EAST-ADL
UML Profile Specification

Version M2.1.10

Revision History

Version	Date	Reason
1.0	2008-03-20	Final PU deliverable of the ATESST project.
2.1.8	2010-06-04	ATESST2 UML profile
M2.1.9	2011-08-29	MAENAD UML profile
M2.1.10	2012-06-15	MAENAD UML profile

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.

Table of Contents – Overview

Table of Contents - Complete

Part I Introduction

The purpose of the EAST-ADL language is to capture automotive electrical and electronic (EE) 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 EE system under development, e.g., requirements, behavior, and verification and validation.

By hosting all aspects of the automotive EE 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.

1 Language Formalism

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

Stereotypes and Enumerations and primitive types are documented in a similar way, an abstract element is tagged with {abstract}.

Headers used in this document, for stereotypes:

Description

A general description of the stereotype.

Extensions

The UML concept(s) that is(are) extended. Note that a stereotype may extend UML via specializations, i.e. the reader should also consider following the stereotypes listed under Generalizations.

Generalizations

The stereotypes that this stereotype specializes.

Properties

Example: /direction : FlowDirection (from SysML::PortAndFlows) [1] {readOnly, composite}

/ = derived

direction = the name of the property.

SysML::FlowDirection = the type of the property.

[1] = the multiplicity.

{readOnly}

There are supporting classes that derive values from the underlying UML model for properties that are derived and readOnly.

(Associations)

The properties of the stereotype that are linked to properties in other stereotypes. Documented as Properties, see above.

Semantics

The meaning of the stereotype.

Constraints

The constraints implemented in the profile.

(Constraints in natural language)

Constraints documented in natural language, not necessarily implemented in the profile.

(Notation)

Icons may be serialized within the profile.

2 Scope

The purpose of this specification is to specify EAST-ADL, a modeling language for electronics systems engineering within the automotive domain. Its intent is to specify the language so that electronics systems engineering modelers may learn to apply and use EAST-ADL, modeling tool vendors may implement and support EAST-ADL and both can provide feedback to improve future versions. EAST-ADL reuses a subset of UML2 and of SysML (the UML profile for system engineering) and provides additional extensions to satisfy the automotive domain requirements.

EAST-ADL has been defined in two steps:

1. A domain model (also called metamodel) related to automotive requirements for system engineering is defined, capturing only the domain specific needs of the language, without adding the UML2 details. The basic concepts of UML2 are used for this purpose, such as classes, compositions and associations.

2. Based on the domain metamodel, a UML2 profile for the domain metamodel is then designed. It consists in implementing in terms of UML2 extensions the domain model previously defined. UML2 extensions, stereotypes with properties and constraints, are then gathered within a UML2 profile: the UML2 profile for EAST-ADL.

2.1 Normative references

The following web sites provide normative documents, which contain provisions which, through reference in this text, constitute provisions of this specification.

- UML2.0 and MARTE, www.omg.org
- SysML, www.omgsysml.org
- AUTOSAR, www.autosar.org

3 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 just like their external interfaces through which they can be connected to communicate with each other, by exchanging data or sending messages.

Additionally, data types are defined in this part of the specification for the exchanged data, and for parameters, are. These definitions conditions are specified that hold for the instances during their lifetime at the execution of the system. For the design of systems of arbitrary size and complexity, the possibility of hierarchical structuring of the instances is provided in the language.

EAST-ADL abstraction layers are used to allow reasoning of 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 layer, with the Vehicle Feature Model describing function decomposition in a feature tree view with a focus on external visibility properties and organization in product line.

- Analysis level with FunctionalAnalysisArchitecture build with an abstract functional definition of the system capturing analysis support of what the system shall do, and ensuring relation with features from the Vehicle layer view. There is an n-to-m mapping between VehicleFeature and Feature entities and FunctionalAnalysisArchitecture entities (i.e., one or several functions may realize one or several features).

- Design level with FunctionalDesignArchitecture representing a decomposition of functionalities denoted in the FunctionalAnalysisArchitecture, 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 for example allocation, efficiency, re-use, or supplier concerns. Again, there is an n-to-m mapping between entities of the FunctionalDesignArchitecture and the one of the FunctionalAnalysisArchitecture. Non-transparent infrastructure functionality of the AUTOSAR Basic SW Architecture, such as mode changes and error handling are also represented at the Design Level.

- Implementation level is based on AUTOSAR concepts.

The Hardware Architecture and Environment Model span several abstraction levels. The Hardware Architecture models Electronic Control Units (ECUs), communication links, sensors and actuators and their connections. The Hardware Architecture is considered already at the analysis level because models of sensors, actuators, and early assumptions of hardware may be needed for the Functional Analysis Architecture

The EnvironmentModel 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 of early analysis models, all the way to the implemented embedded system.

4 EAST-ADL extensions for SystemModeling

The SystemModel is the top level container of an EAST-ADL model. It represents the electronics & software of the vehicle, and its environment, and concepts related to the various abstraction level of models used in EAST-ADL. It is mainly based on both concepts: Models and architectures.

VehicleFeatureModel represents the features of the vehicle, i.e. the externally visible properties

The AnalysisArchitecture is the abstract functional description of the vehicle electronics

The DesignArchitecture contains the functional specification and hardware architecture of the vehicle electronics

The Implementation Architecture contains the software architecture and components and the hardware architecture of the vehicle

The word model vs. architecture is chosen rather informally. Architecture is used where this term is often used in practice, and where the construct is a complete (in some sense) reflection of the aspects that it captures. Model is used in other cases.

These models/architectures contain further elements in a hierarchy.. Relations between these elements over the boundaries between the models/architectures are contained in the SystemModel. This is possible because the SystemModel is a specialization of the Context, and is thus able to contain relations. Typical relations are described in the sub-package CoreConstructs (see definition of Relationship, Realization and Satisfy).

4.1 Overview

The SystemModel is the top level container of an EAST-ADL model. It represents the electronics & software of the vehicle, and its environment, and concepts related to the various abstraction level of models used in EAST-ADL. It is mainly based on both concepts: Models and architectures.

4.2 Profile diagrams

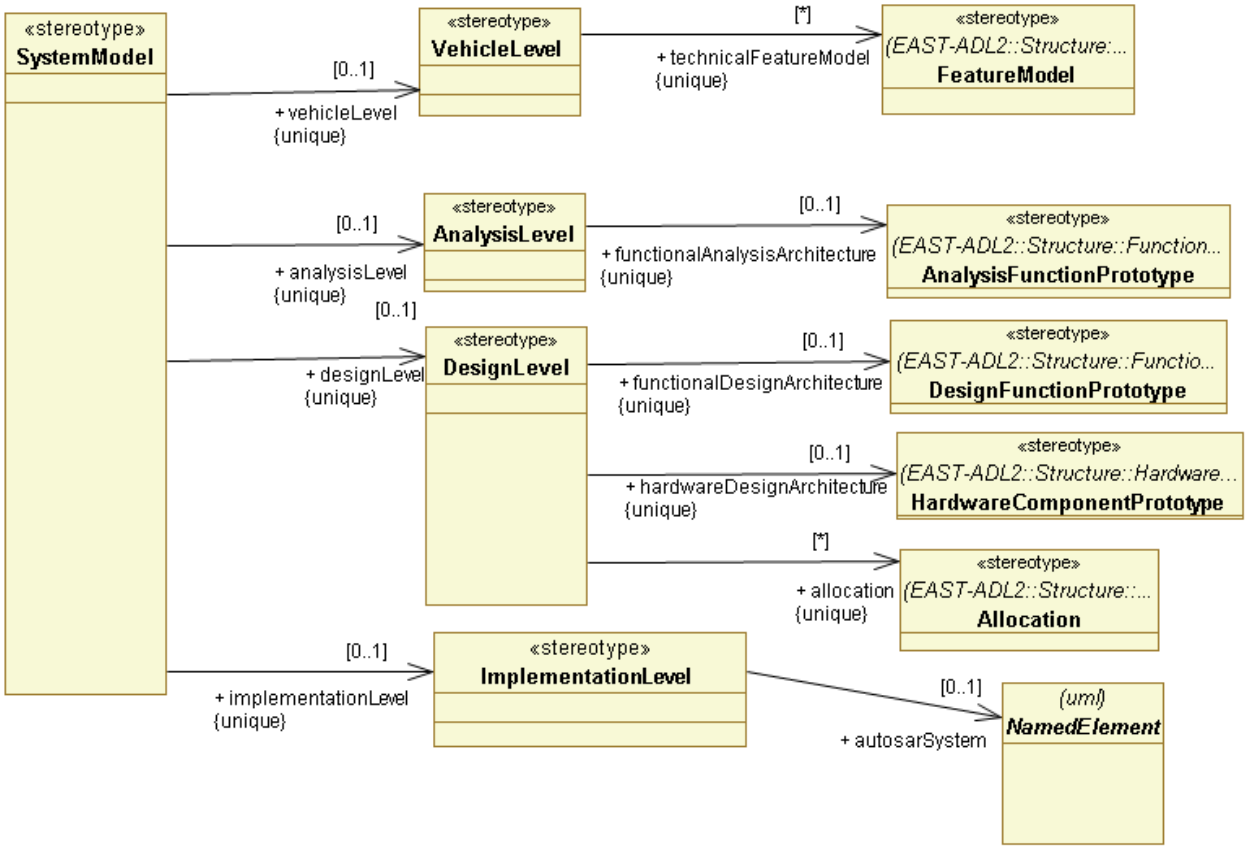

Figure 1. SystemModeling

4.3 Detailed description of UML profile elements

4.3.1 «AnalysisLevel» (from SystemModeling)

Generalizations

- Context

Description

AnalysisLevel represents the vehicle EE system in terms of its abstract functional definition. It includes the functional analysis architecture (FAA) which represents the abstract functional structure.

Extensions

- Class (from UML)
- Package (from UML)

Properties

- functionalAnalysisArchitecture : AnalysisFunctionPrototype [0..1]

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

4.3.2 «DesignLevel» (from SystemModeling)

Generalizations

- Context

Description

DesignLevel represents the vehicle EE 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 Functional Analysis Architecture or a Vehicle level (if no Functional Analysis Architecture 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 EE system. The design architecture includes the Functional Design Architecture 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- 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 Functional Design Architecture parts are typed by FunctionTypes and 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 Functional Design Architecture.

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.

Extensions

- Class (from UML)
- Package (from UML)

Properties

- allocation : Allocation [0..*]
- functionalDesignArchitecture : DesignFunctionPrototype [0..1]
- hardwareDesignArchitecture : HardwareComponentPrototype [0..1]

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

4.3.3 «ImplementationLevel» (from SystemModeling)

Generalizations

- Context

Description

ImplementationLevel represents the software architecture and components, and the hardware architecture of the EE system in the vehicle. The ImplementationLevel is defined by the AUTOSAR System- and SoftwareArchitecture. For example, functions of the Functional Design Architecture 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.

Extensions

- NamedElement (from UML)
- Class (from UML)
- Package (from UML)

Properties

- autosarSystem : NamedElement (from UML) [0..1]

Semantics

No additional semantics

Constraints

No additional constraints

Notation

- Icon: Serialized 

4.3.4 «SystemModel» (from SystemModeling)

Generalizations

- Context

Description

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

Extensions

- Class (from UML)
- Package (from UML)

Properties

- analysisLevel : AnalysisLevel [0..1]
  The AnalysisArchitecture contained in the SystemModel and connected to the EnvironmentModel through ports-connectors
- designLevel : DesignLevel [0..1]

The designArchitecture contained in the SystemModel and connected to the EnvironmentModel through ports-connectors

- implementationLevel : ImplementationLevel [0..1]
  The Implementation Architecture abstraction level.
- vehicleLevel : VehicleLevel [0..1]
  The Vehicle Feature Model contained in the SystemModel.

Semantics

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

Constraints

No additional constraints

4.3.5 «VehicleLevel» (from SystemModeling)

Generalizations

- Context

Description

VehicleLevel represents an arbitrary set of feature models containing only VehicleFeatures.

Extensions

- Package (from UML)
- Class (from UML)

Properties

- technicalFeatureModel : FeatureModel [0..*]

Semantics

The VehicleLevel contains the technical feature models.

Constraints

No additional constraints

Constraints in natural language

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

Notation

- Icon: Serialized 

5 EAST-ADL extensions for FeatureModeling

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.

5.1 Overview

No overview

5.2 Profile diagrams

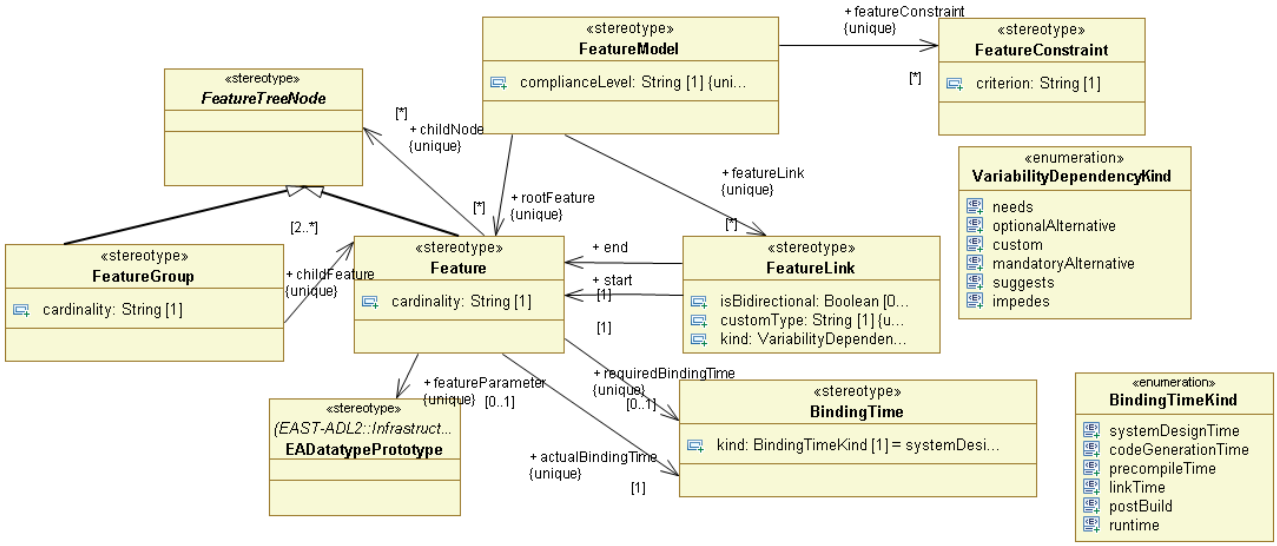

Figure 2. FeatureModeling

5.3 Detailed description of UML profile elements

5.3.1 «BindingTime» (from FeatureModeling)

Generalizations

- EAElement

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 and what the reasons are for a (different) actual binding time.

Extensions

- Class (from UML)

Properties

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

Semantics

No additional semantics

Constraints

No additional constraints

5.3.2 BindingTimeKind (from FeatureModeling)

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 code 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 EE-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

Semantics

No additional semantics

Constraints

No additional constraints

5.3.3 «Feature» (from FeatureModeling)

Generalizations

- FeatureTreeNode
- EAElement

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.

Extensions

- Class (from UML)

Properties

- actualBindingTime : BindingTime [1]
  The actual binding time, independent of the required binding time.
- cardinality : String [1]
  The Cardinality describes for a feature its cardinality. In the context of a feature group it describes the variability behavior of the group (e.g. a cardinality of 1 in a feature group means that one of the child features has to be selected). Cardinalities for features: A cardinality of 0..1 at a feature means that this feature is optional. A cardinality of 1 means that this feature is mandatory and a cardinality of 0..n with n>1 means that this feature may be instantiated more than once in the product to be realized.

  Note that allowing cardinalities >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 a complianceLevel in FeatureModel.
- childNode : FeatureTreeNode [0..*]
- featureParameter : EADatatypePrototype [0..1]
- requiredBindingTime : BindingTime [0..1]
  The required binding time could possibly deviate from the actual binding time but reflects the intended binding time and actual binding time can be later adapted to the required binding time, if surrounding constraints allow a change.

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

5.3.4 «FeatureConstraint» (from FeatureModeling)

Generalizations

- EAElement

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.

Extensions

- Class (from UML)
- Constraint (from UML)

Properties

- criterion : String [1]

Semantics

No additional semantics

Constraints

No additional constraints

5.3.5 «FeatureGroup» (from FeatureModeling)

Generalizations

- FeatureTreeNode

Description

FeatureGroup is a specialization of the FeatureTreeNode, enabling grouping of several Features. It specifies with its cardinality how these grouped features can be combined. For example, a FeatureGroup owning the two Features A and B, with a cardinality of [1] means that A and B are alternative.

Extensions

- Class (from UML)

Properties

- cardinality : String [1]
  The Cardinality describes for a feature group its cardinality. It describes the variability behavior of the group (e.g. a cardinality of 1 in a feature group means that one of the child features has to be selected).
- childFeature : Feature [2..*]

Semantics

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

Constraints

No additional constraints

5.3.6 «FeatureLink» (from FeatureModeling)

Generalizations

- Relationship

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).

Extensions

- Dependency (from UML)
- AssociationClass (from UML)

Properties

- customType : String [1]
  The type of this feature link identified by a String value. The type determines the precise semantics of the relation. There are four predefined types (given a link that starts at feature A and ends at feature B):

  "excludes":

  A and B can never be both selected in a single configuration (always bidirectional).

  "includes":

  if A is selected, then also B must be selected (unidirectional or bidirectional).

  "impedes":

  A and B can usually(!) not be selected in a single configuration, or: you can select A and B but you should have a good reason

  to do so (always bidirectional).

  "suggests":

  if A is selected, then usually(!) also B must be selected, or: you can select A without B but you should have a good reason to do so (unidirectional or bidirectional).

  In addition, each project can decide to use additional link types by defining unique key-words for them. In cases where feature models are shared with third parties (other departments, companies, etc.) a URL name scheme must be used to produce globally unique names, e.g. as for packages in the Java programming language.
- end : Feature [1]
- isBidirectional : Boolean [0..1]
  It must be stated if the link is bidirectional: It exists either unidirectional or bidirectional includes. Also in the case of excludes there exists the unidirectional case (e.g. if the time of exclusion plays a role), whereas the bidirectional case is more common.
- kind : VariabilityDependencyKind [1]
- start : Feature [1]

Semantics

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

Constraints

No additional constraints

Constraints in natural language

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

5.3.7 «FeatureModel» (from FeatureModeling)

Generalizations

- Context

Description

FeatureModel denotes a model owning Features. The FeatureModel can be used to describe variability and commonality of a specified EE-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.

Extensions

- Package (from UML)
- Class (from UML)

Properties

- ~~complianceLevel : String [1]~~
  ~~This specifies that the feature model should comply to a certain, established feature modeling and diagramming technique (such as FODA, Czarnecki 2000, pure:variants).~~
- featureConstraint : FeatureConstraint [0..*]
- featureLink : FeatureLink [0..*]
- rootFeature : Feature [0..*]

Semantics

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

Constraints

No additional constraints

5.3.8 «FeatureTreeNode» (from FeatureModeling) {abstract}

Generalizations

- Context

Description

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

Extensions

No direct extensions

Properties

No additional properties

Semantics

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

Constraints

No additional constraints

5.3.9 VariabilityDependencyKind (from FeatureModeling)

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.

Semantics

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

Constraints

No additional constraints

6 EAST-ADL extensions for VehicleFeatureModeling

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.

6.1 Overview

No overview

6.2 Profile diagrams

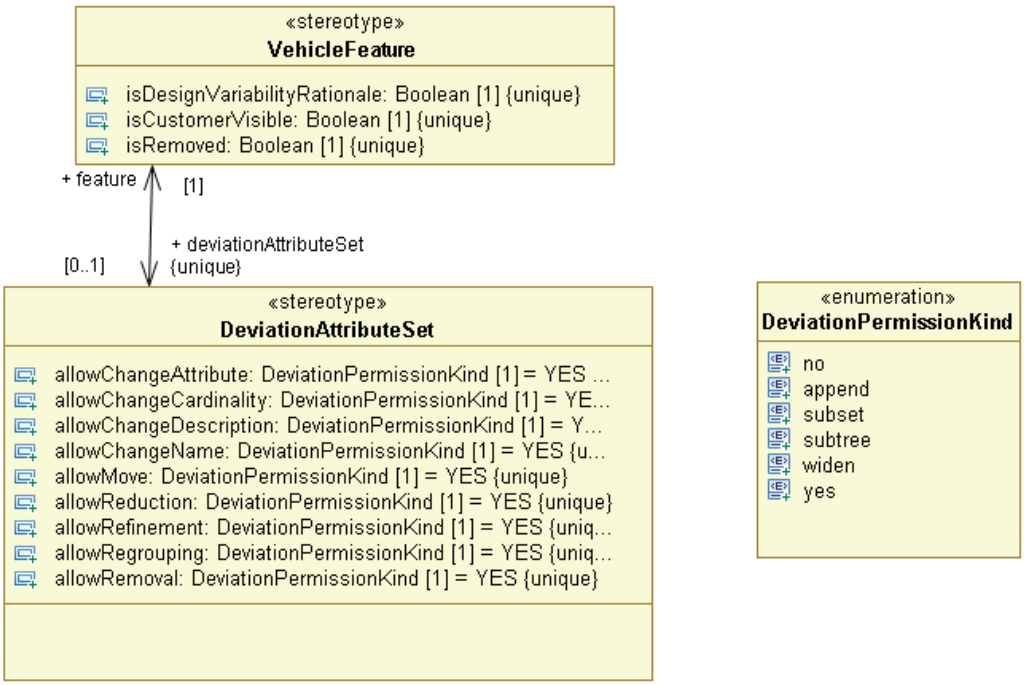

Figure 3. VehicleFeatureModeling

6.3 Detailed description of UML profile elements

6.3.1 «DeviationAttributeSet» (from VehicleFeatureModeling)

Generalizations

- EAElement

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 might be possible.

Extensions

- DataType (from UML)

Properties

- allowChangeAttribute : DeviationPermissionKind = YES [1]
  Allows change of feature attributes.
- allowChangeCardinality : DeviationPermissionKind = YES [1]
  Allows change of feature cardinality (i.e. variabilability of the feature).
- allowChangeDescription : DeviationPermissionKind = YES [1]
  Allows change of the feature description.
- allowChangeName : DeviationPermissionKind = YES [1]
  Allows change of the feature name.
- allowMove : DeviationPermissionKind = YES [1]
  Allows moving of the feature to another place in the feature diagram.
- allowReduction : DeviationPermissionKind = YES [1]
  Reference feature fR has a child without a corresponding referring feature among the children of f.
- allowRefinement : DeviationPermissionKind = YES [1]
  Allows adding of a child feature (without a corresponding feature in the reference model).
- allowRegrouping : DeviationPermissionKind = YES [1]
  Allows to regroup the respective features.
- allowRemoval : DeviationPermissionKind = YES [1]
  Allows the deletion of the feature in the referring model (compared to the reference model).

