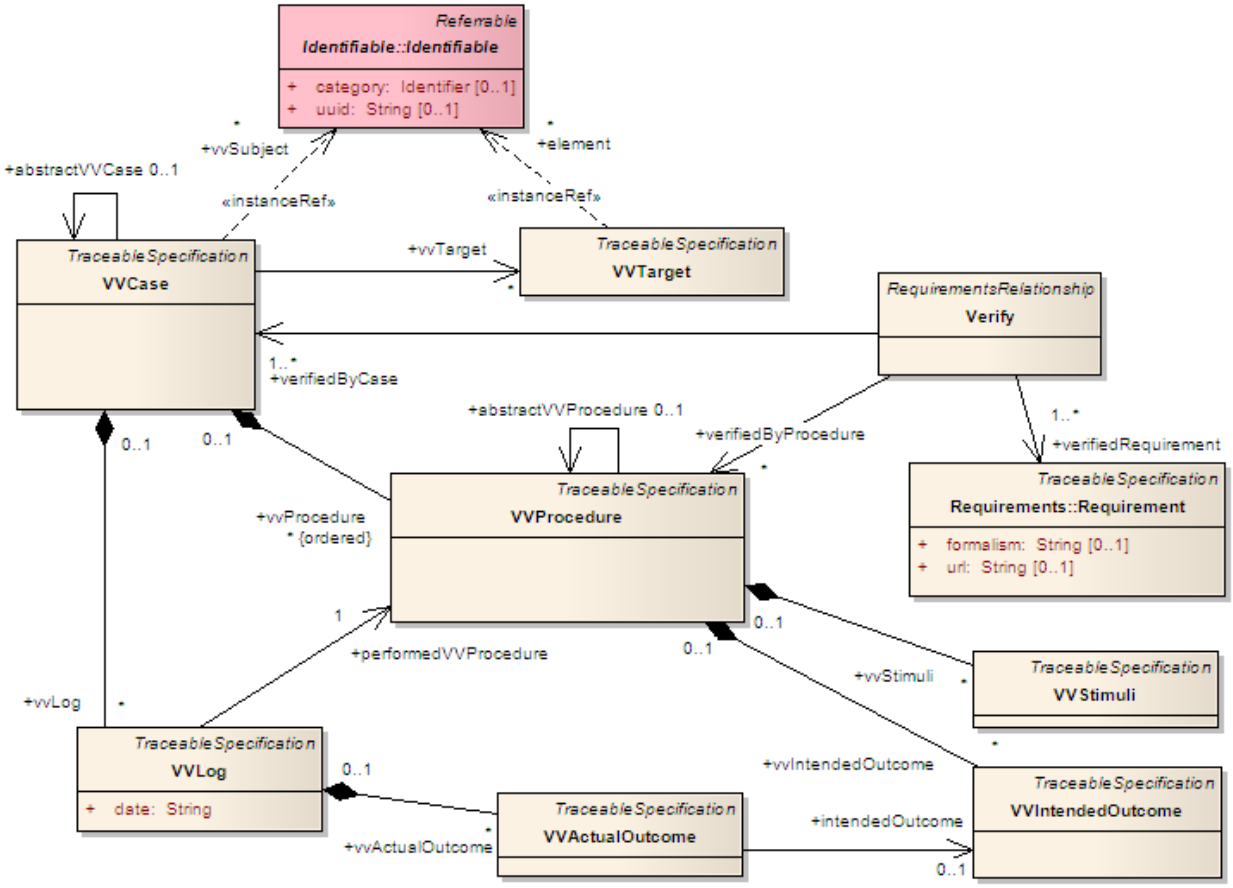

Figure 19. Diagram for Verification & Validation.

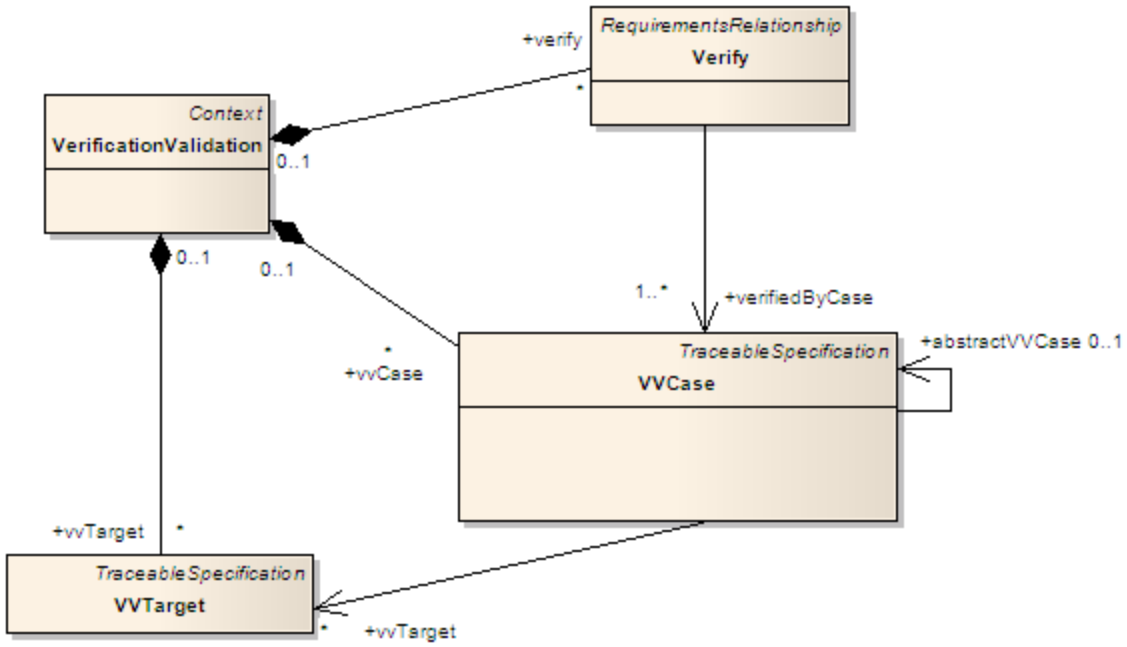

Figure 20. Diagram for Verification and Validation Organization.

13.2 Element Descriptions

13.2.1 VerificationValidation (from VerificationValidation)

Generalizations

- Context (from Elements)

Description

The collection of verification and validation elements. This collection can be used across the EAST-ADL abstraction levels.

Attributes

No additional attributes

Associations

- vvTarget : VVTarget [*] {composite}
- vvCase : VVCase [*] {composite}
- verify : Verify [*] {composite}

Constraints

No additional constraints

Semantics

VerificationValidation is a container element for a set of related vvTarget and vvCase elements and verify relationships.

13.2.2 Verify (from VerificationValidation)

Generalizations

- RequirementsRelationship (from Requirements)

Description

Verify is a relationship metaclass, which signifies a dependency relationship between a Requirement and a VVCase. It shows the relationship when a client VVCase and an optional abstract VVProcedure verifies the supplier Requirement.

Attributes

No additional attributes

Associations

- verifiedRequirement : Requirement [1..*]
  The set of Requirements which the client VVCase verify.
- verifiedByProcedure : VVProcedure [*]
  The abstract VVProcedures used to verify the Requirement.
- verifiedByCase : VVCase [1..*]
  The VVCase that verifies the supplier Requirement

Constraints

No additional constraints

Semantics

The Verify metaclass signifies a refined requirement/verified by relationship between a Requirement and a VVCase, where the modification of the supplier Requirement may impact the verifying client VVCase and optional abstract VVProcedure. The Verify metaclass implies that the semantics of the verifying client VVCase is not complete, without the supplier Requirement.

13.2.3 VVActualOutcome (from VerificationValidation)

Generalizations

- TraceableSpecification (from Elements)

Description

VVActualOutcome represents the actual output of the testing environment as represented by VVTarget when triggered by the VVStimuli of the concrete VVProcedure. This is defined by the association 'performedVVProcedure' of the containing VVLog. It should be equivalent to the VVIntendedOutcome defined by the association 'intendedOutcome'.

Attributes

No additional attributes

Associations

- intendedOutcome : VVIntendedOutcome [0..1]
  Denotes the VVIntendedOutcome that this actual outcome must match.

Constraints

No additional constraints

Semantics

VVActualOutcome represents the actual output of a verification effort as defined by the V&V elements.

13.2.4 VVCase (from VerificationValidation)

Generalizations

- TraceableSpecification (from Elements)

Description

VVCase represents a V&V effort, i.e. it specifies concrete test subjects and targets and provides stimuli and the expected outcome on a concrete technical level.

Attributes

No additional attributes

Associations

- vvProcedure : VVProcedure [*] {ordered} {composite}
  The VVProcedures for this VVCase.
- vvTarget : VVTarget [*]
  The VVTargets for this VVCase. See association ‘vvSubjects’ for more information.
- vvLog : VVLog [*] {composite}
  The VVLogs captured while executing this ConcreteVVCase.
- abstractVVCase : VVCase [0..1]

An abstract VVCase describes "what" needs to be done and is identified from a concrete VVCase.

Dependencies

- vvSubject : Identifiable [*]
  «instanceRef»

Constraints

[1] Only a concrete VVCase can have vvLog.

[2] Only a concrete VVCase can have vvTarget.

[3] Only a concrete VVCase can have an abstractVVCase.

Semantics

VVCase is a grouping element for a set of VVProcedures that together makes up a concrete Verification/Validation effort.

13.2.5 VVIntendedOutcome (from VerificationValidation)

Generalizations

- TraceableSpecification (from Elements)

Description

VVIntendedOutcome represents the expected output of the testing environment represented by VVTarget when triggered by the corresponding VVStimuli of the containing concrete VVProcedure.

Since this entity only occurs on the concrete level (i.e. within the context of a concrete VVCase), the output must be provided in a form that can be directly compared to the output of the VVTarget(s) defined for the containing concrete VVCase.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

VVIntendedOutcome represents the expected output of a Verification/Validation effort.

13.2.6 VVLog (from VerificationValidation)

Generalizations

- TraceableSpecification (from Elements)

Description

Concrete VVCase represents the precise description of a V&V effort on a concrete technical level and thus provides all necessary information to actually perform this V&V effort.

However, it does not represent the actual execution of the effort.

This is the purpose of the VVLog. Each VVLog metaclass represents an execution of a concrete VVCase.

For example, if the HIL test of the wiper system with a certain set of stimuli was performed on Friday afternoon and, for checkup, again on Monday, then there will be one ConcreteVVCase describing the HIL test and two VVLogs describing the test results from Friday and Monday respectively.

Attributes

- date : String [1]
  Date and time when this log was captured.

Associations

- performedVVProcedure : VVProcedure [1]
  Associated procedure.
- vvActualOutcome : VVActualOutcome [*] {composite}
  Set of outcome results.

Constraints

No additional constraints

Semantics

VVLog captures an execution of a ConcreteVVCase.

13.2.7 VVProcedure (from VerificationValidation)

Generalizations

- TraceableSpecification (from Elements)

Description

VVProcedure represents an individual task in a V&V effort (represented by a VVCase), which has to be performed in order to achieve that effort's overall objective. As with VVCases, the definition of VVProcedures is separated in to two levels: an abstract and a concrete level.

The concrete VVProcedure represents such a task on a concrete level It is defined with a concrete testing environment in mind and provides stimuli and the expected outcome of the procedure in a form which is directly applicable to this testing environment.

Attributes

No additional attributes

Associations

- abstractVVProcedure : VVProcedure [0..1]
  An abstract VVProcedure identified from a concrete VVProcedure.
- vvStimuli : VVStimuli [*] {composite}
  Set of involved stimuli.
- vvIntendedOutcome : VVIntendedOutcome [*] {composite}
  Set of intended outcomes.

Constraints

[1] Only a concrete VVProcedure can have vvStimuli.

[2] Only a concrete VVProcedure can have vvIntendedOutcome.

[3] Only a concrete VVProcedure can have an abstractVVProcedure .

Semantics

VVProcedure represents an individual task in a Verifcation/Validation effort.

13.2.8 VVStimuli (from VerificationValidation)

Generalizations

- TraceableSpecification (from Elements)

Description

VVStimuli represents the input values of the testing environment represented by VVTarget in order to perform the corresponding VVProcedure.

Since this entity only occurs on the concrete level (i.e. within the context of a concrete VVCase), the input values must be provided in a form that is directly applicable to the VVTarget(s) defined for the containing concrete VVCase.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

VVStimuli represents the concrete input values used for a VVProcedure during a Verification/Validation effort of a VVTarget.

13.2.9 VVTarget (from VerificationValidation)

Generalizations

- TraceableSpecification (from Elements)

Description

VVTarget represents a concrete testing environment in which a particular V&V activity can be performed. This can be physical hardware or it can be pure software in case of a test by way of design level simulations.

Usually, a VVTarget will identify one or more elements. However, there are cases in which this is not true, for example when a VVTarget represents parts of the system's environment. Therefore the association to element has a minimum cardinality of 0.

VVTargets can be reused across several concrete VVCases.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- element : Identifiable [*]

«instanceRef»

Constraints

No additional constraints

Semantics

VVTarget represents a concrete testing environment in which a particular Verifcation/Validation activity is performed.

Part VI Timing

This part contains the timing constructs for EAST-ADL, which are organized in into events and constraints.

14 Timing

14.1 Overview

The timing package contains constructs for defining timing constraints.

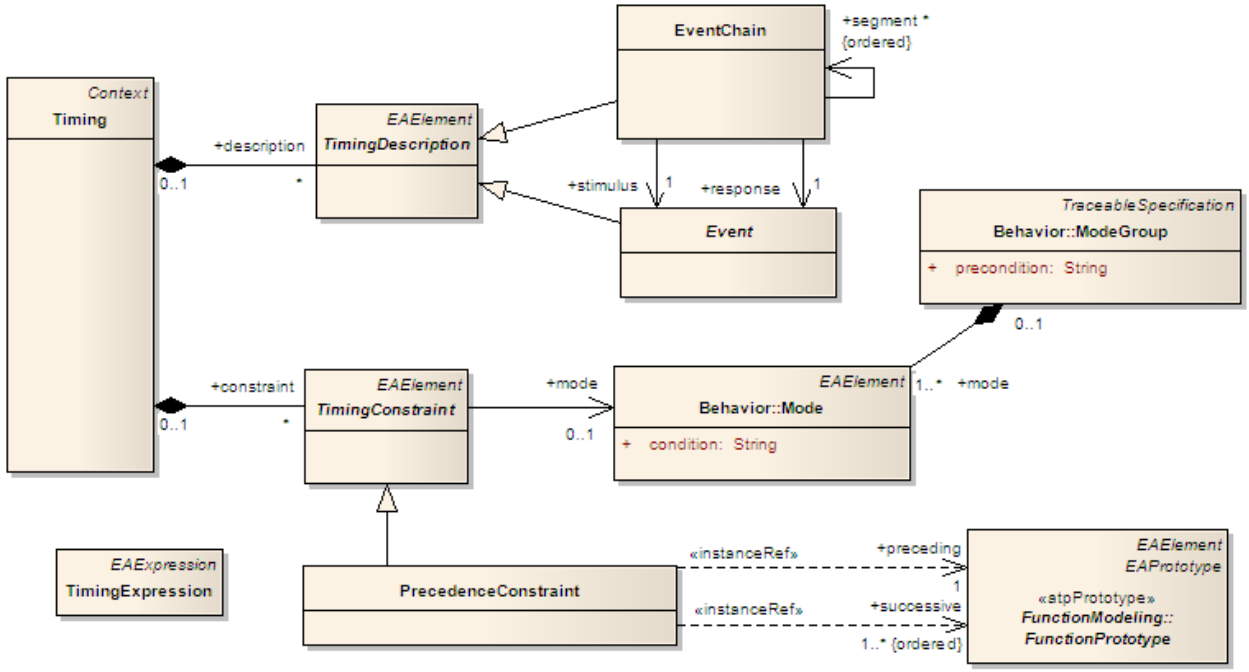

Figure 21. Basic TADL2 elements organized in Timing, with TimingConstraints referring to EAST-ADL Mode.

14.2 Element Descriptions

14.2.1 Event (from Timing) {abstract}

Generalizations

- TimingDescription (from Timing)

Description

The Event class stands for all the forms of identifiable state changes that are possible to constrain with respect to timing using TADL2.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An event denotes a distinct form of state change in a running system, taking place at distinct points in time called occurrence of the event. That is, a running system can be observed by identifying certain forms of state changes to watch for, and for each such observation point, noting the times when changes occur. This notion of observation also applies to a hypothetical predicted run of a system or a system model - from a timing perspective, the only information that needs to be in the output of such a prediction is a sequence of times for each observation point, indicating the times that each event is predicted to occur.

In system models, events appear syntactically as names indicating the state changes of interest. Semantically, an event name is a variable standing for some statically unknown set of occurrences. Note that this connection is purely conceptual; occurrences never exist concretely in any system model as they are a purely semantic notion representing the state changes that can be observed when a system is executed, or simulated, or perhaps only mathematically predicted.

TADL2 assumes that occurrences are characterized by two pieces of information: a timestamp indicating when the corresponding state change occurred, and a color that partitions different event occurrences into groups that should be understood as being causally related. The timestamp is a real value of SI unit seconds, whereas the color value is drawn from some abstract, possibly infinite type whose only restriction is that must support an equality test on its values.

14.2.2 EventChain (from Timing)

Generalizations

- TimingDescription (from Timing)

Description

An EventChain is a container for a pair of events that must be causally related.

Attributes

No additional attributes

Associations

- stimulus : Event [1]
  The event that stimulates the steps to be taken to respond to this event.
- response : Event [1]
  The event that is a response to a stimulus that occurred before.
- segment : EventChain [*] {ordered}
  Referred EventChains in sequence refine this EventChain.

Constraints

No additional constraints

Semantics

A system behavior is consistent with respect to an event chain ec if and only if

for each occurrence x in ec.stimulus,

for each occurrence y in ec.response,

if x.color = y.color then x < y

14.2.3 PrecedenceConstraint (from Timing)

Generalizations

- TimingConstraint (from Timing)

Description

The PrecedenceConstraint represents a particular constraint applied on the execution sequence of functions.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- successive : FunctionPrototype [1..*]
  «instanceRef»
- preceding : FunctionPrototype [1]
  «instanceRef»

Constraints

No additional constraints

Semantics

The semantics for the PrecedenceConstraint metaclass is to define an association relationship between Functions, indicating the association relationship such that all predecessors have completed before the successors are started.

Note: Without a precedence relation, Functions are executed according to their data dependencies, if these are uni-directional. For bi-directional data dependencies, execution order is not defined unless the PrecedenceDependency relationship is used.

14.2.4 Timing (from Timing)

Generalizations

- Context (from Elements)

Description

The collection of timing descriptions, namely events and event chains, and the timing constraints imposed on these events and event chains. This collection can be done across the EAST-ADL abstraction levels.

Attributes

No additional attributes

Associations

- constraint : TimingConstraint [*] {composite}
- description : TimingDescription [*] {composite}

Constraints

No additional constraints

Semantics

-

14.2.5 TimingConstraint (from Timing) {abstract}

Generalizations

- EAElement (from Elements)

Description

This abstract element references a mode in order to indicate that the corresponding TimingConstraint is only valid when the specified mode is active.

Attributes

No additional attributes

Associations

- mode : Mode [0..1]
  Reference to the mode in which the timing constraint is valid.

Constraints

No additional constraints

Semantics

The TimingConstraint does not describe what is classically referred to as a "design" constraint but has the role of a property, requirement, or a validation result. It is a requirement if this TimingConstraint refines a Requirement (by the Refine relationship). The TimingConstraint is a validation result if it realizes a VVActualOutcome, it is an intended validation result if it realizes a VVIntendedOutcome, and in other cases it denotes a property.

14.2.6 TimingDescription (from Timing) {abstract}

Generalizations

- EAElement (from Elements)

Description

An abstract metaclass describing the timing events and their relations by event chains within the timing model.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

14.2.7 TimingExpression (from Timing)

Generalizations

- EAExpression (from Values)

Description

A Timing Expression, denoted by texp, is a term built from an arithmetic expression by applying an optional unit and referencing an optional time base. It stands for a value in the real number system extended with positive and negative infinity.

Grammar:

texp ::= aexp

| aexp UN

| aexp on TB

| aexp UN on TB

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Given a particular variable assignment, the meaning of a timing expression texp in that assignment is a value in the real number system extended with positive and negative infinity. Depending on the form of texp, this value is defined as follows:

- If texp is of the form aexp, its meaning is the meaning of aexp in the given variable assignment.

- If texp is of the form aexp UN, its meaning is r * k, where r is the meaning of aexp in the given variable assignment, and k is the factor of UN in the Universal time base.

- If texp is of the form aexp on TB, its meaning is f (r), where f is the meaning of TB in the given variable assignment, and r is the meaning of aexp in the same assignment.

- If texp is of the form aexp UN on TB, its meaning is f (r * k), where f is the meaning of TB in the given variable assignment, r is the meaning of aexp in the same assignment, k is the factor of UN in DI, and DI is the dimension of TB.

15 TimingConstraints

15.1 Overview

TADL2 offers a palette of means to constrain the time occurrences of events. These can roughly be grouped into restrictions on the recurring delays between a pair of events, restrictions on the repetitions of a single event, and restrictions on the synchronicity of a set of events. All constraints provided by TADL2 are defined in this package.

The semantics of some timing constraint is described by references to other timing constraints in this package. Default attribute values, which apply in a right-to-left manner whenever a constraint argument list is too short to match all defined attributes, are given when applicable.

A helper constraint RepeatConstraint is defined in TADL2, in modeling a RepetitionConstraint with jitter = 0 is used instead.

A system behavior satisfies a RepeatConstraint c if and only if

for each subsequence X of c.event,

if X contains span + 1 occurrences then

e is the distance between the outermost

occurrences in X

and

c.lower <= e <= c.upper

The RepeatConstraint defines the basic notion of repeated occurrences. If the span attribute is 1 and the lower and upper attributes are equal, the accepted behaviors must be strictly periodic. If span is still 1 but lower is strictly less than upper, the pattern may deviate from a periodic one in an accumulating fashion, making the window within which occurrence number N may appear as wide as N(upper-lower) time units. A span attribute greater than 1 similarly constrains every sequence of span+1 occurrences, but places no restriction on the distances within shorter sequences.

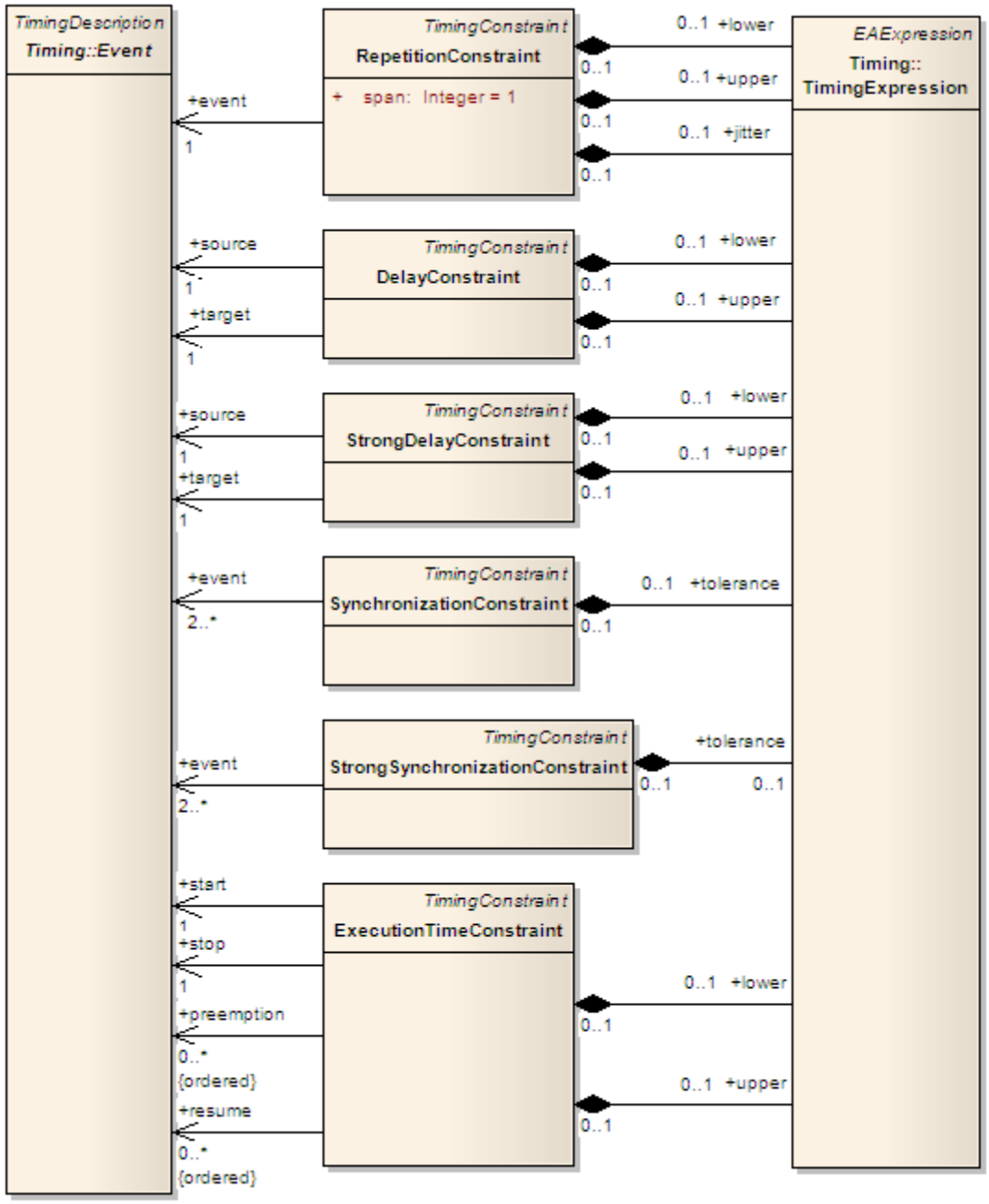

Figure 22. The first of two sets with TADL2 constraints with attributes of type TimingExpression and references to events.

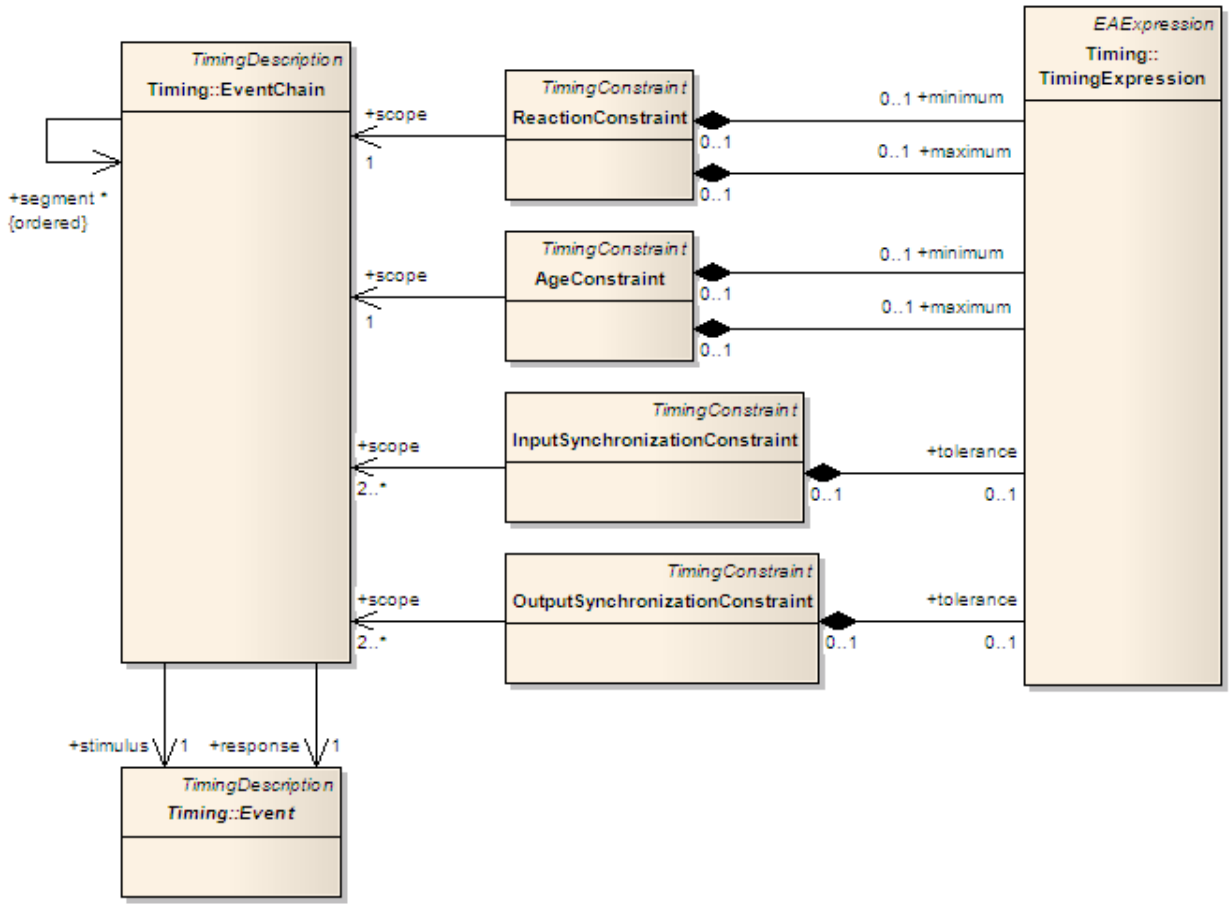

Figure 23. The TADL2 constraints that refer to EventChain, and have attributes of type TimingExpression.