Associations

- feature : VehicleFeature [1]
  The deviation attributes belong to vehicle features that are part of a reference feature model in the context of multi-level feature models. The attribute can constrain the allowed deviation for the respective referring features.

  See VehicleFeature.deviationAttributeSet

Semantics

No additional semantics

Constraints

No additional constraints

6.3.2 DeviationPermissionKind (from VehicleFeatureModeling)

Generalizations

None

Description

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.

Semantics

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

Constraints

No additional constraints

6.3.3 «VehicleFeature» (from VehicleFeatureModeling)

Generalizations

- Feature

Description

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

Extensions

No direct extensions

Properties

- isCustomerVisible : Boolean [1]
  This attribute describes if the feature is customer visible (in contrast to a feature that is e.g. technically driven).
- isDesignVariabilityRationale : Boolean [1]
  A feature being a designVariabilityRationale is from the point of abstraction no real VFM feature but rather captures a variant coming up on a concrete artifact level that needs to be described on the VFM in order to be configured correctly.

  If true, then isCustomerVisible = false
- isRemoved : Boolean [1]
  This attribute describes if the VFMFeature is removed (but kept in the database for tracking of evolution).

Associations

- deviationAttributeSet : DeviationAttributeSet [0..1]
  The deviation attributes belong to vehicle features that are part of a reference feature model in the context of multi-level feature models. The attribute can constrain the allowed deviation for the respective referring features.

  See DeviationAttributeSet.feature

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.

Constraints

No additional constraints

Constraints in natural language

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

Notation

- Icon: Serialized 

7 EAST-ADL extensions for FunctionModeling

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.

7.1 Overview

No overview

7.2 Profile diagrams

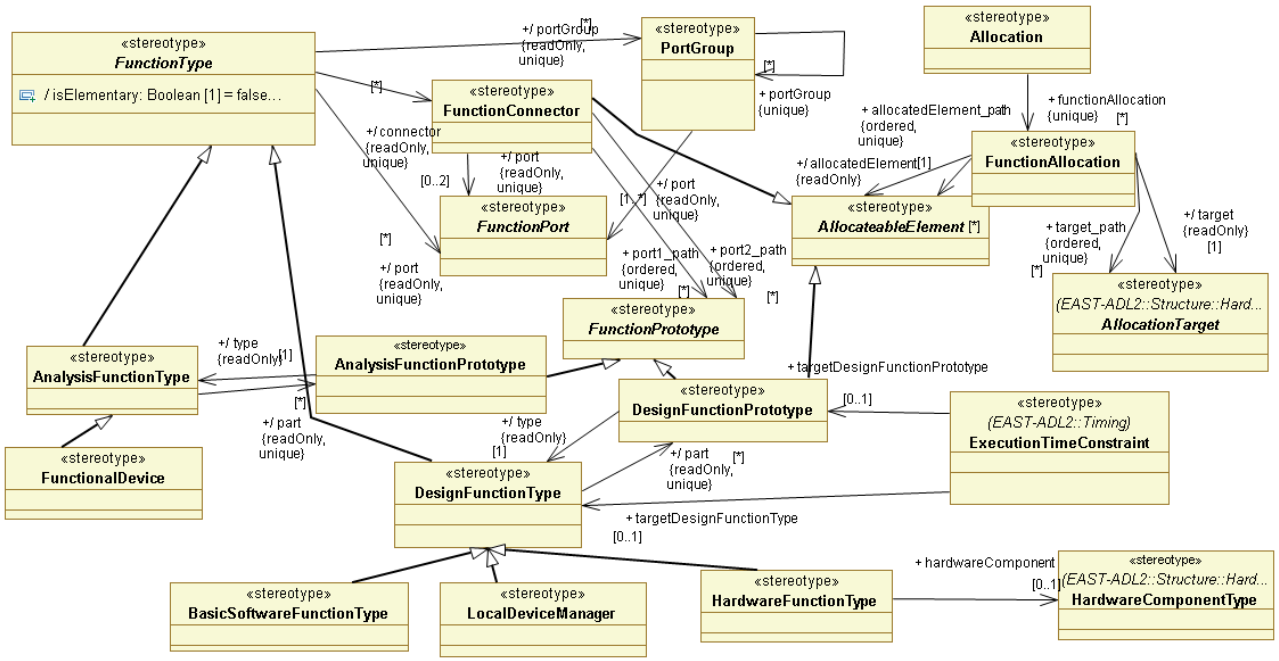

Figure 4. FunctionModeling

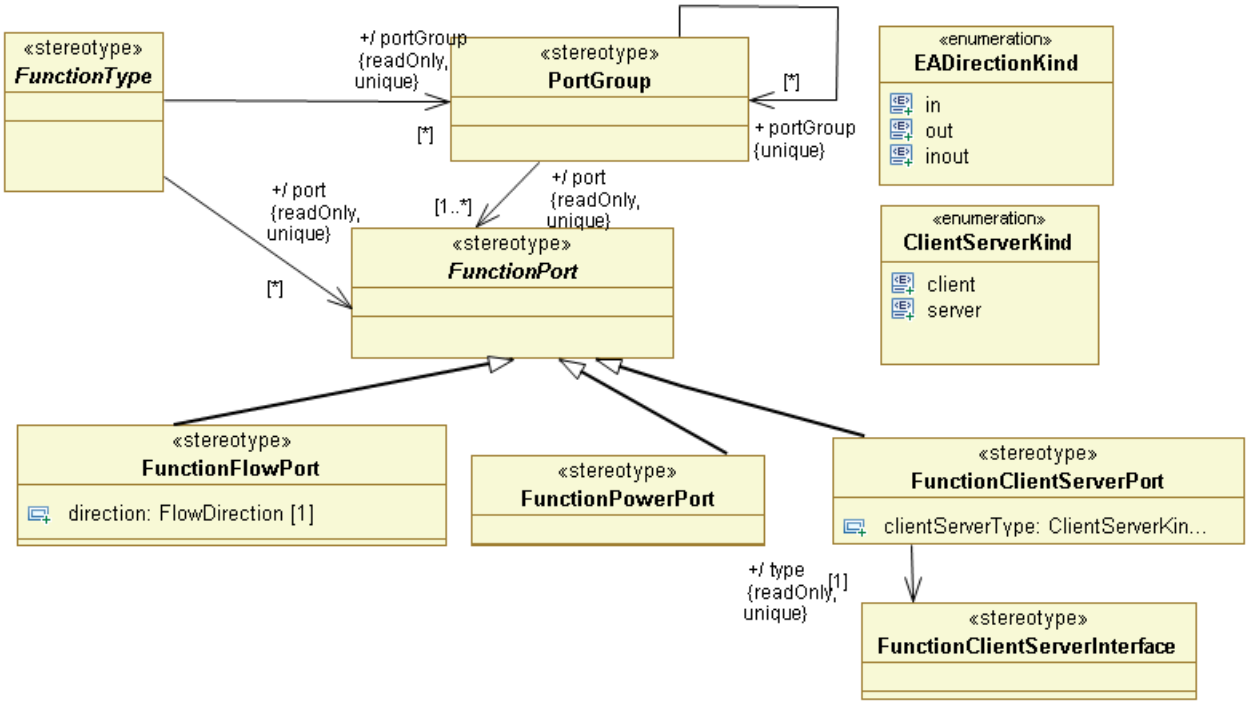

Figure 5. FunctionModeling

7.3 Detailed description of UML profile elements

7.3.1 «AllocateableElement» (from FunctionModeling) {abstract}

Generalizations

None

Description

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

Extensions

No direct extensions

Properties

No additional properties

Semantics

The AllocateableElement abstracts all elements that are allocateable.

Subclasses of the abstract class AllocateableElement add their own semantics.

Constraints

No additional constraints

7.3.2 «Allocation» (from FunctionModeling)

Generalizations

- EAElement

Description

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

Extensions

- Class (from UML)
- Package (from UML)

Properties

- functionAllocation : FunctionAllocation [0..*]

Semantics

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

Constraints

No additional constraints

7.3.3 «AnalysisFunctionPrototype» (from FunctionModeling)

Generalizations

- FunctionPrototype

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.

Extensions

No direct extensions

Properties

- /type : AnalysisFunctionType [1] {readOnly}

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

7.3.4 «AnalysisFunctionType» (from FunctionModeling)

Generalizations

- FunctionType

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 (sub-)AnalysisFunctions. This allows breaking up hierarchically the functionalities provided by the parent AnalysisFunction 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 subAnalysisFunction. If the AnalysisFunction is not decomposed (i.e., if the AnalysisFunction is elementary), then the behavior is describing the behavior of the subAnalysisFunction, which is to be used when building the global behavior of the FunctionalAnalysisArchitecture by composition of the leaf behaviors.

Extensions

No direct extensions

Properties

- /part : AnalysisFunctionPrototype [0..*] {readOnly}

Semantics

The AnalysisFunctionType represents a node in a tree structure corresponding to the functional decomposition of a top level AnalysisFunction. The AnalysisFunction is representing 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.

Constraints

No additional constraints

Constraints in natural language

[1] AnalysisFunctionTypes may only be used on AnalysisLevel.

Notation

- Icon: Serialized 
- Icon: Serialized  component

7.3.5 «BasicSoftwareFunctionType» (from FunctionModeling)

Generalizations

- DesignFunctionType

Description

The BasicSoftwareFunctionType is an abstraction of middleware functionality.

Extensions

No direct extensions

Properties

No additional properties

Semantics

The BasicSoftwareFunctionType is an abstraction of the middleware.

Constraints

No additional constraints

7.3.6 ClientServerKind (from FunctionModeling)

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

Semantics

The ClientServerKind is an enumeration with the two literals "client" and "server".

Constraints

No additional constraints

7.3.7 «DesignFunctionPrototype» (from FunctionModeling)

Generalizations

- FunctionPrototype
- AllocateableElement

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 .

Extensions

No direct extensions

Properties

- /type : DesignFunctionType [1] {readOnly}

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

7.3.8 «DesignFunctionType» (from FunctionModeling)

Generalizations

- FunctionType

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 from the type-prototype pattern used by AUTOSAR.

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

Furthermore, a DesignFunction may be decomposed into (sub-)DesignFunctions. This allows breaking up hierarchically the functionalities provided by the parent DesignFunction 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).

Extensions

No direct extensions

Properties

- /part : DesignFunctionPrototype [0..*] {readOnly}

Semantics

The DesignFunctionType represents a node in a tree structure corresponding to the functional decomposition of a top level DesignFunction. The DesignFunction is representing 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.

Constraints

No additional constraints

Constraints in natural language

[1] DesignFunctionTypes may only be used on DesignLevel.

Notation

- Icon: Serialized 
- Icon: Serialized  component

7.3.9 EADirectionKind (from FunctionModeling)

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

Semantics

The EADirectionKind is an enumeration with the three literals "in", "out", and "inout".

Constraints

No additional constraints

7.3.10 «FunctionalDevice» (from FunctionModeling)

Generalizations

- AnalysisFunctionType

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). 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 between LocalDeviceManagers and Sensors/Actuators that are represented by the FunctionalDevice.

Extensions

No direct extensions

Properties

No additional properties

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).

Constraints

No additional constraints

Constraints in natural language

No additional constraints.

Notation

- Icon: Serialized 

7.3.11 «FunctionAllocation» (from FunctionModeling)

Generalizations

- EAElement

Description

FunctionAllocation represents an allocation constraint binding an AllocateableElement on an AllocationTarget.

The same constraint could be expressed in a textual design constraint.

Extensions

- Dependency (from UML)

Properties

- /allocatedElement : AllocateableElement [1] {readOnly}
- allocatedElement_path : AllocateableElement [0..*] {ordered}
- /target : AllocationTarget [1] {readOnly}
  The ECU where the functionality must be allocated.
- target_path : AllocationTarget [0..*] {ordered}

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.

Constraints

No additional constraints

Notation

- Icon: Serialized

7.3.12 «FunctionClientServerInterface» (from FunctionModeling)

Generalizations

- EAElement

Description

The FunctionClientServerInterface is used to specify the operations in FunctionClientServerPorts.

Extensions

- Interface (from UML)

Properties

- /operation : Operation [0..*] {readOnly}

Semantics

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

Constraints

No additional constraints

7.3.13 «FunctionClientServerPort» (from FunctionModeling)

Generalizations

- FunctionPort

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.

Extensions

- Port (from UML)

Properties

- clientServerType : ClientServerKind [1]
- /type : FunctionClientServerInterface [1] {readOnly}
  The interface of this port.

  {derived from UML::TypedElement::type}

Semantics

The FunctionClientServerPort is a FunctionPort for client-server interaction.

FunctionClientServerPorts are single buffer overwrite and nonconsumable.

Constraints

No additional constraints

Constraints in natural language

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

Notation

- Icon: Serialized 

7.3.14 «FunctionConnector» (from FunctionModeling)

Generalizations

- EAElement
- AllocateableElement

Description

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

Extensions

- Connector (from UML)

Properties

- /port : FunctionPort [0..2] {readOnly}
  The ports that are connected by this connector.

  {derived from UML::Connector::end}
- port1_path : FunctionPrototype [0..*] {ordered}
- port2_path : FunctionPrototype [0..*] {ordered}

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 occurrence of the FunctionType that specifies the occurrence of the FunctionPrototype 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.

Constraints

No additional constraints

Constraints in natural language

[1] Can connect two FunctionFlowPorts of different direction 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 kind 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.

7.3.15 «FunctionFlowPort» (from FunctionModeling)

Generalizations

- FunctionPort
- FlowPort (from SysML::PortAndFlows)

Description

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

Extensions

No direct extensions

Properties

- direction : FlowDirection (from SysML::PortAndFlows) [1]
  Derived, the value is equal to SysML::PortAndFlows::FlowDirection::inout
- /type : EADatatype [1] {readOnly}

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.

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.

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.

Constraints

No additional constraints

Notation

- Icon: Serialized 

7.3.16 «FunctionPort» (from FunctionModeling) {abstract}

Generalizations

- EAElement

Description

The ports conserve variables for component interaction.

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

7.3.17 «FunctionPowerPort» (from FunctionModeling)

Generalizations

- FunctionPort

Description

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

Extensions

- Port (from UML)

Properties

- /type : CompositeDatatype [1] {readOnly}

Semantics

The FunctionPowerPort conserves physical variables in a dynamic process.

The typing Datatype owns two datatypePrototypes 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.

Constraints

No additional constraints

Constraints in natural language

[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.

Notation

- Icon: Serialized 

7.3.18 «FunctionPrototype» (from FunctionModeling) {abstract}

Generalizations

- EAElement

Description

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

The FunctionPrototype is typed by a FunctionType.

FunctionTrigger in the Behavior package is associated to a FunctionPrototype.

Extensions

- Property (from UML)

Properties

No additional properties

Semantics

The FunctionPrototype represents an occurrence of the FunctionType that types it.

Constraints

No additional constraints

Notation

- Icon: Serialized 

7.3.19 «FunctionType» (from FunctionModeling) {abstract}

Generalizations

- Context
- Block (from SysML::Blocks)

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.

Extensions

No direct extensions

Properties

- /connector : FunctionConnector [0..*] {readOnly}
- /isElementary : Boolean = false [1] {readOnly}
  True, when this type does not have any parts.

  Derived from size of UML::StructuredClassifier::ownedConnector and UML::EncapsulatedClassifier::ownedPort
- /port : FunctionPort [0..*] {readOnly}
  Owned in- and out-flow ports.

  {derived from UML::EncapsulatedClassifier::ownedPort}
- /portGroup : PortGroup [0..*] {readOnly}
  Grouping of ports owned by this element.

  {derived from UML::Class::nestedClassifier}

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).

Constraints

No additional constraints

Constraints in natural language

[1] Elementary FunctionTypes shall not have parts.

Notation

- Icon: Serialized  component
- Icon: Serialized  elementary

7.3.20 «HardwareFunctionType» (from FunctionModeling)

Generalizations

- DesignFunctionType

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 DesignFunctionPrototype is typically the end of the ClampConnector on DesignLevel.

Prototypes typed by HardwareComponentType may be allocated to HardwareComponents in which case the HardwareFunctionType must match the HardwareFunctionType of the target HardwareComponent.

DesignFunctionPrototypes typed by HardwareFunctionType may be allocated to HardwareComponents in which case the HardwareFunctionType must match the HardwareFunctionType of the target HardwareComponent.

Extensions

No direct extensions

Properties

- hardwareComponent : HardwareComponentType [0..1]

Semantics

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

HardwareFunctions can be allocated to Sensors or Actuators, i.e. the interfacing element to the plant model.

Constraints

No additional constraints

Constraints in natural language

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

7.3.21 «LocalDeviceManager» (from FunctionModeling)

Generalizations

- DesignFunctionType

Description

The LocalDeviceManager represents a DesignFunction that act as a manager or functional interface to Sensors, Actuators and other devices. It is responsible fort 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.

Extensions

No direct extensions

Properties

No additional properties

Semantics

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

Constraints

No additional constraints

Constraints in natural language

[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 may only interface either Sensors or Actuators.

[3] A LocalDeviceManager shall interface BSWFunctions and DesignFunctions.

Notation

- Icon: Serialized 

7.3.22 «Operation» (from FunctionModeling)

Generalizations

- EAElement

Description

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

Extensions

- Operation (from UML)

Properties

- /argument : EADatatypePrototype [0..*] {ordered, readOnly}
- /return : EADatatypePrototype [0..1] {readOnly}

Semantics

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

Constraints

No additional constraints

7.3.23 «PortGroup» (from FunctionModeling)

Generalizations

- EAElement

Description

The PortGroup metaclass is used to collapse several ports to one. All ports that are part of a port group are graphically represented as a single port. Connectors connected to ports of a port group pair are graphically collapsed to a single line.

The PortGroup has no semantic meaning except that it makes graphical representation of the connected ports easier to read, and provides a means to logically organize several ports to one group.

Connectors are still connected to the contained ports, but tool support may simplify connections by allowing semi-automatic or automatic connection to all ports of a port group.

Extensions

- Class (from UML)
- Port (from UML)

Properties

- /port : FunctionPort [1..*] {readOnly}
  The grouped ports.

  {derived from UML::EncapsulatedClassifier::ownedPort} when this stereotype is applied on a Class. When the stereotype is applied on a Port the value is derived from the ports in the type.
- portGroup : PortGroup [0..*]

Semantics

The PortGroup provides a means to organize ports and connectors. It does not add semantics. In the model, the ports contained in the port group are connected as individual ports.

Constraints

No additional constraints

Notation

- Icon: Serialized 

8 EAST-ADL extensions for HardwareModeling

The package HardwareModeling contains the elements to model physical entities of the embedded EE system. These elements allow capturing the hardware 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.

8.1 Overview

No overview

8.2 Profile diagrams

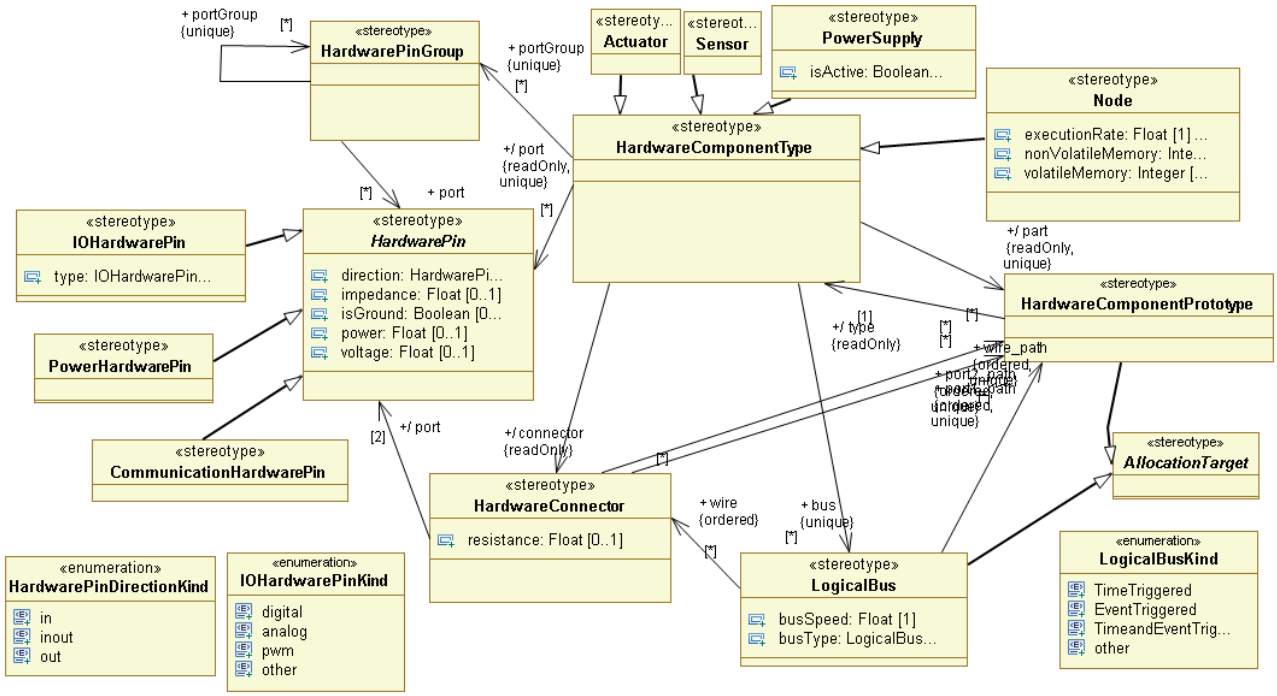

Figure 6. HardwareModeling

8.3 Detailed description of UML profile elements

8.3.1 «Actuator» (from HardwareModeling)

Generalizations

- HardwareComponentType

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.

Extensions

No direct extensions

Properties

No additional properties

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

8.3.2 «AllocationTarget» (from HardwareModeling) {abstract}

Generalizations

- EAElement

Description

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

Extensions

No direct extensions

Properties

No additional properties

Semantics

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

Constraints

No additional constraints

8.3.3 «CommunicationHardwarePin» (from HardwareModeling)

Generalizations

- HardwarePin

Description

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

Extensions

No direct extensions

Properties

No additional properties

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.

Constraints

No additional constraints

Notation

- Icon: Serialized 

8.3.4 «HardwareComponentPrototype» (from HardwareModeling)

Generalizations

- AllocationTarget
- EAElement

Description

Appear as parts 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.

Extensions

- Property (from UML)

Properties

- /type : HardwareComponentType [1] {readOnly}
  The type of the HWElement.