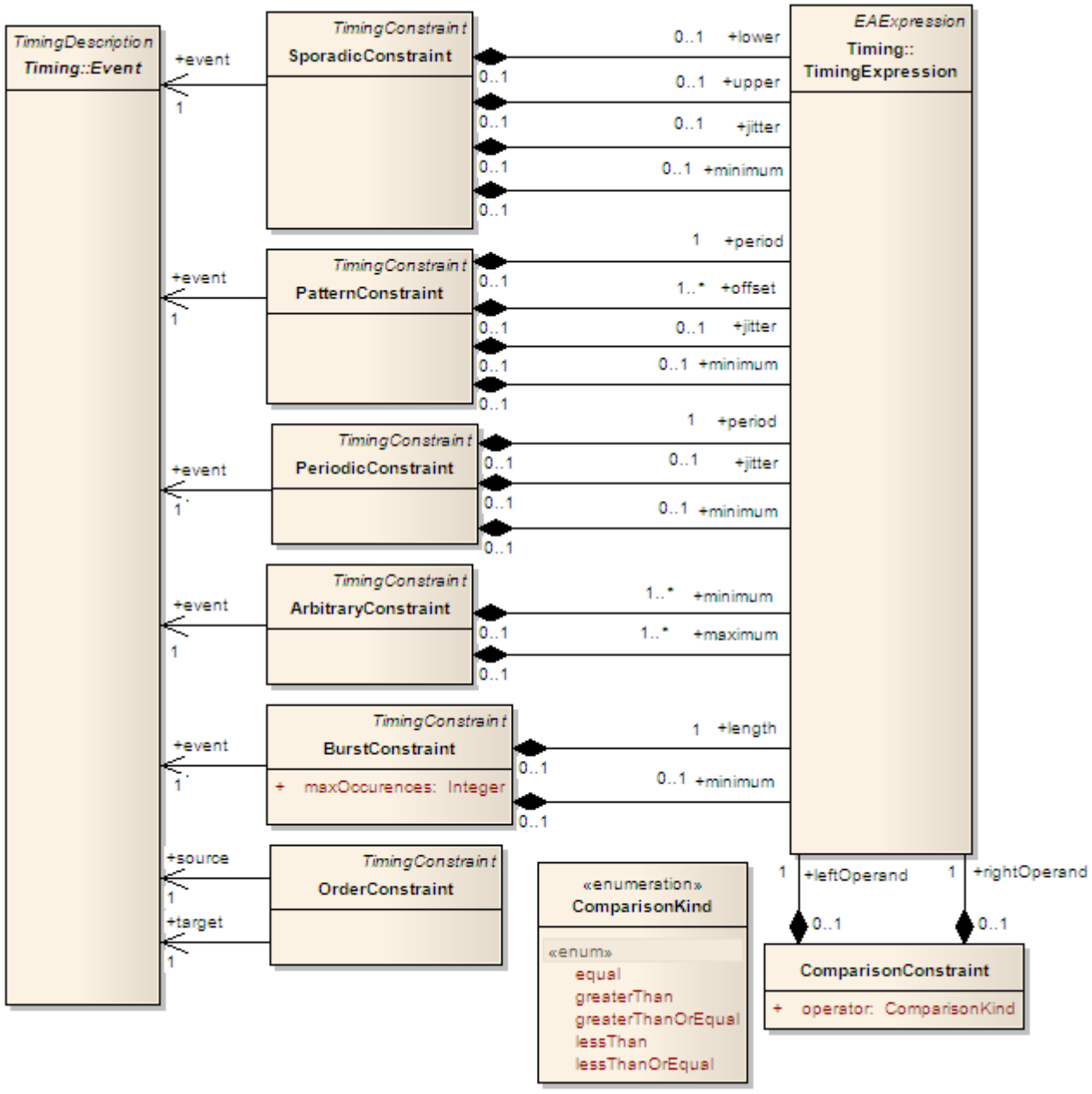

Figure 24. The second of two sets with TADL2 constraints with attributes of type TimingExpression and references to events. Also shown is the ComparisonConstraint with attributes of type TimingExpression.

15.2 Element Descriptions

15.2.1 AgeConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

An AgeConstraint defines how long before each response a corresponding stimulus must have occurred.

This constraint provides an alternative to the ordinary DelayConstraint for situations where the causal relation between event occurrences must be taken into account. It differs from the DelayConstraint in that it applies to an event chain, and only looks at the stimulus occurrences that have the same color as each particular response occurrence. It is the latest of these stimulus occurrences that is required to lie within the prescribed time bounds. If the roles of stimulus and response are swapped, and the time bounds negated, a ReactionConstraint is obtained.

Attributes

No additional attributes

Associations

- scope : EventChain [1]
- maximum : TimingExpression [0..1] {composite}
  Default: infinity
- minimum : TimingExpression [0..1] {composite}
  Default: 0

Constraints

No additional constraints

Semantics

A system behavior satisfies an AgeConstraint c if and only if

for each occurrence y in c.scope.response,

there is an occurrence x in c.scope.stimulus such that

x.color = y.color

and

x is maximal in c.scope.stimulus with that color

and

c.minimum <= y - x <= c.maximum

15.2.2 ArbitraryConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

An ArbitraryConstraint describes an event that occurs irregularly.

An ArbitraryConstraint is equivalent to a combination of Repeat constraints, each one constraining sequences of i+1 occurrences (that is, i repetition spans), with i ranging from 1 to some given n.

Attributes

No additional attributes

Associations

- event : Event [1]
- maximum : TimingExpression [1..*] {composite}
- minimum : TimingExpression [1..*] {composite}

Constraints

[1] The number of elements in minimum and maximum must be equal.

Semantics

A system behavior satisfies an AribtraryConstraint c if and only if

for each c.minimum index i, the same system behavior satisfies

RepeatConstraint { event = c.event,

lower = c.minimum(i),

upper = c.maximum(i),

span = i }

15.2.3 BurstConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A BurstConstraint describes an event that occurs in semi-regular bursts.

A BurstConstraint expresses the maximum number of event occurrences that may appear in any interval of a given length, which is equivalent to constraining the same number of repeat spans (which count one extra occurrence at the end) to have a minimum width of length.

Attributes

- maxOccurences : Integer [1]

Associations

- event : Event [1]
- length : TimingExpression [1] {composite}
- minimum : TimingExpression [0..1] {composite}
  Default: 0

Constraints

No additional constraints

Semantics

A system behavior satisfies a BurstConstraint c if and only if

the same system behavior concurrently satisfies

RepeatConstraint { event = c.event,

lower = c.length,

upper = infinity,

span = c.maxOccurrences }

and

RepeatConstraint { event = c.event,

lower = c.minimum }

15.2.4 ComparisonConstraint (from TimingConstraints)

Generalizations

None

Description

A ComparisonConstraint states that a certain ordering relation must exist between two timing expressions.

This constraint is special in that it does not reference any events. Its main purpose is to express relations between arithmetic variables used in other constraint; for example, stating that the sum of the variables denoting segment delays in a time-budgeting scenario must be less than the maximum end-to-end deadline allowed.

Attributes

- operator : ComparisonKind [1]

Associations

- rightOperand : TimingExpression [1] {composite}
- leftOperand : TimingExpression [1] {composite}

Constraints

No additional constraints

Semantics

A system behavior satisfies a ComparisonConstraint c if and only if

c.leftOperand and c.rightOperand are related according to the ordering relation given by c.operator.

15.2.5 ComparisonKind (from TimingConstraints) «enumeration»

Generalizations

None

Enumeration Literals

- equal
- greaterThan
- greaterThanOrEqual
- lessThan
- lessThanOrEqual

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

15.2.6 DelayConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A DelayConstraint imposes limits between the occurrences of an event called source and an event called target.

This notion of delay is entirely based on the distance between source and target occurrences; whether a matching target occurrence is actually caused by the corresponding source occurrence is of no importance. This means that one-to-many and many-to-one source-target patterns are allowed, and so are stray target occurrences that are not within the prescribed distance of any source occurrence.

Attributes

No additional attributes

Associations

- target : Event [1]
- source : Event [1]
- lower : TimingExpression [0..1] {composite}
  Default: 0
- upper : TimingExpression [0..1] {composite}
  Default: infinity

Constraints

No additional constraints

Semantics

A system behavior satisfies a DelayConstraint c if and only if

for each occurrence x of c.source,

there is an occurrence y of c.target such that

c.lower <= y - x <= c.upper

15.2.7 ExecutionTimeConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

An ExecutionTimeConstraint limits the time between the starting and stopping of an executable entity (function), not counting the intervals when the execution of such an executable entity (function) has been interrupted.

Attributes

No additional attributes

Associations

- preemption : Event [0..*] {ordered}
- stop : Event [1]

- start : Event [1]
- resume : Event [0..*] {ordered}
- upper : TimingExpression [0..1] {composite}
- lower : TimingExpression [0..1] {composite}

Constraints

No additional constraints

Semantics

A system behavior satisfies an ExecutionTimeConstraint c if and only if

for each occurrence x of event c.start,

E is the set of times between x and the next c.stop occurrence, excluding the times between any c.preempt occurrence and its next c.resume occurrence,

and

c.lower <= length of all continuous intervals in E <= c.upper

15.2.8 InputSynchronizationConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

An InputSynchronizationConstraint defines how far apart the responses that belong to a certain stimulus may occur.

This constraint provides an alternative to the ordinary SynchronizationConstraint for situations where the causal relation between event occurrences must be taken into account. It differs from the SynchronizationConstraint in that it applies to a set of event chains, and only looks at the stimulus occurrences that have the same color as each particular response occurrence. It is the latest of these stimulus occurrences for each chain that are required to lie no more than tolerance time units apart. If the roles of stimuli and responses are swapped, an OutputSynchronizationConstraint is obtained.

Attributes

No additional attributes

Associations

- scope : EventChain [2..*]
- tolerance : TimingExpression [0..1] {composite}
  Default: infinity

Constraints

[1] All scopes must reference one common response event.

Semantics

A system behavior satisfies an InputSynchronizationConstraint c if and only if

for each occurrence y in c.scope(1).response,

there is a time t such that for each c.scope index i,

there is an occurrence x in c.scope(i).stimulus such that

y.color = x.color

and

x is maximal in c.scope(i).stimulus with that color

and

0 <= x - t <= c.tolerance

15.2.9 OrderConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

An OrderConstraint imposes an order between the occurrences of an event called source and an event called target.

The OrderConstraint is a minor variant of an application of StrongDelayConstraint with lower set to 0 and upper to infinity; the difference being that the OrderConstraint does not allow matching target and source occurrences to coincide.

Attributes

No additional attributes

Associations

- source : Event [1]
- target : Event [1]

Constraints

No additional constraints

Semantics

A system behavior satisfies an OrderConstraint c if and only if

c.source and c.target have the same number of occurrences,

and for each index i,

if there is an i:th occurrence of c.source at time x, there is

also an i:th occurrence of c.target at time y such that

x < y

15.2.10 OutputSynchronizationConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

An OutputSynchronizationConstraint defines how far apart the responses that belong to a certain stimulus may occur.

This constraint provides an alternative to the ordinary SynchronizationConstraint for situations where the causal relation between event occurrences must be taken into account. It differs from the SynchronizationConstraint in that it applies to a set of event chains, and only looks at the response occurrences that have the same color as each particular stimulus occurrence. It is the earliest of these response occurrences for each chain that are required to lie no more than tolerance time units apart. If the roles of stimuli and responses are swapped, an InputSynchronizationConstraint is obtained.

Attributes

No additional attributes

Associations

- scope : EventChain [2..*]
- tolerance : TimingExpression [0..1] {composite}
  Default: infinity

Constraints

[1] All scopes must reference one common stimulus event.

Semantics

A system behavior satisfies an OutputSynchronizationConstraint c if and only if

for each occurrence x in c.scope(1).stimulus,

there is a time t such that for each c.scope index i,

there is an occurrence y in c.scope(i).response such that

y.color = x.color

and

y is minimal in c.scope(i).response with that color

and

0 <= y - t <= c.tolerance

15.2.11 PatternConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A PatternConstraint describes an event that exhibits a known pattern relative to the occurrences of an imaginary event.

A PatternConstraint requires the constrained event occurrences to appear at a predetermined series of offsets from a sequence of reference points in time that are strictly periodic. The exact placement of these reference points is irrelevant; if one placement exists that is periodic and allows the event occurrences to be reached at the desired offsets, the constraint is satisfied.

Attributes

No additional attributes

Associations

- event : Event [1]

- period : TimingExpression [1] {composite}
- jitter : TimingExpression [0..1] {composite}
  Default: 0
- minimum : TimingExpression [0..1] {composite}
  Default: 0
- offset : TimingExpression [1..*] {composite}

Constraints

No additional constraints

Semantics

A system behavior satisfies a PatternConstraint c if and only if

there is a set of times X such that the same system behavior concurrently satisfies

PeriodicConstraint { event = X,

period = c.period }

and for each c.offset index i,

DelayConstraint { source = X,

target = c.event,

lower = c.offset(i),

upper = c.offset(i) + c.jitter }

and

RepeatConstraint { event = c.event,

lower = c.minimum }

15.2.12 PeriodicConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A PeriodicConstraint describes an event that occurs periodically.

Attributes

No additional attributes

Associations

- event : Event [1]
- minimum : TimingExpression [0..1] {composite}
  Default: 0
- period : TimingExpression [1] {composite}
- jitter : TimingExpression [0..1] {composite}
  Default: 0

Constraints

No additional constraints

Semantics

A system behavior satisfies a PeriodicConstraint c if and only if

the same system behavior satisfies

SporadicConstraint { event = c.event,

lower = c.period,

upper = c.period,

jitter = c.jitter,

minimum = c.minimum }

15.2.13 ReactionConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A ReactionConstraint defines how long after the occurrence of a stimulus a corresponding response must occur.

This constraint provides an alternative to the ordinary DelayConstraint for situations where the causal relation between event occurrences must be taken into account. It differs from the DelayConstraint in that it applies to an event chain, and only looks at the response occurrences that have the same color as each particular stimulus occurrence. It is the earliest of these response occurrences that is required to lie within the prescribed time bounds. If the roles of stimulus and response are swapped, and the time bounds negated, an AgeConstraint is obtained.

Attributes

No additional attributes

Associations

- scope : EventChain [1]
- maximum : TimingExpression [0..1] {composite}
  Default: infinity
- minimum : TimingExpression [0..1] {composite}
  Default: 0

Constraints

No additional constraints

Semantics

A system behavior satisfies a ReactionConstraint c if and only if

for each occurrence x in c.scope.stimulus,

there is an occurrence y in c.scope.response such that

y.color = x.color

and

y is minimal in c.scope.response with that color

and

c.minimum <= y - x <= c.maximum

15.2.14 RepetitionConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A RepetitionConstraint describes the distribution of the occurrences of a single event, including the allowance for jitter.

The RepetitionConstraint extends the basic notion of repeated occurrences by allowing local devitions from the ideal repetitive pattern described by a RepeatConstraint. Its jitter, lower and upper attributes all contribute to the width of the window in which occurrence number N is accepted, according to the formula N(upper-lower) + jitter. That is, with lower = upper, the uncertainty of where occurrence N may be found does not grow with an increasing N, unlike the case when lower differs from upper by a similar amount and jitter is 0. By adjusting all three attributes, a desired balance between accumulating and non-accumulating uncertainties can be obtained.

Attributes

- span : Integer = 1 [1]

Associations

- event : Event [1]
- jitter : TimingExpression [0..1] {composite}
  Default: 0
- lower : TimingExpression [0..1] {composite}
  Default: 0
- upper : TimingExpression [0..1] {composite}
  Default: infinity

Constraints

No additional constraints

Semantics

A system behavior satisfies a RepetitionConstraint c if and only if

the same system behavior concurrently satisfies

RepeatConstraint { event = X,

lower = c.lower,

upper = c.upper,

span = c.span }

and

StrongDelayConstraint { source = X,

target = c.event,

lower = 0,

upper = c.jitter }

15.2.15 SporadicConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A SporadicConstraint describes an event that occurs sporadically.

The SporadicConstraint is just an application of the RepetitionConstraint with a default span attribute of 1, combined with an additional requirement that the effective minimum distance between any two occurrences must be at least the value given by minimum (even if lower-jitter would suggest a smaller value).

Attributes

No additional attributes

Associations

- event : Event [1]
- lower : TimingExpression [0..1] {composite}
  Default: 0
- minimum : TimingExpression [0..1] {composite}
  Default: 0
- upper : TimingExpression [0..1] {composite}
  Default: infinity
- jitter : TimingExpression [0..1] {composite}
  Default: 0

Constraints

No additional constraints

Semantics

A system behavior satisfies a SporadicConstraint c if and only if

the same system behavior concurrently satisfies

RepetitionConstraint { event = c.event,

lower = c.lower,

upper = c.upper,

jitter = c.jitter }

and

RepeatConstraint { event = c.event,

lower = c.minimum }

15.2.16 StrongDelayConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A StrongDelayConstraint imposes limits between each indexed occurrence of an event called source and the identically indexed occurrence of an event called target.

The strong delay notion requires source and target occurrences to appear in lock-step. Only one-to-one source-target patterns are allowed, and no stray target occurrences are accepted.

Strong synchronization differs from the ordinary form of SynchronizationConstraint by grouping event occurrences into synchronization clusters strictly according to their index. This means that multiple occurrences of a single event cannot belong to a single cluster, and clusters may not share occurrences. Strong synchronization tightens the requirements compared to ordinary synchronization in much the same way as StrongDelayConstraint refines the ordinary DelayConstraint.

Attributes

No additional attributes

Associations

- source : Event [1]
- target : Event [1]
- lower : TimingExpression [0..1] {composite}
  Default: 0
- upper : TimingExpression [0..1] {composite}
  Default: infinity

Constraints

No additional constraints

Semantics

A system behavior satisfies a StrongDelayConstraint c if and only if

c.source and c.target have the same number of occurrences,

and for each index i,

if there is an i:th occurrence of c.source at time x

there is also an i:th occurrence of c.target at time y

such that

c.lower <= y - x <= c.upper

15.2.17 StrongSynchronizationConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A StrongSynchronizationConstraint describes how tightly the occurrences of a group of events follow each other.

Attributes

No additional attributes

Associations

- event : Event [2..*]
- tolerance : TimingExpression [0..1] {composite}
  Default: infinity

Constraints

No additional constraints

Semantics

A system behavior satisfies a StrongSynchronizationConstraint c if and only if

there is a set of times X such that for each c.event index i, the same system behavior satisfies

StrongDelayConstraint { source = X,

target = c.event(i),

lower = 0,

upper = c.tolerance }

15.2.18 SynchronizationConstraint (from TimingConstraints)

Generalizations

- TimingConstraint (from Timing)

Description

A SynchronizationConstraint describes how tightly the occurrences of a group of events follow each other.

This form of synchronization only takes the width and completeness of each occurrence cluster into account; it does not care whether som events occur multiple times within a cluster or whether some clusters overlap and share occurrences. In particular, event occurrences are not partitioned into clusters according to their role or what has caused them. Stray occurrences of single events are not allowed, though, since these would just count as incomplete clusters according to this constraint.

Attributes

No additional attributes

Associations

- event : Event [2..*]
- tolerance : TimingExpression [0..1] {composite}
  Default: infinity

Constraints

No additional constraints

Semantics

A system behavior satisfies a SynchronizationConstraint c if and only if

there is a set of times X such that for each c.event index i, the same system behavior concurrently satisfies

DelayConstraint { source = X,

target = c.event(i),

lower = 0,

upper = c.tolerance }

and

DelayConstraint { source = c.event(i),

target = X,

lower = -c.tolerance,

upper = 0}

16 Events

16.1 Overview

This section describes the concept of events for EAST-ADL.

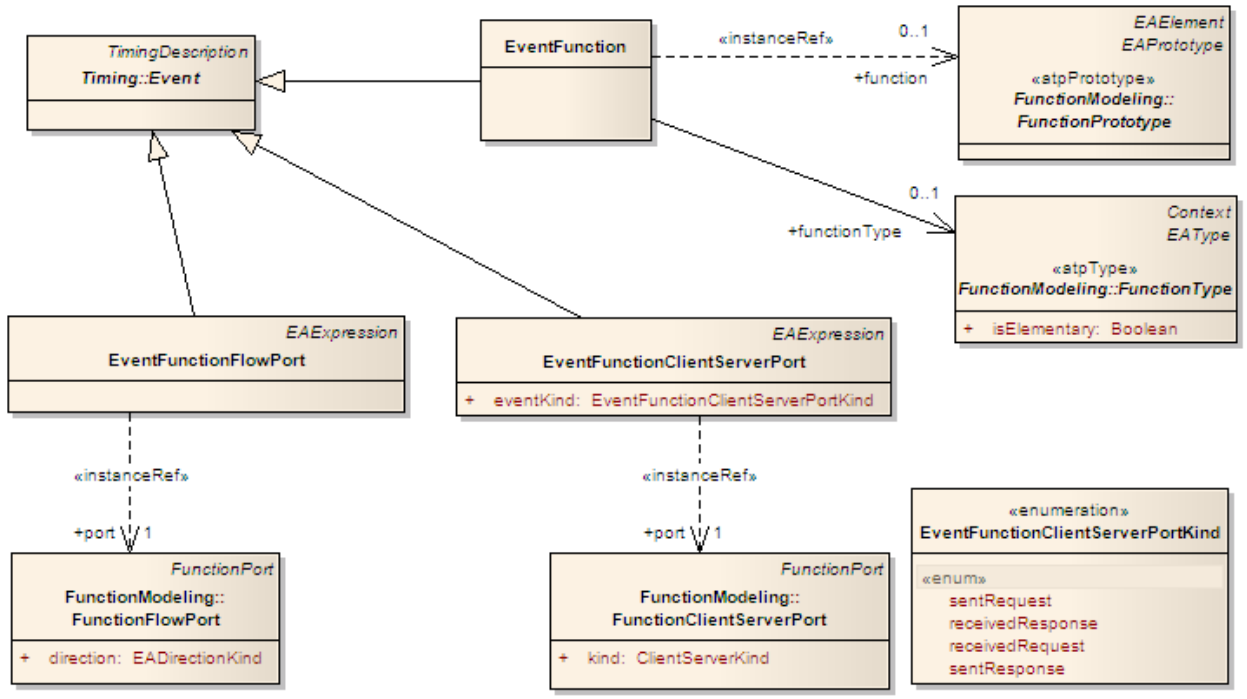

Figure 25. The events for EAST-ADL functional modeling.

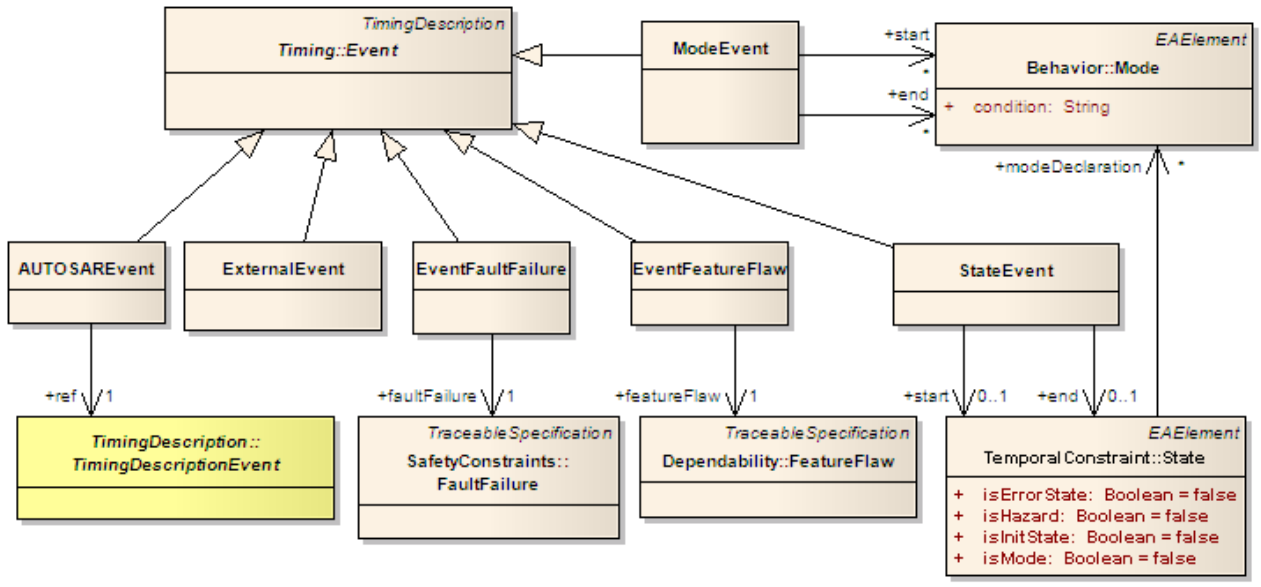

Figure 26. The Events are defined within AUTOSAR and EAST-ADL. These events refer to the structural models of AUTOSAR and EAST-ADL respectively.

16.2 Element Descriptions

16.2.1 AUTOSAREvent (from Events)

Generalizations
- Event (from Timing)

Description

An AUTOSAREvent instance refers to an event of the form defined by AUTOSAR.

Attributes

No additional attributes

Associations
- ref : TimingDescriptionEvent [1]

Constraints

No additional constraints

Semantics

-

16.2.2 EventFaultFailure (from Events)

Generalizations
- Event (from Timing)

Attributes

No additional attributes

Associations
- faultFailure : FaultFailure [1]

Constraints

No additional constraints

Semantics

-

16.2.3 EventFeatureFlaw (from Events)

Generalizations
- Event (from Timing)

Attributes

No additional attributes

Associations
- featureFlaw : FeatureFlaw [1]

Constraints

No additional constraints

Semantics

-

16.2.4 EventFunction (from Events)

Generalizations

- Event (from Timing)

Description

An event of a Function refers to the triggering of the Function, i.e., when the input data is consumed. It can be used in conjunction with FunctionTrigger to define a time-driven triggering for a function. In this case the FunctionTrigger points to the EventFunction of the function and defines a triggerPolicy set to TIME. The timing constraint associated to the EventFunction provides information about the period.

Compare categories of AUTOSAR runnables:

1a triggering only on start and finish (this type of event)

1b triggering allowed anytime during the execution (events on ports, see EventFunctionFlowPort).

Attributes

No additional attributes

Associations

- functionType : FunctionType [0..1]
  The event is valid for all occurences of this function.

Dependencies

- function : FunctionPrototype [0..1]
  «instanceRef»

Constraints

[1] An EventFunction either identifies a FunctionType or a FunctionPrototype as its target function.

Semantics

The EventFunction refers to the triggering event of a referenced functionType or function (prototype). Triggering is the time when the function consumes data.

16.2.5 EventFunctionClientServerPort (from Events)

Generalizations

- Event (from Timing)
- EAExpression (from Values)

Description

Event that refers to the triggering of the Function at a client/server port, i.e., when the input data is sent / received, or when the output data is produced / received.

Attributes

- eventKind : EventFunctionClientServerPortKind [1]

Associations

No additional associations

Dependencies

- port : FunctionClientServerPort [1]
  «instanceRef»

Constraints

[1] eventKind is sentRequest or receivedResponse for a FunctionClientServerPort of type client.
Rationale: Only these values make sense for client ports.

[2] eventKind is receivedRequest or sentResponse for a FunctionClientServerPort of type server.
Rationale: Only these values make sense for server ports.

Semantics

EventFunctionClientServerPort refers to the time when data is sent or received at the ClientServerPort.

16.2.6 EventFunctionClientServerPortKind (from Events) «enumeration»

Generalizations

None

Description

Possible values of eventKind.

Enumeration Literals

- receivedRequest
  Request arrived at server.
- receivedResponse
  Response arrived at client.
- sentRequest
  Request sent from client.
- sentResponse
  Response sent from server.

Associations

No additional associations

Constraints

No additional constraints

Semantics

See each literal.

16.2.7 EventFunctionFlowPort (from Events)

Generalizations

- Event (from Timing)
- EAExpression (from Values)

Description

Event that refers to the triggering of the Function at a flow port, i.e., when data is sent or received.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- port : FunctionFlowPort [1]
  «instanceRef»

Constraints

No additional constraints

Semantics

EventFunctionFlowPort refers to the time when data is sent or received at the FunctionFlowPort.

16.2.8 ExternalEvent (from Events)

Generalizations

- Event (from Timing)

Description

An ExternalEvent instance stands for some particular form of state change.

It is implied that the attribute description uniquely identifies the intended form of state change. It is also assumed that a description string is sufficiently informative to determine an unambiguous set of occurrences for each observation.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

16.2.9 ModeEvent (from Events)

Generalizations

- Event (from Timing)

Description

A mode that identifies when the mode starts or ends.

Attributes

No additional attributes

Associations

- start : Mode [*]
  The mode that is started.
- end : Mode [*]
  The mode that ends.

Constraints

No additional constraints

Semantics

-

16.2.10 StateEvent (from Events)

Generalizations

- Event (from Timing)

Attributes

No additional attributes

Associations

- end : State [0..1]
- start : State [0..1]

Constraints

No additional constraints

Semantics

-

Part VII Dependability

This part contains elements related to Dependability. It is organized as general Dependability providing support for safety information organization according to ISO 26262, ErrorModel, SafetyConstraints and SafetyRequirements. The SafetyCase package supports safety reasoning.

17 Dependability

17.1 Overview

Dependability of a system is the system's ability to ensure service failures are at a level of frequency and severity that is acceptable. Dependability includes several aspects, namely Availability, Reliability, Safety, Integrity and Maintainability. The Dependability package includes support for defining and classifying safety requirements through preliminary Hazard Analysis Risk Assessment, tracing and categorizing safety requirements according to their role in the safety life-cycle, formalizing safety requirements using safety constraints, formalizing and assessing fault propagation through error models, and organizing evidence of safety in a Safety Case.

The support for safety is designed to support the automotive standard for Functional Safety, ISO/DIS 26262.

FunctionalSafetyConcept Functional Safety Concept

FunctionPort -

FunctionPrototype -

FunctionType -

Ground Indirect: Used to provide a ground in a Safety Case

HardwareComponentPrototype -

HardwareComponentType -

HardwarePin -

Hazard Hazard

HazardousEvent HazardousEvent

Identifiable -

InternalFaultPrototype Indirect: Used to declare internal faults of components of the Error Model.

Item Item

LifecycleStageKind Indirect: Used to define the lifecycle stage of a particular safety case.

Mode Used to represent Operating Mode including Safe State,

OperationalSituation OperationalSituation

ProcessFaultPrototype Indirect: Used to represent process faults in components of an error model. This allows declaration of required development rigor in terms of ASIL through a safety constraint.

QuantitativeSafetyConstraint Used to define Failure Rate

Rationale Indirect: Used to declare Rationale in a safety case

Requirement Used to represent Functional, technical, hardware and software requirements

RequirementsContainer Used to organize requirements in a structure

RequirementsRelationship Used to relate between requirements

SafetyCase Indirect: The safety case can be used to organize the ISO26262 related information showing that the system is safe.

SafetyConstraint Used to define the ASIL level on a particular fault or failure

SafetyGoal Safety Goal

SeverityClassKind Severity enumeration S0, S1, S2 or S3

SystemModel -

TechnicalSafetyConcept Container element for the Technical Safety Requirements allocated to architectural elements, that together form the Technical Safety Concept

TraceableSpecification -

UseCase Used in the role Operational Situation for Hazardous event

Warrant Indirect: Represents warrant in a safety case

VehicleFeature A set of Vehicle Features, realized by architectural elements, makes up the Item

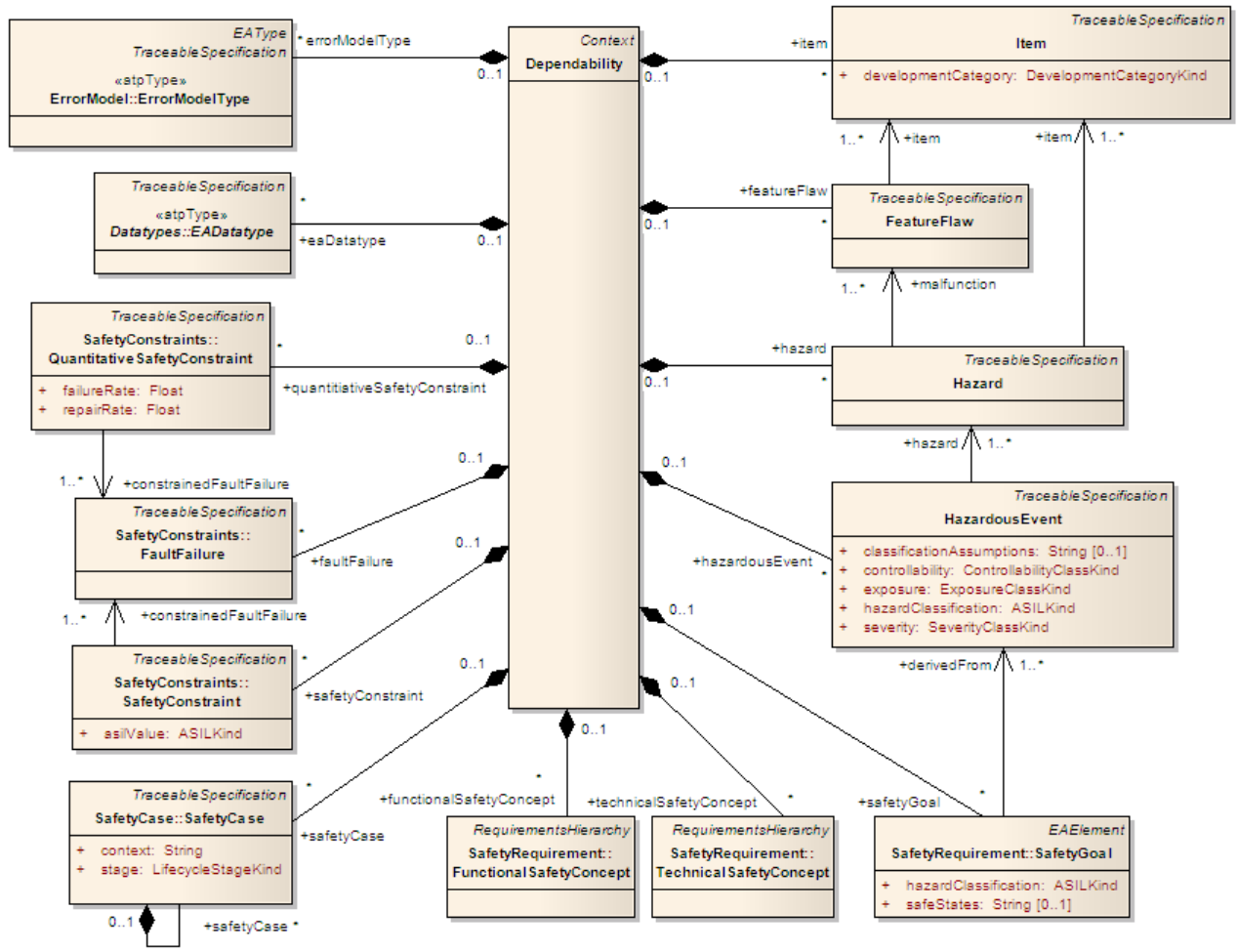

Figure 27. Diagram for organization of dependability related information.

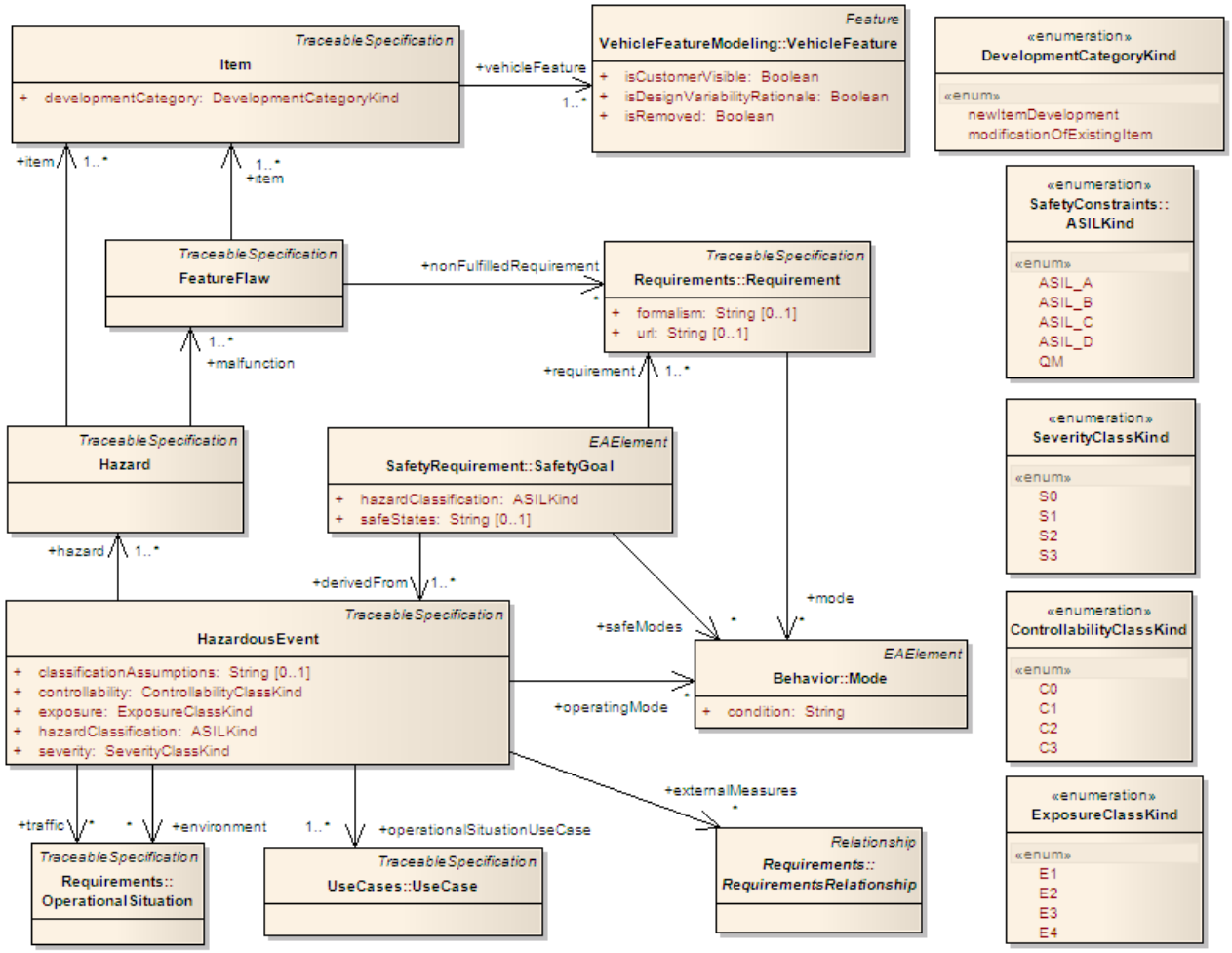

Figure 28. Diagram for Dependability.

17.2 Element Descriptions

17.2.1 ControllabilityClassKind (from Dependability) «enumeration»

Generalizations

None

Description

The ControllabilityClassKind is an enumeration metaclass with enumeration literals indicating controllability attributes C0, C1, C2 or C3 in accordance with ISO26262.

Enumeration Literals

- C0
  Controllable in general.
- C1
  Simply controllable.
- C2

Normally controllable.

- C3
  Difficult to control or uncontrollable.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics are defined at each enumeration literal and fully defined in the ISO26262 standard.

17.2.2 Dependability (from Dependability)

Generalizations

- Context (from Elements)

Description

The collection of dependability related information. This includes safety requirements, safety cases, safety constraints, and error modeling. This collection can be used across the EAST-ADL abstraction levels.

Attributes

No additional attributes

Associations

- technicalSafetyConcept : TechnicalSafetyConcept [*] {composite}
- eaDatatype : EADatatype [*] {composite}
  Datatypes defined in this context.
- safetyCase : SafetyCase [*] {composite}
- quantitiativeSafetyConstraint : QuantitativeSafetyConstraint [*] {composite}
- hazard : Hazard [*] {composite}
- functionalSafetyConcept : FunctionalSafetyConcept [*] {composite}
- faultFailure : FaultFailure [*] {composite}
- errorModelType : ErrorModelType [*] {composite}
- featureFlaw : FeatureFlaw [*] {composite}
- item : Item [*] {composite}
- safetyGoal : SafetyGoal [*] {composite}
- safetyConstraint : SafetyConstraint [*] {composite}
- hazardousEvent : HazardousEvent [*] {composite}

Constraints

No additional constraints

Semantics

Dependability is a container element that collects elements related to dependabilty. It is possible to have several Dependability elements to organize related dependability information in dedicated containers.

17.2.3 DevelopmentCategoryKind (from Dependability) «enumeration»

Generalizations

None

Description

DevelopmentCategoryKind is an enumeration with enumeration literals indicating whether the item is a modification of an existing item or if it is a new development.

Enumeration Literals

- modificationOfExistingItem
  In case of a modification the relevant lifecycle sub-phases and activities shall be determined.
- newItemDevelopment
  In case of a new development, the entire lifecycle shall be passed through.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics are defined at each enumeration literal and fully defined in the ISO26262 standard.

17.2.4 ExposureClassKind (from Dependability) «enumeration»

Generalizations

None

Description

The ExposureClassKind is an enumeration metaclass with enumeration literals indicating the probability attributes E1, E2, E3 or E4 in accordance with ISO26262.

Enumeration Literals

- E1
  Rare events. Situations that occur less often than once a year for the great majority of drivers
- E2
  Sometimes. Situations that occur a few times a year for the great majority of drivers
- E3
  Quite often. Situations that occur once a month or more often for an average driver
- E4
  Often. All situations that occur during almost every drive on average

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics are defined at each enumeration literal and fully defined in the ISO26262 standard.

17.2.5 FeatureFlaw (from Dependability)

Generalizations

- TraceableSpecification (from Elements)

Description

FeatureFlaw denotes an abstract failure of a set of items, i.e. an inability to fulfill one or several of its requirements.

Attributes

No additional attributes

Associations

- nonFulfilledRequirement : Requirement [*]
  Identifies the requirements that are not fulfilled.
- item : Item [1..*]
  The item(s) for which the FeatureFlaw is identified

Constraints

No additional constraints

Semantics

FeatureFlaw represents functional anomalies derivable from each foreseeable source. nonFulfilledRequirements identifies those requirements that correspond to the FeatureFlaw.

17.2.6 Hazard (from Dependability)

Generalizations

- TraceableSpecification (from Elements)

Description

The Hazard metaclass represents a condition or state in the system that may contribute to accidents. The Hazard is caused by malfunctioning behavior of E/E safety-related systems including interaction of these systems.

The Hazard does not address hazards such as electric shock, fire, smoke, heat, radiation, toxicity, flammability, reactivity, corrosion, release of energy, and similar hazards unless directly caused by malfunctioning behavior of safety related electrical/electronic systems.

Attributes

No additional attributes

Associations

- item : Item [1..*]
  The item for which the Hazard is identified

- malfunction : FeatureFlaw [1..*]
  The deviation of the item's operation compared to specified behavior.

Constraints

No additional constraints

Semantics

The Hazard element represents a condition or state in the system that may contribute to accidents. The associated malfunction identifies the FeatureFlaw that corresponds to the Hazard.

17.2.7 HazardousEvent (from Dependability)

Generalizations

- TraceableSpecification (from Elements)

Description

The HazardousEvent metaclass represents a combination of a Hazard and a specific situation, the latter being characterized by operating mode and operational situation in terms of a particular use case, environment and traffic.

Attributes

- classificationAssumptions : String [0..1]
  The classificationAssumptions attribute denotes assumptions concerning the classification of the Hazard.
- controllability : ControllabilityClassKind [1]
  The controllability by the driver or other traffic participants defined by the enumeration C0, C1, C2 or C3 in accordance with ISO26262.
- exposure : ExposureClassKind [1]
  The probability of exposure of the operational situations defined by the probability attributes E1, E2, E3 or E4 in accordance with ISO26262.
- hazardClassification : ASILKind [1]
  The ASIL-Level shall be determined for each hazardous event using the estimation parameters in accordance with ISO26262.
- severity : SeverityClassKind [1]
  The severity of potential harm defined by the severity attributes S0, S1, S2 or S3 in accordance with ISO26262.

Associations

- operatingMode : Mode [*]
  OperatingMode denotes the Operating mode of the item.
- externalMeasures : RequirementsRelationship [*]
- traffic : OperationalSituation [*]
  A definition of the traffic situation in terms of adjacent vehicles, pedestrians and other dynamic aspects. Represents the external and dynamic aspects of the vehicle operating situation .
- environment : OperationalSituation [*]
  A definition of the road environment in terms of road conditions, lanes, geometry, etc. Represents the external and static aspects of the vehicle operating situation .
- operationalSituationUseCase : UseCase [1..*]
  Operational situation with respect to the activities of actors, typically the driver.

- hazard : Hazard [1..*]
  The Hazard that together with the operational situation constitutes the HazardousEvent.

Constraints

No additional constraints

Semantics

The HazardousEvent denotes a combination of a Hazard and an operational situation. The controllability and severity attributes shall be consistent with the operational situation and operational scenario, and the Exposure shall reflect the likelihood of the operational situation and scenario.

17.2.8 Item (from Dependability)

Generalizations

- TraceableSpecification (from Elements)

Description

The Item entity identifies the scope of safety information and the safety assessment, i.e. the part of the system onto which the ISO26262 related information applies. Safety analyses are carried out on the basis of an item definition and the safety concepts are derived from it.

Attributes

- developmentCategory : DevelopmentCategoryKind [1]
  The Item entity identifies the scope of safety information and the safety assessment, i.e. the part of the system onto which the ISO26262 related information applies. Safety analyses are carried out on the basis of an item definition and the safety concepts are derived from it.

Associations

- vehicleFeature : VehicleFeature [1..*]

Constraints

No additional constraints

Semantics

Item represents the scope of safety information and the safety assessment through its reference to one or several Features.

17.2.9 SeverityClassKind (from Dependability) «enumeration»

Generalizations

None

Description

The SeverityClassKind is an enumeration metaclass with enumeration literals indicating the severity attributes S0, S1, S2 or S3 in accordance with ISO26262.

Enumeration Literals

- S0
  No injuries.
- S1

Light and moderate injuries.

- S2
  Severe and life-threatening injuries (survival probable).
- S3
  Life-threatening injuries (survival uncertain), fatal injuries.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics are defined at each enumeration literal and fully defined in the ISO26262 standard.

18 ErrorModel

18.1 Overview

The EAST-ADL sub-package for error modeling provides support for safety engineering by representing possible incorrect behaviors of a system in its operation (e.g., component errors and their propagations).

Abnormal behaviors of architectural elements as well as their instantiations in a particular product context can be represented. This forms a basis for safety analysis through external techniques and tools. Through the integration with other language constructs, definitions of error behaviors and hazards can be traced to the specifications of safety requirements, and further to the subsequent functional and non-functional requirements on error handling and hazard mitigations as well as to the necessary V&V efforts.

Error behaviors are treated as a separated view, orthogonal to the nominal architecture model. This separation of concern in modeling is considered necessary in order to avoid the undesired effects of error modeling, such as the risk of mixing nominal and erroneous behavior in regards to comprehension, reuse, and system synthesis (e.g., code generation).

A key element of the Error Model is the distinction between Fault and Failure. The terms are stated from the perspective of the component: An incoming flaw represent a Fault for the component that may or may not result in a component failure. An internal flaw is Fault that may or may not result in a component failure. A flaw that is propagated out of the component is a Failure.

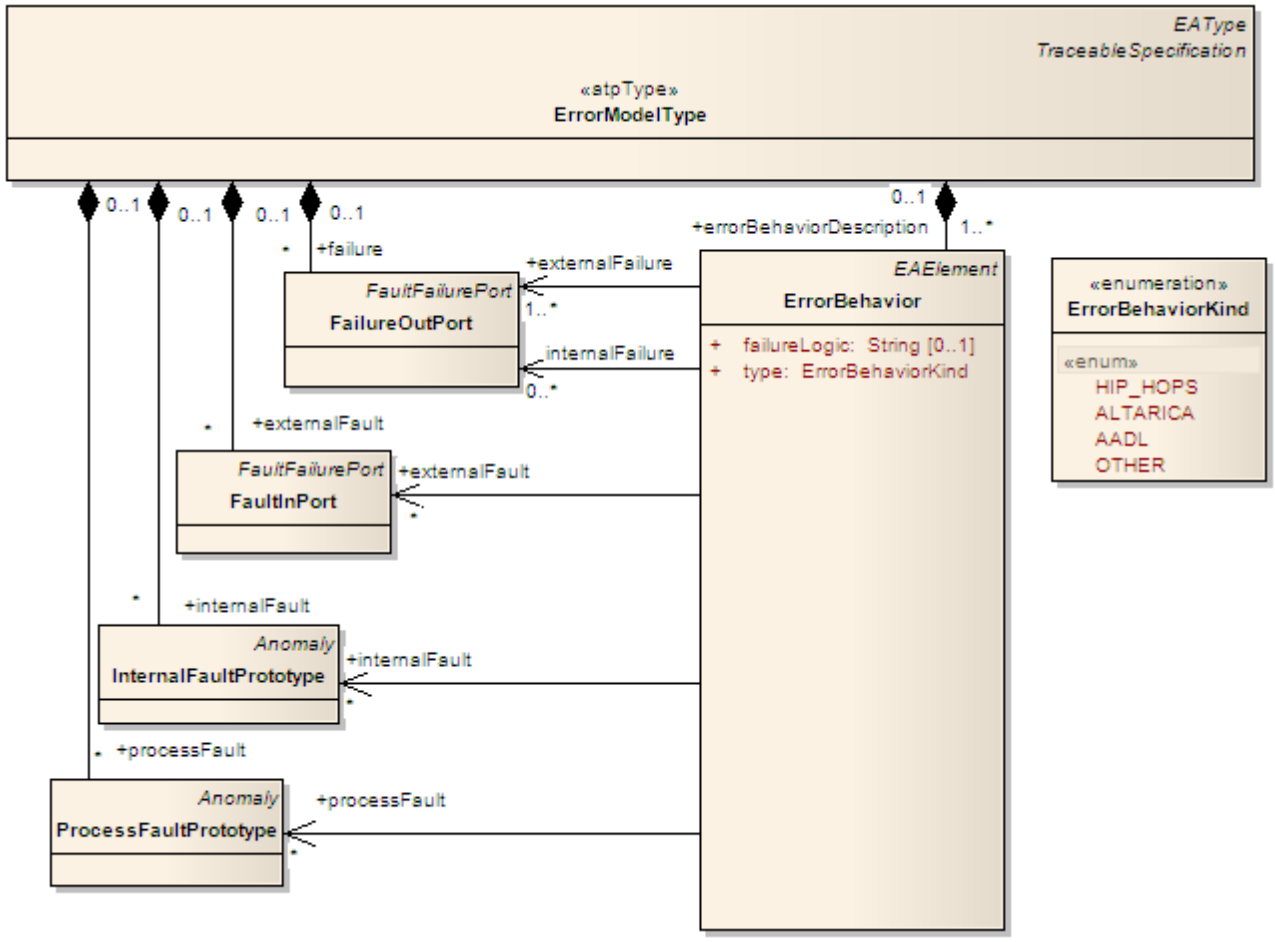

Figure 29. Diagram for ErrorBehavior.

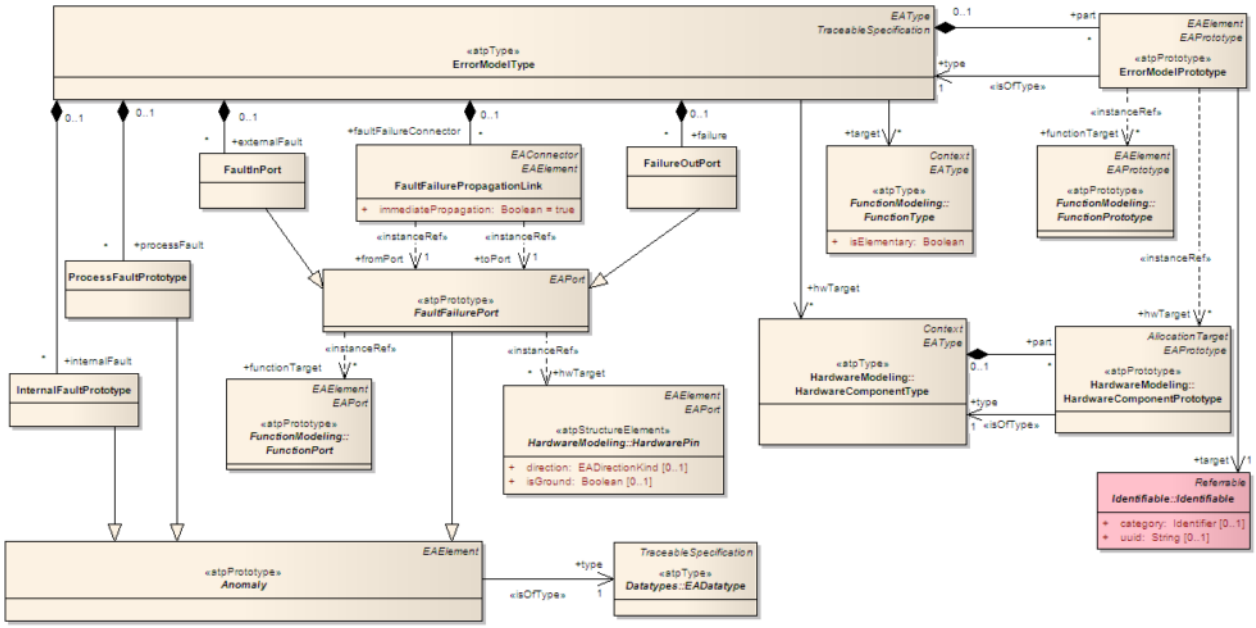

Figure 30. The EAST-ADL metaclasses for defining the error model structure.

18.2 Element Descriptions

18.2.1 Anomaly (from ErrorModel) {abstract} «atpPrototype»

Generalizations

- EAElement (from Elements)

Description

The Anomaly metaclass represents a Fault that may occur internally in an ErrorModel or be propagated to it, or a Failure that is propagated out of an Error Model. The anomaly may represent different Faults or Failures depending on the range of its EADatatype. Typically, the EADatatype is an Enumeration, for example:

BrakeAnomaly:

- BrakePressureTooLow

Semantics="brake pressure is below 20% of requested value"

- Omission

Semantics="brake pressure is below 10% of maximal brake pressure"

- Comission

Semantics="brake pressure exceeds requested value with more than 10% of maximal brake pressure"

Semantics may also be a more formal expression defining in the type of the nominal datatype what value range is considered a fault. This depends on the user and tooling available.

Attributes

No additional attributes

Associations

- type : EADatatype [1]
  «isOfType»
  The declaration of port type.

Constraints

No additional constraints

Semantics

An anomaly refers to a condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, etc., or from someone's perceptions or experiences (ISO26262). The set of available faults or failures represented by the Anomaly is defined by its EADatatype, typically an enumeration type like {omission, commission}. It is an abstract class further specialized with metaclasses for different types of fault/failure.

18.2.2 ErrorBehavior (from ErrorModel)

Generalizations

- EAElement (from Elements)

Description

ErrorBehavior represents the descriptions of failure logics or semantics that the target element identified by the ErrorModelType exhibits. Typically the target is a system, a function, a software component, or a hardware device.

Each ErrorBehavior description relates the occurrences of internal faults and incoming external faults to failures. The faults and failures that the errorBehavior propagates to and from the target element are declared through the ports of the error model.

Attributes

- failureLogic : String [0..1]
  The specification of error behavior based on an external formalism or the path to the file containing the external specification.
- type : ErrorBehaviorKind [1]
  The type of formalism applied for the error behavior description.

Associations

- internalFault : InternalFaultPrototype [*]
  internalFaults that influence the errorBehavior
- processFault : ProcessFaultPrototype [*]
  processFaults that may affect the errorBehavior
- externalFailure : FailureOutPort [1..*]
  Failures that may result from the ErrorBehavior
- externalFault : FaultInPort [*]
  external(incoming) faults that influence the errorBehavior.

Constraints

No additional constraints

Semantics

ErrorBehavior defines the error propagation logic of its containing ErrorModelType.

The ErrorBehavior description represents the error propagations from internal faults or incoming faults to external failures. Faults are identified by the internalFault and externalFault associations respectively. The propagated failures are identified by the externalFailure association.

The ErrorBehavior is defined in the failureLogic string, either directly or as a URL referencing an external specification.

The failureLogic can be based on different formalisms, depending on the analysis techniques and tools available. This is indicated by its type:ErrorBehaviorKind attribute. The failureLogic attribute contains the actual failure propagation logic.