  {derived from UML::TypedElement::type}

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

8.3.5 «HardwareComponentType» (from HardwareModeling)

Generalizations

- Context

Description

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

Extensions

- Class (from UML)

Properties

- bus : LogicalBus [0..*]
- /connector : HardwareConnector [0..*] {readOnly}
  The HWConnectors.

  {derived from UML::StructuredClassifier::ownedConnector}
- /part : HardwareComponentPrototype [0..*] {readOnly}
  The HWElementPrototypes.

  {derived from UML::Classifier::attribute}
- /port : HardwarePin [0..*] {readOnly}
  The Ports.

  {derived from UML::EncapsulatedClassifier::ownedPort}
- portGroup : HardwarePinGroup [0..*]

Semantics

The HardwareElementType 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 an HWFunctionType referencing the HardwareElementType. This is typically connected through its ports to the environment model to participate in the end-to-end behavioral definition of a function.

Constraints

No additional constraints

8.3.6 «HardwareConnector» (from HardwareModeling)

Generalizations

- EAElement

Description

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

Extensions

- Connector (from UML)

Properties

- /port : HardwarePin [2]
- port1_path : HardwareComponentPrototype [0..*] {ordered}
- port2_path : HardwareComponentPrototype [0..*] {ordered}
- ~~resistance : Float [0..1]~~
  ~~The resistance of the HardwareConnector in Ohms.~~

Semantics

The connector joins the two referenced ports electrically, with a resistance defined by the resistance attribute.

Constraints

No additional constraints

8.3.7 «HardwarePin» (from HardwareModeling) {abstract}

Generalizations

- EAElement

Description

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

Semantics

Hardware pin represents an electrical connection point.

Extensions

- Port (from UML)

Properties

- direction : EADirectionKind~~HardwarePinDirectionKind~~ [1]
  The direction of current through the pin.
- impedance : Float [0..1]
  The internal impedance in Ohms to ground of the component as seen through this pin.
- isGround : Boolean [0..1]
  Indicates that the pin is connected to ground.
- power : Float [0..1]
  The maximal power in watts that can be provided by this pin or that is consumed.
- voltage : Float [0..1]
  The maximal voltage in Volts provided by the pin. Shall not be defined if isGround=TRUE.

Semantics

No additional semantics

Constraints

No additional constraints

~~8.3.8 HardwarePinDirectionKind (from HardwareModeling)~~

~~**Generalizations**~~

~~None~~

~~**Description**~~

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

~~**Enumeration Literals**~~

- ~~in~~
- ~~inout~~
- ~~out~~

~~**Semantics**~~

~~The HardwarePinDirectionKind is an enumeration with the three literals "in", "out", and "inout".~~

~~**Constraints**~~

~~No additional constraints~~

~~8.3.9~~8.3.8 «HardwarePinGroup» (from HardwareModeling)

Generalizations

- EAElement

Description

The HardwarePinGroup provides means to organize hardware pins to improve readability of the component interface and connectors between components. Tools may show the set of ports in the pin group as a single pin, join connectors that go between pins in pin groups to a single line.

Extensions

- Port (from UML)
- Class (from UML)

Properties

- port : HardwarePin [0..*]
- portGroup : HardwarePinGroup [0..*]

Semantics

A HardwarePinGroup has no semantics, but is only a grouping mechanism that may affect visualization and port operations in tools.

Constraints

No additional constraints

~~8.3.10~~8.3.9 «IOHardwarePin» (from HardwareModeling)

Generalizations

- HardwarePin

Description

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

Extensions

No direct extensions

Properties

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

Semantics

The IOHardwarePin represents an electrical pin or connection point.

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~8.3.11~~8.3.10 IOHardwarePinKind (from HardwareModeling)

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.

Semantics

No additional semantics

Constraints

No additional constraints

~~8.3.12~~8.3.11 «LogicalBus» (from HardwareModeling)

Generalizations

- AllocationTarget

Description

The LogicalBus represents logical communication channels. It serves as an allocation target for connectors, i.e. the data exchanged between functions in the FunctionalDesignArchitecture.

Extensions

- Class (from UML)

Properties

- busSpeed : Float [1]
- busType : LogicalBusKind [1]
- wire : HardwareConnector [0..*] {ordered}
- wire_path : HardwareComponentPrototype [0..*] {ordered}

Semantics

The LogicalBus represents a logical connection that carries data from any sender to all receivers. Senders and receivers are identified by the wires of the LogicalBus, i.e. the associated HardwareConnectors. The available busSpeed represents the maximum amount of useful data that can be carried. The busSpeed has already deducted speed reduction resulting from frame overhead, timing effects, etc.

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~8.3.13~~8.3.12 LogicalBusKind (from HardwareModeling)

Generalizations

None

Description

LogicalBusKind 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

Semantics

No additional semantics

Constraints

No additional constraints

~~8.3.14~~8.3.13 «Node» (from HardwareModeling)

Generalizations

- HardwareComponentType

Description

Node represents the computer nodes of the embedded EE 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-single core ECU is represented, it is sufficient to have a single, non-hierarchical Node.

Extensions

No direct extensions

Properties

- executionRate : Float = 1 [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.

- nonVolatileMemory : Integer [1]
  The size in Bytes of the Node's Non-Volatile memory (ROM, NRAM, EPROM, etc .
- volatileMemory : Integer [0..1]
  The size in Bytes of the Node's Volatile memory (RAM)

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 that is processed 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.

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~8.3.15~~8.3.14 «PowerHardwarePin» (from HardwareModeling)

Generalizations

- HardwarePin

Description

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

Extensions

No direct extensions

Properties

No additional properties

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~8.3.16~~8.3.15 «PowerSupply» (from HardwareModeling)

Generalizations

- HardwareComponentType

Description

PowerSupply represents a hardware element that supplies power.

Extensions

No direct extensions

Properties

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

Semantics

PowerSupply denotes a power source that may be active (e.g., a battery) or passive (main relay).

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~8.3.17~~8.3.16 «Sensor» (from HardwareModeling)

Generalizations

- HardwareComponentType

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.

Extensions

No direct extensions

Properties

No additional properties

Semantics

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

Constraints

No additional constraints

Notation

- Icon: Serialized 

9 EAST-ADL extensions for Environment

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.

9.1 Overview

No overview

9.2 Profile diagrams

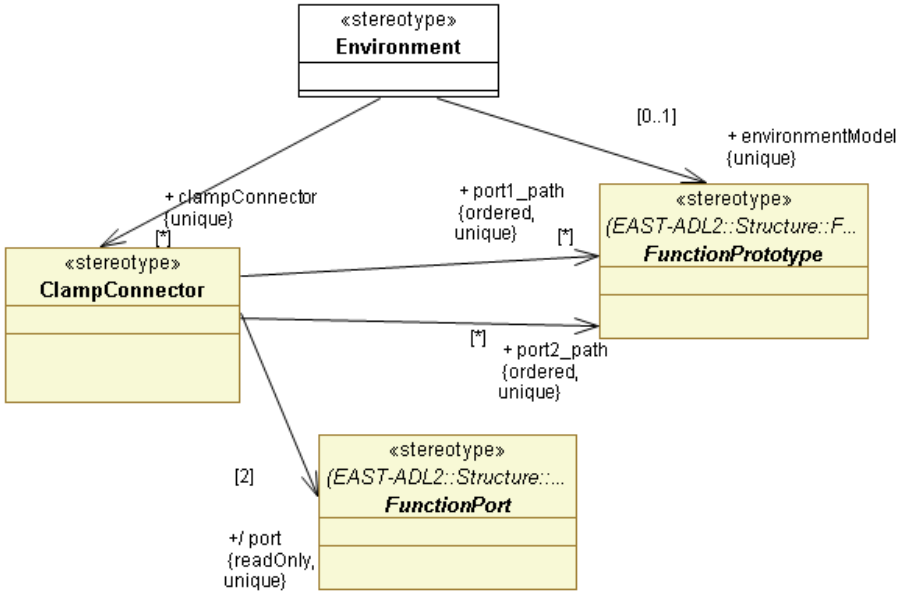

Figure 7. Environment

9.3 Detailed description of UML profile elements

9.3.1 «ClampConnector» (from Environment)

Generalizations

- EAElement

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.

Extensions

- Connector (from UML)

Properties

- /port : FunctionPort [2] {readOnly}
- port1_path : FunctionPrototype [0..*] {ordered}
- port2_path : FunctionPrototype [0..*] {ordered}

Semantics

No additional semantics

Constraints

No additional constraints

Constraints in natural language

[1] Can connect two FunctionFlowPorts of different direction.

[2] Can connect two ClientServerPorts of different kind.

[3] Can connect two FunctionFlowPorts with direction inout.

[4] Cannot connect ports in the same SystemModel.

9.3.2 «Environment» (from Environment)

Generalizations

- Context

Description

The collection of the environment functional description. 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.

Extensions

- Class (from UML)
- Package (from UML)

Properties

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

Semantics

No additional semantics

Constraints

No additional constraints

Notation

- Icon: Serialized 

Part III Behavioral Constructs

10 EAST-ADL extensions for Behavior

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 fresher data arriving on ports. The function then performs some calculation and finally outputs some data on the output ports. The execution is non-concurrent: only one behavior is active at any point in time and not preemptable.

A FunctionBehavior in EAST-ADL is mainly a reference point to some description provided else where in a tool-dependent format, as depicted in the Diagram for FunctionBehavior below. This enables to re-use current behavior descriptions contained in the tools currently used by automotive companies and suppliers. Given that, requirement and traceability information can be provided for behavior in relation to the rest of the EAST-ADL model. A list of typical tool format 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 behaviors can be used. Yet it shall 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 first following Diagram of the behavior of a function) relate to the organization of the triggering of behaviors 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.

The triggering of behavior itself, defined by 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 constraint that are defined in separate sections of the language (see Events, Time and TimingConstraints sections).

10.1 Overview

No overview

10.2 Profile diagrams

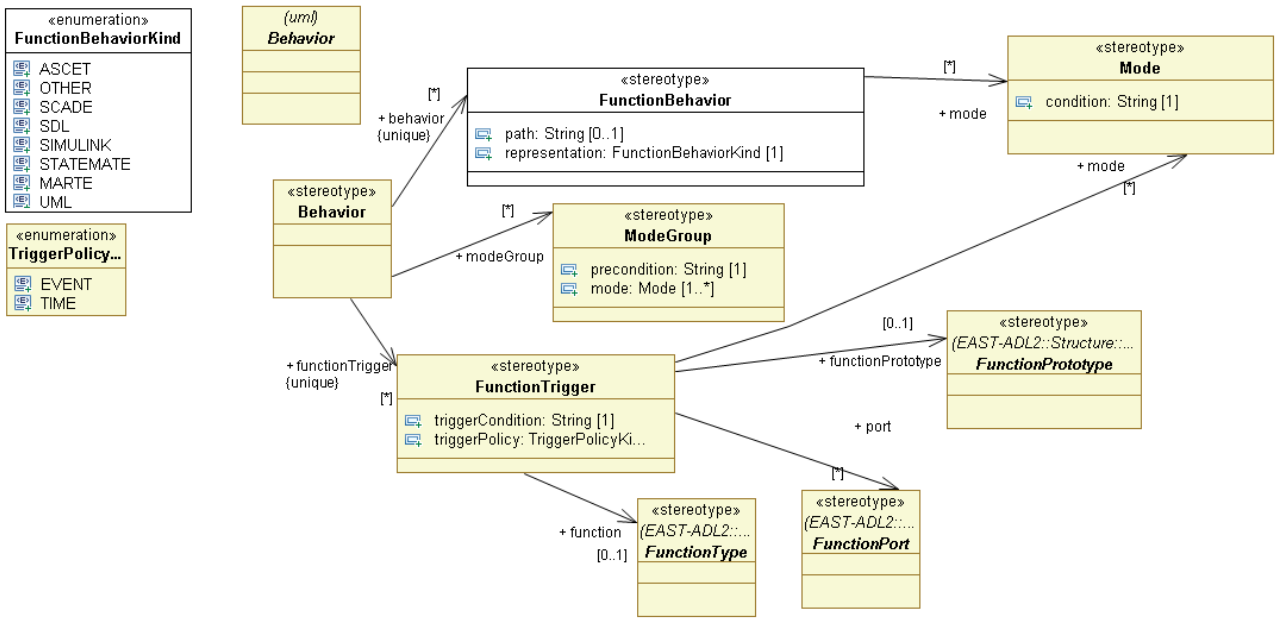

Figure 8. Behavior

10.3 Detailed description of UML profile elements

10.3.1 «Behavior» (from Behavior)

Generalizations

- Context

Description

Behavior is a container of FunctionBehaviors, it enables to regroup 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 done across the EAST-ADL abstraction levels.

Extensions

- Class (from UML)
- Package (from UML)

Properties

- behavior : FunctionBehavior [0..*]
- functionTrigger : FunctionTrigger [0..*]
- modeGroup : ModeGroup [0..*]

Semantics

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

Constraints

No additional constraints

10.3.2 «FunctionBehavior» (from Behavior)

Generalizations

- Context

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 with 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 that 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 be referring 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, etc. must prevent two behaviors being potentially active.

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 in parts, the behavior is a specification for the composed behavior of the FunctionType.

Extensions

- Behavior (from UML)

Properties

- function : FunctionType [0..1]
- mode : Mode [0..*]
- path : String [0..1]
  The path to the file or model entity containing the ExternalBehavior
- representation : FunctionBehaviorKind [1]
  The type of behavior that the ExternalBehavior represents.

Semantics

Though the representation provided to a FunctionBehavior follows the semantics of the behavioral representation used (for instance SIMULINK, ASCET, etc.). Externally, in relation to the EAST-ADL model, however, 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 specific and considered part of the execution of the FunctionBehavior.

Constraints

No additional constraints

Notation

- Icon: Serialized

10.3.3 FunctionBehaviorKind (from Behavior)

Generalizations

None

Description

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

Enumeration Literals

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

Semantics

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

It shall 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.

Constraints

No additional constraints

10.3.4 «FunctionTrigger» (from Behavior)

Generalizations

- EAElement

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, one or several ports of the Function triggers the function, i.e. activates the function. In both cases, a triggerCondition is provided in the form of a Boolean expression that must evaluate to true for the function to execute. The triggerCondition syntax and grammar is unspecified.

In addition a FunctionTrigger may refer to a list of Modes in which the trigger will be considered as potentially active. Because of FunctionBehaviors may also refer to Modes, it is thus possible to arrange various function configurations for which different sets of triggers and behaviors are active. And this, at various level 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.

Extensions

- Class (from UML)

Properties

- function : FunctionType [0..1]
- functionPrototype : FunctionPrototype [0..1]
- mode : Mode [0..*]
- port : FunctionPort [0..*]
- triggerCondition : String [1]
  An OCL expression that allows release of the FunctionType only if it evaluates to TRUE.
- triggerPolicy : TriggerPolicyKind [1]
  Defines whether time or trigger events on ports makes the Function execute

Semantics

Association Mode defines in which modes the trigger is active

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.

Constraints

No additional constraints

Constraints in natural language

[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 and at least one of them shall trigger the function

Notation

- Icon: Serialized 

10.3.5 «Mode» (from Behavior)

Generalizations

- EAElement

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 to logically organize a set of triggers and behaviors over a set of functions. Modes are both referred to by FunctionTriggers and FunctionBehaviors, thus capturing this organization (see FunctionTrigger and FunctionBehavior).

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

Extensions

- Class (from UML)

Properties

- condition : String [1]

Semantics

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

Constraints

No additional constraints

10.3.6 «ModeGroup» (from Behavior)

Generalizations

- TraceableSpecification

Description

ModeGroups serve as container of Modes. The set of 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.

Extensions

- Class (from UML)

Properties

- mode : Mode [1..*]
- precondition : String [1]

Semantics

No additional semantics

Constraints

No additional constraints

10.3.7 TriggerPolicyKind (from Behavior)

Generalizations

None

Description

TriggerPolicyKind represents an enumeration for triggering policies.

Enumeration Literals

- EVENT
  Triggering by event.
- TIME
  Triggering by time.

Semantics

The TriggerPolicyKind contains EVENT and TIME as possible triggering policies.

Constraints

No additional constraints

Part IV Variability

11 EAST-ADL extensions for Variability

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.

11.1 Overview

No overview

11.2 Profile diagrams

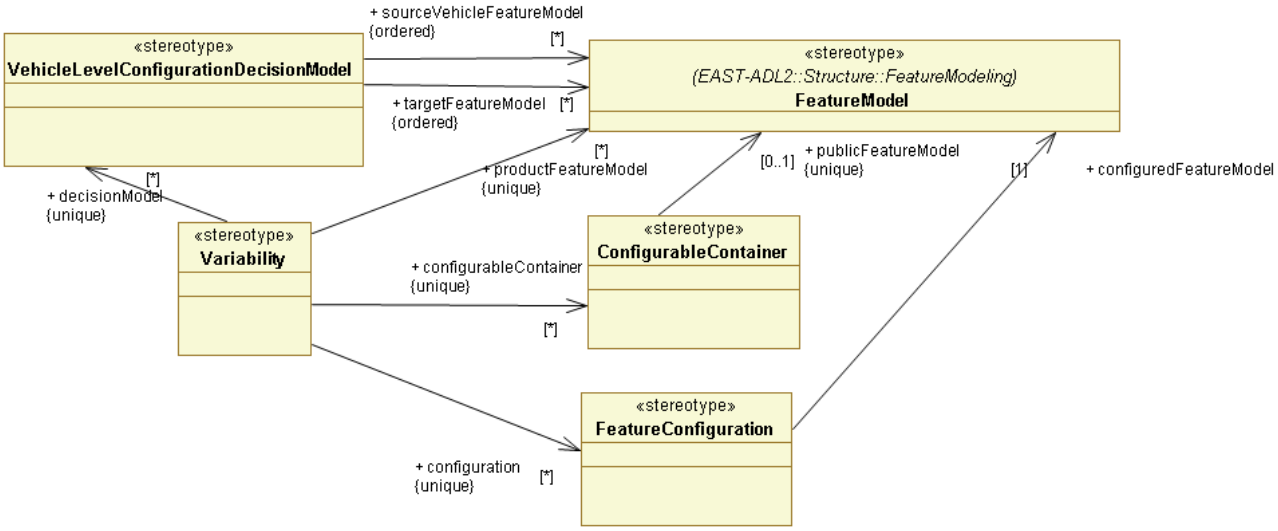

Figure 9. Variability

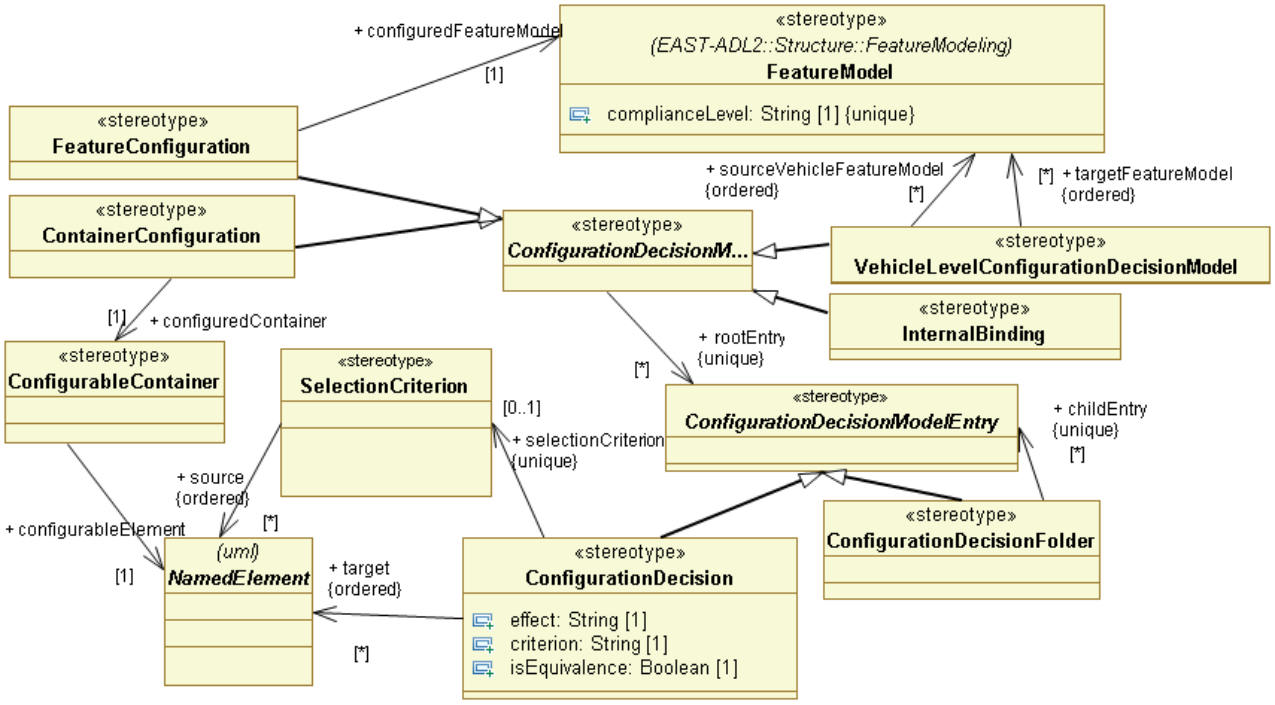

Figure 10. Variability2

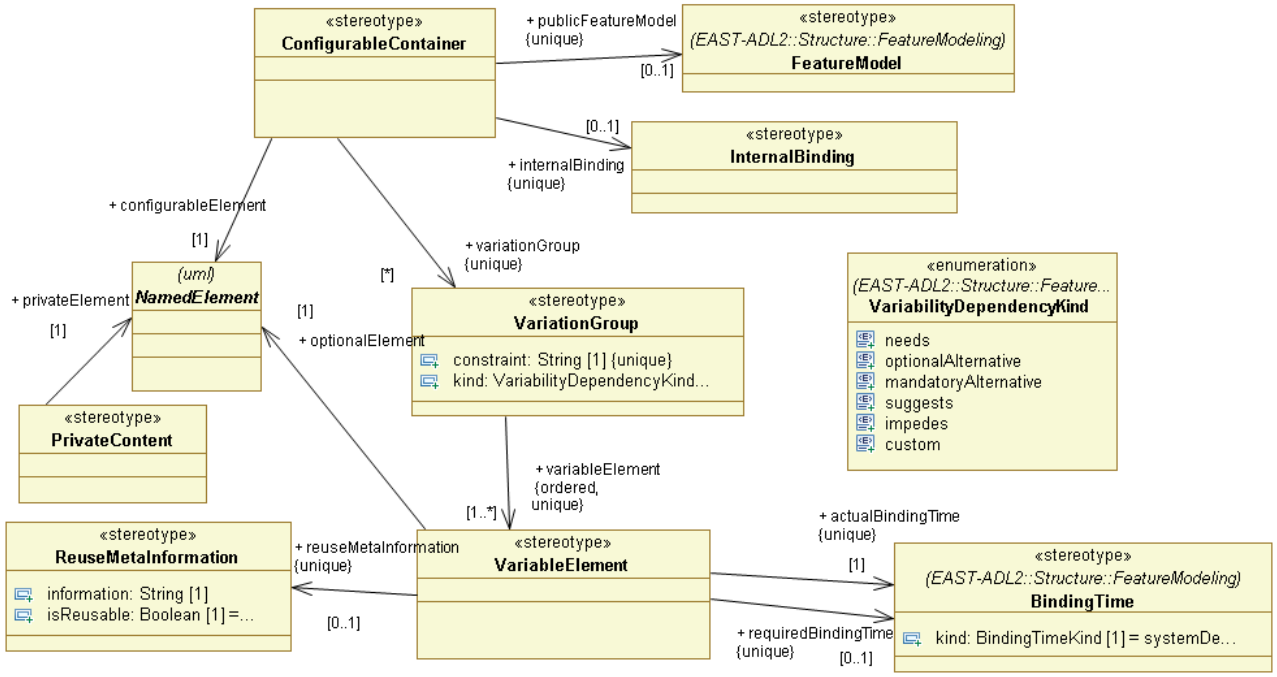

Figure 11. Variability3

11.3 Detailed description of UML profile elements

11.3.1 «ConfigurableContainer» (from Variability)

Generalizations

- EAElement

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".