18.2.3 ErrorBehaviorKind (from ErrorModel) «enumeration»

Generalizations

None

Description

The ErrorBehaviorKind metaclass represents an enumeration of literals describing various types of formalisms used for specifying error behavior.

Enumeration Literals

- AADL
  A specification of error behavior according to the external formalism AADL.
- ALTARICA
  A specification of error behavior according to the external formalism ALTARICA.
- HIP_HOPS
  A specification of error behavior according to the external formalism HiP-HOPS.
- OTHER
  A specification of error behavior according to other user defined formalism.

Associations

No additional associations

Constraints

No additional constraints

Semantics

ErrorBehaviorKind represents different formalisms for ErrorBehavior. The semantics are defined at each enumeration literal.

18.2.4 ErrorModelPrototype (from ErrorModel) «atpPrototype»

Generalizations

- EAElement (from Elements)
- EAPrototype (from Elements)

Description

The ErrorModelPrototype is used to define hierarchical error models allowing additional detail or structure to be described in the error model of a particular target. A hierarchal structure can also be defined when several ErrorModels are integrated into a larger ErrorModel representing a system integrated from several targets.

Typically the target is a system/subsystem, a function, a software component, or a hardware device.

Attributes

No additional attributes

Associations

- type : ErrorModelType [1]
  «isOfType»
  The ErrorModelType that types the ErrorModelPrototype.
- target : Identifiable [1]
  - The target element (i.e., a system, a function, a component, or hardware device) owning the anomalies.

  ARElement can also be the target or ErrorModelType.

Dependencies

- functionTarget : FunctionPrototype [*]
  «instanceRef»
- hwTarget : HardwareComponentPrototype [*]
  «instanceRef»

Constraints

No additional constraints

Semantics

An ErrorModelPrototype represents an occurrence of the ErrorModelType that types it.

18.2.5 ErrorModelType (from ErrorModel) «atpType»

Generalizations

- EAType (from Elements)
- TraceableSpecification (from Elements)

Description

ErrorModelType and ErrorModelPrototype support the hierarchical composition of error models based on the type-prototype pattern also adopted for the nominal architecture composition. The purpose of the error models is to represent information relating to the anomalies of a nominal model element.

An ErrorModelType represents the internal faults and fault propagations of the nominal element that it targets.

Typically the target is a system/subsystem, a function, a software component, or a hardware device.

ErrorModelType inherits the abstract metaclass TraceableSpecification, allowing the ErrorModelType to be referenced from its design context in a similar way as requirements, test cases and other specifications.

Attributes

No additional attributes

Associations

- target : FunctionType [*]
  The nominal FunctionType whose ErrorModel is defined by the ErrorModelType
- hwTarget : HardwareComponentType [*]
- internalFault : InternalFaultPrototype [*] {composite}
  An internal fault that the ErrorModelType may propagate or mask
- faultFailureConnector : FaultFailurePropagationLink [*] {composite}
  The contained links for internal propagations of faults/failures between the subordinate error models.
- processFault : ProcessFaultPrototype [*] {composite}
  A processFault that affects the ErrorModelType. Process faults cannot be masked, and propagate to all defined failures.
- part : ErrorModelPrototype [*] {composite}
  The contained error models forming a hierarchy.
- failure : FailureOutPort [*] {composite}
  A failureOutPort represent a propagated Failure
- externalFault : FaultInPort [*] {composite}
  An external fault that the ErrorModelType may propagate or mask
- errorBehaviorDescription : ErrorBehavior [1..*] {composite}
  The description of failure logic of the target element.

Constraints

[1] An ErrorModelType without part shall have one errorBehaviorDescription.

Semantics

The ErrorModelType represents a specification of the faults and fault propagations of its target element.

Both types and prototypes may be targets, and the following cases are relevant:

- One nominal type:

The ErrorModelType represents the identified nominal type wherever this nominal type is instantiated.

- Several nominal types:

The ErrorModelType represents the identified nominal types individually, i.e. the same error model applies to all nominal types and is reused.

- One nominal prototype:

The ErrorModelType represents the identified nominal prototype whenever its context, i.e. its top-level composition is instantiated.

- Several nominal prototypes with instanceref:

The ErrorModelType represents the identified set of nominal prototypes (together) whenever their context, i.e. their top-level composition, is instantiated.

The fault propagation of an errorModelType is defined by its contained parts, the ErrorModelPrototypes and their connections. In case it contains both parts and an errorBehaviorDescription, the errorBehaviorDescription shall be consistent with the parts.

FaultFailurePropagationLinks define valid propagation paths in the ErrorModelType. In case the contained FaultInPorts and FailureOutPorts reference nominal ports, the connectivity of the nominal model may serve as a pattern for connecting ports in the ErrorModelType.

The ErrorModelType contains internalFaults and externalFaults, representing faults that are either propagated to externalFailures or masked, according to the definition of its fault propagation.

A processFault represents a flaw introduced during design, and may lead to any of the failures represented by the ErrorModelType. A processFault therefore has a direct propagation to all failures and cannot be masked.

18.2.6 FailureOutPort (from ErrorModel)

Generalizations

- FaultFailurePort (from ErrorModel)

Description

The FailureOutPort represents a propagation point for failures that propagate out from the containing ErrorModelType.The EADatatype of the FailureOutPort defines the range of valid failures.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] The direction of the nominal port must be 'out'.

Semantics

The value range of a FailureOutPort represents failures that can propagate to FaultInPorts in other ErrorModels. The value range is defined by the FailureOutPort's EADatatype.

If nominal Ports HWTargets or FunctionTargets are referenced, the failures of the FailureOutPort correspond to data on these nominal ports.

18.2.7 FaultFailurePort (from ErrorModel) {abstract} «atpPrototype»

Generalizations

- Anomaly (from ErrorModel)
- EAPort (from Elements)

Description

Abstract port for Faults and Failures.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- functionTarget : FunctionPort [*]
  «instanceRef»
- hwTarget : HardwarePin [*]
  «instanceRef»

Constraints

No additional constraints

Semantics

FaultFailurePort is abstract. Semantics is defined on its specializations.

18.2.8 FaultFailurePropagationLink (from ErrorModel)

Generalizations

- EAConnector (from Elements)
- EAElement (from Elements)

Description

The FaultFailurePropagationLink metaclass represents the links for the propagations of faults/failures across system elements. In particular, it defines that one error model provides the faults/failures that another error model receives.

A fault/failure link can only be applied to compatible ports, either for fault/failure delegation within an error model or for fault/failure transmission across two error models. A FaultFailurePropagationLink can only connect fault/failure ports that have compatible types.

Attributes

- immediatePropagation : Boolean = true [1]

Associations

No additional associations

Dependencies

- fromPort : FaultFailurePort [1]
  «instanceRef»
- toPort : FaultFailurePort [1]
  «instanceRef»

Constraints

[1] Only compatible fromPort-toPort pairs may be connected.

[2] Two fault/failure ports are compatible if the EADatatype of the fromPort represents a subset of the Fault/Failure set represented by the toPort's EADatatype.

Semantics

The FaultFailurePropagationLink defines a Failure propagation path, from the fromPort on one error model to the toPort of another error model.

18.2.9 FaultInPort (from ErrorModel)

Generalizations

- FaultFailurePort (from ErrorModel)

Description

The FaultInPort represents a propagation point for faults that propagate to the containing ErrorModelType. The EADatatype of the FaultInPort defines the range of valid failures.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] The direction of the nominal port must be 'in'.

Semantics

The value range of a FaultInPort represents faults propagated from a FailureOutPort in another ErrorModel. The value range is defined by the FaultInPort's EADatatype.

If nominal Ports HWTarget or FunctionTarget are referenced, the faults on the FaultInPort correspond to data on these nominal ports.

18.2.10 InternalFaultPrototype (from ErrorModel)

Generalizations

- Anomaly (from ErrorModel)

Description

The InternalFault metaclass represents the particular internal conditions of the target component/system that are of particular concern for its fault/failure definition.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The system anomaly represented by an InternalFault, which when activated, can cause errors and failures of the target element.

18.2.11 ProcessFaultPrototype (from ErrorModel)

Generalizations

- Anomaly (from ErrorModel)

Description

The ProcessFaultPrototype metaclass represents the anomalies that the target component/system can have due to design or implementation flaws (e.g., incorrect requirements, buffer size configuration, scheduling, etc.).

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

The ProcessFaultPrototype represent general internal anomalies of a component that are introduced during design and implementation.

19 SafetyConstraints

19.1 Overview

The SafetyConstraints package contains constructs for defining safety constraints.

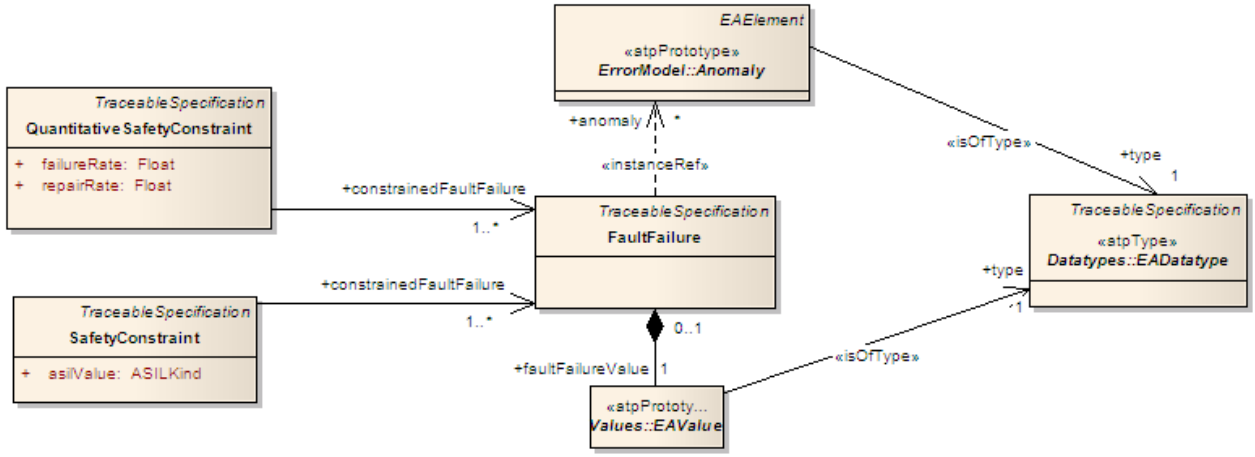

Figure 31. Diagram for SafetyConstraints.

19.2 Element Descriptions

19.2.1 ASILKind (from SafetyConstraints) «enumeration»

Generalizations

None

Description

The ASILKind is an enumeration metaclass with enumeration literals indicating the level of safety integrity in accordance with ISO26262.

Enumeration Literals

- ASIL_A
  ASIL A, Lowest Safety Integrity Level.
- ASIL_B
  ASIL B, second lowest Safety Integrity Level.
- ASIL_C
  ASIL C, second highest Safety Integrity Level.
- ASIL_D
  ASIL D, Highest Safety Integrity Level.
- QM
  Quality Management only, no requirement according to ISO 26262.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics is defined at each enumeration literal and fully defined in the ISO26262 standard.

19.2.2 FaultFailure (from SafetyConstraints)

Generalizations

- TraceableSpecification (from Elements)

Description

The FaultFailure represents a certain fault or failure on its referenced Anomal(ies). The faultFailureValue specifies the value of the Anomaly that corresponds to the condition represented by the FaultFailure. Alternatively, a boolean expression over the referenced anomalies defines the condition represented by the FaultFailure.

Attributes

No additional attributes

Associations

- faultFailureValue : EAValue [1] {composite}
  The faultFailureValue defines the value that anomal(ies) should have or the expression that the anomal(ies) should fulfill.

Dependencies

- anomaly : Anomaly [*]
  «instanceRef»

Constraints

[1] faultFailureValue shall have the same datatype as the referenced Anomal(ies) or be of type EABoolean.

Semantics

A FaultFailure represents a fault or failure on the referenced Anomal(ies). The Faultfailure condition is satisfied when a) faultFailureValue is an EAValue and at least one of the referenced anomal(ies) is equal to this value or b) when faultFailureValue is a boolean EAExpression and the referenced anomal(ies) satisfies the expression, i.e. it evaluates to true.

19.2.3 QuantitativeSafetyConstraint (from SafetyConstraints)

Generalizations

- TraceableSpecification (from Elements)

Description

The QuantitativeSafetyConstraint metaclass represents the quantitative integrity constraints on a fault or failure. Thus, the system has the same or better performance with respect to the constrained fault or failure, and depending on the role this is either a requirement or a property.

Attributes

- failureRate : Float [1]
  failureRate denotes the number of failures per unit time, i.e. the density of probability of failure divided by probability of survival for a hardware element (ISO26262 definition). For exponential failure distributions it is often denoted by lambda.
- repairRate : Float [1]
  repairRate denotes the number of repairs per unit time. For exponential repair distributions it is often denoted by mu.

Associations

- constrainedFaultFailure : FaultFailure [1..*]
  A QuantitativeSafetyConstraint defines quantitative bounds on the constrainedFaultFailure in terms of the failure and repair rates, failureRate and repairRate. The rates are exponentially distributed (user defined attributes may be used to specify alternative distributions and additional quantitative parameters).

Constraints

No additional constraints

Semantics

A QuantitativeSafetyConstraint provides information about the probabilistic estimates of target faults/failures, further specified by the failureRate and repairRate attribute.

19.2.4 SafetyConstraint (from SafetyConstraints)

Generalizations

- TraceableSpecification (from Elements)

Description

The SafetyConstraint metaclass represents the qualitative integrity constraints on a fault or failure. Thus, the system has the same or better performance with respect to the constrained fault or failure, and depending on the role this is either a requirement or a property.

Attributes

- asilValue : ASILKind [1]
  The ASIL level of the target fault or failure.

Associations

- constrainedFaultFailure : FaultFailure [1..*]
  The constrained fault or failure.

Constraints

No additional constraints

Semantics

A SafetyConstraint defines qualitative bounds on the constrainedFaultFailure in terms of safety integrity level, asilValue.

Depending on role, the SafetyConstraint may define a required or an actual safety integrity level.

20 SafetyRequirement

20.1 Overview

The SafetyRequirement package contains constructs for organizing ISO 26262 safety requirements.

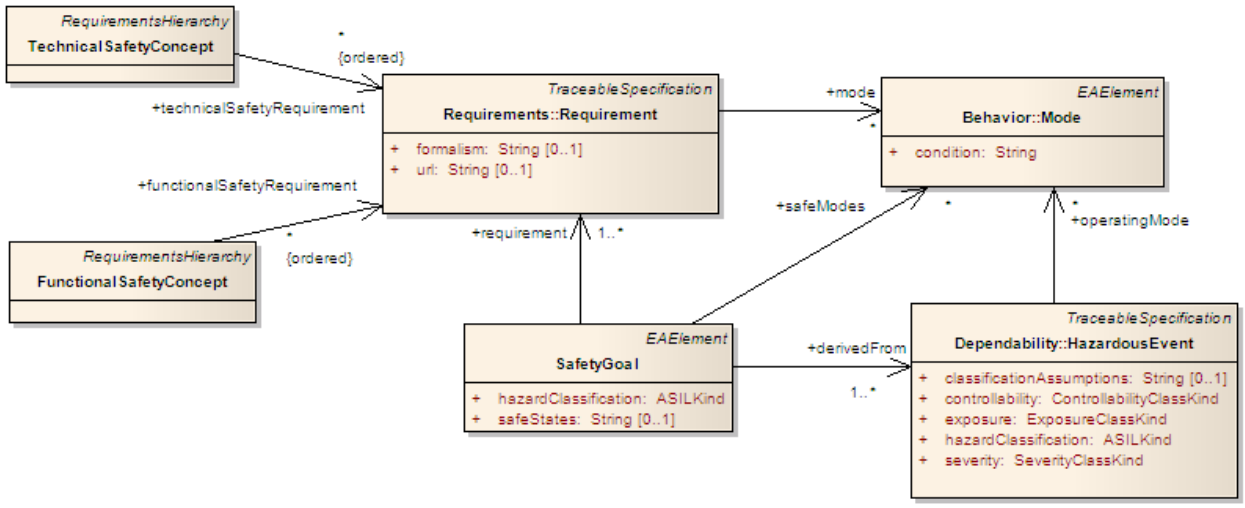

Figure 32. Diagram for Safety Concepts.

20.2 Element Descriptions

20.2.1 FunctionalSafetyConcept (from SafetyRequirement)

Generalizations

- RequirementsHierarchy (from Requirements)

Description

FunctionalSafetyConcept represents the set of functional safety requirements that together fulfils a SafetyGoal in accordance with ISO 26262.

To comply with the SafetyGoals, the FunctionalSafetyConcept specifies the basic safety mechanisms and safety measures in the form of functional safety requirements.

Attributes

No additional attributes

Associations

- functionalSafetyRequirement : Requirement [*] {ordered}
  Represents a functional safety requirement that describes the measures for complying with the safety goals and the corresponding ASIL.

Constraints

[1] Contained functionalSafetyRequirements shall not be of type SafetyGoal.

Semantics

The collection of requirements in the FunctionalSafetyConcept defines the requirements necessary to make the Item safe. The requirements are abstract and do not specify technical details.

20.2.2 SafetyGoal (from SafetyRequirement)

Generalizations

- EAElement (from Elements)

Description

SafetyGoal represents the top-level safety requirement defined in ISO26262. Its purpose is to define how to avoid its associated HazardousEvents, or reduce the risk associated with the hazardous event to an acceptable level.

The SafetyGoal is defined through one or several associated requirement elements.

An ASIL shall be assigned to each SafetyGoal, to represent the integrity level at which the SafetyGoal must be met.

Similar SafetyGoals can be combined into one SafetyGoal. If different ASILs are assigned to similar SafetyGoals, the highest ASIL shall be assigned to the combined SafetyGoal.

For every SafetyGoal, a safe state should be defined, either textually or by referencing a specific mode. The safe state is a system state to be maintained or to be reached when a potential source of its hazardous event is detected.

Attributes

- hazardClassification : ASILKind [1]
- safeStates : String [0..1]
  For every SafetyGoal, a safe state should be defined, in order to declare a system state to be maintained or to be reached when the failure is detected and so to allow a failure mitigation action without any violation of the associated SafetyGoal.

Associations

- safeModes : Mode [*]
  The safe modes identified for the SafetyGoal
- requirement : Requirement [1..*]
- derivedFrom : HazardousEvent [1..*]
  The HazardousEvent which the SafetyGoal shall address

Constraints

No additional constraints

Semantics

SafetyGoal represents a safety Goal according to ISO26262. Requirements define the SafetyGoal, and HazardousEvents identify the responsibility of each SafetyGoal. hazardClassification defines the integrity classification of the SafetyGoal, and safeStates may be defined by a string or formalized through associated Modes.

20.2.3 TechnicalSafetyConcept (from SafetyRequirement)

Generalizations

- RequirementsHierarchy (from Requirements)

Description

TechnicalSafetyConcept represents the set of technical safety requirements that together fulfils a FunctionalSafetyConcept and SafetyGoal in accordance with ISO 26262.

These are derived from FunctionalSafetyConcepts i.e. TechnicalSafetyRequirements are derived from FunctionalSafetyRequirements.

Attributes

No additional attributes

Associations

- technicalSafetyRequirement : Requirement [*] {ordered}

Constraints

No additional constraints

Semantics

The TechnicalSafetyConcept consists of the technical safety requirements and details the functional safety concept considering the functional concept and the preliminary architectural design. It corresponds to the Technical Safety Concept of ISO26262.

21 SafetyCase

21.1 Overview

Safety is a property of a system that is difficult to verify quantitatively since no clear measurement method exists that can be applied during the development. Not even exhaustive testing is feasible, as faults in electronics can have an intensity of 10^-9 faults/hour and still pose an unacceptable risk. Hence, it is only when sufficient field data have been collected from a system used in a particular context that it can be said to be safe enough. Nonetheless, safety must be addressed and assessed during development, restricted to qualitative reasoning about the safety of a product. A structured engineering method is thus needed to approach this problem. One such method is the so called safety case, which came originally from the nuclear industry.

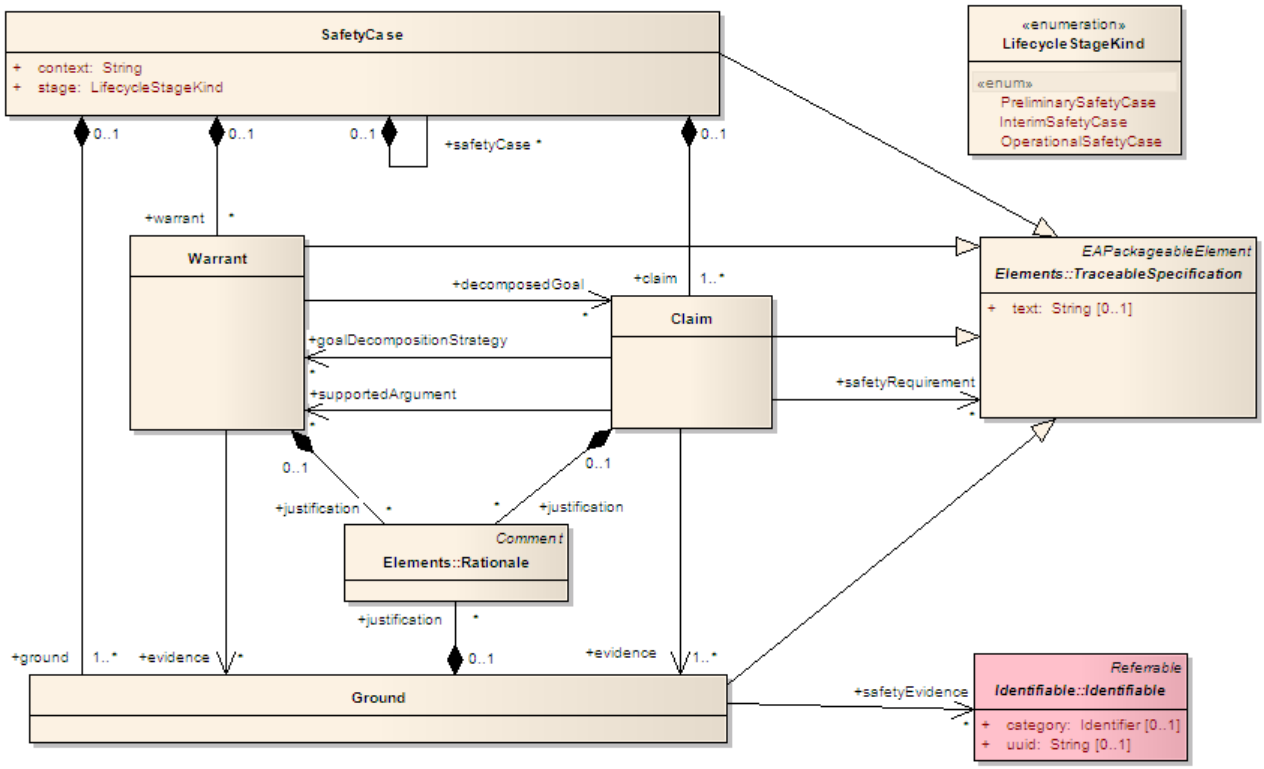

Figure 33.

21.2 Element Descriptions

21.2.1 Claim (from SafetyCase)

Generalizations

- TraceableSpecification (from Elements)

Description

Claim represents a statement, the truth of which needs to be confirmed.

Claim has associations to the strategy for goal decomposition and to supported arguments. It also holds associations to the evidences for the SafetyCase.

Attributes

No additional attributes

Associations

- goalDecompositionStrategy : Warrant [*]
  Strategies can be used to add further detail to a goal decomposition.
- supportedArgument : Warrant [*]
  Supported argument for the Claim.
- evidence : Ground [1..*]
  An evidence provides the backing for stating that a requirement (Claim) has been meet.
- justification : Rationale [*] {composite}
  Justification can be used wherever it is felt to be valuable to provide the rationale behind the Claim.
- safetyRequirement : TraceableSpecification [*]
  Safety requirements and objectives in the SystemModel.

Constraints

No additional constraints

Semantics

Goal-based development provides the claim what should be achieved.

Goal is what the argument must show to be true.

21.2.2 Ground (from SafetyCase)

Generalizations

- TraceableSpecification (from Elements)

Description

Claim is based on Grounds (evidences) - specific facts about a precise situation that clarify and make good the Claim.

Ground represents statements that explain how the SafetyCase Ground clarifies and make good the Claim.

Ground has associations to the entities that are the evidences in the SafetyCase.

Attributes

No additional attributes

Associations

- justification : Rationale [*] {composite}
  Justification can be used wherever it is considered valuable to provide the rationale behind the Ground.
- safetyEvidence : Identifiable [*]
  Safety evidence in the SystemModel. May also refer to elements in the AUTOSAR model.

Constraints

No additional constraints

Semantics

Ground (evidence) is information that supports the Claim that the safety requirements and objectives are met i.e. used as the basis of the safety argument.

Solution is evidence that the sub-goals have been met. This can be achieved by decomposing all goal claims to a level where direct reference to evidences was considered possible.

The evidences address different aspects of the goal. It always has to be ensured that each of them is defensible enough to confirm the underlying statement.

21.2.3 LifecycleStageKind (from SafetyCase) «enumeration»

Generalizations

None

Description

The SafetyCase should be initiated at the earliest possible stage in the safety program so that hazards are identified and dealt with while the opportunities for their exclusion exist.

The LifecycleStageKind is an enumeration metaclass with enumeration literals indicating safety case life cycle stage.

Enumeration Literals

- InterimSafetyCase
  The interim safety case is situated after the first system design and tests
- OperationalSafetyCase
  The operational safety case is prior to in-service use
- PreliminarySafetyCase
  The preliminary safety case is started when development of the system is started.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The safety case is one incremental safety case, rather than several complete new ones. The safety case lifecycle stage has the following meanings:

- The preliminary safety case is started when development of the system is started. After this stage discussions with the customer can commence about possible safety issues (hazards).

- The interim safety case is situated after the first system design and tests.

- The operational safety case is prior to in-service use.

21.2.4 SafetyCase (from SafetyCase)

Generalizations

- TraceableSpecification (from Elements)

Description

SafetyCase represents a safety case that communicates a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a given context.

Safety Cases are used in safety related systems, where failures can lead to catastrophic or at least dangerous consequences.

Attributes

- context : String [1]
  Description of how the SafetyCase Warrant (argument) relates to, and depends upon, information from other viewpoints.
- stage : LifecycleStageKind [1]
  Safety case life cycle stage (preliminary, interim or operational)

Associations

- safetyCase : SafetyCase [*] {composite}
  Sub SafetyCase
- warrant : Warrant [*] {composite}
  Argumentation of the facts to the Claim in general ways.
- ground : Ground [1..*] {composite}
  Explains how the SafetyCase Ground clarifies and make good the Claim.
- claim : Claim [1..*] {composite}
  A statement the truth of which needs to be confirmed.

Constraints

No additional constraints

Semantics

The SafetyCase element is a container element for warrant, type and claim that together represent evidence of safety for the system or item in its context.

21.2.5 Warrant (from SafetyCase)

Generalizations

- TraceableSpecification (from Elements)

Description

Warrant represents argumentation of the facts to the Claim in general ways.

The Warrant entity has associations with the decomposed goals and with the evidences for the SafetyCase.

Attributes

No additional attributes

Associations

- evidence : Ground [*]

Explains how the SafetyCase Ground clarifies and justifies the Claim.

- decomposedGoal : Claim [*]
  A statement which needs to be confirmed
- justification : Rationale [*] {composite}
  Justification can be used wherever it is felt to be valuable to provide the rationale behind the Warrant.

Constraints

No additional constraints

Semantics

The overall objective of an argument is to lead the evidence to the claim.

Arguments are actions of inferring a conclusion from premised propositions. An argument is considered valid if the conclusion can be logically derived from its premises. An argument is considered sound if it is valid and all premises are true.

A goal decomposition strategy breaks down a goal into a number of sub-goals. It is recommended that the strategies are of specific form.

Part VIII Generic Constraints

This part contains support for GenericConstraints, i.e. those that do not belong to the predefined timing and safety constraints.

22 GenericConstraints

22.1 Overview

The main concept in this package is GenericConstraint which denotes a property, requirement, or a validation result for the identified element of the model. The kind of GenericConstraint is described as one of the predefined GenericConstraintKind literals.

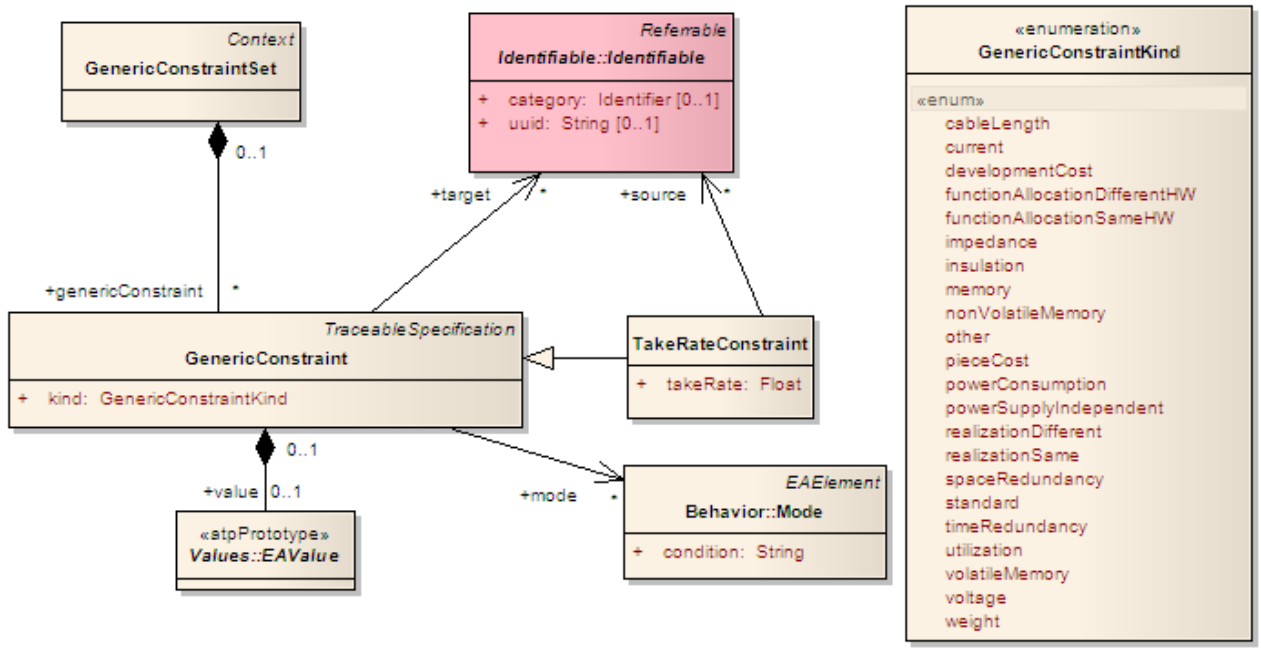

Figure 34. Diagram of GenericConstraint.

22.2 Element Descriptions

22.2.1 GenericConstraint (from GenericConstraints)

Generalizations

- TraceableSpecification (from Elements)

Description

The GenericConstraint denotes a property, requirement, or a validation result for the identified element of the model. The kind of GenericConstraint is described as one of the GenericConstraintKind literals.

Example: If the attribute genericConstraintType is cableLength, the value could be "5 meters" (value of a numerical datatype with unit "meters").

Attributes

- kind : GenericConstraintKind [1]
  The type of the GenericConstraint, see GenericConstraintKind.

Associations

- mode : Mode [*]
  The mode where this GenericConstraint is valid.
- target : Identifiable [*]
  The subject of the GenericConstraint.
- value : EAValue [0..1] {composite}
  The concrete value of the GenericConstraint according to the semantics of the genericConstraintType.

Constraints

No additional constraints

Semantics

The GenericConstraint does not describe what is classically referred to as a "design" constraint but has the role of a property, requirement, or a validation result. It is a requirement if this GenericConstraint refines a Requirement (by the Refine relationship). The GenericConstraint is a validation result if it realizes a VVActualOutcome, it is an intended validation result if it realizes a VVIntendedOutcome, and in other cases it denotes a property.

22.2.2 GenericConstraintKind (from GenericConstraints) «enumeration»

Generalizations

None

Description

Enumeration for different type of constraints.

Enumeration Literals

- cableLength
  The length of the cable. Recommended quantity is length.
- current
  The electrical current of the target. Recommended quantity is Electric current.
- developmentCost
  The overall development cost. Recommended quantity is time.
- functionAllocationDifferentHW
  The referenced elements shall be allocated to different HW elements during design and implementation.
- functionAllocationSameHW
  The referenced elements shall be allocated to the same HW elements during design and implementation.
- impedance
  The internal impedance in Ohms to ground of the component as seen through a targetPin or between a pair of targetPins. Recommended quantity is M L^2 t^-3 I^-2.
- insulation
  The insulation resistance of the target.
- memory
- nonVolatileMemory
  The size in Bytes of the Node's Non-Volatile memory (ROM, NRAM, EPROM, etc.).

- other
- pieceCost
  The costs per piece.
- powerConsumption
  The power consumption of the unit. Recommended quantity is power M L^2 t^-3.
- powerSupplyIndependent
  The targets (the DesignFunctions) shall be allocated to Nodes with independent power supplies.

  This constraint needs to be implemented by appropriate FunctionAllocations in the DesignLevel.
- realizationDifferent
  The referenced elements shall be realized by different functional, logical or software element during design and implementation.
- realizationSame
  The referenced elements shall be realized by the same functional, logical or software element during design and implementation.
- spaceRedundancy
  The targets are replicated for redundancy, genericConstraintValue times.
- standard
  The standard (e.g., ISO26262) that is the basis for development of the target.
- timeRedundancy
  The targets are executed with time redundancy, genericConstraintValue times.
- utilization
- volatileMemory
  The size in Bytes of the Node's Volatile memory (RAM)
- voltage
  The voltage between the targets. Recommended quantity is voltage M L^2 t^-3 I^-1.
- weight
  The physical weight of the unit. Recommended quantity is mass.

Associations

No additional associations

Constraints

No additional constraints

Semantics

The semantics is defined on each literal.

22.2.3 GenericConstraintSet (from GenericConstraints)

Generalizations

- Context (from Elements)

Description

The collection of generic constraints. This collection can be used across the EAST-ADL abstraction levels.

Attributes

No additional attributes

Associations

- genericConstraint : GenericConstraint [*] {composite}

Constraints

No additional constraints

Semantics

GenericConstraintSet is a container element for GenericConstraints and has no specific semantics.

22.2.4 TakeRateConstraint (from GenericConstraints)

Generalizations

- GenericConstraint (from GenericConstraints)

Description

The TakeRateConstraint defines the ratio between the number of configurations that includes the target elements and the number of configurations that include the source. If several source elements are referenced, it would be the configurations in which all these exist.

TakeRateConstraint complements configuration decisions, as the latter defines the rules for actual configuration. TakeRateConstraint defines expected rates of configurations and the set of constraints should be consistent with the configuration decisions. Also, the set of TakeRateConstraints shall be consistent among themselves.

Attributes

- takeRate : Float [1]
  The rate of target compared with source configurations.

Associations

- source : Identifiable [*]
  The elements that are compared with the elements identified by target (see GenericConstraint).

Constraints

[1] The cardinality of target is > 0

Semantics

The TakeRate constraint defines frequency of configurations. Let sourceamount and targetamount be the number of system configurations where all source and target elements, respectively, are included. takeRate= targetamount/sourceamount. If no source is associated, takeRate=targetamount.

23 Part IX Infrastructure

This part contains the EAST-ADL infrastructure which is Datatypes, Elements and User attributes.

24 Datatypes

24.1 Overview

The Datatypes subpackage of EAST-ADL defines EAST-ADL general-purpose datatypes that may be used to type structural constructs in several different modeling diagrams.

The purpose of the metaclasses in the Datatypes subpackage is to specify the concepts for the specific domain.

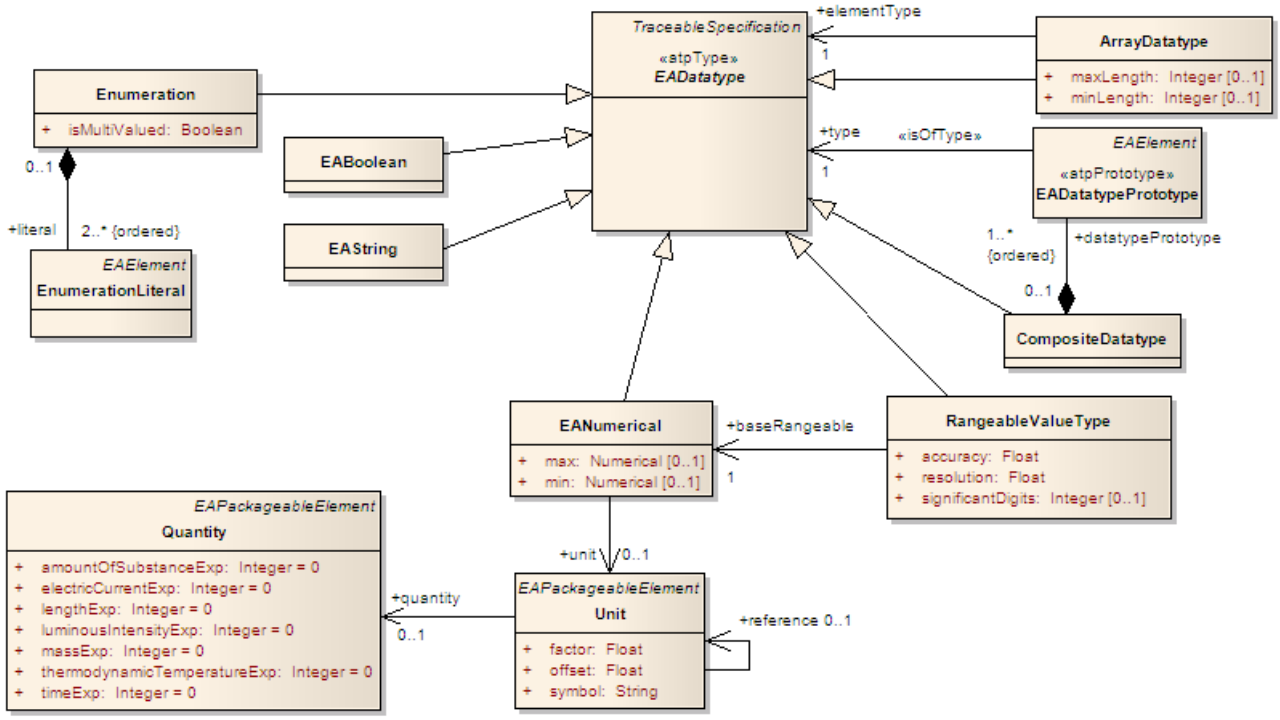

Figure 35. Diagram for Datatypes.

24.2 Element Descriptions

24.2.1 ArrayDatatype (from Datatypes)

Generalizations

- EADatatype (from Datatypes)

Description

Specification of an array of the typing EADatatype. All elements of the ArrayDatatype have the same datatype.

Attributes

- maxLength : Integer [0..1]
  The maximum number of values in this array. Unbounded if not provided.

- minLength : Integer [0..1]
  The minimum number of values in this array.

Associations

- elementType : EADatatype [1]
  The type of all elements in this array.

Constraints

No additional constraints

Semantics

ArrayDatatype is a datatype for an array of datatypes of the same type.

24.2.2 CompositeDatatype (from Datatypes)

Generalizations

- EADatatype (from Datatypes)

Description

A CompositeDatatype represents a non-scalar datatype. Take as an example a CompositeDatatype "MyCountries" that can refer, e.g., to an Enumeration "CountryEnumeration" {USA, Canada, Japan, EU} via two EADatatypePrototypes (record variables): FirstCountry and SecondCountry. Then an attribute typed by this CompositeDatatype "MyCountries" may have a value like: (EU (identified as FirstCountry), Japan (identified as SecondCountry)).

Attributes

No additional attributes

Associations

- datatypePrototype : EADatatypePrototype [1..*] {ordered} {composite}
  The record variable owned by the CompositeDatatype.

Constraints

No additional constraints

Semantics

A CompositeDatatype represents a non-scalar datatype. The contained datatypePrototypes act as record variables to identify the ordered datatype instances of the tuple (the CompositeDatatype).

24.2.3 EABoolean (from Datatypes)

Generalizations

- EADatatype (from Datatypes)

Description

A EABoolean value denotes a logical condition that is either 'true' or 'false'.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

EABoolean is the primitive type that holds two literals: true, false.

24.2.4 EADatatype (from Datatypes) {abstract} «atpType»

Generalizations

- TraceableSpecification (from Elements)

Description

The EADatatype is a metaclass, which signifies a type whose instances are identified only by their value. The EADatatype metaclass represents the description of the value set for some variable, parameter etc. without a description of how these possible values are represented at implementation level. The implementation representation is defined at implementation level by the AUTOSAR concept PrimitiveTypeWithSemantics, and the implemented datatype shall be associated with a Realization relationship. The realizing datatype must match the EADatatype regarding range, resolution, unit, and dimension.

Attributes

No additional attributes

Associations

No additional associations

Constraints

[1] In the case of an AR implementation, an EADatatype is realized generally by PrimitiveTypeWithSemantics, which has to be consistent w.r.t. range, resolution, etc.

Semantics

EADatatype metaclass is a special kind of classifier, similar to a class. It differs from the class in that instances of a data type are identified only by their value.

24.2.5 EADatatypePrototype (from Datatypes) «atpPrototype»

Generalizations

- EAElement (from Elements)

Description

The EADatatypePrototype represents a typed variable. An example is a composite datatype ColorValue with parts R, G, and B of type integer. ColorValue would contain three prototypes only to be able to reference the record parts by name.

Attributes

No additional attributes

Associations

- type : EADatatype [1]
  «isOfType»
  The type of the EADatatypePrototype.

Constraints

No additional constraints

Semantics

The EADatatypePrototype represents a typed variable. It acts as an appearance of a datatype.

24.2.6 EANumerical (from Datatypes)

Generalizations

- EADatatype (from Datatypes)

Description

Datatype for numerical values.

Attributes

- max : Numerical [0..1]
  The maximal value of the range.
- min : Numerical [0..1]
  The minimum value of the range.

Associations

- unit : Unit [0..1]
  - The unit of data.

  Example: For temperature the unit may be "degree Celsius".

Constraints

No additional constraints

Semantics

EANumerical has attributes for modeling of the allowed range.

24.2.7 EAString (from Datatypes)

Generalizations

- EADatatype (from Datatypes)

Description

A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters. An instance of EAString defines a piece of text. The semantics of the string itself depends on its purpose. It can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

EAString is the primitive type that defines a sequence of characters in some suitable character set used to display information.

24.2.8 Enumeration (from Datatypes)

Generalizations

- EADatatype (from Datatypes)

Description

An enumeration is a datatype whose values are enumerated in the model as enumeration literals. Enumeration is a kind of datatype, whose instances may be any of a number of user-defined enumeration literals.

Attributes

- isMultiValued : Boolean [1]
  This boolean attribute is true, if multiple enumeration values can be selected. It is false, if only one enumeration value is allowed to be selected.

Associations

- literal : EnumerationLiteral [2..*] {ordered} {composite}
  The literal (value) of the enumeration.

Constraints

No additional constraints

Semantics

Enumeration is a kind of datatype, whose instances may be any number > 1 of user-defined enumeration literals. Enumerations contain at least two literals, otherwise it would be a constant. The contained literals need to be ordered.

24.2.9 EnumerationLiteral (from Datatypes)

Generalizations

- EAElement (from Elements)

Description

An enumeration literal is a user-defined data value for an enumeration.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

An EnumerationLiteral defines an element of the run-time extension of an enumeration data type. An EnumerationLiteral has a name (inherited from EAElement) that can be used to identify it within its Enumeration datatype. The EnumerationLiteral name is scoped and must therefore be unique within its Enumeration. The run-time values corresponding to EnumerationLiterals can be compared for equality.

24.2.10 Quantity (from Datatypes)

Generalizations

- EAPackageableElement (from Elements)

Description

A Quantity describes a physical dimension by exponents of the available attributes.

Some examples of Quantity are:

name = "Length" and lengthExp = "1"

name = "Angle" and all attribues = 0, i.e. angle is without dimension.

name = "Acceleration" and lengthExp = 1 and timeExp =-2.

Attributes

- amountOfSubstanceExp : Integer = 0 [1]
- electricCurrentExp : Integer = 0 [1]
- lengthExp : Integer = 0 [1]
- luminousIntensityExp : Integer = 0 [1]
- massExp : Integer = 0 [1]
- thermodynamicTemperatureExp : Integer = 0 [1]
- timeExp : Integer = 0 [1]

Associations

No additional associations

Constraints

No additional constraints

Semantics

The Quantity describes a physical dimension for use in numerical datatypes and expressions to facilitate dimension consistency and control.

24.2.11 RangeableValueType (from Datatypes)

Generalizations

- EADatatype (from Datatypes)

Description

The RangeableValueType is a specific datatype applicable for numerical datatypes. It describes the accuracy, resolution, and the significant digits of the baseRangeable datatype.

Attributes

- accuracy : Float [1]
  The accuracy of the data (e.g., the FunctionFlowports input or output).

  Example: An accuracy of 0.5 of the temperature means a communicated value of 19 represents an actual temperature of 19 +/- 0.5 degrees.
- resolution : Float [1]
  The resolution of the data expressed as the size of the minimum difference between data values.

  Example: A resolution of 0.1 means that temperature may be represented in increments of 0.1 degrees.
- significantDigits : Integer [0..1]
  The number of significant digits, e.g., for the speed case: if the speed is a one digit number (e.g., 5 km/h), then this digit is significant, if the speed is a two digits number (e.g., 15 km/h), then the first digit is significant (here: 1), if the speed is a three digits number (e.g., 215 km/h), then the first two digits are significant (here: 21). Significant means here, that the respective digits are reliable.

Associations

- baseRangeable : EANumerical [1]
  The datatype with additional attributes specified by this concept.

Constraints

No additional constraints

Semantics

The RangeableValueType adds the ability to describe the accuracy, resolution, and the significant digits of the baseRangeable datatype.

24.2.12 Unit (from Datatypes)

Generalizations

- EAPackageableElement (from Elements)

Description

A Unit describes a unit used for numerical values of a datatype. It may relate to another unit to enable conversions. It may also reference a quantity to give a dimension of the unit.

As a unit conversion example:

The Unit with name Second has the factor 1000, and the reference Millisecond, i.e.:

second = 1000 * millisecond

Moreover the Unit may be given a symbol and an offset, for example:

The Unit Fahrenheit with factor 1.8 and offset 32 gives with the reference to Celsius the definition of Fahrenheit:

F = C*9/5 + 32

Attributes

- factor : Float [1]
- offset : Float [1]
- symbol : String [1]

Associations

- quantity : Quantity [0..1]
  The (physical) quantity, e.g., "Speed", "Temperature".
- reference : Unit [0..1]

Constraints

No additional constraints

Semantics

Unit descibes the unit of typed numerical values.

25 Values

25.1 Overview

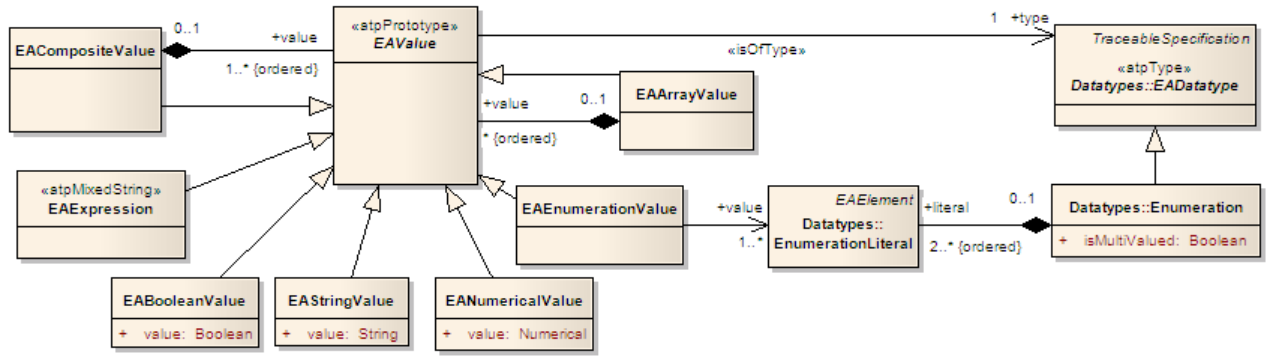

Figure 36.

25.2 Element Descriptions

25.2.1 EAArrayValue (from Values)

Generalizations

- EAValue (from Values)

Description

Used to hold the values in an array.

Attributes

No additional attributes

Associations

- value : EAValue [*] {ordered} {composite}
  The values in this array, each is typed by the same EADatatype as the type of the EAArrayDatatype.

Constraints

[1] Shall be typed by an ArrayDatatype.

Semantics

-

25.2.2 EABooleanValue (from Values)

Generalizations

- EAValue (from Values)

Description

Used to model a boolean value.

Attributes

- value : Boolean [1]

Associations

No additional associations

Constraints

[1] Shall be typed by an EABoolean.

Semantics

The semantics of this value is defined by the element typed by the typing EABoolean.

25.2.3 EACompositeValue (from Values)

Generalizations

- EAValue (from Values)

Description

Used to model values in a record.

Attributes

No additional attributes

Associations

- value : EAValue [1..*] {ordered} {composite}
  The ordered set of values, each typed by the same EADatatype as the corresponding EADatatypePrototype in the CompositeDatatype.

Constraints

[1] Shall be typed by an CompositeDatatype.

[2] The values in this EACompositeValue shall be typed and ordered in the same way as the EADatatypePrototypes in the typing CompositeDatatype.

Semantics

The semantics of this value is defined by the element typed by the typing CompositeDatatype.

25.2.4 EAEnumerationValue (from Values)

Generalizations

- EAValue (from Values)

Description

Used to model a value for an Enumeration or several values in a multivalued EnumerationValueType
.

Attributes

No additional attributes

Associations

- value : EnumerationLiteral [1..*]
  The enumeration value.

Constraints

[1] Shall be typed by an Enumeration or an EnumerationValueType.

Semantics

The semantics of this value is defined by the element typed by the typing Enumeration or the semantics defined in the EnumerationValueType.

25.2.5 EAExpression (from Values) «atpMixedString»

Generalizations

- EAValue (from Values)

Description

The mixed string EAExpression allow for modeling of expressions with references to elements in the model. Specializations within the metamodel define their syntax and the referred metaclasses used in the expressions.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Used for modeling of expressions with references to model elements. Different typing of the expression is possible, if e.g. typed by an EABooleanDatatype the evaluated expression results in a boolean value.

25.2.6 EANumericalValue (from Values)

Generalizations

- EAValue (from Values)

Description

Used to model a numerical value.

Attributes

- value : Numerical [1]

Associations

No additional associations

Constraints

[1] Shall be typed by an EANumerical or a RangeableValueType.

Semantics

The semantics of this value is defined by the element typed by the type EADatatype.

25.2.7 EAStringValue (from Values)

Generalizations

- EAValue (from Values)

Description

Used to model a string value.

Attributes

- value : String [1]

Associations

No additional associations

Constraints

[1] Shall be typed by an EAString.

Semantics

The semantics of this value is defined by the element typed by the typing EAString.

25.2.8 EAValue (from Values) {abstract} «atpPrototype»

Generalizations

None

Description

EAValue is an abstract element with concrete elements used to store typed values in the model. Some of the specializations correspond to UML2 literal specifications EAValue corresponds to UML2 Value Specification which is a typed element.

The EAValue does not have a name and is contained where a value is modeled.

Attributes

No additional attributes

Associations

- type : EADatatype [1]
  «isOfType»
  The type of the value.

Constraints

No additional constraints

Semantics

The semantics of this element is defined by the element typed by the corresponding EADatatype.

26 Elements

26.1 Overview

The Element subpackage of the Infrastructure package of the EAST-ADL specifies the most basic abstract structural constructs in EAST-ADL.

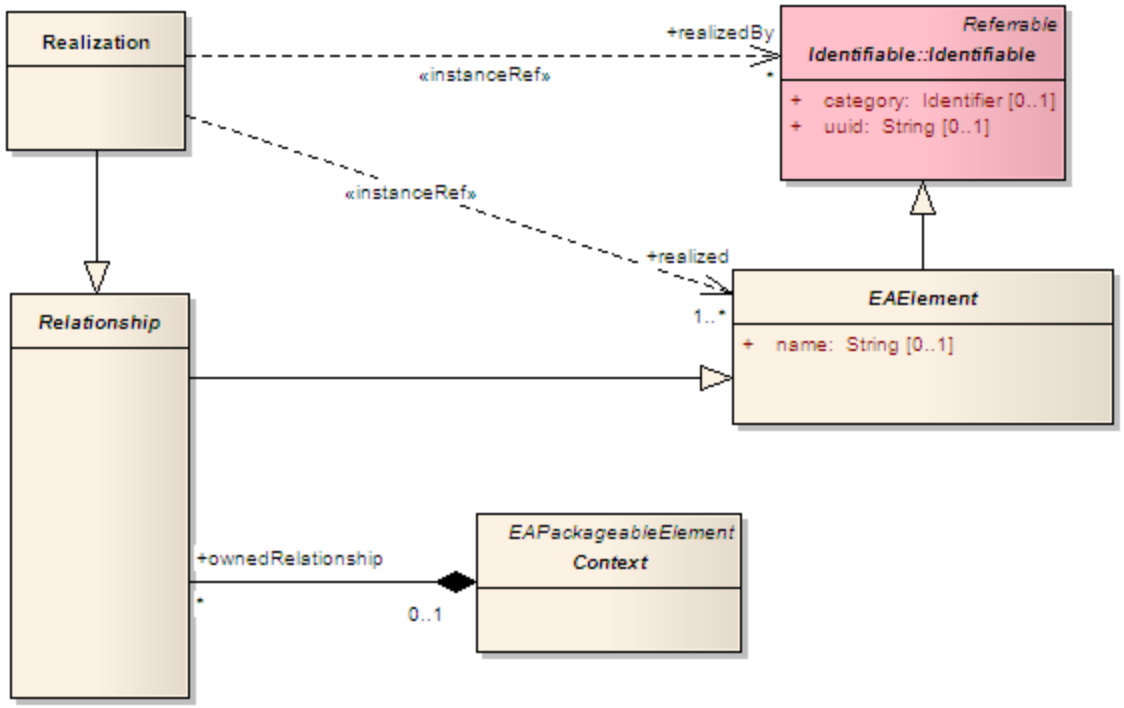

Figure 37. Diagram for RelationshipModeling.

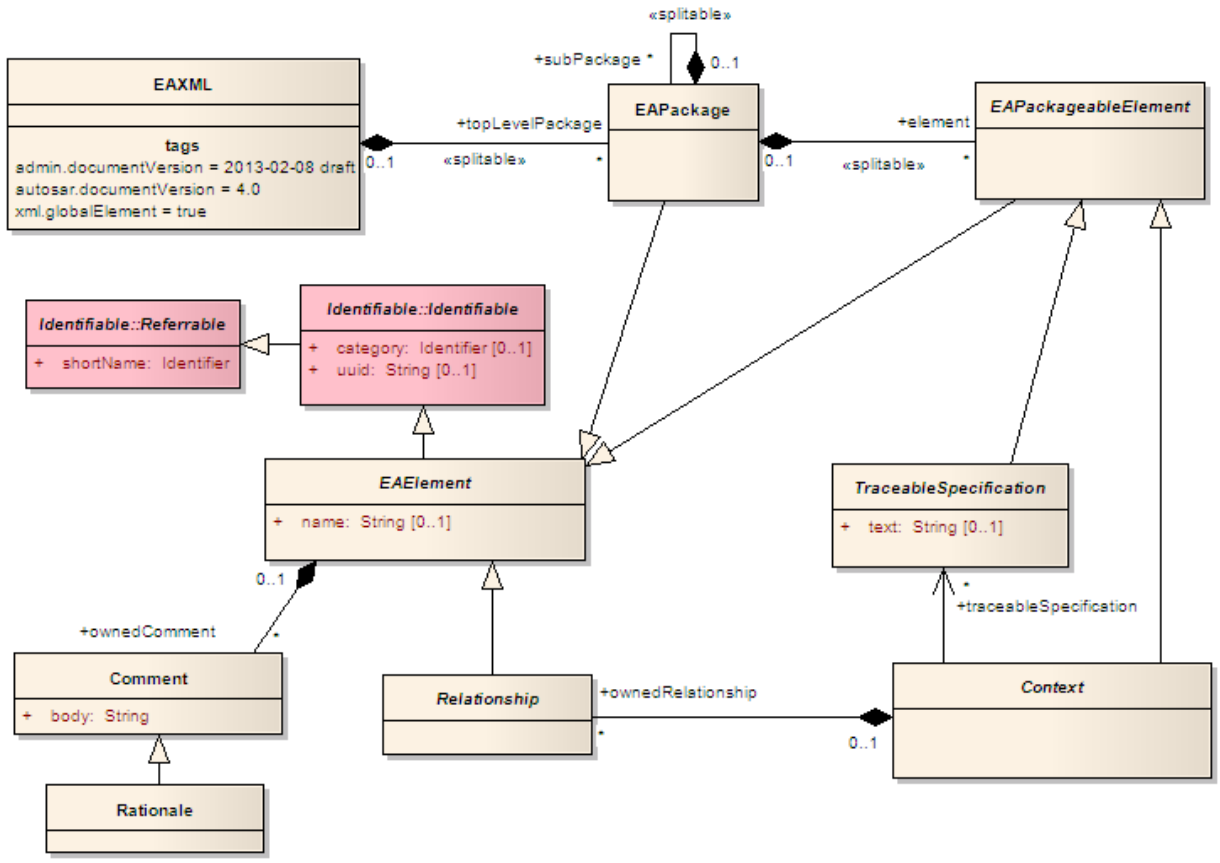

Figure 38. Diagram for Elements.