Extensions

- Class (from UML)
- Package (from UML)
- NamedElement (from UML)

Properties

- configurableElement : NamedElement (from UML) [1]
- internalBinding : InternalBinding [0..1]
  The PDM of the configurable container.
- publicFeatureModel : FeatureModel [0..1]
  The local feature model of the configurable container.
- variationGroup : VariationGroup [0..*]
  The variation groups that define certain constraints between this VariableContainer's variable elements.

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.

Constraints

No additional constraints

Constraints in natural language

[1] Identifies one FunctionType or one HardwareComponentType.

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

11.3.2 «ConfigurationDecision» (from Variability)

Generalizations

- ConfigurationDecisionModelEntry

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.) in dependency of 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[+]".

Extensions

- NamedElement (from UML)

Properties

- criterion : String [1]
  The inclusionCriterion gives the criterion to select the respective products (e.g. Northern American cars).
- effect : String [1]
  The rationale gives the reason for the specified product decision, especially for the inclusion criterion and the selection of included and excluded features.
- isEquivalence : Boolean [1]
  Means that the included and excluded features are selected if and only if the specified inclusion criterion holds.
- selectionCriterion : SelectionCriterion [0..1]
- target : NamedElement (from UML) [0..*] {ordered}

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.

Constraints

No additional constraints

11.3.3 «ConfigurationDecisionFolder» (from Variability)

Generalizations

- ConfigurationDecisionModelEntry

Description

ConfigurationDecisionFolder represents a grouping for ConfigurationDecisions.

Extensions

No direct extensions

Properties

- childEntry : ConfigurationDecisionModelEntry [0..*]

Semantics

ConfigurationDecisionFolder is a grouping entity for ConfigurationDecisions.

Constraints

No additional constraints

11.3.4 «ConfigurationDecisionModel» (from Variability) {abstract}

Generalizations

- EAElement

Description

A ConfigurationDecisionModel defines how to configure m so-called target feature models, depending on a given configuration of n so-called 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, only for those used on vehicle level they are defined explicitly; 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).

Extensions

- Package (from UML)
- Class (from UML)

Properties

- rootEntry : ConfigurationDecisionModelEntry [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

11.3.5 «ConfigurationDecisionModelEntry» (from Variability) {abstract}

Generalizations

- EAElement

Description

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

Extensions

- Class (from UML)

Properties

- isActive : Boolean = true [1]
  If active==TRUE then the entry is selected for the ProductDecisionModel.

Semantics

No additional semantics

Constraints

No additional constraints

11.3.6 «ContainerConfiguration» (from Variability)

Generalizations

- ConfigurationDecisionModel

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 contained other 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 as 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.

Extensions

No direct extensions

Properties

- configuredContainer : ConfigurableContainer [1]

Semantics

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

Constraints

No additional constraints

11.3.7 «FeatureConfiguration» (from Variability)

Generalizations

- ConfigurationDecisionModel

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 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), we here 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 are no source feature model to which the criterion can refer, all ConfigurationDecisions in a FeatureConfiguration must have "true" as their criterion.

Extensions

No direct extensions

Properties

- configuredFeatureModel : FeatureModel [1]

Semantics

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

Constraints

No additional constraints

11.3.8 «InternalBinding» (from Variability)

Generalizations

- ConfigurationDecisionModel

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' in the above sense (also called variable content) refer to the documentation of meta-class ConfigurableContainer.

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

11.3.9 «PrivateContent» (from Variability)

Generalizations

- EAElement

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 play together.

Extensions

- Class (from UML)
- NamedElement (from UML)

Properties

- privateElement : NamedElement (from UML) [1]

Semantics

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

Constraints

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

11.3.10 «ReuseMetaInformation» (from Variability)

Generalizations

- TraceableSpecification

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). This kind of information can be stored in this information.

Extensions

- Class (from UML)

Properties

- information : String [1]
  The reuse information is stored in this attribute.

- isReusable : Boolean = true [1]
  This Boolean attributes just says if the entity itself can essentially be reused or not. Specific information or constraints on reuse are in the information attribute. Default value is TRUE.

Semantics

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

Constraints

No additional constraints

11.3.11 «SelectionCriterion» (from Variability)

Generalizations

- EAElement

Description

A mixed string description, identifying the source elements.

Extensions

- NamedElement (from UML)
- Class (from UML)

Properties

- source : NamedElement (from UML) [0..*] {ordered}

Semantics

No additional semantics

Constraints

No additional constraints

11.3.12 «Variability» (from Variability)

Generalizations

- Context

Description

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

Extensions

- Class (from UML)
- Package (from UML)

Properties

- configurableContainer : ConfigurableContainer [0..*]
- configuration : FeatureConfiguration [0..*]
- decisionModel : VehicleLevelBinding~~VehicleLevelConfigurationDecisionModel~~ [0..*]
- productFeatureModel : FeatureModel [0..*]
- variableElement : VariableElement [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

11.3.13 «VariableElement» (from Variability)

Generalizations

- EAElement

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, by that 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 play together.

Extensions

- NamedElement (from UML)
- Class (from UML)

Properties

- actualBindingTime : BindingTime [1]
- optionalElement : NamedElement (from UML) [1]
- requiredBindingTime : BindingTime [0..1]
- reuseMetaInformation : ReuseMetaInformation [0..1]

Semantics

Marks the element identified by association optionalElement as optional.

Constraints

No additional constraints

Constraints in natural language

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

11.3.14 «VariationGroup» (from Variability)

Generalizations

- EAElement

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.).

Extensions

- Class (from UML)

Properties

- constraint : String [1]
  Only defined iff kind=="custom". An OCL constraint specifying how the VariableElements in the variation group can be combined.
- kind : VariabilityDependencyKind [1]
  The kind of the variation group (see enumeration VariationGroupKind).
- variableElement : VariableElement [1..*] {ordered}

Semantics

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

Constraints

No additional constraints

11.3.15 «VehicleLevelBinding~~VehicleLevelConfigurationDecisionModel~~» (from Variability)

Generalizations

- ConfigurationDecisionModel

Description

This class represents a ConfigurationDecisionModel on 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 VehicleLevelConfigurationDecisionModel may be used to bridge the gap from vehicle level variability management to that on 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).

Extensions

No direct extensions

Properties

- sourceVehicleFeatureModel : FeatureModel [0..*] {ordered}
- targetFeatureModel : FeatureModel [0..*] {ordered}

Semantics

No additional semantics

Constraints

No additional constraints

Constraints in natural language

[1] The sourceVehicleFeatureModels shall only contain VehicleFeatures.

[2] The sourceVehicleFeatureModels shall be different from the targetFeatureModels

Part V Requirements

12 EAST-ADL extensions for Requirements

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 level 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. Support for requirements modeling is provided by the EAST-ADL on two levels: a generic level and specializing levels (e.g. Dependability.SafetyRequirement) where specialized requirement entities are provided which are intended for certain special uses.

Elements inspired by SysML are Requirement, Satisfy, Refine, DeriveRequirement, (Verify)

12.1 Overview

No overview

12.2 Profile diagrams

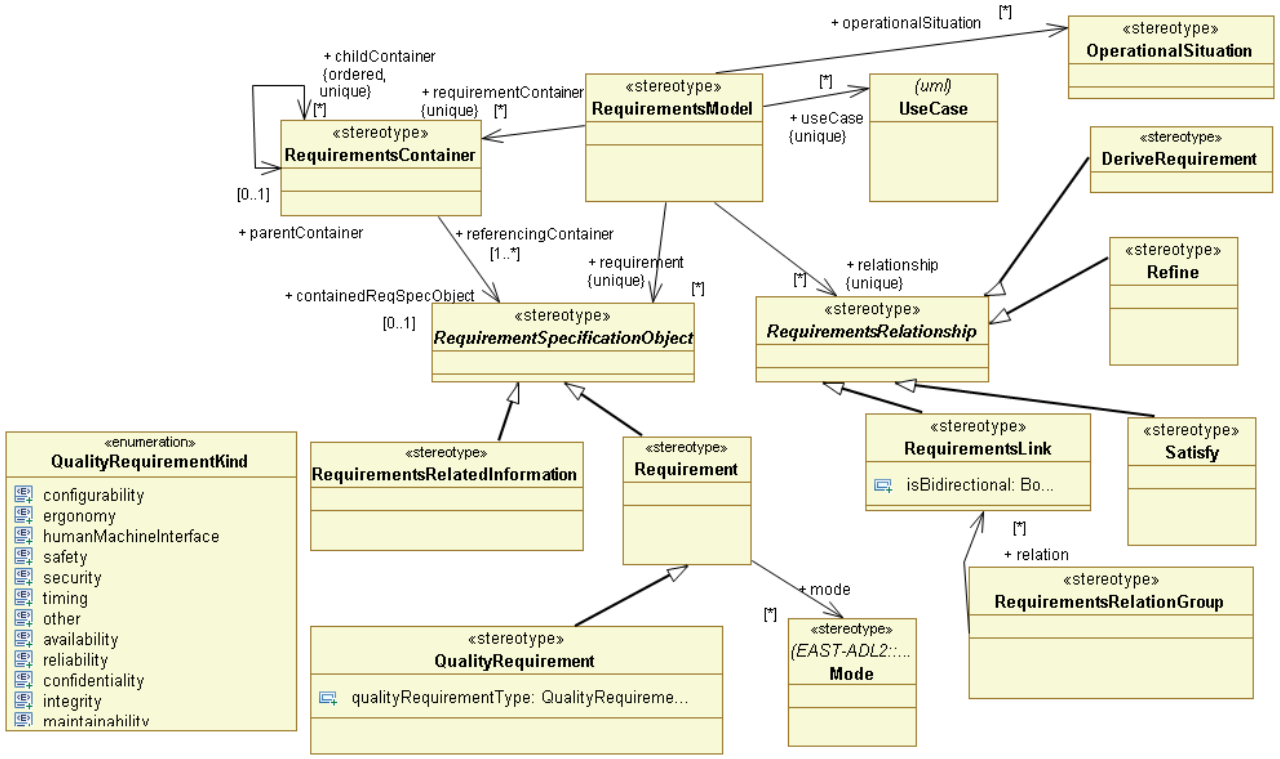

Figure 12. Requirements

12.3 Detailed description of UML profile elements

12.3.1 «Actor» (from Requirements)

Generalizations

- TraceableSpecification

Description

No additional description

Extensions

- Actor (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.1~~12.3.2 «DeriveRequirement» (from Requirements)

Generalizations

- RequirementsRelationship
- DeriveReqt (from SysML::Requirements)

Description

DeriveReqt signifies a dependency relationship in-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). It inherits from SysML::DeriveReqt which extends Dependency.

Extensions

No direct extensions

Properties

- /derived : Requirement [1..*] {readOnly}
  The set of Requirements derived from the supplier Requirement.

  {derived from UML::DirectedRelationship::target}
- /derivedFrom : Requirement [1..*] {readOnly}
  The set of Requirements that the client Requirement are derived from.

  {derived from UML::DirectedRelationship::source}

Semantics

DeriveReqt signifies a derived/derived by relationship between Requirements, where the modification of the supplierRequirement may impact the derived client Requirement. DeriveReqt implies the semantics that the derived client Requirement is not complete, without the supplier Requirement.

Constraints

No additional constraints

Notation

- Icon: Serialized 

12.3.3 «Extend» (from Requirements)

Generalizations

- Relationship

Description

No additional description

Extensions

- Extend (from UML)

Properties

- extendedCase : UseCase [1]
- extensionLocation : ExtensionPoint [1..*]

Semantics

No additional semantics

Constraints

No additional constraints

12.3.4 «ExtensionPoint» (from Requirements)

Generalizations

- RedefinableElement

Description

No additional description

Extensions

- ExtensionPoint (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

12.3.5 «Include» (from Requirements)

Generalizations

- Relationship

Description

No additional description

Extensions

- Include (from UML)

Properties

- addition : UseCase [1]

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.2~~12.3.6 «OperationalSituation» (from Requirements)

Generalizations

- TraceableSpecification

Description

No additional description

Extensions

- Class (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.3~~12.3.7 «QualityRequirement» (from Requirements)

Generalizations

- Requirement

Description

QualityRequirements are the kind of requirements that are used to introduce externally visible properties of the system considered as a whole.

The attribute qualityRequirementType allows a more specific classification.

Extensions

No direct extensions

Properties

- qualityRequirementType : QualityRequirementKind [1]
  The specific type of quality requirement

Semantics

No additional semantics

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~12.2.4~~12.3.8 QualityRequirementKind (from Requirements)

Generalizations

None

Description

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

Enumeration Literals

- availability
- confidentiality
- configurability
- ergonomy
- humanMachineInterface
- integrity
- maintainability
- other
- reliability

- safety
- security
- timing

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.5~~12.3.9 «Refine» (from Requirements)

Generalizations

- RequirementsRelationship
- Refine (from Standard)

Description

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

Extensions

- NamedElement (from UML)
- Dependency (from UML)

Properties

- /refinedBy : NamedElement (from UML) [1..*] {readOnly}
  List of Entity participating to the refinement of the refined requirements.

  {derived from UML::Dependency::client}
- refinedBy_path : NamedElement (from UML) [0..*] {ordered}
- /refinedRequirement : Requirement [1..*] {readOnly}
  List of refined requirements.

  {derived from UML::DirectedRelationship::target}

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.

Constraints

No additional constraints

Constraints in natural language

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

Notation

- Icon: Serialized 

12.3.10 «RedefinableElement» (from Requirements) {abstract}

Generalizations

None

Description

No additional description

Extensions

- RedefinableElement (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.6~~12.3.11 «Requirement» (from Requirements)

Generalizations

- RequirementSpecificationObject
- Requirement (from SysML::Requirements)

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 to a Context (by inheriting from TraceableSpecification or it may be included in a RequirementContainer, 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.

Extensions

No direct extensions

Properties

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

Semantics

The Requirement metaclass applies to the EAElement that is associated to the Requirement through the Satisfy relation.

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~12.2.7~~12.3.12 «RequirementsContainer» (from Requirements)

Generalizations

- TraceableSpecification

Description

RequirementContainer represents a larger unit or module of specification information. It is used to share several Requirements which are semantically related to each other. Also, a RequirementContainer structure will be used for structuring requirement specification objects (Requirements, Rationals etc.). Thus, to preserve the ordering of requirement specification objects the ordering of child containers is very important here.

In addition to sharing related Requirements, the RequirementContainer allows to define relations between its contained Requirements and also a grouping of them.

Furthermore, the RequirementContainer allows introducing additional user attribute definitions by way of UserAttributeElementTypes or UserAttributeTemplates which are valid only locally inside this RequirementContainer. 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.

An EAST-ADL system model may contain a forest of RequirementContainer (see parent child relationship). Only non root RequirementContainer which have a parentContainer are allowed to reference a RequirementSpecificationObject.

The RequirementContainer with its parent child containment relationship and the reference to RequirementSpecificationObject is the basis element for structuring requirement information into a forest structure.

Extensions

- Class (from UML)
- Package (from UML)

Properties

- containedReqSpecObject : RequirementSpecificationObject [0..1]

Associations

- childContainer : RequirementsContainer [0..*] {ordered}
  Sub containers of a requirement container. Sub containers may have references (each time max. 1) to requirement specification objects. To preserve the original ordering of requirement specifiaction objects, the ordering of Sub containers is very important here.

  See RequirementsContainer.parentContainer
- parentContainer : RequirementsContainer [0..1]
  The Parent container of a container. If there is no parent, the container is a root container and thus cannot have a reference to a requirement specification object.

  See RequirementsContainer.childContainer

Semantics

No additional semantics

Constraints

No additional constraints

Constraints in natural language

[1] Only non root RequirementContainer (parentContainer must be set) which have a parentContainer are allowed to reference a RequirementSpecificationObject.

~~12.2.8~~12.3.13 «RequirementsLink» (from Requirements)

Generalizations

- RequirementsRelationship

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 RequirementsGroup 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.

Extensions

No direct extensions

Properties

- isBidirectional : Boolean [1]
- source : Requirement [1..*]
- target : Requirement [1..*]

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.9~~12.3.14 «RequirementsModel» (from Requirements)

Generalizations

- Context

Description

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

Extensions

- UseCase (from UML)
- Package (from UML)

Properties

- operationalSituation : OperationalSituation [0..*]
- ~~relationship : RequirementsRelationship [0..*]~~
- requirement : RequirementSpecificationObject [0..*]
- requirementContainer : RequirementsContainer [0..*]
- useCase : UseCase (from UML) [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.10~~12.3.15 «RequirementSpecificationObject» (from Requirements) {abstract}

Generalizations

- TraceableSpecification

Description

In general it is a standard practice (e.g. using IBM Rational DOORS) to define requirements and also rationales, explanations and other requirement related information as direct successors or predecessors of an appropriate requirement. Thus, requirements and requirement related information are generalized to RequirementSpecificationObject which in turn can be referenced by the structuring container structure (RequirementContainer).

Extensions

No direct extensions

Properties

- referencingContainer : RequirementsContainer [1..*]

~~No additional properties~~

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.11~~12.3.16 «RequirementsRelatedInformation» (from Requirements)

Generalizations

- RequirementSpecificationObject

Description

This is a placeholder for all objects which are not Requirements (such as Rational, Explanations, Related Material etc...). E.g. an element of type RequirementsRelatedInformation which is a

rational of an element of type Requirement will directly succeeding this requirement as a sibling element (see structuring of requirement elements via RequirementContainer).

Extensions

- Class (from UML)

Properties

No additional properties

Semantics

This metaclass can be used to represents information this is not a requirement but is related to requirements and is often provided together with a set of requirements in a requirements specification.

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~12.2.12~~12.3.17 «RequirementsRelationGroup» (from Requirements)

Generalizations

- TraceableSpecification

Description

RequirementsRelationGroup represents a group of relations between Requirements.

Extensions

- Class (from UML)
- Package (from UML)

Properties

- relation : RequirementsLink [0..*]
  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 ReqRelationGroups.

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.13~~12.3.18 «RequirementsRelationship» (from Requirements) {abstract}

Generalizations

- Relationship

Description

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 ReqGroup 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 n:m relations can be expressed with instances of this class.

The semantic of a concrete requirement relation is not defined by the EAST-ADL and therefore needs to 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 uaType (user attributeable element type) can be used. Certain types will be used for certain relation semantics.

3) ReqRelationGroups 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

Extensions

- Class (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

~~12.2.14~~12.3.19 «Satisfy» (from Requirements)

Generalizations

- RequirementsRelationship
- Satisfy (from SysML::Requirements)

Description

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

Extensions

- NamedElement (from UML)

Properties

- /satisfiedBy : NamedElement (from UML) [0..*] {readOnly}

  {derived from UML::Dependency::client}
- satisfiedBy_path : NamedElement (from UML) [0..*] {ordered}
- /satisfiedRequirement : Requirement [0..*] {readOnly}
  List of satisfied ADL requirements, which are satisfied by the client ADL entities.

  {derived from UML::DirectedRelationship::target}
- satisfiedUseCase : UseCase (from UML) [0..*]
  List of satisfied use cases, which are satisfied by the client entities or satisfied by the client AUTOSAR elements.
- satisfiedUseCase : UseCase [0..*]

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.

Constraints

No additional constraints

Constraints in natural language

[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!

12.3.20 «UseCase» (from Requirements)

Generalizations

- TraceableSpecification

Description

No additional description

Extensions

- UseCase (from UML)

Properties

- extend : Extend [0..*] {composite}
- extensionPoint : ExtensionPoint [0..*] {composite}
- include : Include [0..*] {composite}

Semantics

No additional semantics

Constraints

No additional constraints

13 EAST-ADL extensions for VerificationValidation

A multitude of different verification and validation (V&V) techniques, methods and tools are applied during the development of EE-Systems. Different techniques are applicable at different abstraction levels. Also, the technique of choice depends on the properties to validate and/or verify. Furthermore, each partner in a project may develop and employ his own V&V processes and activities. Hence it is impossible to introduce in the EAST-ADL a way to model all the objects that can be required by all the possible V&V techniques. As a consequence, EAST-ADL furnishes just the means for planning, organizing and describing V&V activities on a fairly abstract level, and for defining 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.). The common parts of all V&V techniques are described by the EAST-ADL, which includes: the results expected from the V&V activities, the actual results which were obtained when applying the V&V techniques, 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.

Single V&V techniques may be used only 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 achieve a complete verification and validation of a given system. Often these techniques and activities are employed and performed by many different teams and departments, frequently even by different companies. This raises the demand for an overall planning and organization of all V&V related information.

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

(1) On the abstract level, verification and validation information is defined without referring to a concrete testing environment and without specifying stimuli and 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.

In accordance to the "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 for particular testing environments. So all abstract VVCases and VVProcedures for a certain system together form sort of 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 to all conceivable testing environments.