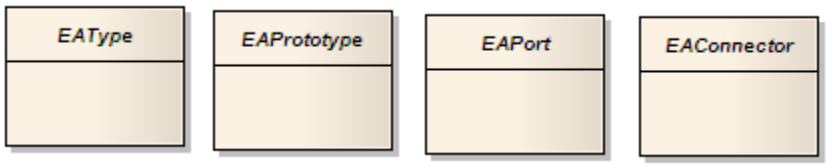

Figure 39.

26.2 Element Descriptions

26.2.1 Comment (from Elements)

Generalizations

None

Description

Comment represents a textual annotation.

Attributes

- body : String [1]
  Specifies a string that is the comment.

Associations

No additional associations

Constraints

No additional constraints

Semantics

Comment represents a textual annotation that applies to the containing or associated element.

26.2.2 Context (from Elements) {abstract}

Generalizations

- EAPackageableElement (from Elements)

Description

Context represents a simple and practical way to allocate TraceableSpecifications to a specific EAST-ADL model context, and to let this specific model context own Relationships.

Attributes

No additional attributes

Associations

- ownedRelationship : Relationship [*] {composite}
  Relationship(s) owned by this context.
- traceableSpecification : TraceableSpecification [*]
  Traceable specification(s) identified by this context.

Constraints

No additional constraints

Semantics

See Relationship and TraceableSpecification.

26.2.3 EAConnector (from Elements) {abstract}

Generalizations

None

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

26.2.4 EAElement (from Elements) {abstract}

Generalizations

- Identifiable (from Identifiable)

Description

The EAElement is an abstract metaclass that represents an arbitrary named entity in the domain model. It specializes AUTOSAR Identifiable which has the shortName attribute used for identification of the element within the namespace in which it is defined.

The abbreviation EA in the name of this metaclass is short for EAST-ADL.

Attributes

- name : String [0..1]
  Optional descriptive name of the EAElement, this name does not have the length restrictions as found for the AUTOSAR Identfiable shortName.

Associations

- ownedComment : Comment [*] {composite}
  Comment owned by this EAElement.

Constraints

No additional constraints

Semantics

Also the EAElement can be used to extend the EAST-ADL approach to other languages and standards by adding a generalize relation from the respective (non EAST-ADL) element to the EAElement.

26.2.5 EAPackage (from Elements)

Generalizations

- EAElement (from Elements)

Description

Used for organization of the packageable elements in the model.

Attributes

No additional attributes

Associations

- element : EAPackageableElement [*] {composite}
  «splitable»
  Contained packageable elements.
- subPackage : EAPackage [*] {composite}
  «splitable»
  Contained packages.

Constraints

No additional constraints

Semantics

EAPackages can be organized hierarchically, where each level may contain a number of EAPackageableElements.

26.2.6 EAPackageableElement (from Elements) {abstract}

Generalizations

- EAElement (from Elements)

Description

Elements that are packageable may be directly contained in a package.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Elements specializing EAPackageableElement can be created directly within an EAPackage.

26.2.7 EAPort (from Elements) {abstract}

Generalizations

None

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

26.2.8 EAPrototype (from Elements) {abstract}

Generalizations

None

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

26.2.9 EAType (from Elements) {abstract}

Generalizations

None

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

26.2.10 EAXML (from Elements)

Generalizations

None

Description

The root element of an exchanged XML file which contains an EAST-ADL model.

Attributes

No additional attributes

Associations

- topLevelPackage : EAPackage [*] {composite}
  «splitable»
  Contained top level packages.

Constraints

No additional constraints

Semantics

EAXML represents the root element of an EAST-ADL XML file.

26.2.11 Rationale (from Elements)

Generalizations

- Comment (from Elements)

Description

Rationale represents a justification to any model element.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

Rationale represents a justification to any model element.

26.2.12 Realization (from Elements)

Generalizations

- Relationship (from Elements)

Description

The Realization is a relationship which relates two or more elements across boundaries of the EAST-ADL abstraction levels.

It identifies an element that serves as a specification within this realization relationship and on the other side it identifies an element that is supposed to realize this specification on a lower abstraction level or an implementation.

Attributes

No additional attributes

Associations

No additional associations

Dependencies

- realized : EAElement [1..*]
  «instanceRef»
- realizedBy : Identifiable [*]
  «instanceRef»

Constraints

[1] The realizedBy elements shall be on a lower abstraction level than the realized relements.

[2] The realizedBy or realized elements shall be structural or behavioral.

Semantics

The Realization is a relationship which identifies one or several abstract elements that are realized by one or several concrete elements. The realizedBy elements together represents a realization of

the group of realized elements and is collectively responsible for meeting the specification of the realized elements, including (derivations of) its requirements.

26.2.13 Relationship (from Elements) {abstract}

Generalizations

- EAElement (from Elements)

Description

The Relationship is an abstract metaclass which represents a relationship between arbitrary elements.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

In many cases, Contexts such as functions and sensors need to have requirements and other specification elements allocated to them. In other cases, the relationship between an element and the related specification element is specific for a certain Context: for example a Requirement on a sensor is only applicable in certain hardware architectures. These relationships are modeled by concrete specializations of Relationship.

See Context and TraceableSpecification.

26.2.14 TraceableSpecification (from Elements) {abstract}

Generalizations

- EAPackageableElement (from Elements)

Description

The TraceableSpecification is an abstract metaclass which is used to allow its specializations to be allocated to a Context.

Attributes

- text : String [0..1]
  An optional description attribute that provides textual representation, or a reference to the textual representation, of the Traceable Specification in a specific formalism.

Associations

No additional associations

Constraints

No additional constraints

Semantics

TraceableSpecification is specialized by requirements, test cases and other specifications, that can be allocated to a Context, for example to a sensor or to an entire HW architecture.

See Context and Relationship.

27 UserAttributes

27.1 Overview

User attributes in EAST-ADL are primarily intended to provide a mechanism for augmenting the elements of an EAST-ADL model with customized meta-information. All instances of metaclass Identifiable can have user attributes attached to them. The scope and structuring of this meta-information can be defined on a per-project basis by defining user attributes for certain types of EAST-ADL elements with UserElementTypes and UserAttributeDefinitions.

Since EAST-ADL Requirements are, in their most general form, simple objects with all information contained in user-customized, project-specific attributes, the concept of user attributes is also suitable for defining those attributes of requirements. In that sense, basic Requirements in EAST-ADL can be seen as "empty" elements which only provide a node to which user attributes can be attached in order to supply the Requirement with all necessary information, including its main textual description. However, in the case when the Requirement is the context in which the available user attributes are defined, the containing RequirementsModel of the Requirements is the point where user element types are stored (c.f. association "requirementType" of RequirementsModel) and these are only applicable within this container. In this use case, UserElementType corresponds to ReqIF's SpecType.

The role of user attributes within the overall EAST-ADL is thus twofold: they (1) provide a means to customize the language to specific company and project needs and (2) constitute an important part of the requirements support of the language.

User attribute values are characterized by user attribute definitions contained in UserElementTypes. The latter are identified by a key attribute. Whenever interoperability with third parties is required, an internet domain naming scheme should be used in order to produce universally unique keys, similar to package names in the Java programming language. For example, a company with a home page URL of "www.example.com" could use the key "com.example.MyPort" for a user element type representing a custom type of port. For more details refer to the documentation of attribute key in metaclass UserElementType.

In order to attach a user attribute value to an instance of a subclass of Identifiable (e.g. DesignFunctionType "WiperSystem") a UserAttributedElement is created that points to this instance (i.e. instance "WiperSystem") via association attributedElement and contains the actual value as an instance of metaclass EAValue. In addition, the UserAttributedElement has to point to a UserElementType containing an appropriate UserAttributeDefinition in order to identify the user attribute, i.e. specify the meaning of the value and its type. For more details, refer to the documentation of UserAttributedElement.

User attributes in EAST-ADL serve a similar purpose to stereotypes in UML2 but are intended as a more light-weight and simpler mechanism, especially with respect to tool implementation.

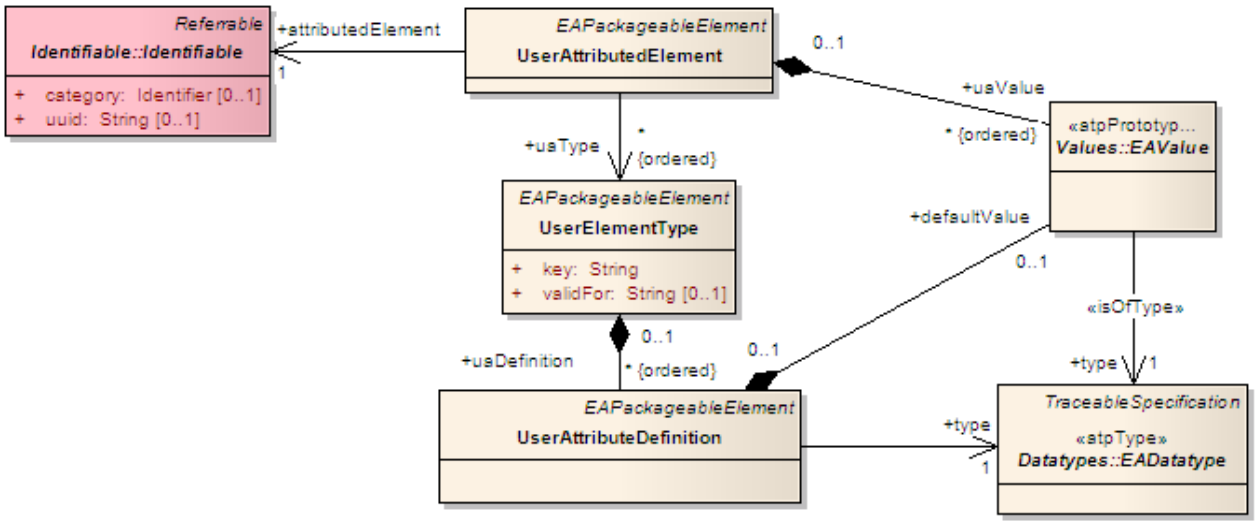

Figure 40. Diagram for User Attributes.

27.2 Element Descriptions

27.2.1 UserAttributeDefinition (from UserAttributes)

Generalizations

- EAPackageableElement (from Elements)

Description

UserAttributeDefinition defines a certain user attribute.

The name of a UserAttributeDefinition should be used in editing tools as a label for the input field representing the user attribute and its description should be presented to the user to explain the meaning of this user attribute.

To identify a user attribute in a universally unique way, its short name is appended to the key of the containing UserElementType after appending a "." character (dot) as a separator. For example, if a UserAttributeDefinition with short name "MyStatus" is contained in a UserElementType with key "com.myCompany.myDepartment.myProject.MyPort", then the user attribute represented by this UserAttributeDefinition has the key "com.myCompany.myDepartment.myProject.MyPort.MyStatus".

Attributes

No additional attributes

Associations

- type : EADatatype [1]
  The type of the user attribute. This type defines the set of legal values for the given user attribute.
- defaultValue : EAValue [0..1] {composite}
  Optional default value of the UserAttributeDefinition. The EAValue shall be typed by the same EADatatype as the UserAttributeDefinition.

Constraints

No additional constraints

Semantics

UserAttributeDefinition defines a user defined attribute.

27.2.2 UserAttributedElement (from UserAttributes)

Generalizations

- EAPackageableElement (from Elements)

Description

UserAttributedElement is used to attach user attribute values to any EAST-ADL or AUTOSAR element, i.e. all instances of all subclasses of Identifiable. What user attributes a certain element should be supplied with can be defined beforehand with UserElementTypes.

According to a common EAST-ADL meta-modeling pattern, the meta-classes that are attributable, i.e. to which user attributes may be attached, do not inherit from meta-class UserAttributedElement but instead UserAttributedElement points to these meta-classes via association "attributedElement" (for example, to allow attaching user attributes to AUTOSAR Identifiable that cannot inherit from EAST-ADL infrastructure meta-classes).

The actual values are given as a contained instance of EAValue and are provided with a definition through the UserAttributeDefinitions in the UserElementType. If more than one value is contained, then the same number of UserElementTypes/UserAttributeDefinitions must be referenced and the order of values and definitions must be consistent (see constraint no. 2 below).

Example: let us assume that a DesignFunctionType "WiperSystem" should be provided with the value "OK" for a user attribute "Status". This is achieved by creating an instance of UserAttributedElement pointing via association "attributedElement" to instance "WiperSystem", pointing via instance "uaType" to the UserElementType with a UserAttributeDefinition "Status" and containing via containment association "uaValue" an EAStringValue "OK".

Attributes

No additional attributes

Associations

- uaValue : EAValue [*] {ordered} {composite}
  An ordered set of values attached to the element given by association "attributedElement". These values must conform in number, order and datatype to the user attribute definitions of the UserElementTypes given by association "uaType" (in depth-first order).
- uaType : UserElementType [*] {ordered}
  The custom type(s) of this element.
- attributedElement : Identifiable [1]
  The element to which one or more user attribute values are attached.

Constraints

[1] The associations "uaValue" and the uaDefinitions of all "uaType"s must refer to the same number of elements.

[2] The order of associations "uaValue" and "uaType" / "uaDefinition" must be consistent, i.e. the n-th EAValue must correspond to the n-th UserAttributeDefinition when listing all UserElementTypes' definitions in depth-first order.

Semantics

UserAttributedElement can be annotated with user attributes.

27.2.3 UserElementType (from UserAttributes)

Generalizations

- EAPackageableElement (from Elements)

Description

UserElementType defines a certain set of user attributes, i.e. it states that all Identifiables of a certain kind (c.f. the validFor attribute) may be provided with a user attribute value of some datatype. For example, it can be specified that all AnalysisFunctionPrototypes may be amended with an attribute "Status".

The name of a UserElementType should be used in editing tools as a label for the input field representing the user attribute and its description should be presented to the user to explain the meaning of this user attribute.

Attributes

- key : String [1]
  The globally unique identifier of the user element type. Any string may be used as key as long as it is globally unique.

  However, there is a recommended procedure for building globally unique keys for user attributes, similar to package naming conventions in the Java programming language:

  (1) use an internet domain name which is sufficiently specific so that you have control over who will use it for user attribute key generation (e.g. "myDepartment.myCompany.com")

  (2) reverse it as in Java package names (e.g. "com.myCompany.myDepartment")

  (3) optionally append additional, dot-separated names for the specific context in which the user attribute is to be used (e.g. "myProject" which results in "com.myCompany.myDepartment.myProject")

  (4) add a last segment that names the user element type and is sufficiently descriptive to explain its purpose (e.g. "MyPort").

  In this example, the key of our status attribute would be "com.myCompany.myDepartment.myProject.MyPort".

  In general, the last segment of the key, i.e. everything following the last dot, should be sufficient to identify the attribute in its usual, most specific context of use. Therefore, implementations may use this last segment as an abbreviated name of the user attribute, e.g. for presenting it in a GUI. But note that the name of the UserElementType should usually be used (if defined).
- validFor : String [0..1]
  Comma-separated list of metaclass names this user element type is applicable to. If undefined, then this type is applicable to all subclasses of metaclass Identifiable. White-space may appear before and after metaclass names and commas.

  Example: If UserElementType 'MyFunction' has its validFor attribute set to "FunctionalDevice, LocalDeviceManager", then the contained UserAttributeDefinitions are only applicable to functional devices and local device managers, i.e. only instances of FunctionalDevice and LocalDeviceManager may be adorned with the 'MyFunction' user element type.

Associations

- uaDefinition : UserAttributeDefinition [*] {ordered} {composite}
  The definitions of user attributes for this UserElementType.

Constraints

[1] The short names of all UserAttributeDefinitions (i.e. value of attribute "shortName" in UserAttributeDefinition, which is inherited from meta-class Referrable) referred to by association "uaDefinition" must be unique within this UserElementType. In other words, no two UserAttributeDefinitions referred to by association "uaDefinition" must have the same short name.

Semantics

UserElementType represents a user defined type of the specified EAST-ADL or AUTOSAR metaclass.

Part X Annexes

This part contains the EAST-ADL Annexes. The first annex is about notation followed by element packages that are preliminary and subject to further refinement before inclusion in the main language.

28 Annex A: Notation

This annex lists the elements with defined notations to be used when the element is shown in a diagram. For those elements that are not listed here the general notation is a solid-outline rectangle with the metaclass name at the top right. The rectangle contains the user defined name of the element.

28.1.1 Actuator (from HardwareModeling)

Actuator is shown as a solid-outline rectangle with double vertical borders. The rectangle contains the name, and its ports or port groups on the perimeter.

28.1.2 AnalysisLevel (from SystemModeling)

The Analysis Architecture is shown as a solid-outline rectangle containing the name, with its ports or port groups on the perimeter. Contained entities may be shown with their connectors (White-box view).

28.1.3 ArrayDatatype (from Datatypes)

The datatype ArrayDatatype is denoted using the rectangle symbol with keyword «Datatype ArrayDatatype».

28.1.4 CommunicationHardwarePin (from HardwareModeling)

CommunicationHardwarePin is shown as a solid square with a C inside. Its name may appear outside the square.

28.1.5 CompositeDatatype (from Datatypes)

The datatype CompositeDatatype is denoted using the rectangle symbol with keyword «Datatype CompositeDatatype».

28.1.6 DeriveRequirement (from Requirements)

A DeriveRequirement relationship is shown as a dashed arrow between two Requirements. The Requirement at the tail of the arrow (the derived Requirement) depends on the Requirement at the arrowhead (the Requirement derived from).

28.1.7 DesignLevel (from SystemModeling)

The DesignLevel is shown as a solid-outline rectangle containing the name, with its ports or port groups on the perimeter. Contained entities may be shown with their connectors and allocations (White-box view).

28.1.8 EABoolean (from Datatypes)

The datatype EABoolean is denoted using the rectangle symbol with keyword «Datatype Boolean».

28.1.9 EADatatype (from Datatypes)

The EADatatype is denoted using the rectangle symbol with keyword «Datatype».

28.1.10 EANumerical (from Datatypes)

The datatype EANumerical is denoted using the rectangle symbol with keyword «Datatype Numerical».

28.1.11 EAString (from Datatypes)

The datatype EAString is denoted using the rectangle symbol with keyword «Datatype String».

28.1.12 ElectricalComponent (from HardwareModeling)

ElectricalComponent is shown as a solid-outline rectangle. The rectangle contains the name, and its ports or port groups on the perimeter.

28.1.13 Enumeration (from Datatypes)

The datatype Enumeration is denoted using the rectangle symbol with keyword «Datatype Enumeration».

28.1.14 EnumerationLiteral (from Datatypes)

An EnumerationLiteral is typically shown as a name, one per line, in the compartment of the Enumeration notation.

28.1.15 FunctionAllocation (from FunctionModeling)

A FunctionAllocation is shown as a dependency (dashed line) with an "allocation" keyword attached to it.

28.1.16 FunctionBehavior (from Behavior)

FunctionBehavior appears as a solid-outline rectangle with "Behavior" at the top right. The rectangle contains the name.

28.1.17 FunctionConnector (from FunctionModeling)

FunctionConnector is shown as a solid line

28.1.18 FunctionPrototype (from FunctionModeling)

Shall be shown in the same style as the class specified as type, however it shall be clear that this is a part.

28.1.19 FunctionType (from FunctionModeling)

The FunctionType is shown as a solid-outline rectangle containing the name, with its FunctionPorts or PortGroups on the perimeter. Contained entities may be shown with their FunctionConnectors (White-box view).

28.1.20 HardwareComponentPrototype (from HardwareModeling)

Shall be shown in the same style as the class specified as type, however it shall be clear that this is a part.

28.1.21 Hazard (from Dependability)

The Hazard is shown as a solid-outline rectangle with "Haz" at the top right. It contains the name of the Hazard and optionally the name of the source entity.

28.1.22 HazardousEvent (from Dependability)

The HazardousEvent is shown as a solid-outline rectangle with "Haz" at the top right. It contains the name of the HazardousEvent and optionally the name of the source entity.

28.1.23 ImplementationLevel (from SystemModeling)

The ImplementationLevel is shown as a solid-outline rectangle containing the name.

28.1.24 IOHardwarePin (from HardwareModeling)

IOHardwarePin is shown as a solid square with an IO inside. Its name may appear outside the square.

28.1.25 Node (from HardwareModeling)

Node is shown as a solid-outline rectangle with Node at the top right. The rectangle contains the name, and its ports or port groups on the perimeter.

28.1.26 PortGroup (from FunctionModeling)

FunctionConnectors connected to FunctionPorts of a PortGroup are graphically collapsed into a single line.

The PortGroup is rendered as its contained ports, but with a double outline.

28.1.27 PowerHardwarePin (from HardwareModeling)

PowerHardwarePin is shown as a solid square with PWR inside. Its name may appear outside the square.

28.1.28 PrecedenceConstraint (from Timing)

PrecedenceConstraint is shown as a dashed arrow with "Precedes" next to it. It points from preceeding to the successive entity.

28.1.29 RangeableValueType (from Datatypes)

The datatype RangeableValueType is denoted using the rectangle symbol with keyword «Datatype RangeableValueType».

28.1.30 Realization (from Elements)

A Realization relationship is shown as a dashed line with a triangular arrowhead at the end that corresponds to the realized entity. The entity at the tail of the arrow (the realizing EAElement or the realizing ARElement) depends on the entity at the arrowhead (the realized EAElement).

28.1.31 Refine (from Requirements)

A Refine relationship is shown as a dashed arrow between the Requirements and EAElement. The entity at the tail of the arrow (the refining EAElement) depends on the Requirement at the arrowhead (the refined Requirement).

28.1.32 Requirement (from Requirements)

Requirement is shown as a solid rectangle with Req top right and its name.

28.1.33 RequirementsHierarchy (from Requirements)

RequirementsHierarchy is shown as a solid-outline rectangle containing the name. Contained entities may also be shown inside (White-box view)

28.1.34 SafetyGoal (from SafetyRequirement)

SafetyGoal is a box with text SafetyGoal at the top left.

28.1.35 Satisfy (from Requirements)

A Satisfy relationship is shown as a dashed line with an arrowhead at the end that corresponds to the satisfied Requirement or UseCaseUseCase. The entity at the tail of the arrow (the satisfying EAElement or the satisfying ARElement) depends on the entity at the arrowhead (the satisfied Requirement or UseCaseUseCase).

28.1.36 Sensor (from HardwareModeling)

Sensor is shown as an oval. The circle contains the name, and its ports or port groups on the perimeter.

28.1.37 SystemModel (from SystemModeling)

The default notation for a SystemModel is a solid-outline rectangle containing the SystemModel's name, and with compartments separating by horizontal lines containing features or other members of the SystemModel. Contained entities may also be shown with their connectors (White-box view).

28.1.38 VehicleLevel (from SystemModeling)

The VehicleLevel is shown as a solid-outline rectangle containing the name. Contained entities may be shown (White-box view).

28.1.39 Verify (from VerificationValidation)

A Verify relationship is shown as a dashed arrow between the Requirements and VVCase.

-

29 Annex B: Needs

This annex contains preliminary extensions to EAST-ADL for the modeling of stakeholder needs and related information. It is fully aligned with the language but not yet validated and ready for inclusion in the base specification.

30 Needs

30.1 Overview

This annex contains preliminary extensions to EAST-ADL for the modeling of stakeholder needs and related information. It is fully aligned with the language but not yet validated and ready for inclusion in the base specification.

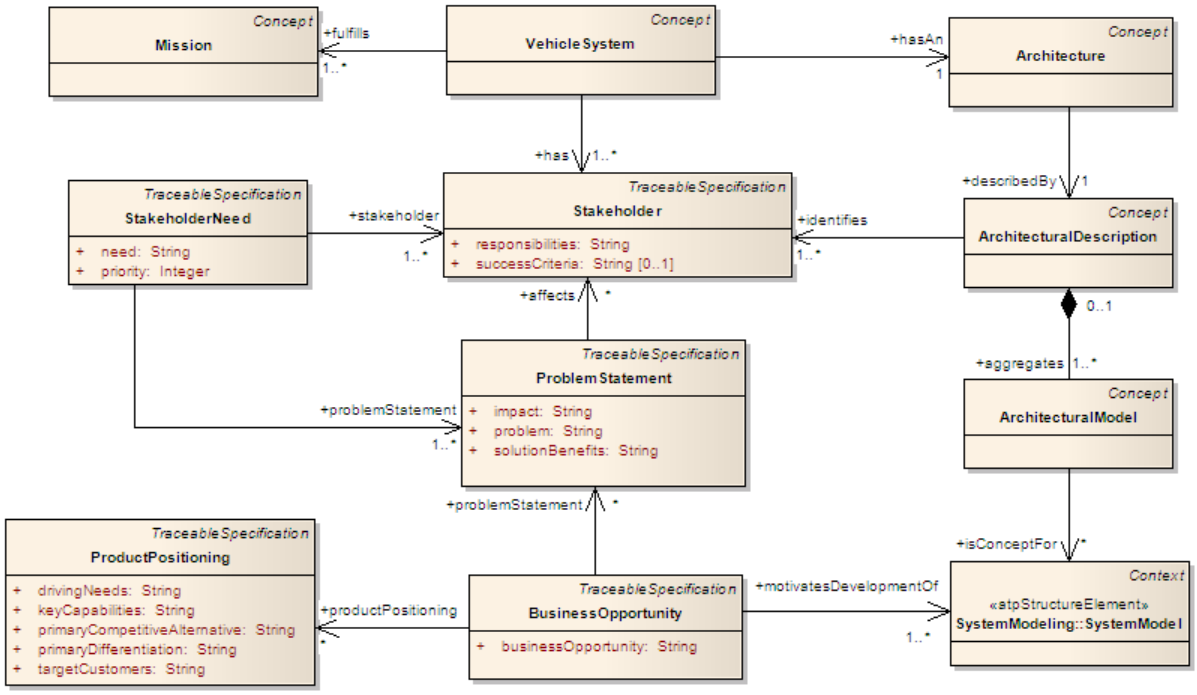

Figure 41. Diagram for Needs.

30.2 Element Descriptions

30.2.1 ArchitecturalDescription (from Needs)

Generalizations

- Concept (from Needs)

Description

A collection of products to document an architecture. [IEEE 1471]

Attributes

No additional attributes

Associations

- aggregates : ArchitecturalModel [1..*] {composite}
- identifies : Stakeholder [1..*]

Constraints

No additional constraints

Semantics

-

30.2.2 ArchitecturalModel (from Needs)

Generalizations

- Concept (from Needs)

Description

A view may consist of one or more architectural models. Each such architectural model is developed using the methods established by its associated architectural viewpoint. An architectural model may participate in more than one view. [IEEE 1471]

Attributes

No additional attributes

Associations

- isConceptFor : SystemModel [*]

Constraints

No additional constraints

Semantics

-

30.2.3 Architecture (from Needs)

Generalizations

- Concept (from Needs)

Description

The fundamental organization of a system embodied by its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. [IEEE 1471]

Attributes

No additional attributes

Associations

- describedBy : ArchitecturalDescription [1]

Constraints

No additional constraints

Semantics

-

30.2.4 BusinessOpportunity (from Needs)

Generalizations

- TraceableSpecification (from Elements)

Description

The business opportunity represents a brief description of the business opportunity being met by developing the electrical/electronic system which establishes traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

Attributes

- businessOpportunity : String [1]
  This attribute holds a brief description of the business opportunity being met by developing the electrical/electronic system. This redefines the text attribute in TraceableSpecification.

Associations

- motivatesDevelopmentOf : SystemModel [1..*]
  The SystemModel that the BusinessOpportunity motivates development of.
- problemStatement : ProblemStatement [*]
  Optional relation to brief statements summarizing the problem being solved.
- productPositioning : ProductPositioning [*]
  The optional ProductPositioning provides an overall statement summarizing, at the highest level, the unique position the product intends to fill in the marketplace.

Constraints

No additional constraints

Semantics

-

30.2.5 Concept (from Needs) {abstract}

Generalizations

- EAElement (from Elements)

Description

An abstract or general idea inferred or derived from specific instances. [Webster]

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

30.2.6 Mission (from Needs)

Generalizations

- Concept (from Needs)

Description

A mission is a use or operation for which a system is intended by one or more stakeholders to meet some set of objectives. [IEEE 1471]

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

30.2.7 ProblemStatement (from Needs)

Generalizations

- TraceableSpecification (from Elements)

Description

The problem statement represents a brief statement summarizing the problem being solved which gives the opportunity to establish traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

The problem statement could be extended with further modeling of dependencies between different problems and deduction of root problems

Attributes

- impact : String [1]
  The impact of the problem
- problem : String [1]
  The brief problem statement. This redefines the text attribute in TraceableSpecification.
- solutionBenefits : String [1]
  Lists some key benefits of a successful solution.