13.1 Overview

No overview

13.2 Profile diagrams

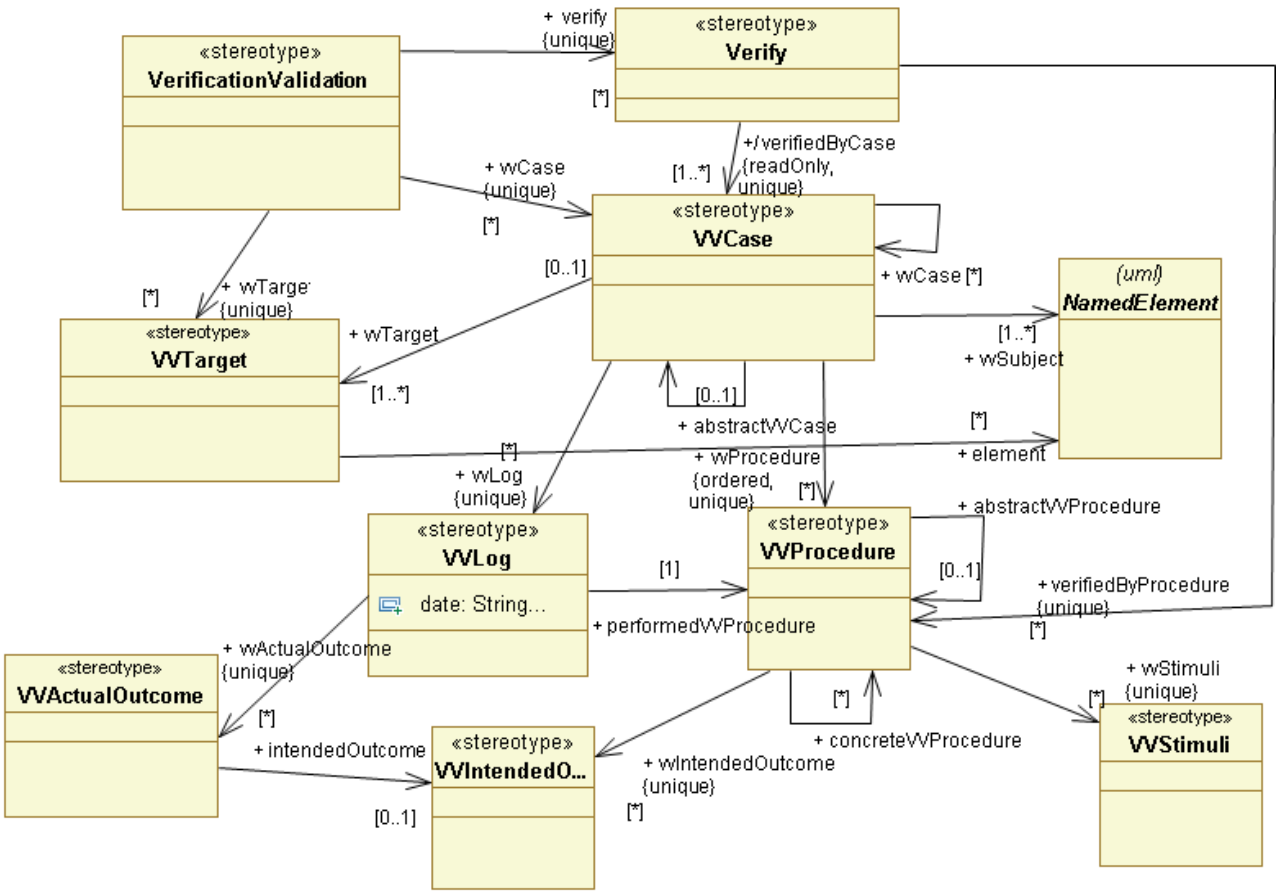

Figure 13. VerificationValidation

13.3 Detailed description of UML profile elements

13.3.1 «VerificationValidation» (from VerificationValidation)

Generalizations

- Context

Description

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

Extensions

- Class (from UML)
- Package (from UML)

Properties

- verify : Verify [0..*]
- vvCase : VVCase [0..*]
- vvTarget : VVTarget [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

13.3.2 «Verify» (from VerificationValidation)

Generalizations

- RequirementsRelationship
- Verify (from SysML::Requirements)

Description

The Verify is a relationship metaclass, which signifies a dependency relationship between a Requirement and a VVCase, showing the relationship when a client VVCase verifies the supplier Requirement.

Extensions

No direct extensions

Properties

- /verifiedByCase : VVCase [1..*] {readOnly}
  The verification that verifies the supplier requirement(s).

  {derived from UML::DirectedRelationship::source}
- verifiedByProcedure : VVProcedure [0..*]
  The procedures used to verify the requirements.
- /verifiedRequirement : Requirement [1..*] {readOnly}
  The set of requirements which the client VV cases verify. {derived from UML::DirectedRelationship::target}

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. The Verify metaclass implies the semantics that the verifying client VVCase is not complete, without the supplier Requirement.

Constraints

No additional constraints

13.3.3 «VVActualOutcome» (from VerificationValidation)

Generalizations

- TraceableSpecification

Description

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

Extensions

- Class (from UML)

Properties

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

Semantics

No additional semantics

Constraints

No additional constraints

13.3.4 «VVCase» (from VerificationValidation)

Generalizations

- TraceableSpecification

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.

Extensions

- Class (from UML)
- NamedElement (from UML)

Properties

- abstractVVCase : VVCase [0..1]
- vvCase : VVCase [0..*]
- vvLog : VVLog [0..*]
- vvProcedure : VVProcedure [0..*] {ordered}
  The abstract VV procedures for this AbstractVVCase.
- vvSubject : NamedElement (from UML) [1..*]
- vvTarget : VVTarget [1..*]

Semantics

No additional semantics

Constraints

No additional constraints

Notation

- Icon: Serialized 

13.3.5 «VVIntendedOutcome» (from VerificationValidation)

Generalizations

- TraceableSpecification

Description

Expected output of the testing environment represented by VVTarget when triggered by the corresponding VVStimuli of the containing ConcreteVVProcedure.

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

Extensions

- Class (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

13.3.6 «VVLog» (from VerificationValidation)

Generalizations

- TraceableSpecification

Description

ConcreteVVCase 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 a certain execution of a ConcreteVVCase.

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 result from Friday and Monday respectively.

Extensions

- Class (from UML)

Properties

- date : String [1]
  Date and time when this log was captured.
- performedVVProcedure : VVProcedure [1]
- vvActualOutcome : VVActualOutcome [0..*]
  Set of outcome results.

Semantics

No additional semantics

Constraints

No additional constraints

13.3.7 «VVProcedure» (from VerificationValidation)

Generalizations

- TraceableSpecification

Description

VVProcedure represents an individual task in the context of an overall V&V effort (represented by a VVCase), which has to be performed in order to achieve that effort's overall objective. Just as is

the case for VVCases, the definition of VVProcedures is separated in two levels: an abstract and a concrete level represented by the entities AbstractVVProcedure and ConcreteVVProcedure.

The concreteVVProcedure metaclass represents such a task on a concrete level, i.e. it is defined with a concrete testing environment in mind and provides stimuli and an expected outcome of the procedure in a form which is directly applicable to this testing environment.

Extensions

- Class (from UML)

Properties

- abstractVVProcedure : VVProcedure [0..1]
- concreteVVProcedure : VVProcedure [0..*]
- vvIntendedOutcome : VVIntendedOutcome [0..*]
- vvStimuli : VVStimuli [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

Notation

- Icon: Serialized 

13.3.8 «VVStimuli» (from VerificationValidation)

Generalizations

- TraceableSpecification

Description

VVStimuli represents the input values to 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 ConcreteVVCase), the input values must be provided in a form such that they are directly applicable to the VVTarget(s) defined for the containing ConcreteVVCase.

Extensions

- Class (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

13.3.9 «VVTarget» (from VerificationValidation)

Generalizations

- TraceableSpecification

Description

VVTarget represents a concrete testing environment in which or on 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 be a realization of 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 ConcreteVVCases.

Extensions

- Class (from UML)
- NamedElement (from UML)

Properties

- element : NamedElement (from UML) [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

14 EAST-ADL extensions for Interchange

The interchange part of an EAST-ADL system model is for exchanging model data with external stakeholders. E.g. it provides elements (see RIFArea) for importing resp. exporting requirements specifications into resp. out of an EAST-ADL system model.

14.1 Overview

No overview

14.2 Profile diagrams

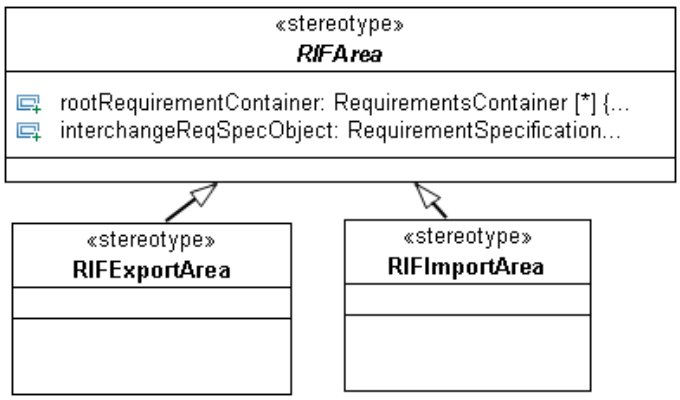

Figure 14. Interchange

14.3 Detailed description of UML profile elements

14.3.1 «RIFArea» (from Interchange) {abstract}

Generalizations

- Context

Description

An extra allocated part of the EAST-ADL System Model that contains Requirement Specific Data (Container, Reqs etc...) from RIF Import resp. RIF Export.

Especially for the context of requirement engineering and considering the possibility to import/export requirement related data via RIF, the feature uuid will be used to check that two elements are semantically the same and thus should stay together in reference via a Multi-Level reference link.

Since requirement data to be imported/exported will be put into RIFArea, requirement data elements which are not inside RIFArea and have semantically equal element in the RIFAreas (such elements have the same uuid value) will be connected with the appropriate elements in the RIFArea using Multi-Level reference links.

Extensions

- Class (from UML)

Properties

- interchangeReqSpecObject : RequirementSpecificationObject [0..*]
- rootRequirementContainer : RequirementsContainer [0..*] {ordered}

Semantics

No additional semantics

Constraints

No additional constraints

14.3.2 «RIFExportArea» (from Interchange)

Generalizations

- RIFArea

Description

Contains (clones of) requirement specific data to be exported in RIF format.

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

14.3.3 «RIFImportArea» (from Interchange)

Generalizations

- RIFArea

Description

Contains requirement specific data to be imported from an external RIF file.

If an element will be imported from external the uuid will be taken from the given external exchange data file, because the identifier is global unique and shall not be changed somewhere.

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

Part VI Timing

15 EAST-ADL extensions for Timing

This section contains the UML-profile specification, specifying stereotypes in the UML-profile, defined from the domain model classes in the Timing package. It includes specification details for each stereotype. If the stereotype has properties, which may be referred to as tag definitions, or if the stereotype has constraints, this section also includes specification details for these properties and constraints.

15.1 Overview

No overview

15.2 Profile diagrams

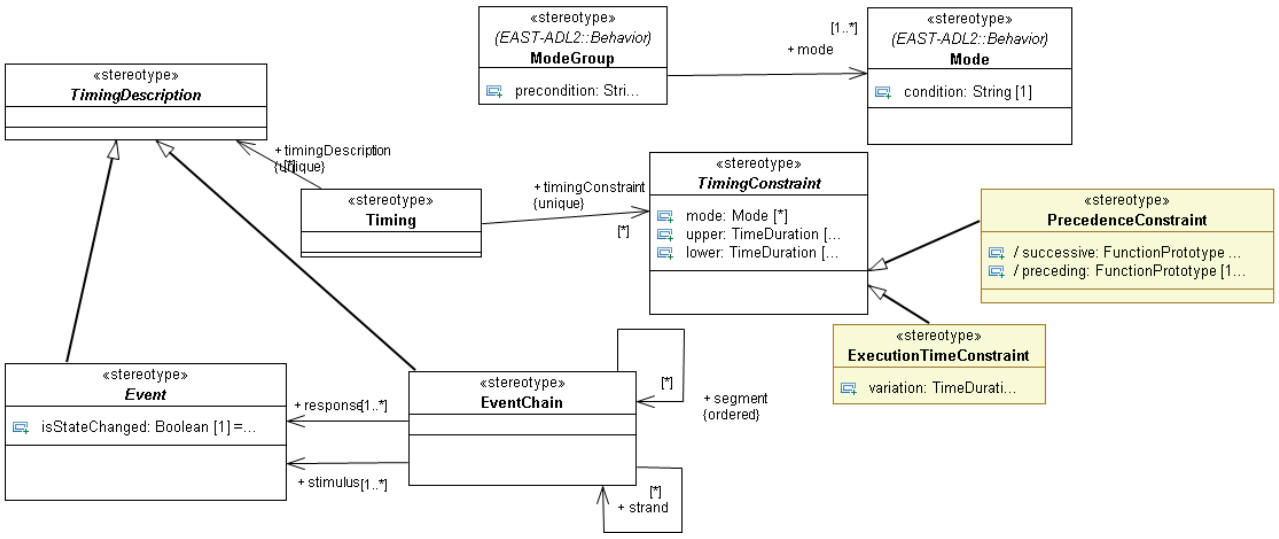

Figure 15. Timing

15.3 Detailed description of UML profile elements

15.3.1 «Event» (from Timing) {abstract}

Generalizations

- TimingDescription

Description

An Event (E) is supposed to denote a distinct form of state change in a running system, taking place at distinct points in time called occurrences of the event. An event may also report a [current] state. In that case, the event occurs periodically. For example, the "driver door has been opened" is an event indicating a state change; whereas the "driver door is open" is an event reporting a state.

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.

The occurrence of an event either stimulates an execution, or is caused by an execution [as a response to another event that occurred before]. In the first case the event is called Stimulus (S) and in the latter case it is called Response (R). Stimuli always precede responses; and responses in turn always succeed stimuli.

An event occurs instantaneously, which means that an event occurs at a time instant without any duration. In addition, an event can appear any number of times and the subsequent occurrences may follow a specific pattern, like periodic, sporadic, or in sudden bursts. Each of these occurrences has a unique time instant.

The distinction between an event and its occurrence is usually obvious from the considered context (causal and temporal). The event is not defined by its occurrences, but rather by a description expressing its purpose.

Extensions

No direct extensions

Properties

- isStateChanged : Boolean (from UML) = true [1]
  This attribute indicates whether the event reports a state change or a [current] state. If the boolean value is TRUE, then the event reports a state change (no over-undersampling).

  If the boolean value is FALSE, then the event reports a [current] state.

  By default, the value of this attribute is TRUE.

Semantics

No additional semantics

Constraints

No additional constraints

Constraints in natural language

[1] In the case that the event reports a [current] state (isStateChange is FALSE), the event must have a periodic event model [or a pattern model]. Rationale: The [current] state shall be reported consistently and periodically.

15.3.2 «EventChain» (from Timing)

Generalizations

- TimingDescription

Description

The purpose of event chains is to describe the temporal behavior of a number of steps to be taken to respond to one or more events accordingly. [An event chain is also used to express that a temporal requirement/constraint is imposed on a number of steps to respond to one or more events accordingly (-> requirement).] Such events could be observed in a given system and are categorized into stimuli and responses.

Event chains can refer to other event chains which are then called event chain segments or strands. Segments are sequential event chains refining an EventChain, while strands define parallel event chains that refine an EventChain. An EventChain can be both a segment and a

strand at the same time. An event chain respectively event chain segment can be atomic which means it is not refined to other event chains.

Extensions

- Class (from UML)

Properties

- response : Event [1..*]
  The Response element is the entity to describe an event that is a response to a stimulus that occurred before.
- segment : EventChain [0..*] {ordered}
  Referred EventChains that are not parallel and in sequence refines this EventChain.
- stimulus : Event [1..*]
  The Stimulus element is the entity to describe an event that stimulates the steps to be taken to respond to this event.
- strand : EventChain [0..*]
  Parallel EventChains refining this EventChain.

Semantics

An EventChain references two groups of events: stimulus and response. The semantics is that each event in the stimulus group somehow causes, or at least affects the value of all events in the response group. However, since questions about causality and value influence clearly involve the semantics of the underlying structural model, this aspect of an EventChain is semantically outside its scope. Instead, delay constraint semantics are defined solely in terms of the times at which the stimulus and response events occur, independently of whether there actually exists a causal connection between these events in the structural model.

Constraints

No additional constraints

Constraints in natural language

[1] The cardinality of strand shall be either 0 or greater than 1. Rationale: Only values > 1 express true parallelism.

15.3.3 «ExecutionTimeConstraint» (from Timing)

Generalizations

- TimingConstraint

Description

ExecutionTimeConstraint expresses the execution time of a function under the assumption of a nominal CPU that executes 1 "function second" per second. Function allocation will decide the actual execution time by multiplication with the relative speed of the host CPU.

Example:

The ECU is 20% 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 120 MHz)

The function is activated by a time trigger or a port trigger. The function starts execution some time after activation, depending on e.g. interference and blocking from other functions on the same resource

Immediately on start, the function reads input data on all ports. Functions write data at the latest when the execution time has elapsed (which is after the execution time plus any blocking and interference time).

Extensions

No direct extensions

Properties

- targetDesignFunctionPrototype : DesignFunctionPrototype [0..1]
- targetDesignFunctionType : DesignFunctionType [0..1]
- variation : TimeDuration [1]

Semantics

lower (from TimingConstraint) denotes the minimal best case execution time.

upper (from TimingConstraint) denotes the maximal worst case execution time.

variation denotes the allowed variation in execution time, i.e. maximal minimal execution time.

Example:

lower=5

upper=10

variation=2

best case execution time of 6 and worst case of 7 is within this constraint

best case execution time of 6 and worst case of 9 violates this constraint

If a measured value is characterized, variation is not used, as it is always upper-lower, e.g. lower=6 and upper=9 above. In this example, the ExecutionTimeConstraint would be a Realization of a VVActualOutcome.

Constraints

No additional constraints

Constraints in natural language

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

[2] variation shall be a value between 0 and upper-lower.

Notation

- Icon: Serialized 

15.3.4 «PrecedenceConstraint» (from Timing)

Generalizations

- TimingConstraint

Description

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

Extensions

- Dependency (from UML)

Properties

- /preceding : FunctionPrototype [1] {readOnly}
  The function prototype that must be executed first.

  {derived from UML::DirectedRelationship::source}
- preceding_path : FunctionPrototype [0..*] {ordered}
- /successive : FunctionPrototype [1..*] {ordered, readOnly}
  The function prototypes that must be executed after preceding was executed.

  {derived from UML::DirectedRelationship::target}
- successive_path : FunctionPrototype [0..*] {ordered}

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.

Constraints

No additional constraints

Notation

- Icon: Serialized 

15.3.5 «TimeDuration» (from Timing)

Generalizations

- EAElement

Description

CseCodeType

0:	1 µsec	Time	
1:	10 µsec	Time	
2:	100 µsec	Time	
3:	1 msec	Time	
4:	10 msec	Time	
5:	100 msec	Time	
6:	1 sec	Time	
7:	10 sec	Time	
8:	1 min	Time	
9:	1 h	Time	
10:	1 d	Time	
100:	Angular degrees	Angle	
101:	Revolutions 360 degrees	Angle	
102:	Cycle 720 degrees	Angle	e.g. in case of IC engines
103:	Cylinder segment	Combustion	e.g. in case of IC engines

998: When frame available Time Source defined in the ASAP 2 keyword, FRAME

999: Always if there is new value Calculation of a new upper range limit after receiving a new partial value, e.g. when calculating a complex trigger condition

1000: Non deterministic Without fixed scaling

If, for example, the value in swCseCodeFactor is 360 and the value in swCseCode is 100, this is equivalent to the value 1 in swCseCodeFactor and the value 101 in swCseCode.

CseCodeType is from AUTOSAR and MSR/ASAM.

Note that we have set the cseCodeType for 1 µsec to 0 (error in AUTOSAR R3). And have changed cseCodeType 2 to 100 µsec (error in MSR).

Extensions

- DataType (from UML)

Properties

- cseCode : Integer (from UML) [1]
  Within TIMMO this is normally time, note that when it is expressed as angle it can be converted to time.
- cseCodeFactor : Integer (from UML) = 1 [1]
  Is normally equal to 1.
- value : Float [1]
  The actual value complemented with the cseCode.

Semantics

No additional semantics

Constraints

No additional constraints

15.3.6 «Timing» (from Timing)

Generalizations

- Context

Description

The collection of timing constraints and their descriptions in the form of events and event chains. This collection can be done across the EAST-ADL abstraction levels.

Extensions

- Class (from UML)
- Package (from UML)

Properties

- timingConstraint : TimingConstraint [0..*]
- timingDescription : TimingDescription [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

15.3.7 «TimingConstraint» (from Timing) {abstract}

Generalizations

- EAElement

Description

TimingConstraint is an abstract entity that identifies a mode.

Extensions

- Class (from UML)
- Constraint (from UML)

Properties

- lower : TimeDuration [0..1]
- mode : Mode [0..*]
  The mode where the TimingConstraint is valid.
- upper : TimeDuration [0..1]

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.

Constraints

No additional constraints

Constraints in natural language

[1] upper shall be greater or equal to lower.

15.3.8 «TimingDescription» (from Timing) {abstract}

Generalizations

- EAElement

Description

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

Extensions

- Class (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

16 EAST-ADL extensions for TimingConstraints

This section describes the timing constraints.

16.1 Overview

No overview

16.2 Profile diagrams

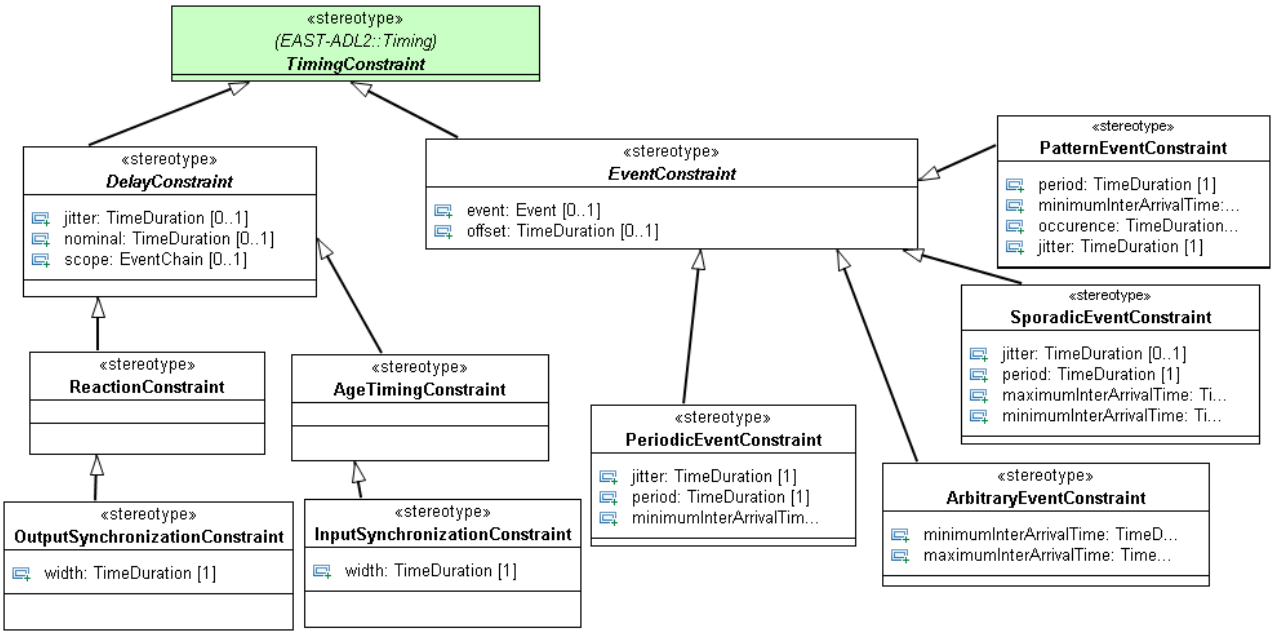

Figure 16. TimingConstraints

16.3 Detailed description of UML profile elements

16.3.1 «AgeTimingConstraint» (from TimingConstraints)

Generalizations

- DelayConstraint

Description

Different tolerances on over-/undersampling can be identified when the solution has been modeled.