Associations

- affects : Stakeholder [*]
  The Stakeholders affected by the problem.

Constraints

No additional constraints

Semantics

-

30.2.8 ProductPositioning (from Needs)

Generalizations

- TraceableSpecification (from Elements)

Description

The problem positioning represents an overall brief statement summarizing, at the highest level, the unique position the product intends to fill in the marketplace which gives the opportunity to establish traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

Positioning is assumed to belong to a particular context, typically a system, but also for a smaller part of a system.

Attributes

- drivingNeeds : String [1]
  Brief statement of key benefit; that is, the compelling need for the product.
- keyCapabilities : String [1]
  Brief statement of the key capabilities
- primaryCompetitiveAlternative : String [1]
  Brief statement of primary competitive alternative
- primaryDifferentiation : String [1]
  Brief statement of primary differentiation
- targetCustomers : String [1]
  Brief statement of target customers.

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

30.2.9 Stakeholder (from Needs)

Generalizations

- TraceableSpecification (from Elements)

Description

The stakeholder represents various roles with regard to the creation and use of architectural descriptions. Stakeholders include clients, users, the architect, developers, and evaluators. [IEEE 1471]

Attributes

- responsibilities : String [1]
  Summarize the Stakeholder's key responsibilities with regard to the electrical/electronic system being developed; that is, their interest as a Stakeholder.
- successCriteria : String [0..1]
  Describes how the Stakeholder defines success.

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

30.2.10 StakeholderNeed (from Needs)

Generalizations

- TraceableSpecification (from Elements)

Description

Stakeholder needs represent a list of the key problems as perceived by the stakeholder, and it gives the opportunity to establish traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

Attributes

- need : String [1]
  The brief need statement. Redefines text.
- priority : Integer [1]
  The priority of the need.

Associations

- problemStatement : ProblemStatement [1..*]
  The ProblemStatement that provide statements summarizing the problem being solved.
- stakeholder : Stakeholder [1..*]
  Role with regard to the creation and use of architectural description.

Constraints

No additional constraints

Semantics

-

30.2.11 VehicleSystem (from Needs)

Generalizations

- Concept (from Needs)

Description

A collection of components organized to accomplish a specific function or set of functions. [IEEE 1471]

Attributes

No additional attributes

Associations

- hasAn : Architecture [1]

- fulfills : Mission [1..*]
- has : Stakeholder [1..*]

Constraints

No additional constraints

Semantics

-

31 Annex C: BehaviorDescription

This annex contains preliminary extensions to EAST-ADL for the modeling of behavior description. It is fully aligned with the language but not yet validated and ready for inclusion in the base specification.

32 BehaviorDescription

32.1 Overview

The Behavior Description Annex provides the language support for allowing a more precise declaration of various behavior concerns, such as assumed or implied by requirements and quality constraints, assigned to system environment, functions and components, or test procedures. It also constitutes the basis for consolidating such concerns in a common system design context and a gateway for supporting model transformations from EAST-ADL to external methods and tools for ensuring the analytical leverage.

In particular, this architectural language support would give the following benefits.

1. The modeling support for behavior constraint descriptions would allow the system developers to refine textual requirements by formalizing the statements of use case and operational scenarios and thereby to elicit, validate, and derive related concerns on the basis of particular architectural design assumptions.

2. As a part of the overall language support for managing the traceability of requirement satisfactions, the descriptions of behavior constraints can be used to specify the required conformity of IP-protected black-box functions or components. This constitutes a basis for having a more precise reasoning about the compositional effect (i.e. the compositionality and composability).

3. By managing all behavioral specifications in a common architectural context, EAST-ADL behavior descriptions provide the fundamental support for assessing the correctness and completeness of refinements of system artifacts across multiple levels of abstraction for final code generation.

4. By having all behavior specifications in the same context of architecture design, this language package also provides a basis for capturing the design of advanced mode logics such as in regard to fault-tolerance and quality-of-services. For such advanced features, the ability of describing, predicting and managing the system wide effects of modes is of critical importance. This means that one should be able to relate system modes control with system operational situations, application behaviors, the schemes of execution and resource deployment, etc.

5. When targeting error specifications, the behavior descriptions can be used to refine the definitions of estimated failure modes by providing a precise specification of faulty conditions in value and time and capturing the transitions between nominal states and errors.

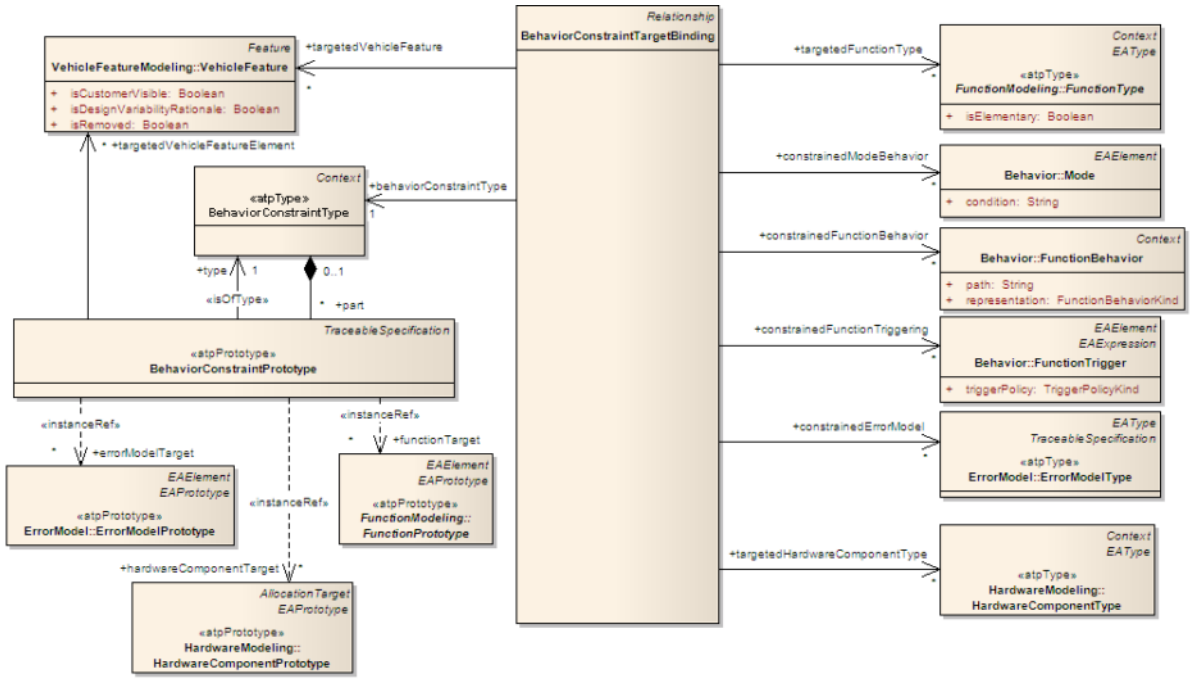

Figure 42. Diagram for dependencies of BehaviorConstraints.

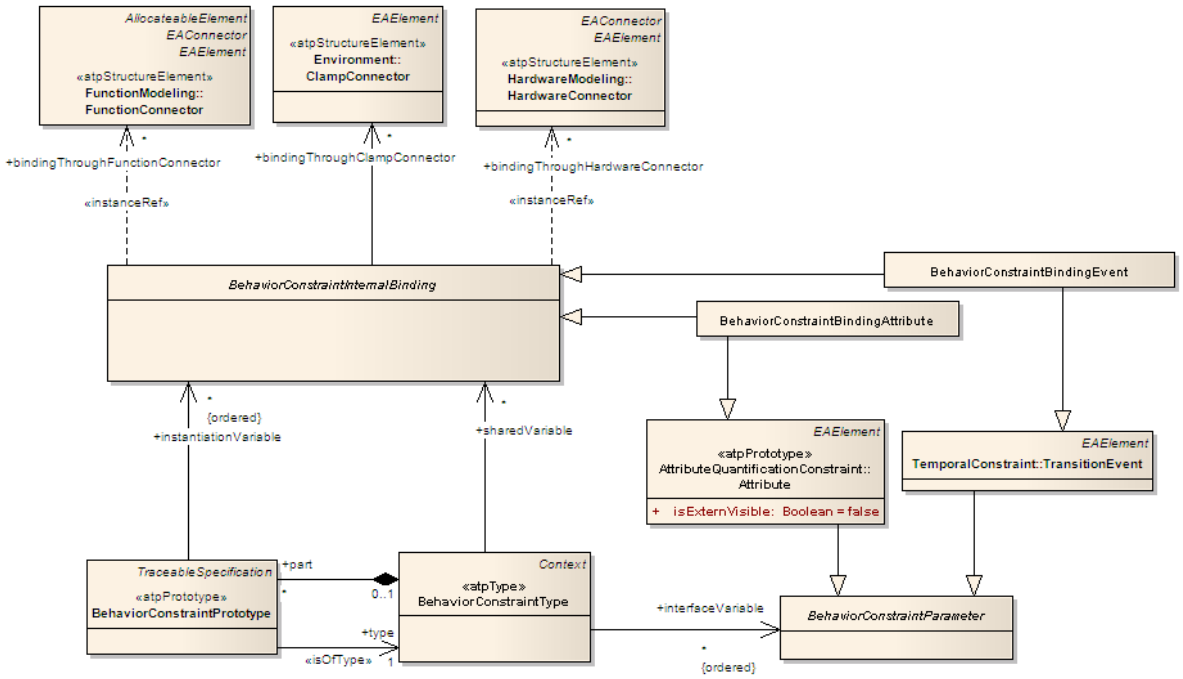

Figure 43. BehaviorConstraintParameterBinding

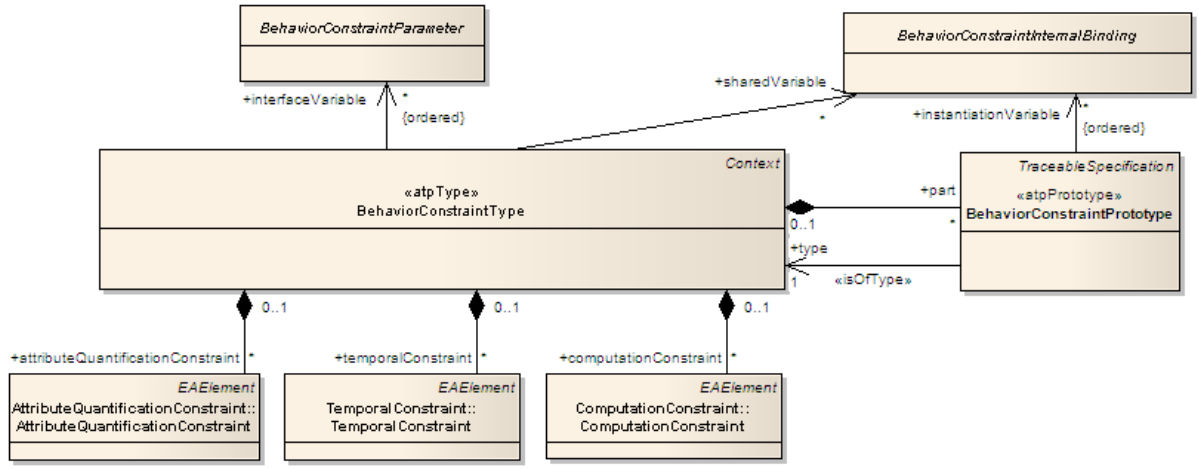

Figure 44. Diagram for organization in BehaviorConstraints.

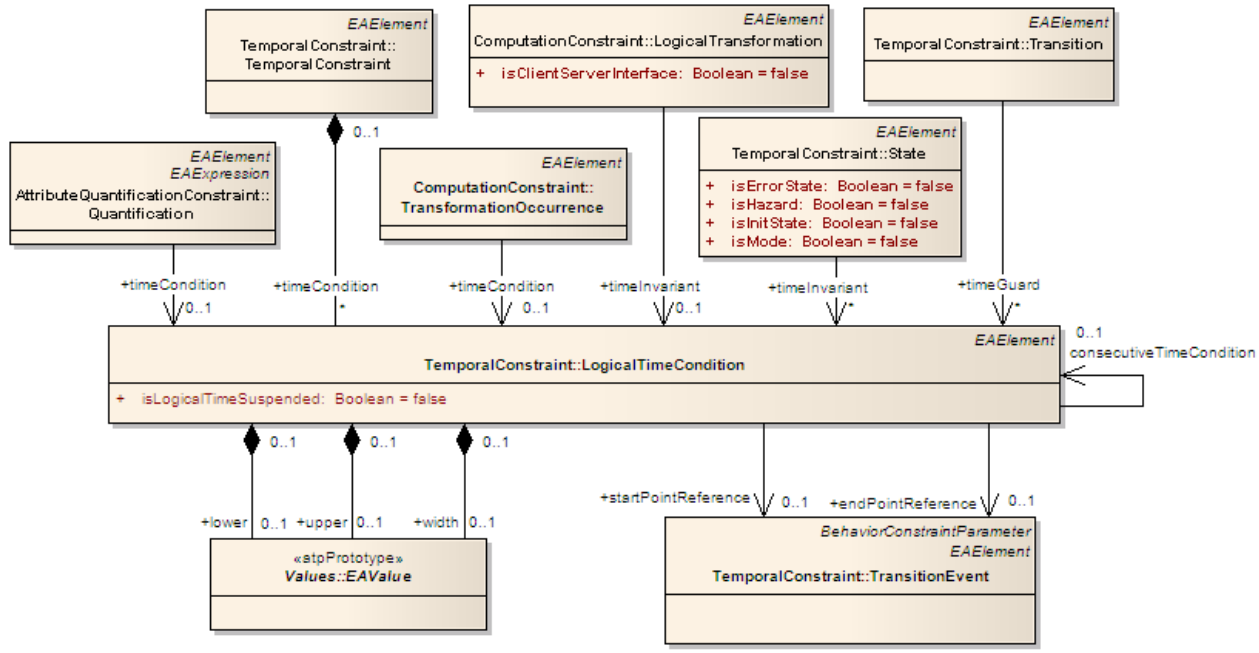

Figure 45. LogicalTimeConditionMappingToTiming

32.2 Element Descriptions

32.2.1 BehaviorConstraintBindingAttribute (from BehaviorDescription)

Generalizations

- BehaviorConstraintInternalBinding (from BehaviorDescription)
- Attribute (from AttributeQuantificationConstraint)

Description

BehaviorConstraintBindingEvent is a specialization of BehaviorConstraintBindingParameter.

It allows a behavior constraint type to declare the value attributes to be shared of its prototypes.

See also BehaviorConstraintBindingParameter.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

32.2.2 BehaviorConstraintBindingEvent (from BehaviorDescription)

Generalizations

- BehaviorConstraintInternalBinding (from BehaviorDescription)
- TransitionEvent (from TemporalConstraint)

Description

BehaviorConstraintBindingEvent is a specialization of BehaviorConstraintBindingParameter. It allows a behavior constraint type to declare the discrete events to be shared by its prototypes.

See also BehaviorConstraintBindingParameter.

Attributes

No additional attributes

Associations

No additional associations

Constraints

No additional constraints

Semantics

-

32.2.3 BehaviorConstraintInternalBinding (from BehaviorDescription) {abstract}

Generalizations

None

Description

BehaviorConstraintInternalBinding is the modeling construct for the declaration of parameters to be shared by the parts (i.e. behavior constraint prototypes) of a behavior constraint type. In other words, a behavior constraint type uses such parameters to bind the parameters of its parts (BehaviorConstraintType.part:BehaviorConstraintPrototype.instantiationVariable). For such a binding, the declarations of prototype instantiation (BehaviorConstraintType.part:BehaviorConstraintPrototype.instantiationVariable) refer directly to

the part binding parameters of the instantiation context (BehaviorConstraintType.partBindingParameter)

Each binding parameter can have a structural correspondence (bindingThroughFunctionConnector, bindingThroughClampConnector, bindingThrough-LogicalBus, or bindingThrough-HardwareConnector), stating the structural channels through which the binding takes place.

In the meta-model, the abstract binding parameter is further specialized into

* BehaviorConstraintBindingAttribute - the contextual parameters that are value attributes.

* BehaviorConstraintBindingEvent - the contextual parameters that are discrete events.

Attributes

No additional attributes

Associations

- bindingThroughClampConnector : ClampConnector [*]

Dependencies

- bindingThroughFunctionConnector : FunctionConnector [*] «instanceRef»
- bindingThroughHardwareConnector : HardwareConnector [*] «instanceRef»

Constraints

When a binding of behavior constraint prototypes go across different system functions or components, there should be at least one corresponding structural communication connector through which such bindings can take place (i.e. bindingThroughFunctionConnector, bindingThroughClampConnector, bindingThrough-LogicalBus, or bindingThrough-HardwareConnector).

Semantics

A BehaviorConstraintBindingParameter is an event- or data- channel connecting behaviors. See also Attribute and TransitionEvent.

32.2.4 BehaviorConstraintParameter (from BehaviorDescription) {abstract}

Generalizations

None

Description

BehaviorConstraintParameter is the modeling construct for the declarations of the parameters that a behavior constraint type offer for its instantiations. During the instantiation, a behavior constraint prototype declares the particular contextual parameters to be bound (BehaviorConstraintPrototype.BehaviorinstantiatedWithParameter) with the parameters of its corresponding behavior constraint types (BehaviorConstraintPrototype.type:BehaviorConstraintType.parameter). This allows thereby the values of those contextual parameters to be assigned to the parameters of prototypes.

Attributes

No additional attributes

Associations

No additional associations

Constraints

See Attribute and TransitionEvent.

Semantics

See Attribute and TransitionEvent.

32.2.5 BehaviorConstraintPrototype (from BehaviorDescription) «atpPrototype»

Generalizations

- TraceableSpecification (from Elements)

Description

BehaviorConstraintPrototype is the modeling construct for declaring the instantiated occurrence(s) of a behavior constraint type (BehaviorConstraintPrototype.type) in particular behavior specification context where the behavior constraint type acts as part.

BehaviorConstraintPrototype.instantiationVariable {ordered} is declared by BehaviorConstraintPrototype.type.interfaceVariable {ordered}

Attributes

No additional attributes

Associations

- targetedVehicleFeatureElement : VehicleFeature [*]
  The corresponding vehicle feature elements of a behavior constraint prototype.
- instantiationVariable : BehaviorConstraintInternalBinding [*] {ordered}
  The contextual parameters used for the prototype instantiation.
- type : BehaviorConstraintType [1]
  «isOfType»
  The behavior constraint type instantiated by the prototype.

Dependencies

- functionTarget : FunctionPrototype [*]
  «instanceRef»
- hardwareComponentTarget : HardwareComponentPrototype [*]
  «instanceRef»
- errorModelTarget : ErrorModelPrototype [*]
  «instanceRef»

Constraints

[1] A BehaviorConstraintPrototype must has a type (BehaviorConstraintPrototype.type) and a context where it acts as part (BehaviorConstraintType.part).

BehaviorConstraintType.part:BehaviorConstraintPrototype.instantiationVariable can only be a subset of the BehaviorConstraintType.interfaceVariable.

Semantics

See BehaviorConstraintType.

32.2.6 BehaviorConstraintTargetBinding (from BehaviorDescription)

Generalizations

- Relationship (from Elements)

Attributes

No additional attributes

Associations

- targetedVehicleFeature : VehicleFeature [*]
  The target vehicle feature of a behavior constraint description .
- targetedFunctionType : FunctionType [*]
  The tareget function of a behavior constraint description.
- targetedHardwareComponentType : HardwareComponentType [*]
- constrainedModeBehavior : Mode [*]
  The mode definition being refined by a behavior constraint description
- constrainedFunctionBehavior : FunctionBehavior [*]
  The function behavior being refined by a behavior constraint description
- constrainedFunctionTriggering : FunctionTrigger [*]
  The function triggering being refined by a behavior constraint description
- constrainedErrorModel : ErrorModelType [*]
  The error model being refined by a behavior constraint description
- behaviorConstraintType : BehaviorConstraintType [1]

Constraints

No additional constraints

Semantics

-

32.2.7 BehaviorConstraintType (from BehaviorDescription) «atpType»

Generalizations

- Context (from Elements)

Description

The specification of behavior constraints provides the modeling support for formalizing, integrating, and managing various behavioral concerns in a common context of system architecture design. A behavior constraint can be annotated either for refining requirements or for precisely defining the behavioral properties of design and analysis artifacts.

According to the fundamental needs of system design and analysis, an EAST-ADL behavior constraint specification is subdivided into three categories (i.e. AttributeQuantificationConstraint, TemporalConstraint, and ComputationConstraint). It is up to the users of EAST-ADL language, according to their particular design and analysis contexts, to decide the exact types and degree of constraints to be applied.

A behavior constraint specification has both type and prototype(s) based on a type-prototype pattern for composition. The behavior constraint type specification establishes a template for a range of behavioral concerns that share some common declarations and semantics. A behavior constraint type can have parameters (i.e. events and data) for its instantiations in particular contexts. A

behavior constraint type can also have internal parameters shareable by its own prototypes. The behavior constraint prototype specifications declare the particular instantiations of the type. During an instantiation, the parameters of behavior constraint type are bound to some parameters of the contexts (which are the partBindingParameter of the contextual behavior constraint types). Through such binding declarations, the prototypes of behavior constraint types (i.e. their instantiations) are connected to the contextual parameters.

EAST-ADL associates behavior constraints to the requirements, design or analysis artifacts. Due to such associations, a behavior constraint specification can get many different roles in system development and thereby be composed with or related to other behavior constraints in many different ways.

1. When associated to requirements with a Refine relationship, a behavior constraint specification refines the textual requirement descriptions.

2. When associated to functions and function behaviors, a behavior constraint specification defines the behavioral properties that have to be satisfied for the reasoning of system design (i.e. the compositionality and composability) and realizations.

3. When associated to modes, a behavior constraint specification defines the behavioral concerns of system modes, including their relations to other system application and execution behaviors.

4. When associated to error models, a behavior constraint specification refines the definitions of estimated failure modes by providing a precise specification of faulty conditions in value and time and constitutes a basis for capturing the transitions between nominal states and errors.

Attributes

No additional attributes

Associations

- sharedVariable : BehaviorConstraintInternalBinding [*]
  Parameters that a behavior constraint type has for binding its parts (i.e. prototypes).
- interfaceVariable : BehaviorConstraintParameter [*] {ordered}
  The parameters that a behavior constraint type offer at its interface for its instantiation
- part : BehaviorConstraintPrototype [*] {composite}
  Other behavior constraints that are instantiated as the internal parts.
- attributeQuantificationConstraint : AttributeQuantificationConstraint [*] {composite}
  The attribute quantification constraints underlying a behavior constraint specficaiton.
- computationConstraint : ComputationConstraint [*] {composite}
  The computation constraints underlying a behavior constraint specficaiton.
- temporalConstraint : TemporalConstraint [*] {composite}
  The temporal constraints underlying a behavior constraint specficaiton.

Constraints

[1] A behavior constraint references at least one requirement, vehicle feature, mode, function type, function behavior, function trigger, or error behavior definition.

Semantics

The EAST-ADL support for explicit behavior description is fundamentally a Hybrid-System Model, i.e. an aggregation of AttributeQuantificationConstraint, TemporalConstraint, and ComputationConstraint.

A behavior constraint type is instantiated with prototypes.

33 AttributeQuantificationConstraint

33.1 Overview

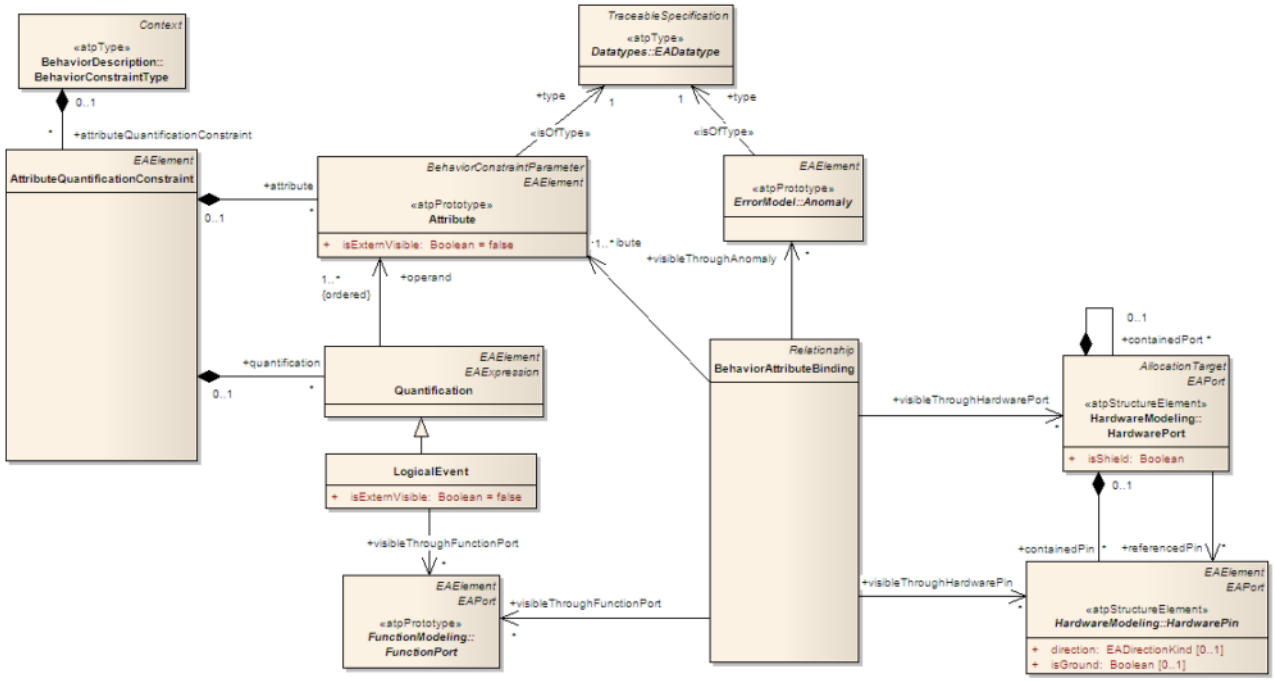

Figure 46. AttributeQuantificationConstraint

33.2 Element Descriptions

33.2.1 Attribute (from AttributeQuantificationConstraint) «atpPrototype»

Generalizations

- EAElement (from Elements)
- BehaviorConstraintParameter (from BehaviorDescription)

Description

The attribute (Attribute) denotes a parameter or argument of a behavior constraint specification. An attribute can be a constant, simple, or complex data, given by the corresponding EAST-ADL data types (EADataType) for the related meta-information like unit, valid range, required accuracy, etc.

An attribute can represent an in-, out-, or local-quantity to be processed. If an attribute is externally visible (isExternVisble = true), it denotes an input or output variable and has associated structural ports given by the function ports for the external accesses.

Attributes are instantiation parameters (BehaviorInstantiationParameter), to which certain values can be assigned when a behavior constraint type is instantiated as behavior constraint instances (i.e. prototypes) in certain specification contexts.

Attributes

- isExternVisible : Boolean = false [1]

Associations

- type : EADatatype [1]
  «isOfType»
  The type of the attribute.

Constraints

An attribute must be typed by EADataType.

Semantics

The attributes of a behavior constraint specification is a subset of elements in the vector space of R^n, where R is the real number and n is a natural number defining the dimension of vector space.

33.2.2 AttributeQuantificationConstraint (from AttributeQuantificationConstraint)

Generalizations

- EAElement (from Elements)

Description

Attribute quantification constraints (AttributeQuantificationConstraint) are concerned with the value conditions of attributes underlying a behavior on a timeline. They are useful for declaring the variables (e.g. the input-, output- and internal variables of a function), their expected values and logical relations. An attribute quantification constraint can be expressed either by simple equations like F = m*a, V >= 90, or dynamics models. When necessary, the straints on computational operations for data transformations and value assignment can be declared through the computation constraints (ComputationConstraint).

Attributes

No additional attributes

Associations

- quantification : Quantification [*] {composite}
  The attributes quantification of a behavior constraint.
- attribute : Attribute [*] {composite}
  The value attributes of a behavior constraint.

Constraints

No additional constraints

Semantics

The attribute quantification constraint specification is a pair/tuple of two sets: 1. the set of attributes for the behavior being specified; 2. the set of quantification statements over the attributes.

33.2.3 BehaviorAttributeBinding (from AttributeQuantificationConstraint)

Generalizations

- Relationship (from Elements)

Attributes

No additional attributes

Associations

- visibleThroughFunctionPort : FunctionPort [*]
  The corresponding function ports for an attribute.
- visibleThroughHardwarePin : HardwarePin [*]
  A pin in the mapped HardwareComponentType.
- visibleThroughHardwarePort : HardwarePort [*]
  A port in the mapped HardwareComponentType.
- visibleThroughAnomaly : Anomaly [*]
  An anomaly in the mapped ErrorFunctionType.
- attribute : Attribute [1..*]

Constraints

No additional constraints

Semantics

-

33.2.4 LogicalEvent (from AttributeQuantificationConstraint)

Generalizations

- Quantification (from AttributeQuantificationConstraint)

Description

Logical Event is the modeling construct for the declarations of the value conditions that, when fulfilled, may trigger state transitions. If a logical event is externally visible (isExternVisble == true), it is disseminated through function ports.