An age constraint is of interest in control engineering when looking back through the system.

In case of over- or undersampling, there is no one-to-one relation possible between the occurrences of stimuli and responses of the associated event chain. Thus, the age constraint defines the semantic of which delay must be constrained.

The attribute upper is applicable in worst-case analysis.

The attribute lower is applicable in best-case analysis.

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

16.3.2 «ArbitraryEventConstraint» (from TimingConstraints)

Generalizations

- EventConstraint

Description

The Arbitrary Event Model describes that an event occurs occasionally, singly, irregular or randomly. The primary purpose of this event model is to abstract event occurrences captured by data acquisition tools (background debugger, trace analyzer, etc.) during the operation of a system.

Extensions

No direct extensions

Properties

- maximumInterArrivalTime : TimeDuration [1..*]
  The set of maximum inter-arrival times specifies the maximum inter-arrival time between two and more subsequent occurrences of the event. The first element in the set specifies the maximum inter-arrival time between two subsequent occurrences of the event among the given occurrences. The second element in the set specifies the maximum inter-arrival time between three subsequent occurrences of the event among the given occurrences; and so forth.
- minimumInterArrivalTime : TimeDuration [1..*]
  The set of minimum inter-arrival times specifies the minimum inter-arrival time between two and more subsequent occurrences of the event. The first element in the set specifies the minimum inter-arrival time between two subsequent occurrences of the event among the given occurrences. The second element in the set specifies the minimum inter-arrival time between three subsequent occurrences of the event among the given occurrences; and so forth.

Semantics

No additional semantics

Constraints

No additional constraints

Constraints in natural language

[1] The number of elements in the sets minimum inter-arrival time and maximum inter-arrival time must be the same. Rationale: Consistent specification of arrival times.

16.3.3 «DelayConstraint» (from TimingConstraints) {abstract}

Generalizations

- TimingConstraint

Description

DelayConstraints give bounds on system timing attributes, i.e. end-to-end delays, periods, etc.

A DelayConstraint can specify one or several of an upper bound, a lower bound or a nominal value and jitter. The jitter is evenly distributed around the nominal (nominal - jitter/2 .. nominal + jitter/2). The bound will be measured in a given unit.

Extensions

No direct extensions

Properties

- jitter : TimeDuration [0..1]
  Jitter is the range within which a value varies.
- nominal : TimeDuration [0..1]
  The recurring distance between each occurrence.
- scope : EventChain [0..1]

Semantics

Lower (from TimingConstraint) denotes the minimum value of the given bound.

Upper (from TimingConstraint) denotes the maximum value of the given bound.

Variation around the nominal value can be expressed by means of an upper and lower bound, or by means of a jitter value.

For example, [lower=10, upper=20, nominal=15] is equal to [nominal=15, jitter=10].

Constraints

No additional constraints

Constraints in natural language

[1] At least Upper or Nominal must be specified. Rationale: At least one value is needed to work with.

Notation

- Icon: Serialized 

16.3.4 «EventConstraint» (from TimingConstraints) {abstract}

Generalizations

- TimingConstraint

Description

An EventConstraint describes the basic characteristics of the way an event occurs over time.

Extensions

No direct extensions

Properties

- event : Event [0..1]

- offset : TimeDuration [0..1]
  In addition an event model may specify an offset, which delays the start of the first period - the occurrence of the very first event - by the given amount of time.

Semantics

No additional semantics

Constraints

No additional constraints

16.3.5 «InputSynchronizationConstraint» (from TimingConstraints)

Generalizations

- AgeTimingConstraint

Description

InputSynchronizationConstraint is a language entity that expresses a timing constraint on the input synchronization among the set of stimulus events.

Extensions

No direct extensions

Properties

- width : TimeDuration [1]
  The width of the sliding window.

Semantics

The parameters of InputSynchronizationConstraint, see TimingConstraint, constrain the time from the first stimulus until last stimulus (i.e., maximum skew between these stimuli). A join point is identified by the response event in the scope EventChain.

Constraints

No additional constraints

Constraints in natural language

[1] The set of FunctionFlowPorts referenced by the events should contain only FlowPorts with direction = in. The rationale for this is that the events shall relate to data on FunctionFlowPorts which is considered (or shall be) temporally consistent.

[2] The scope EventChain shall contain exactly one response Event.

[3] The semantics of this constraint requires that there is more than one stimulus Event in the scope EventChain (each refering to a different FlowPort with direction = in).

Notation

- Icon: Serialized 

16.3.6 «OutputSynchronizationConstraint» (from TimingConstraints)

Generalizations

- ReactionConstraint

Description

OutputSynchronizationConstraint is a language entity that expresses a timing constraint on the output synchronization among the set of response events.

Extensions

No direct extensions

Properties

- width : TimeDuration [1]
  The width of the sliding window.

Semantics

The parameters of OutputSynchronizationConstraint, see TimingConstraints, constrain the time from the first response until last response (i.e., maximum skew between these responses). A fork point is identified by the stimulus event in the scope EventChain.

Constraints

No additional constraints

Constraints in natural language

[1] The set of FunctionFlowPorts referenced by the events should contain only OutFlowPorts. The rationale for this is that the events shall relate to data on FunctionFlowPorts which is considered (or shall be) temporally consistent.

[2] The scope EventChain shall contain exactly one stimulus Event.

[3] The semantics of this constraint require that there is more than one response Events in the scope EventChain.

Notation

- Icon: Serialized 

16.3.7 «PatternEventConstraint» (from TimingConstraints)

Generalizations

- EventConstraint

Description

The [Concrete] Pattern Event Constraint describes that an event occurs following a known pattern. The pattern event model is characterized by the following parameters:

Extensions

No direct extensions

Properties

- jitter : TimeDuration [1]
  The jitter specifies maximal possible time interval the occurrence of the events within the given period can vary (formerly: can be delayed).
- minimumInterArrivalTime : TimeDuration [1]
  The minimum inter-arrival time specifies the minimal possible time interval between two consecutive occurrences of the event within the given period.
- occurence : TimeDuration [1..*] {ordered}
  The set occurrence (1..n) specifies the offset for each occurrence of the event in the specified period. Each occurrence is specified from the beginning of the period

- period : TimeDuration [1]
  The period specifies the time interval within the event occurs any number of times following a pattern.

Semantics

No additional semantics

Constraints

No additional constraints

16.3.8 «PeriodicEventConstraint» (from TimingConstraints)

Generalizations

- EventConstraint

Description

The PeriodicEventConstraint describes that an event occurs periodically.

Extensions

No direct extensions

Properties

- jitter : TimeDuration [1]
  The jitter specifies the maximal possible time interval the occurrence of an event can vary (formerly: be delayed).
- minimumInterArrivalTime : TimeDuration [1]
  The minimum inter-arrival time specifies the minimal possible time interval between two consecutive occurrences of an event.
- period : TimeDuration [1]
  The period specifies the ideal time interval between two subsequent occurrences of the event.

Semantics

No additional semantics

Constraints

No additional constraints

Notation

- Icon: Serialized 

16.3.9 «ReactionConstraint» (from TimingConstraints)

Generalizations

- DelayConstraint

Description

ReactionConstraint is used to impose a timing constraint on an event chain in order to specify bounds for reacting on the occurrence of a stimulus or stimuli. The intention of this constraint is to look forward in time.

Compare AgeTimingConstraint.

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

16.3.10 «SporadicEventConstraint» (from TimingConstraints)

Generalizations

- EventConstraint

Description

The Sporadic Event Constraint describes that an event occurs occasionally. In general it is supposed that the event eventually occurs. Indeed, it is also known that some of the events do not occur for whatsoever reasons.

Note! The parameters minimum inter-arrival time and maximum inter-arrival time must reference the same point in time. Typically, this is the point in time that specifies the beginning of the period subject to consideration.

Extensions

No direct extensions

Properties

- jitter : TimeDuration [0..1]
  The optional parameter jitter specifies the maximal possible time interval the occurrence of an event can vary (formerly: be delayed). By nature, a sporadic event is supposed to occur at any time, thus it is one of the characteristic that the occurrence is jittery.
- maximumInterArrivalTime : TimeDuration [0..1]
  The optional parameter maximum inter-arrival time specifies the maximal possible time interval between two consecutive occurrences of an event.

  The maximum inter-arrival time may be used to describe different cases:

  (1) The maximum inter-arrival time is equal to the duration of the period.

  (2) The maximum inter-arrival time is used to specify a point in time within the period that immediately follows the period subject to consideration.

  (3) The maximum inter-arrival time is used to specify a point in time within one of the subsequent periods that follow the period subject to consideration.
- minimumInterArrivalTime : TimeDuration [1]
  The minimum inter-arrival time specifies is the minimal possible time interval between two consecutive occurrences of an event.
- period : TimeDuration [1]
  The period specifies the ideal time interval between two subsequent occurrences of the event.

Semantics

No additional semantics

Constraints

No additional constraints

17 EAST-ADL extensions for Events

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

17.1 Overview

No overview

17.2 Profile diagrams

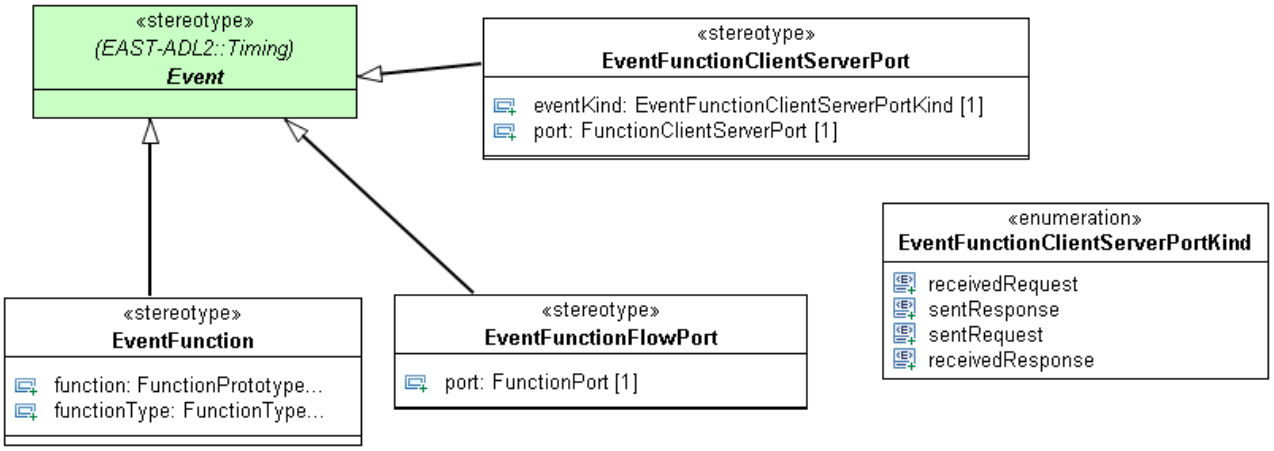

Figure 17. Events

17.3 Detailed description of UML profile elements

17.3.1 «EventFunction» (from Events)

Generalizations

- Event

Description

An event of a Function refers to the triggering of the Function, i.e., when the input data is consumed, data transformation is performed on that input data by the function, and output data is produced. It is used in conjunction with FunctionTrigger (see that concept) 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 EventInFlowPort)

Extensions

No direct extensions

Properties

- function : FunctionPrototype [0..1]
- function_path : FunctionPrototype [0..*] {ordered}
- functionType : FunctionType [0..1]

Semantics

No additional semantics

Constraints

No additional constraints

Constraints in natural language

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

17.3.2 «EventFunctionClientServerPort» (from Events)

Generalizations

- Event

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.

Extensions

No direct extensions

Properties

- eventKind : EventFunctionClientServerPortKind [1]
- port : FunctionClientServerPort [1]
- port_path : FunctionPrototype [0..*] {ordered}

Semantics

No additional semantics

Constraints

No additional constraints

Constraints in natural language

[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.

17.3.3 EventFunctionClientServerPortKind (from Events)

Generalizations

None

Description

Possible values of eventKind.

Enumeration Literals

- receivedRequest
- receivedResponse

- sentRequest
- sentResponse

Semantics

No additional semantics

Constraints

No additional constraints

17.3.4 «EventFunctionFlowPort» (from Events)

Generalizations

- Event

Description

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

Extensions

No direct extensions

Properties

- port : FunctionPort [1]
- port_path : FunctionPrototype [0..*] {ordered}

Semantics

No additional semantics

Constraints

No additional constraints

Part VII Dependability

18 EAST-ADL extensions for Dependability

Dependability of a system is the ability to avoid service failures that are more frequent and more severe than 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 role in 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.

18.1 Overview

No overview

18.2 Profile diagrams

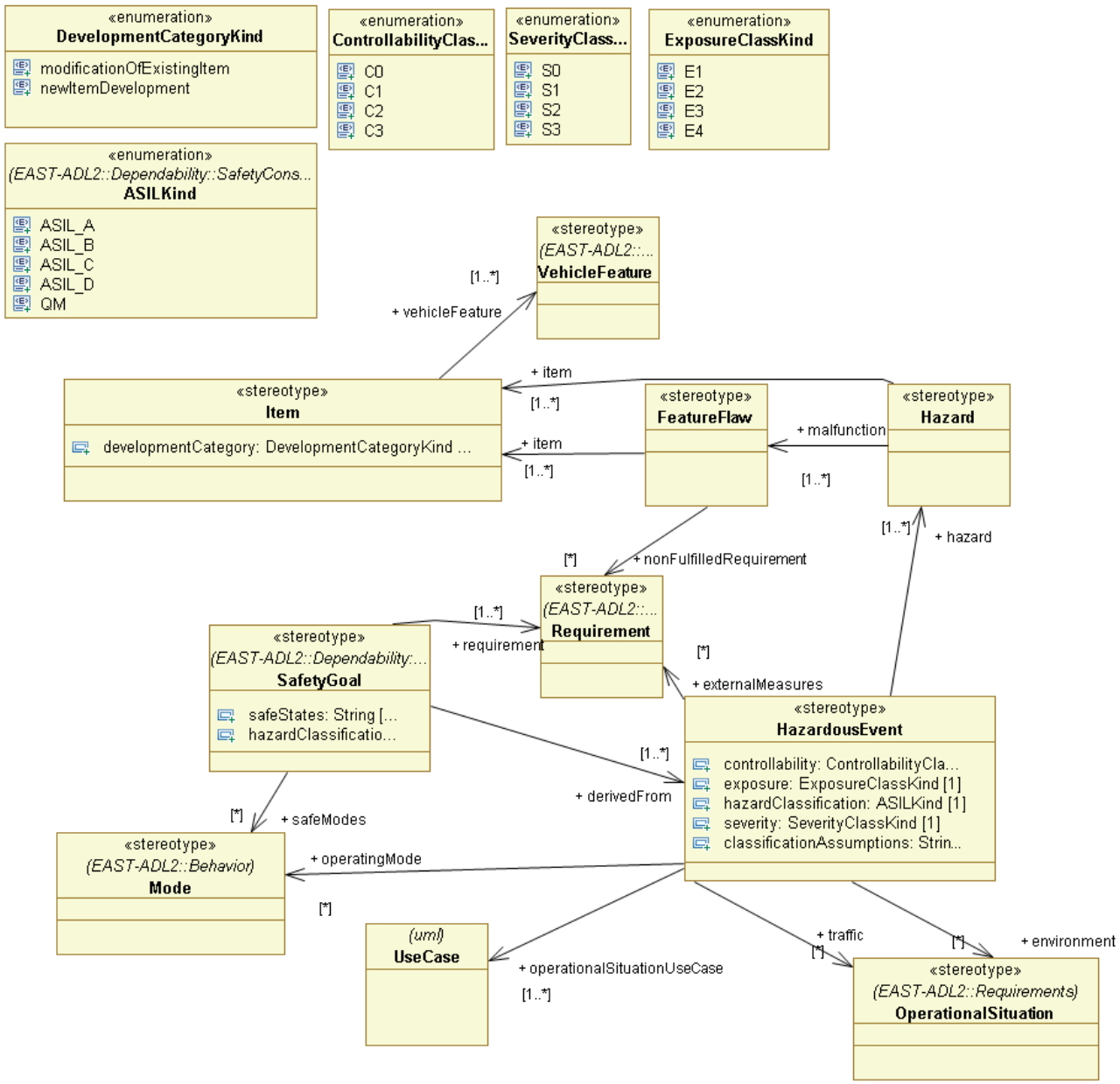

Figure 18. Dependability

18.3 Detailed description of UML profile elements

18.3.1 ControllabilityClassKind (from Dependability)

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

Semantics

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

Constraints

No additional constraints

18.3.2 «Dependability» (from Dependability)

Generalizations

- Context

Description

No additional description

Extensions

- Class (from UML)
- Package (from UML)

Properties

- eADatatype : EADatatype [0..*]
- errorModelType : ErrorModelType [0..*]
- faultFailure : FaultFailure [0..*]
- featureFlaw : FeatureFlaw [0..*]
- functionalSafetyConcept : FunctionalSafetyConcept [0..*]
- hazardousEvent : HazardousEvent [0..*]
- item : Item [0..*]
- quantitativeSafetyConstraint : QuantitativeSafetyConstraint [0..*]
- safetyCase : SafetyCase [0..*]
- safetyConstraint : SafetyConstraint [0..*]
- safetyGoal : SafetyGoal [0..*]
- technicalSafetyConcept : TechnicalSafetyConcept [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

18.3.3 DevelopmentCategoryKind (from Dependability)

Generalizations

None

Description

DevelopmentCategoryKind in 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.

Semantics

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

Constraints

No additional constraints

18.3.4 ExposureClassKind (from Dependability)

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 yhan 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

Semantics

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

Constraints

No additional constraints

18.3.5 «FeatureFlaw» (from Dependability)

Generalizations

- TraceableSpecification

Description

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

Extensions

- Class (from UML)

Properties

- item : Item [1..*]
- nonFulfilledRequirement : Requirement [0..*]

Semantics

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

Constraints

No additional constraints

18.3.6 «Hazard» (from Dependability)

Generalizations

- TraceableSpecification

Description

The Hazard represents a condition or state in the system that may contribute to accidents. It is usually a failure of some kind, but may also be a result of nominal operation.

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

The Hazard metaclass is contained in the context, as Hazard specializes TraceableSpecification. The context describes the element of the system where this hazard occur.

Extensions

- Class (from UML)

Properties

- item : Item [1..*]
- malfunction : FeatureFlaw [1..*]

Semantics

No additional semantics

Constraints

No additional constraints

18.3.7 «HazardousEvent» (from Dependability)

Generalizations

- TraceableSpecification

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.

Extensions

- Class (from UML)
- UseCase (from UML)

Properties

- classificationAssumptions : String [0..1]
- controllability : ControllabilityClassKind [1]
- environment : OperationalSituation [0..*]
- exposure : ExposureClassKind [1]
- externalMeasures : Requirement [0..*]
- hazard : Hazard [1..*]
- hazardClassification : ASILKind [1]
- operatingMode : Mode [0..*]
- operationalSituationUseCase : UseCase (from UML) [1..*]
- severity : SeverityClassKind [1]
- traffic : OperationalSituation [0..*]

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.

Constraints

No additional constraints

18.3.8 «Item» (from Dependability)

Generalizations

- EAElement

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.

Extensions

- Class (from UML)

Properties

- developmentCategory : DevelopmentCategoryKind [1]
  It shall be determined whether the item is a modification of an existing item or if it is a new development.
- vehicleFeature : VehicleFeature [1..*]

Semantics

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

Constraints

No additional constraints

18.3.9 SeverityClassKind (from Dependability)

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

Semantics

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

Constraints

No additional constraints

19 EAST-ADL extensions for ErrorModel

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, forming 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 handing 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 some undesired effects of error modeling, such as the risk of mixing nominal and erroneous behavior in regards to the comprehension, reuse, and system synthesis (e.g., code generation).

19.1 Overview

No overview

19.2 Profile diagrams

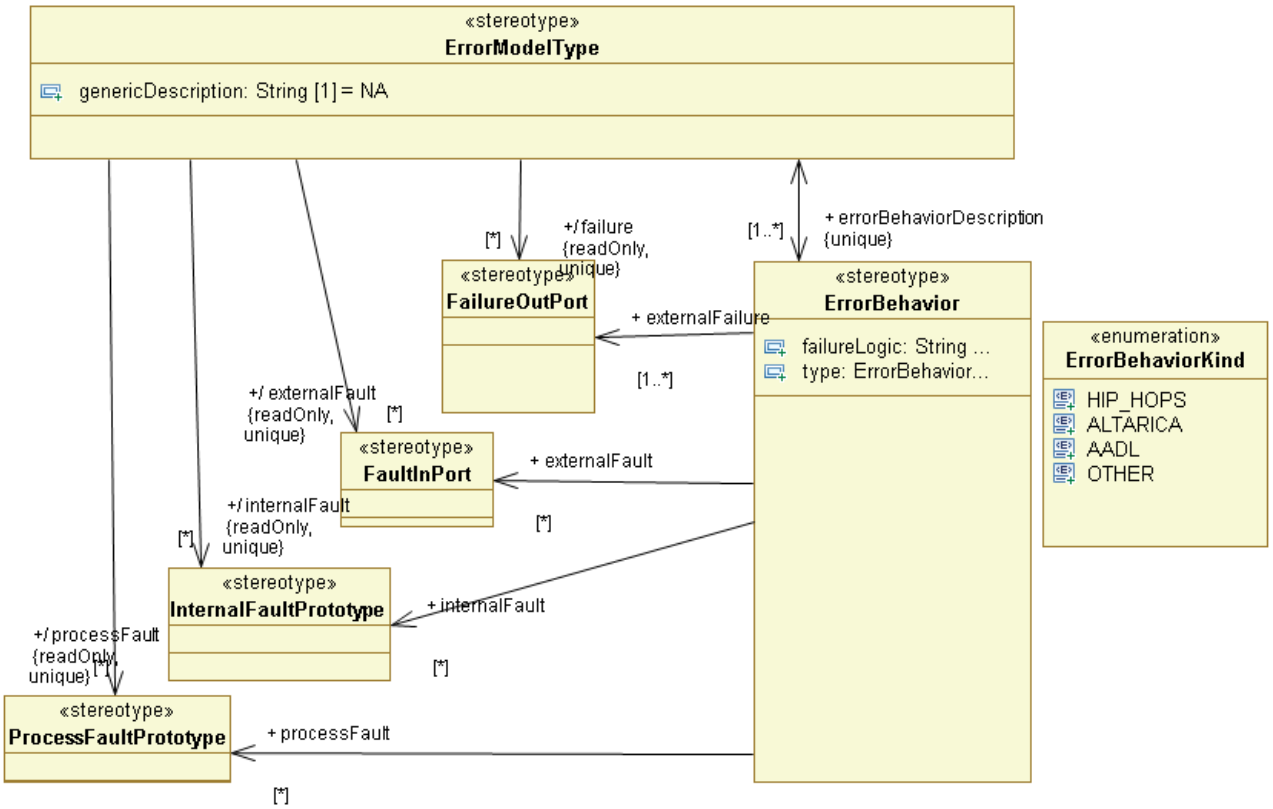

Figure 19. ErrorModel

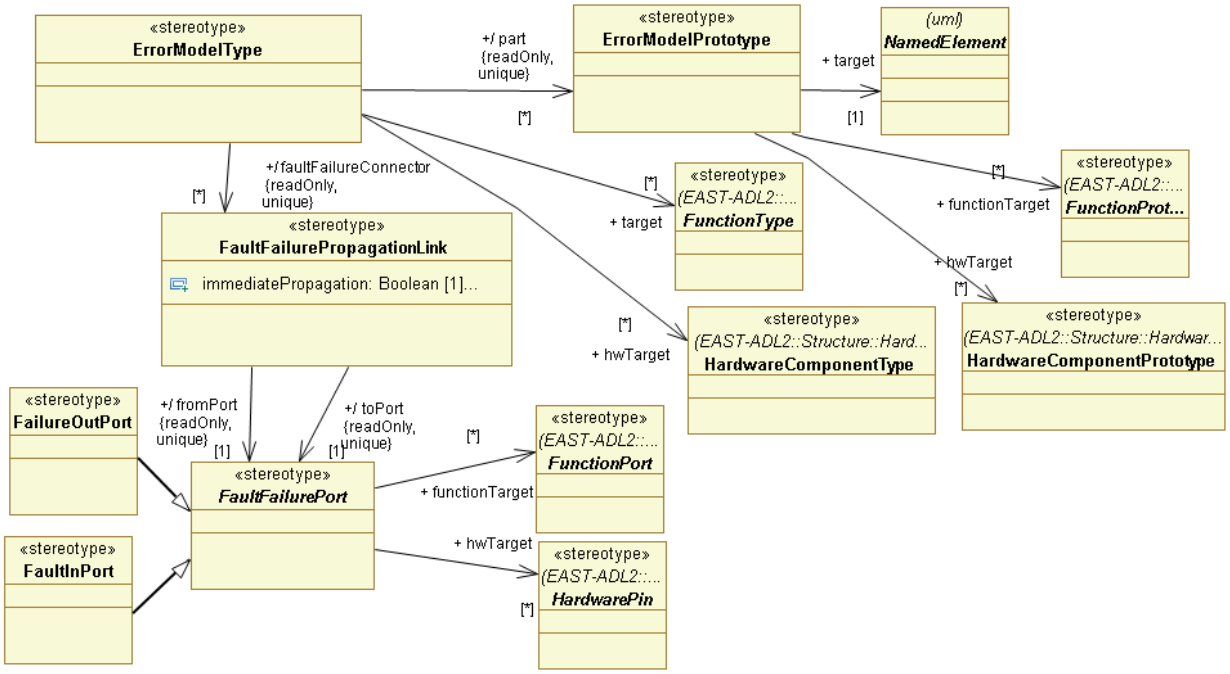

Figure 20. ErrorModel2

19.3 Detailed description of UML profile elements

19.3.1 «Anomaly» (from ErrorModel)

Generalizations

- EAElement

Description

The Anomaly metaclass represents a Fault that may occur internally in an ErrorModel or being 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.

Extensions

- Property (from UML)

Properties

- genericDescription : String [1]
- type : EADatatype [1]

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.

Constraints

No additional constraints

19.3.2 «ErrorBehavior» (from ErrorModel)

Generalizations

- EAElement

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 propagated to and from the target element are declared through the ports of the error model.

Extensions

- Behavior (from UML)

Properties

- externalFailure : FailureOutPort [1..*]
- externalFault : FaultInPort [0..*]
- failureLogic : String (from UML) [0..1]
  The error logic description based on an external formalism or the path to the file or model entity containing the external error logic description.
- internalFault : InternalFaultPrototype [0..*]
  The occurrences of internal faults.
- processFault : ProcessFaultPrototype [0..*]
- type : ErrorBehaviorKind [1]
  The type of formalism applied for the error behavior description.

Associations

- owner : ErrorModelType [0..1]
  See ErrorModelType.errorBehaviorDescription

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.

Constraints

No additional constraints

Notation

- Icon: Serialized 

19.3.3 ErrorBehaviorKind (from ErrorModel)

Generalizations

None

Description

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

Enumeration Literals

- AADL
- ALTARICA
- HIP_HOPS
- OTHER

Semantics

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

Constraints

No additional constraints

19.3.4 «ErrorModelPrototype» (from ErrorModel)

Generalizations

- EAElement

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.

The ErrorModelPrototype is used to define hierarchical error models allowing additional detail or structure to the error model of a particular target. A hierarchal structure can also be defined when several ErrorModels are integrated to 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.

Extensions

- Property (from UML)
- NamedElement (from UML)

Properties

- functionTarget : FunctionPrototype [0..*]
- functionTarget_path : FunctionPrototype [0..*] {ordered}
- hwTarget : HardwareComponentPrototype [0..*]
- hwTarget_path : HardwareComponentPrototype [0..*] {ordered, composite}
- target : NamedElement (from UML) [1]
- /type : ErrorModelType [1] {readOnly}
  {derived from UML::TypedElement::type}

Semantics

An ErrorModelPrototype represents a unique compositional occurrence of the ErrorModelType that types it in the containing ErrorModelType.

Constraints

No additional constraints

Notation

- Icon: Serialized 

19.3.5 «ErrorModelType» (from ErrorModel)

Generalizations

- TraceableSpecification

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.

Extensions

- Class (from UML)

Properties

- /externalFault : FaultInPort [0..*] {readOnly}
- /failure : FailureOutPort [0..*] {readOnly}
- /faultFailureConnector : FaultFailurePropagationLink [0..*] {readOnly}
  The links for the error propagations between subordinate error models. {derived from UML::StructuredClassifier::ownedConnector}
- genericDescription : String = NA [1]
- hwTarget : HardwareComponentType [0..*]
- /internalFault : InternalFaultPrototype [0..*] {readOnly}
- /part : ErrorModelPrototype [0..*] {readOnly}

{derived from UML::Classifier::attribute}

- /processFault : ProcessFaultPrototype [0..*] {readOnly}
- target : FunctionType [0..*]

Associations

- errorBehaviorDescription : ErrorBehavior [1..*]
  See ErrorBehavior.owner

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 thus has a direct propagation to all externalFailures and cannot be masked.

Constraints

No additional constraints

Constraints in natural language

An ErrorModelType without part shall have one errorBehaviorDescription

Notation

- Icon: Serialized 

19.3.6 «FailureOutPort» (from ErrorModel)

Generalizations

- FaultFailurePort

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.

Extensions

No direct extensions

Properties

No additional properties

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.

Constraints

No additional constraints

Constraints in natural language

[1] The direction of the nominal port must be out.

Notation

- Icon: Serialized 

19.3.7 «FaultFailurePort» (from ErrorModel) {abstract}

Generalizations

- Anomaly

Description

No additional description

Extensions

- Port (from UML)

Properties

- functionTarget : FunctionPort [0..*]
- functionTarget_path : FunctionPrototype [0..*] {ordered}
- hwTarget : HardwarePin [0..*]
- hwTarget_path : HardwareComponentPrototype [0..*] {ordered}

Semantics

No additional semantics

Constraints

No additional constraints

19.3.8 «FaultFailurePropagationLink» (from ErrorModel)

Generalizations

- EAElement

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.

Extensions

- Connector (from UML)

Properties

- /fromPort : FaultFailurePort [1] {readOnly}
- fromPort_path : ErrorModelPrototype [0..*] {ordered}
- immediatePropagation : Boolean = true [1]
- /toPort : FaultFailurePort [1] {readOnly}
- toPort_path : ErrorModelPrototype [0..*] {ordered}

Semantics

The FaultFailurePropagationLink defines a Failure propagation path, from the fromPort on one error model to the toPort of another error model.

Constraints

No additional constraints

Constraints in natural language

[1] Only compatible fromPort-toPort pairs may be connectedNo additional constraints

[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.

19.3.9 «FaultInPort» (from ErrorModel)

Generalizations

- FaultFailurePort

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.

Extensions

No direct extensions

Properties

No additional properties

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.

Constraints

No additional constraints

Constraints in natural language

[1] The direction of the nominal port must be in.

Notation

- Icon: Serialized 

19.3.10 «InternalFaultPrototype» (from ErrorModel)

Generalizations

- Anomaly

Description

The InternalFault metaclass represents the particular internal conditions of the target component/system that are of particular concern for its fault/failure definition.

Extensions

- Class (from UML)
- Property (from UML)
- Event (from UML)

Properties

No additional properties

Semantics

The system anomaly represented by an InternalFault, which when activated, can cause errors and failures of the target element.

Constraints

No additional constraints

19.3.11 «ProcessFaultPrototype» (from ErrorModel)

Generalizations

- Anomaly

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.).

Extensions

- Property (from UML)
- Event (from UML)

Properties

No additional properties

Semantics

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.).

Constraints

No additional constraints

20 EAST-ADL extensions for SafetyConstraints

This section contains the UML-profile specification, specifying stereotypes in the UML-profile, defined from the domain model classes in the SafetyConstraints package. It includes specification details for each stereotype. If the stereotype has properties, which may be referred to as tag definitions, or if the stereotype has constraints, this section also includes specification details for these properties and constraints.

20.1 Overview

No overview

20.2 Profile diagrams

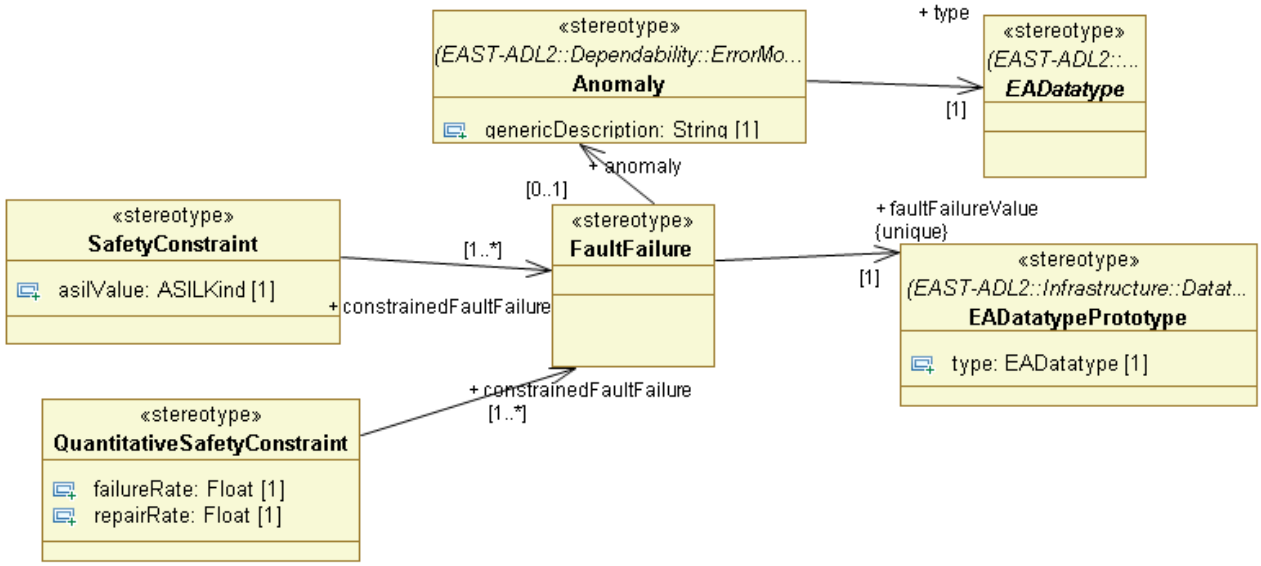

Figure 21. SafetyConstraints

20.3 Detailed description of UML profile elements

20.3.1 ASILKind (from SafetyConstraints)

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.

Semantics

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

Constraints

No additional constraints

20.3.2 «FaultFailure» (from SafetyConstraints)

Generalizations

- TraceableSpecification

Description

The FaultFailure represents a certain fault or failure on its referenced Anomaly. The faultFailureValue specifies the value of the Anomaly that the FaultFailure corresponds to, i.e. one of the possible values of the Anomaly.

Extensions

- Class (from UML)

Properties

- anomaly : Anomaly [0..1]
- faultFailureValue : String [1]

Semantics

A FaultFailure is defined as a certain value, faultFailureValue, occurring at the referenced Anomaly.

Constraints

No additional constraints

20.3.3 «QuantitativeSafetyConstraint» (from SafetyConstraints)

Generalizations

- TraceableSpecification

Description

The QuantitativeSafetyConstraint metaclass represents the quantitative integrity constraints on a fault or failure. Thus, the system has 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.

Extensions

- Class (from UML)

- Constraint (from UML)

Properties

- constrainedFaultFailure : FaultFailure [1..*]
- failureRate : Float [1]
- repairRate : Float [1]

Semantics

A QuantitativeSafetyConstraint provides information about the probabilistic estimates of target faults/failures, further specified by the failureRate and repairRate attribute.

Constraints

No additional constraints

20.3.4 «SafetyConstraint» (from SafetyConstraints)

Generalizations

- TraceableSpecification

Description

The SafetyConstraint metaclass represents the qualitative integrity constraints on a fault or failure. Thus, the system has 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.

Extensions

- Class (from UML)
- Constraint (from UML)

Properties

- asilValue : ASILKind [1]
- constrainedFaultFailure : FaultFailure [1..*]

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.

Constraints

No additional constraints

21 EAST-ADL extensions for SafetyRequirement

This subprofile defines a set of stereotypes concerning the definition of safety requirements inked to ISO26262 norm.

21.1 Overview

This subprofile defines a set of stereotypes concerning the definition of safety requirements linked to the ISO26262 norm.

21.2 Profile diagrams

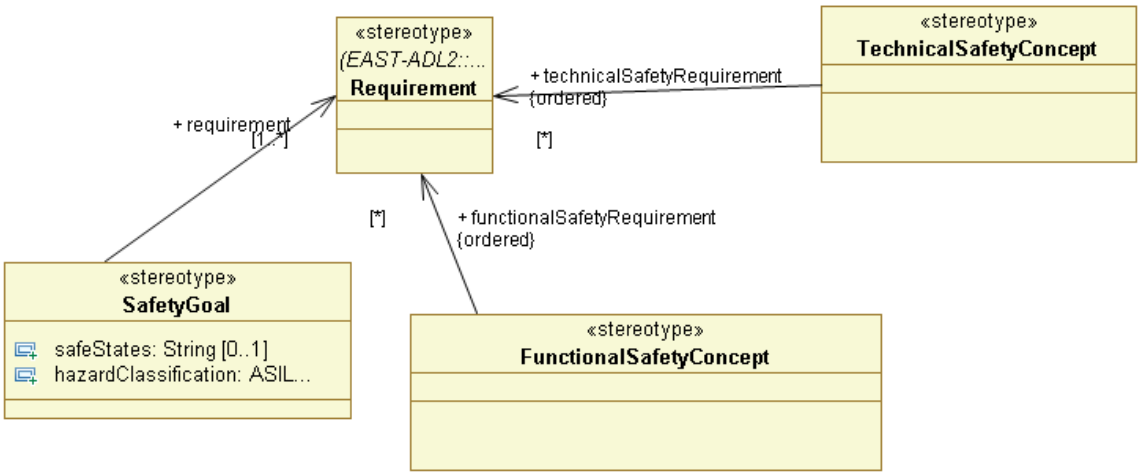

Figure 22. SafetyRequirement

21.3 Detailed description of UML profile elements

21.3.1 «FunctionalSafetyConcept» (from SafetyRequirement)

Generalizations

- RequirementsContainer

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.

Extensions

No direct extensions

Properties

- functionalSafetyRequirement : Requirement [0..*] {ordered}

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.

Constraints

No additional constraints

Constraints in natural language

[1] Contained functionalSafetyRequirements shall not be of type SafetyGoal.

21.3.2 «SafetyGoal» (from SafetyRequirement)

Generalizations

- EAElement

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.

Extensions

- Class (from UML)

Properties

- derivedFrom : HazardousEvent [1..*]
- hazardClassification : ASILKind [1]
- requirement : Requirement [1..*]
- safeModes : Mode [0..*]
- safeStates : String [0..1]
  For every safety goal, 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 safety goal.

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.

Constraints

No additional constraints

21.3.3 «TechnicalSafetyConcept» (from SafetyRequirement)

Generalizations

- RequirementsContainer

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.

Extensions

No direct extensions

Properties

- technicalSafetyRequirement : Requirement [0..*] {ordered}
  technicalSafety Requirements.

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.

Constraints

No additional constraints

22 EAST-ADL extensions for SafetyCase

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 enough 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.

22.1 Overview

No overview

22.2 Profile diagrams

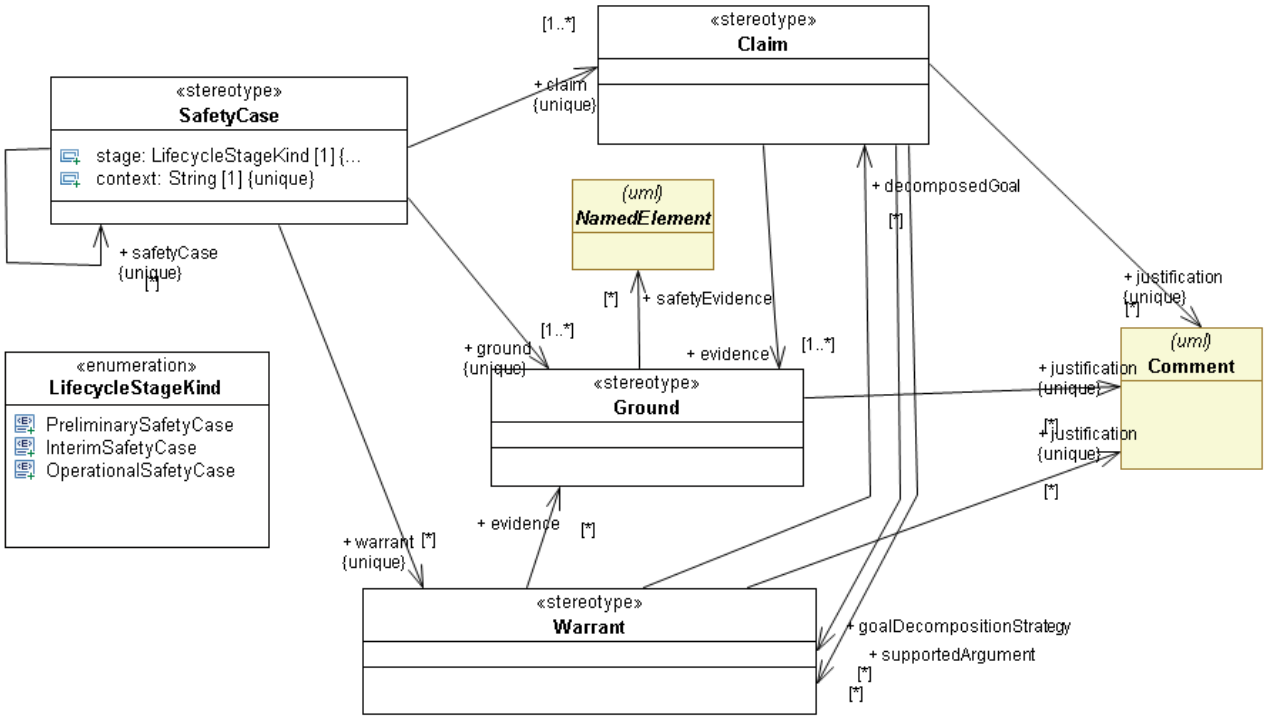

Figure 23. SafetyCase

22.3 Detailed description of UML profile elements

22.3.1 «Claim» (from SafetyCase)

Generalizations

- TraceableSpecification

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.

Extensions

- Class (from UML)
- Comment (from UML)

Properties

- evidence : Ground [1..*]
- goalDecompositionStrategy : Warrant [0..*]
- justification : Comment (from UML) [0..*]
- safetyRequirement : TraceableSpecification [0..*]
  Safety requirements and objectives in system model.
- supportedArgument : Warrant [0..*]

Semantics

Goal-based development provides the claim what should be achieved.

Goal is what the argument must show to be true.

Constraints

No additional constraints

22.3.2 «Ground» (from SafetyCase)

Generalizations

- TraceableSpecification

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.

Extensions

- NamedElement (from UML)
- Class (from UML)
- Comment (from UML)

Properties

- justification : Comment (from UML) [0..*]
- safetyEvidence : NamedElement (from UML) [0..*]
  Safety evidence in system model.

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 felt 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.

Constraints

No additional constraints

22.3.3 LifecycleStageKind (from SafetyCase)

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
- OperationalSafetyCase
- PreliminarySafetyCase

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.

Constraints

No additional constraints

22.3.4 «SafetyCase» (from SafetyCase)

Generalizations

- TraceableSpecification

Description

SafetyCase represents a safety case that communicates a clear, comprehensive and defensible argument that a system is acceptable 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.

Extensions

- Class (from UML)

Properties

- claim : Claim [1..*]
- context : String [1]
- ground : Ground [1..*]

- safetyCase : SafetyCase [0..*]
  Attached SafetyCases
- stage : LifecycleStageKind [1]
- warrant : Warrant [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

22.3.5 «Warrant» (from SafetyCase)

Generalizations

- TraceableSpecification

Description

Warrant represents argumentation of the facts to the Claim in general ways.