Attributes

- isExternVisible : Boolean = false [1]

Associations

- visibleThroughFunctionPort : FunctionPort [*]

Constraints

see Quantification.

Semantics

see Quantification.

33.2.5 Quantification (from AttributeQuantificationConstraint)

Generalizations

- EAElement (from Elements)
- EAExpression (from Values)

Description

A quantification is a statement over the attributes about their value condition or relation.

Together with the attribute definitions, it also provides the support for annotating acausal dynamic behavior constraints in terms of continuous-time and discrete-time dynamics models. In the development of embedded systems, such acausal specifications of behaviors are necessary for the definitions of system environments (e.g. the physical plants), electrical and electronics devices (e.g. the transfer functions of actuators)

Attributes

No additional attributes

Associations

- operand : Attribute [1..*] {ordered}
  The operands of quantification expressions.
- timeCondition : LogicalTimeCondition [0..1]

Constraints

[1] A quantification is applied to at least one attribute.

Semantics

The quantification is a tuple of: 1. the operands of quantification expressions given by attributes; 2. the time conditions of concern; 3. the actual expressions of properties over single or multiple attributes.

EAST-ADL does not define logic and arithmetic operators for the expressions of parameter conditions but would support the definitions in future extensions.

34 ComputationConstraint

34.1 Overview

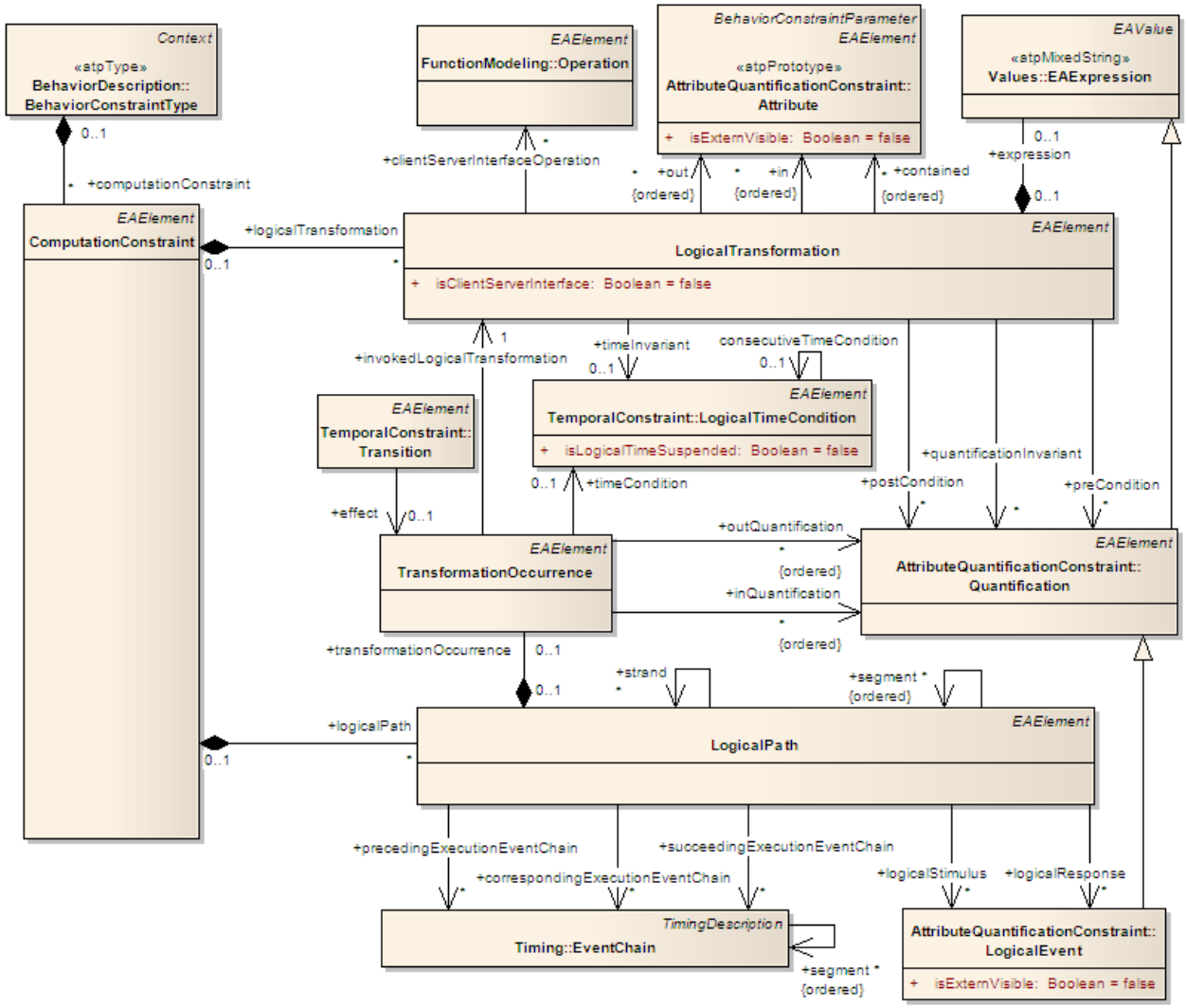

Figure 47. ComputationConstraint

34.2 Element Descriptions

34.2.1 ComputationConstraint (from ComputationConstraint)

Generalizations

- EAElement (from Elements)

Description

Computation constraints (ComputationConstraint) provide the language support for specifying the restrictions on data processing, especially when the details of design are not available (e.g. for the reasons of software component IP-protection). The descriptions can be related both to the expected logical transformations of data and to the expected cause-effect flow of events.

Attributes

No additional attributes

Associations

- logicalTransformation : LogicalTransformation [*] {composite}
  The required computation activities.
- logicalPath : LogicalPath [*] {composite}
  The paths of quantities across the required computation activities.

Constraints

[1] A computation constraint contains at least one transformation or one flow definition.

Semantics

The EAST-ADL computation constraint is a pair/tuple of: 1. a set of restrictions on the logical transformations of data, 2. a set of restrictions on the cause-effect paths logical transformations.

34.2.2 LogicalPath (from ComputationConstraint)

Generalizations

- EAElement (from Elements)

Description

The EAST-ADL logical path (LogicalPath) is a set of restrictions on the cause-effect flows of some observable logical and executional events. It provides the modeling support for annotating the expected cause-effect traces across a system or a component.

A logical path specifies the overall causality of computation by relating execution events with logical transformations and logical events. An execution event can be the triggering of function, port reading or writing, which constitutes the basis for the description of execution control using timing event chains (Timing::EventChain). Compared to such execution events, the logical transformation and logical events are only concerned with the computational logic. The specification of logical path allows the internal causality of the computations of a function/component to be captured and merged explicitly with the related external execution events.

Logical paths can be combined in parallel (strand) or in sequence (segment).

Attributes

No additional attributes

Associations

- precedingExecutionEventChain : EventChain [*]
  The preceding execution event chains.
- succeedingExecutionEventChain : EventChain [*]
  The succeeding execution event chains.
- correspondingExecutionEventChain : EventChain [*]
  The corresponding execution event chains.

- logicalResponse : LogicalEvent [*]
  The logical stimulus of a logical path.
- logicalStimulus : LogicalEvent [*]
  The logical response of a logical path.
- transformationOccurrence : TransformationOccurrence [0..1] {composite}
  The activation of logical transformations in a logical path.
- strand : LogicalPath [*]
  Other logical paths in parallel (strand).
- segment : LogicalPath [*] {ordered}
  Other subordinate logical pathes in sequence.

Constraints

No additional constraints

Semantics

A logical path is a set of restrictions on the cause-effect flows of computation. When applied to a function/component, a logical path defines the correspondence from a triple of logical stimulus (logicalStimulus), logical transformation (transformationOccurrance), and logical response (logicalResponse), to a triple of preceding execution event chains (precedingEventChain), the corresponding execution event chains (correspondingExecutionEventChain), and the succeeding execution event chains (succeedingEventChain).

By describing the internal causality of a function/component, a logical path may refine an execution event chain (correspondingExecutionEventChain), which is primarily used to capture the causality of triggering, port reading and writing events.

34.2.3 LogicalTransformation (from ComputationConstraint)

Generalizations

- EAElement (from Elements)

Description

A logical transformation (LogicalTransformation) is a set of restrictions on the computation activity for some data. That is, given some in-&local-data that meet certain preconditions, such a computation activity always maps such data to some out-data that meet the related postconditions if the time-&value-invariants are not violated during the computation.

Each computation activity executes some functions for the mapping of quantities, which can be based on arithmetic, Boolean- or string-related calculations. The expressions (Expression) for any further lower level details of a transformation can be based on an external language, such as the MISRA-C.

A logical transformation can define the following conditions of computation activity:

1. The pre-conditions, i.e. the quantifications that must be satisfied just prior to data-processing,

2. The value-invariants, i.e. the value conditions that must be satisfied throughout the execution of data-processing,

3. The time-invariants, i.e. the time conditions that must be satisfied throughout the execution of data-processing,

4. The post-conditions, i.e. the quantifications that must be satisfied just after the execution of data-processing.

Attributes

- isClientServerInterface : Boolean = false [1]

Associations

- clientServerInterfaceOperation : Operation [*]
  The client-server interface that a logical transformation description is applied to(isClientServerInterface=true)
- expression : EAExpression [0..1] {composite}
- contained : Attribute [*] {ordered}
  The data that are used both as internal attribute of the transformation.
- out : Attribute [*] {ordered}
  The output data of the transformation.
- in : Attribute [*] {ordered}
  The input data of the transformation.
- quantificationInvariant : Quantification [*]
  The parameter conditions that must remain unchanged by the execution of the transformation.
- preCondition : Quantification [*]
  The parameter conditions that must hold before the execution of the transformation.
- postCondition : Quantification [*]
  The parameter conditions that must hold after the execution of the transformation.
- timeInvariant : LogicalTimeCondition [0..1]
  The duration bounds when the transformation takes place.

Constraints

[1] If a logical transformation description is applied to a client-server interface (isClientServerInterface=true), it has at least one corresponding operation specified in a client-server interface definition (FunctionModelling::Operation).

Semantics

The computation activity of a logical transformation can be activated in logical paths or in state transitions. The execution follows the run-to-completion assumption. This means that the execution is only possible when the previous execution instance of the same transformation is fully completed. For a system function, the amount of time to execute its transformations is constrained by the EAST-ADL function event in the timing package.

34.2.4 TransformationOccurrence (from ComputationConstraint)

Generalizations

- EAElement (from Elements)

Description

A transformation occurrence (TransformationOccurrence) denotes the activations of logical transformations due to state transitions or logical paths. A transformation occurrence can also have a time condition (timeCondition), stating the time instances when the invocation happens. If a logical transformation is invoked, its in-data will be assigned with particular values by the invocation context (inQuantification). As the consequence of transformation, the out-data will also be assigned with particular value (outQuantification).

Attributes

No additional attributes

Associations

- inQuantification : Quantification [*] {ordered}
- outQuantification : Quantification [*] {ordered}
- invokedLogicalTransformation : LogicalTransformation [1]
  The definitions of logical transformations to be invoked
- timeCondition : LogicalTimeCondition [0..1]

Constraints

No additional constraints

Semantics

A logical transformation can only occur in a state transition or a logical path. In such an occurrence, a set of logical transformations are invoked. Given some particular quantifications of in-data (inQuantification) and time conditions (timeCondition), some the particular quantifications of out-data will be satisfied after the invocation.

35 TemporalConstraint

35.1 Overview

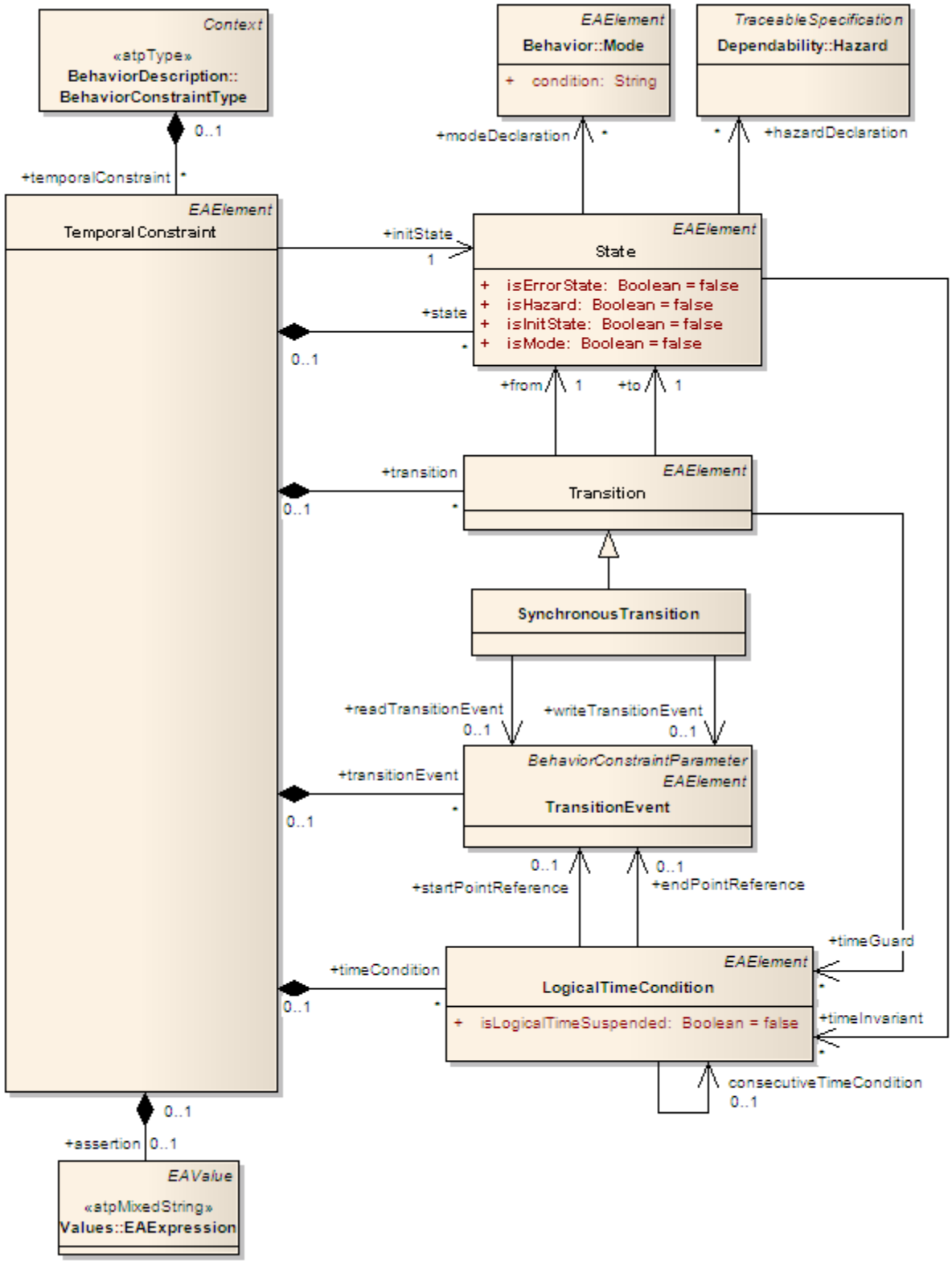

Figure 48. TemporalConstraint

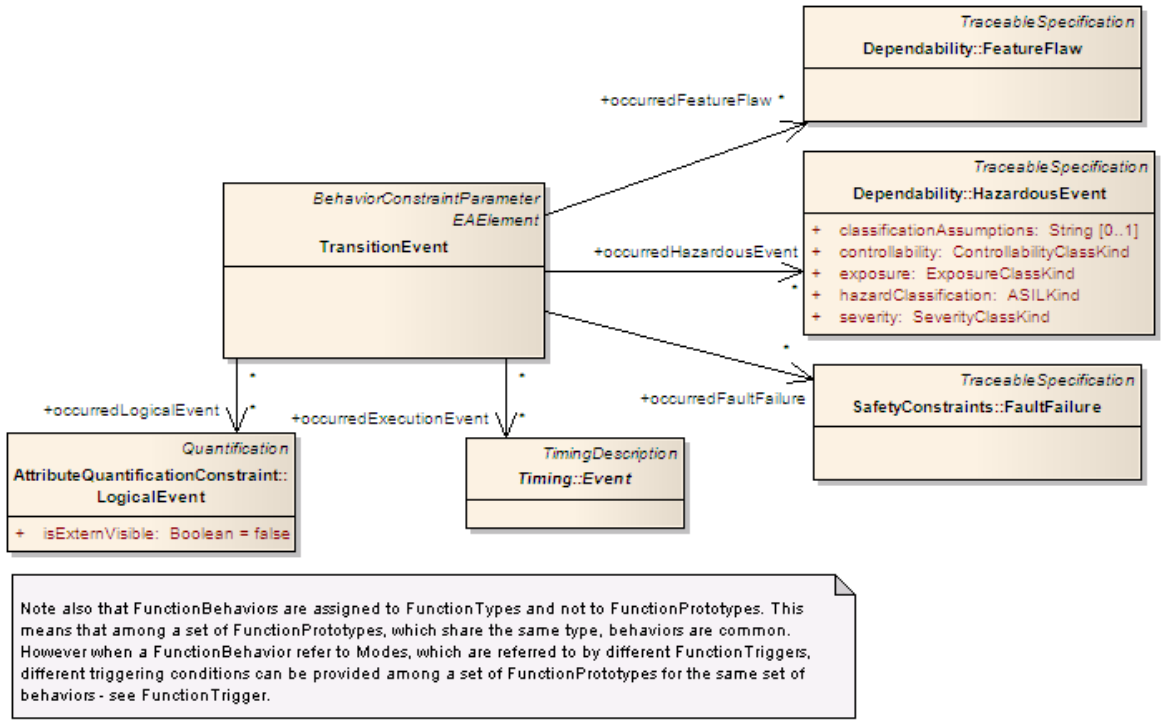

Figure 49. EventOccurrenceMappingToEvents

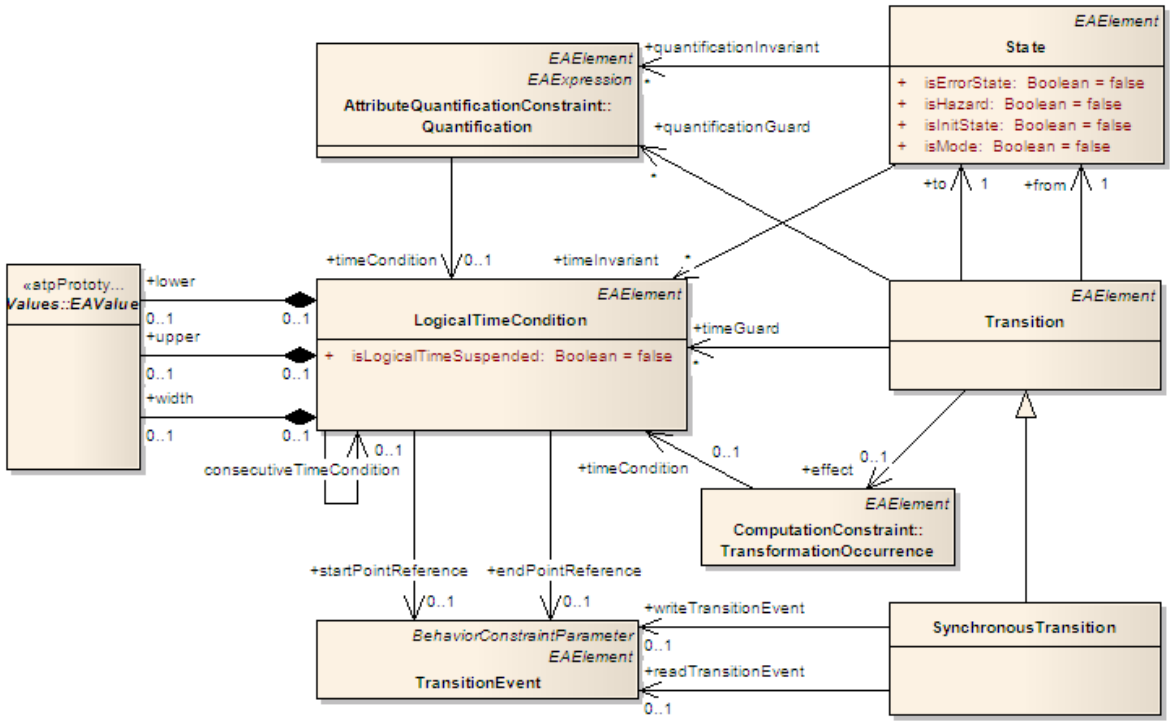

Figure 50. Transition

35.2 Element Descriptions

35.2.1 LogicalTimeCondition (from TemporalConstraint)

Generalizations

- EAElement (from Elements)

Description

The logical time condition is an abstract notion of time for the descriptions of behavior constraints. Declarations of such time conditions can be used to define the time basis of continuous- and discrete-time dynamics or the timing concerns in statemachine or data-processing related behaviors.

The semantics of logical time conditions can be further refined by associating such conditions to the occurrences of execution events (TransitionEvents), such as for defining the change of an environmental condition or the triggering of a function. This makes it possible to precisely define the reference points of a time interval (i.e. startPointReference and endPointReference).

A time condition can have a consecutive time condition on the same time line. E.g. if condition1=[t1, t2], then the consecutive time condition is condition2=[t2, t3].

With EAST-ADL, the expression of the value of a logical time condition is based on the Timing::TimeDuration in the format of CseCode as in AUTOSAR and MSR/ASAM. For descriptions where the notion of time proceeding is not of interest, a time condition with isLogicalTimeSuspended=true has to be explicitly declared and used.

Attributes

- isLogicalTimeSuspended : Boolean = false [1]

Associations

- width : EAValue [0..1] {composite}
- lower : EAValue [0..1] {composite}
- upper : EAValue [0..1] {composite}
- endPointReference : TransitionEvent [0..1]
- startPointReference : TransitionEvent [0..1]
  The event occurance used as a reference point for the start of measuring the time duration.

Constraints

No additional constraints

Semantics

A logical time condition (LTC) is an infinite sequence of time intervals

35.2.2 State (from TemporalConstraint)

Generalizations

- EAElement (from Elements)

Description

A system or component can have a finite set of discrete states. Each state defines a situation where certain value invariant (quantificationInvariant) and/or time invariant (timeInvariant) hold.

A state s is an initial state when isInitState=true.

In the context of system design, a state s can represent one or multiple operation modes when isMode=true; or one or multiple errors in the system when isError=true, or hazards when isHazard=true.

Attributes

- isErrorState : Boolean = false [1]
- isHazard : Boolean = false [1]
- isInitState : Boolean = false [1]
  Indicating an initial state when the value is true.
- isMode : Boolean = false [1]

Associations

- modeDeclaration : Mode [*]
  The operation modes represented by a state (when isMode=true)
- hazardDeclaration : Hazard [*]
  The hazards represented by a state (when isHazard=true).
- quantificationInvariant : Quantification [*]
  The value invariants of a state, i.e. the value conditions that must hold in a state.
- timeInvariant : LogicalTimeCondition [*]
  The time invariants of a state, i.e. the time conditions that must hold in a state.

Constraints

No additional constraints

Semantics

Each state defines a situation where certain value- and/or time-conditions in terms of state invariants hold.

35.2.3 SynchronousTransition (from TemporalConstraint)

Generalizations

- Transition (from TemporalConstraint)

Description

SynchronousTransition denotes a specialization of discrete transitions (Transition) of which the firing can be synchronized by explicit rendezvous events.

Attributes

No additional attributes

Associations

- writeTransitionEvent : TransitionEvent [0..1]
- readTransitionEvent : TransitionEvent [0..1]

Constraints

For behavior constraint descriptions that target application software functions, SynchronousTransition should not be applied.

Semantics

When all the given guard conditions are met, a transition will be fired to respond to the occurrence of an event (which is indicated by the role readEventOccurrences?) or to signal the occurrence of an event (which is indicated by the role writeEventOccurrance!). A transition, when fired, will lead to the

exit of the associated "from" state and an entrance to the associated "to" state, while invoking one or more logical transformations (TransformationOccurrance!) as the effects of the transition.

35.2.4 TemporalConstraint (from TemporalConstraint)

Generalizations

- EAElement (from Elements)

Description

Temporal constraints (TemporalConstraint) provide the language support for capturing the concerns relating to discrete behavior, which emphasizes the dependency that a behavior has in regard to its own history and other behaviors on a timeline. They are useful for precisely defining requirements or design solutions.

A temporal constraint consists of a set of states of discrete behavior, a set of occurrences of discrete events, a set of discrete transitions; and a set of time intervals that constitute the logical time basis of discrete behavior

Attributes

No additional attributes

Associations

- assertion : EAExpression [0..1] {composite}
- transitionEvent : TransitionEvent [*] {composite}
  The events, when occurred, fire the transitions of discrete behaviors.
- timeCondition : LogicalTimeCondition [*] {composite}
- transition : Transition [*] {composite}
  Owned transitions.
- initState : State [1]
  A state s is an initial state when isInitState=true
- state : State [*] {composite}
  Owned states.

Constraints

A Temporal constraint has a single initial state.

Semantics

The definition of temporal constraint is based on a generic definition of automata. That is, a temporal constraint is a tuple of: 1. a set of states of discrete behavior; 2. a set of occurrences of discrete events; 3. a set of discrete transitions; and 4. a set of time intervals that constitute the logical time basis of discrete behavior.

The execution has the following pattern: In one state, read certain parameter, upon certain parameter condition(s) and event occurrence(s), do certain transitions(s) to go to another state. Only one state is active during the operation.

35.2.5 Transition (from TemporalConstraint)

Generalizations

- EAElement (from Elements)

Description

Discrete transitions (Transition) describe the possible switches between discrete states due to the occurrences of discrete events or due to the violations of a state invariant in time or in value quantification.

See also Transition.

Attributes

No additional attributes

Associations

- quantificationGuard : Quantification [*]
  The value guard conditions of a transition.
- effect : TransformationOccurrence [0..1]
  The transformations to be activated when the transition is fired.
- timeGuard : LogicalTimeCondition [*]
  The time guard conditions of a transition.
- to : State [1]
  The target state of the transition.
- from : State [1]
  The source state of the transition.

Constraints

A transition connects one or two states. This means that the from and to roles can be applied to two distinct states or a single state.

Semantics

When all the given guard conditions are met, a transition will be fired. A transition, when fired, will lead to the exit of the associated "from" state and an entrance to the associated "to" state, while invoking one or more logical transformations (TransformationOccurrance!) as the effects of the transition.

35.2.6 TransitionEvent (from TemporalConstraint)

Generalizations

- EAElement (from Elements)
- BehaviorConstraintParameter (from BehaviorDescription)

Description

A transition event denotes the occurrence, i.e. the actual happening, of certain logical, execution specific, and erroneous conditions, that fires the transitions of discrete behavior.

* An occurred logical event (occurredLogicalEvent) denotes a logical condition (e.g. when the measured value of vehicle speed is below 30 km/h) that takes place at a particular time instance and becomes valid in a certain time interval according to the definition of corresponding quantification. Logical events of input or output variables (defined through Attribute) can be communicated through the corresponding ports.

* An occurred execution specific event (occurredExecutionEvent) denotes a distinct form of condition change in system execution at distinct points in time, such as at the triggering of a function, or at the receiving/sending of data from/to ports.

* The occurrence of a fault, a failure, or a hazard (occurredFeatureFlaw, occurredHazardousEvent, or occurredFaultFailure) denotes a distinct form of deviation from nominal behaviors in certain time condition, of which the estimated existences are expressed by the corresponding anomaly or hazard definition.

Attributes

No additional attributes

Associations

- occurredExecutionEvent : Event [*]
- occurredFeatureFlaw : FeatureFlaw [*]
- occurredHazardousEvent : HazardousEvent [*]
- occurredFaultFailure : FaultFailure [*]
  The fault(s)/failure(failures) represented by the parameter condition.
- occurredLogicalEvent : LogicalEvent [*]

Constraints

The set of occurred erroneous events ((occurredFeatureFlaw, occurredHazardousEvent, or occurredAnomaly) is a symmetric set difference of feature flaws (Dependability::FeatureFlaw), system hazards (Dependability::HazardeousEvent, and system faults/failures (ErrorModel::Anomaly) as such concepts only differ in scope or in abstraction level.

Semantics

A transition between two states of discrete behavior can be fired to respond to the occurrence of an event (which is indicated by the role readEventOccurrences?) or to signal the occurrence of an event (which is indicated by the role writeEventOccurrance!).