The Warrant entity has associations to the decomposed goals and to the evidences for the SafetyCase.

Extensions

- Class (from UML)
- Comment (from UML)

Properties

- decomposedGoal : Claim [0..*]
- evidence : Ground [0..*]
- justification : Comment (from UML) [0..*]

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.

Constraints

No additional constraints

Part VIII Generic Constraints

23 EAST-ADL extensions for GenericConstraints

This section contains the UML-profile specification, specifying stereotypes in the UML-profile, defined from the domain model classes in the GenericConstraints package. It includes specification details for each stereotype. If the stereotype has properties, which may be referred to as tag definitions, or if the stereotype has constraints, this section also includes specification details for these properties and constraints.

23.1 Overview

No overview

23.2 Profile diagrams

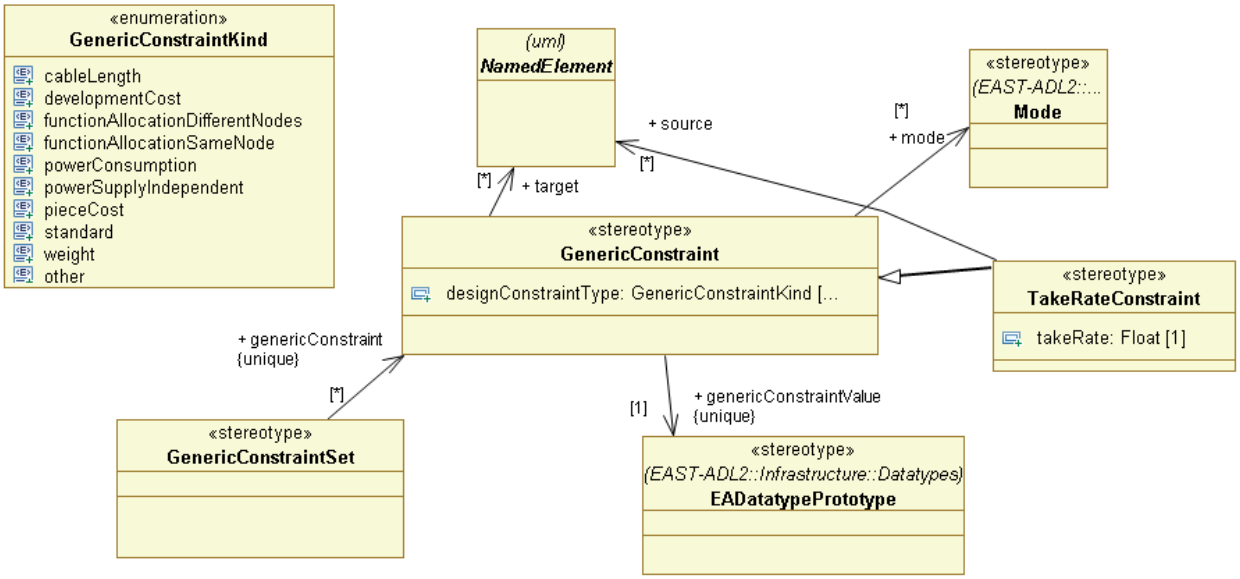

Figure 24. GenericConstraints

23.3 Detailed description of UML profile elements

23.3.1 «GenericConstraint» (from GenericConstraints)

Generalizations

- TraceableSpecification

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 genericConstraintValue could be "5 meters" (value of a numerical datatype with unit "meters").

Extensions

- Constraint (from UML)
- Class (from UML)
- NamedElement (from UML)

Properties

- designConstraintType : GenericConstraintKind [0..1]
  The specific type of design constraint.
- genericConstraintValue : String [1]
- mode : Mode [0..*]
- target : NamedElement (from UML) [0..*]

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.

Constraints

No additional constraints

Notation

- Icon: Serialized 

23.3.2 GenericConstraintKind (from GenericConstraints)

Generalizations

None

Description

Enumeration for different type of constraints.

Enumeration Literals

- cableLength
- developmentCost
- functionAllocationDifferentNodes
- functionAllocationSameNode
- other
- pieceCost
- powerConsumption
- powerSupplyIndependent
- standard
- weight

Semantics

No additional semantics

Constraints

No additional constraints

23.3.3 «GenericConstraintSet» (from GenericConstraints)

Generalizations

- Context

Description

The collection of generic constraints. This collection can be done across the EAST-ADL abstraction levels.

Extensions

- Package (from UML)
- Class (from UML)

Properties

- genericConstraint : GenericConstraint [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

23.3.4 «TakeRateConstraint» (from GenericConstraints)

Generalizations

- GenericConstraint

Description

No additional description

Extensions

- NamedElement (from UML)

Properties

- source : NamedElement (from UML) [0..*]
- takeRate : Float [1]

Semantics

No additional semantics

Constraints

No additional constraints

Part IX Infrastructure

24 EAST-ADL extensions for Datatypes

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.

24.1 Overview

No overview

24.2 Profile diagrams

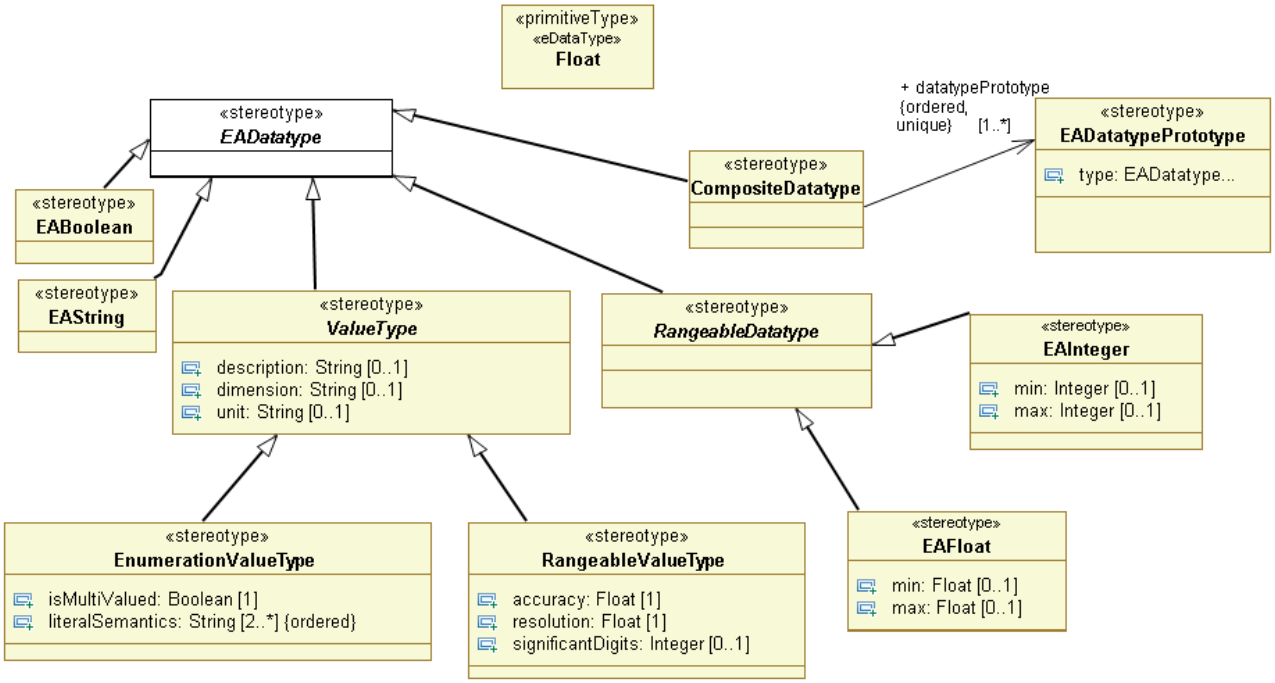

Figure 25. Datatypes

24.3 Detailed description of UML profile elements

24.3.1 «CompositeDatatype» (from Datatypes)

Generalizations

- EADatatype

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)).

Extensions

No direct extensions

Properties

- datatypePrototype : EADatatypePrototype [1..*] {ordered}

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).

Constraints

No additional constraints

24.3.2 «EABoolean» (from Datatypes)

Generalizations

- EADatatype

Description

No additional description

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

24.3.3 «EADatatype» (from Datatypes) {abstract}

Generalizations

- TraceableSpecification

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 on implementation level. The implementation representation is defined on 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.

Extensions

- DataType (from UML)

Properties

No additional properties

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.

Constraints

No additional constraints

Constraints in natural language

[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.

Notation

- Icon: Serialized 

24.3.4 «EADatatypePrototype» (from Datatypes)

Generalizations

- EAElement

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. The EADatatypePrototype is also used to represent argument and return values of operations or to represent a parameter.

Extensions

- Property (from UML)
- Parameter (from UML)

Properties

- type : EADatatype [1]

Semantics

The EADatatypePrototype represents a typed variable. It acts as an occurrence of a datatype.

Constraints

No additional constraints

24.3.5 «EAFloat» (from Datatypes)

Generalizations

- RangeableDatatype

Description

No additional description

Extensions

No direct extensions

Properties

- max : Float [0..1]
- min : Float [0..1]

Semantics

No additional semantics

Constraints

No additional constraints

24.3.6 «EAInteger» (from Datatypes)

Generalizations

- RangeableDatatype

Description

No additional description

Extensions

No direct extensions

Properties

- max : Integer [0..1]
- min : Integer [0..1]

Semantics

No additional semantics

Constraints

No additional constraints

24.3.7 «EAString» (from Datatypes)

Generalizations

- EADatatype

Description

No additional description

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

24.3.8 «Enumeration» (from Datatypes)

Generalizations

- EADatatype

Description

No additional description

Extensions

No direct extensions

Properties

- literal : EnumerationLiteral [2..*] {composite}

Semantics

No additional semantics

Constraints

No additional constraints

24.3.9 «EnumerationLiteral» (from Datatypes)

Generalizations

- EAElement

Description

No additional description

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

~~24.3.8~~24.3.10 «EnumerationValueType» (from Datatypes)

Generalizations

- ValueType

Description

The EnumerationValueType is a specific ValueType applicable for Enumerations. It provides the possibility to describe semantics of the baseEnumeration's literals and the information, if multiple values of the baseEnumeration may be selected or not.

Extensions

- Enumeration (from UML)

Properties

- isMultiValued : Boolean [1]
- literalSemantics : String [2..*] {ordered}

Semantics

The EnumerationValueType adds the ability to describe semantics of the baseEnumeration's literals and if multiple values of the baseEnumeration may be selected or not.

Constraints

No additional constraints

~~24.3.9~~24.3.11 Float (from Datatypes)

Generalizations

None

Description

An instance of Float is an element from the set of real numbers. The value must comply with IEEE 754 and is limited to what can be expressed by a 64 bit binary representation.

Type

uml:PrimitiveType

Properties

No additional properties

Semantics

Float has the semantics of the Float datatype as defined by IEEE Standard for Floating-Point Arithmetic (IEEE 754).

Constraints

No additional constraints

~~24.3.10~~24.3.12 «RangeableDatatype» (from Datatypes) {abstract}

Generalizations

- EADatatype

Description

The stereotype RangeableDatatype reflects numeric datatypes that may have a range (between a minimal and a maximal value). An example for a RangeableDatatype is the Celsius temperature scale with minValue = -273.15.

Extensions

No direct extensions

Properties

No additional properties

Semantics

The stereotype RangeableDatatype reflects numeric datatypes that may have a range (between a minimal and a maximal value).

Constraints

No additional constraints

~~24.3.11~~24.3.13 «RangeableValueType» (from Datatypes)

Generalizations

- ValueType

Description

The RangeableValueType is a specific ValueType applicable for RangeableDatatypes. It provides the possibility to describe the accuracy, resolution, and the significant digits of the baseRangeable datatypes.

Extensions

No direct extensions

Properties

- accuracy : Float [1]
- resolution : Float [1]
- significantDigits : Integer [0..1]

Semantics

The RangeableValueType adds the ability to describe the accuracy, resolution, and the significant digits of the baseRangeable datatype.

Constraints

No additional constraints

~~24.3.12~~24.3.14 «ValueType» (from Datatypes) {abstract}

Generalizations

- EADatatype

Description

From SysML:

A ValueType defines types of values that may be used to express information about a system, but cannot be identified as the target of any reference. Since a value cannot be identified except by means of the value itself, each such value within a model is independent of any other, unless other forms of constraints are imposed. Value types may be used to type properties, operation parameters, or potentially other elements within SysML. SysML defines ValueType as a stereotype of UML DataType to establish a more neutral term for system values that may never be given a concrete data representation. For example, the SysML "Real" ValueType expresses the mathematical concept of a real number, but does not impose any restrictions on the precision or scale of a fixed or floating-point representation that expresses this concept. More specific value types can define the concrete data representations that a digital computer can process, such as conventional Float, Integer, or String types. SysML ValueType adds an ability to carry a unit of measure or dimension associated with the value. A dimension is a kind of quantity that may be stated in terms of defined units, but does not restrict the selection of a unit to state the value. A unit is a particular value in terms of which a quantity of the same dimension may be expressed. A SysML ValueType may define its own properties and/or operations, just as for a UML DataType.

Extensions

No direct extensions

Properties

- ~~description~~ semantics : String [0..1]

- dimension : String [0..1]
- unit : String [0..1]

Semantics

The abstract ValueType defines types of values that may be used to express information about a system. The ValueType adds an ability to carry a description, a dimension associated with the value, and a unit of measure. A dimension is a kind of quantity that may be stated in terms of defined units, but does not restrict the selection of a unit to state the value. A unit is a particular value in terms of which a quantity of the same dimension may be expressed.

Logical and physical datatypes cannot be distinguished on the type. The context (e.g., EnvironmentModel or FunctionalAnalysisArchitecture) decides if a speed datatype is physical or logical. On AnalysisLevel or DesignLevel, physical datatypes shall not be interpreted in the implementation sense as this would include int32, coding formula, etc.

Constraints

No additional constraints

25 EAST-ADL extensions for Elements

This section contains the UML-profile specification, specifying stereotypes in the UML-profile, defined from the metaclasses in the Infrastructure::Elements subprofile. It includes specification details for each stereotype. If the stereotype has properties, which may be referred to as tag definitions, or if the stereotype has constraints, this section also includes specification details for these properties and constraints.

25.1 Overview

The Infrastructure::Elements subprofile of EAST-ADL defines general-purpose relationship constructs that may be used to model dependencies between structural constructs.

The purpose of the stereotypes in this subprofile is to specify rigorously ("formally") the various relationships that may exist between basic constructs.

25.2 Profile diagrams

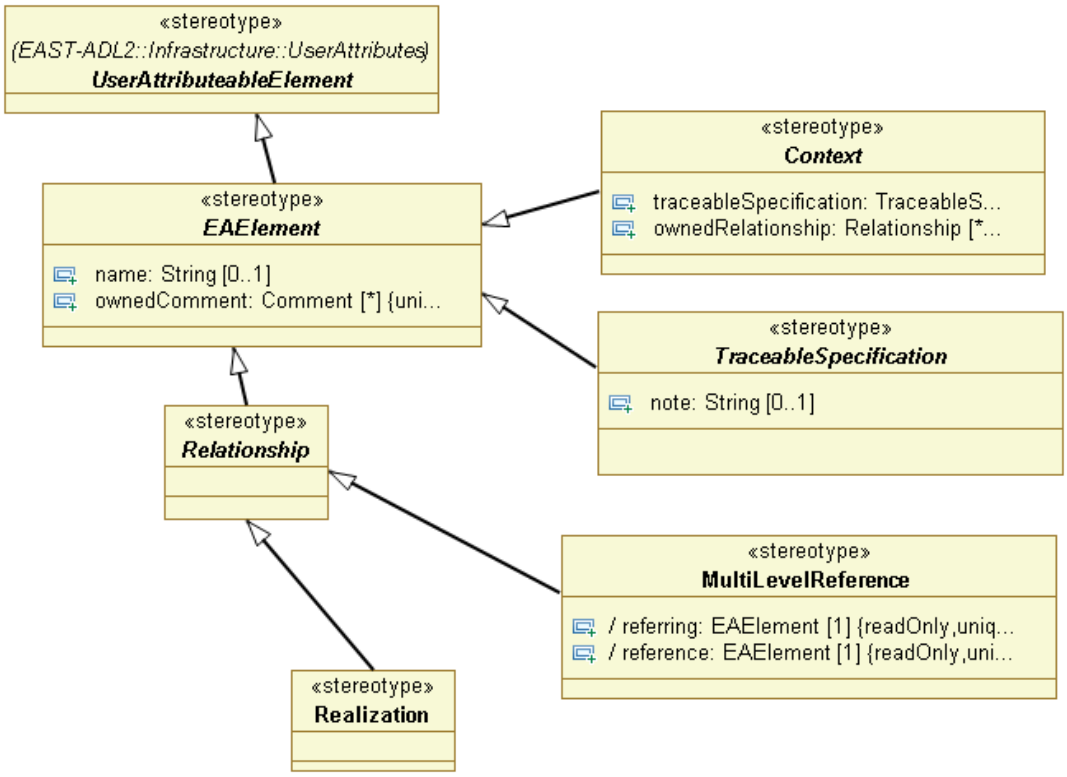

Figure 26. Elements

25.3 Detailed description of UML profile elements

25.3.1 «Comment» (from Elements)

Generalizations

None

Description

No additional description

Extensions

- Comment (from UML)

Properties

- body : String [1]

Semantics

No additional semantics

Constraints

No additional constraints

~~25.3.1~~25.3.2 «Context» (from Elements) {abstract}

Generalizations

- EAElement

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.

Extensions

No direct extensions

Properties

- ownedRelationship : Relationship [0..*]
  Relationship(s) associated to this context.
- traceableSpecification : TraceableSpecification [0..*]
  Traceable specification(s) allocated to this context.

Semantics

See Relationship and TraceableSpecification.

Constraints

No additional constraints

~~25.3.2~~25.3.3 «EAElement» (from Elements) {abstract}

Generalizations

~~UserAttributeableElement~~
None

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.

Extensions

- NamedElement ~~Comment~~ (from UML)

Properties

- name : String [0..1]
  The name of the EAElement.
- ownedComment : Comment ~~(from UML)~~ [0..*]
  ~~Comment(s) owned in this context.~~

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.

Constraints

No additional constraints

25.3.4 «EAPackage» (from Elements)

Generalizations

- EAElement

Description

No additional description

Extensions

- Package (from UML)

Properties

- element : EAPackageableElement [0..*] {composite}
- subPackages : EAPackage [0..*] {composite}

Semantics

No additional semantics

Constraints

No additional constraints

25.3.5 «EAPackageableElement» (from Elements) {abstract}

Generalizations

- EAElement

Description

No additional description

Extensions

- PackageableElement (from UML)

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

~~25.3.3~~25.3.6 «MultiLevelReference» (from Elements)

Generalizations

- Relationship

Description

MultiLevelReference gives the possibility to establish reference links (Multi-Level Concept) between model elements.

Extensions

- Dependency (from UML)

Properties

- /reference : EAElement [1] {readOnly}
  Referencing the source element of a Multi-Level reference link.
- /referring : EAElement [1] {readOnly}
  Referencing the target element of a Multi-Level reference link.

Semantics

No additional semantics

Constraints

No additional constraints

25.3.7 «Rationale» (from Elements)

Generalizations

- Rationale (from SysML::ModelElements)

Description

No additional description

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

~~25.3.4~~25.3.8 «Realization» (from Elements)

Generalizations

- Relationship

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.

Extensions

- NamedElement (from UML)
- Realization (from UML)
- NamedElement (from UML)

Properties

- /realized : EAElement [1..*] {readOnly}
  The set of ADL entities, which are realized by the set of client ADL entities or realized by the set of client AUTOSAR elements.

  {derived from UML::DirectedRelationship::target}
- realized_path : NamedElement (from UML) [0..*] {ordered}
- /realizedBy : NamedElement (from UML) [0..*] {readOnly}
  The set of client ADL entities, realizing the set of supplier ADL entities.

  {derived from UML::Dependency::client}
- realizedBy_path : NamedElement (from UML) [0..*] {ordered}

Semantics

The modification of the supplier realized element impact the realizing client entity. The Realization metaclass implies the semantics that the realizing client is not complete, without the supplier.

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~25.3.5~~25.3.9 «Relationship» (from Elements) {abstract}

Generalizations

- EAElement

Description

The Relationship is an abstract metaclass which represents a relationship between arbitrary elements.

Extensions

No direct extensions

Properties

No additional properties

Semantics

In many cases, Contexts such as functions and sensors need to have requirements and other specification elements allocated them. In other cases, the relation 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 architecture. These relationships are modeled by concrete specializations of Relationship.

See Context and TraceableSpecification.

Constraints

No additional constraints

~~25.3.6~~25.3.10 «TraceableSpecification» (from Elements) {abstract}

Generalizations

- EAPackageableElement
- ~~EAElement~~

Description

The TraceableSpecification is an abstract metaclass which is used to allow its specializations to be allocated to a Context.

Extensions

No direct extensions

Properties

- note : String [0..1]

Semantics

TraceableSpecification is specialized by requirements, test cases and other specifications, that there by can be allocated to a Context, for example to a sensor or to an entire HW architecture.

See Context and Relationship.

Constraints

No additional constraints

26 EAST-ADL extensions for UserAttributes

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 EAElement 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 within UATemplates.

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 perfectly suitable to define 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 case of requirements the context in which the available user attributes are defined is different: here the container of the requirements is the point where user attribute definitions are store and these are then applicable only within this container.

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.

The mechanism of user attributes was optimized for flexibility and simplicity. In particular, the actual attributes attached to an element and/or their values may well conflict the attribute definitions in effect for this element. For example, it is perfectly legal to not provide an attribute value if an attribute definition was specified or, the other way round, to provide a value for an undefined attribute. The attribute definitions are merely meant as a guideline for the engineer and as a basis for optionally checking if all attribute values are correct with respect to attribute definitions (by way of appropriate tool support). With this conception of attribute values and definitions, many intricacies and difficult situations during the creation and evolution of a model are circumvented and complex interdependencies between parts of the model are avoided. For example, it is made sure that a model and all its user attribute values can be safely viewed and edited even if the attribute definitions (i.e. UATemplates) for the model are temporarily unavailable or permanently lost.

26.1 Overview

The stereotypes defined in this subprofile provide a set of constructs to help user define their own attributes. The core construct in EAST-ADL, the EAElement, inherits from UserAttributableElement stereotype so that virtually any types of EAST-ADL entities might be enhanced with user-defined attributes. Of course in a UML model one is allowed to add attributes to the classes and UML elements on which stereotypes are applied, yet this mechanism enables to distinguish between attributes meant to be interpreted as compliant with EAST-ADL language and other if any.

26.2 Profile diagrams

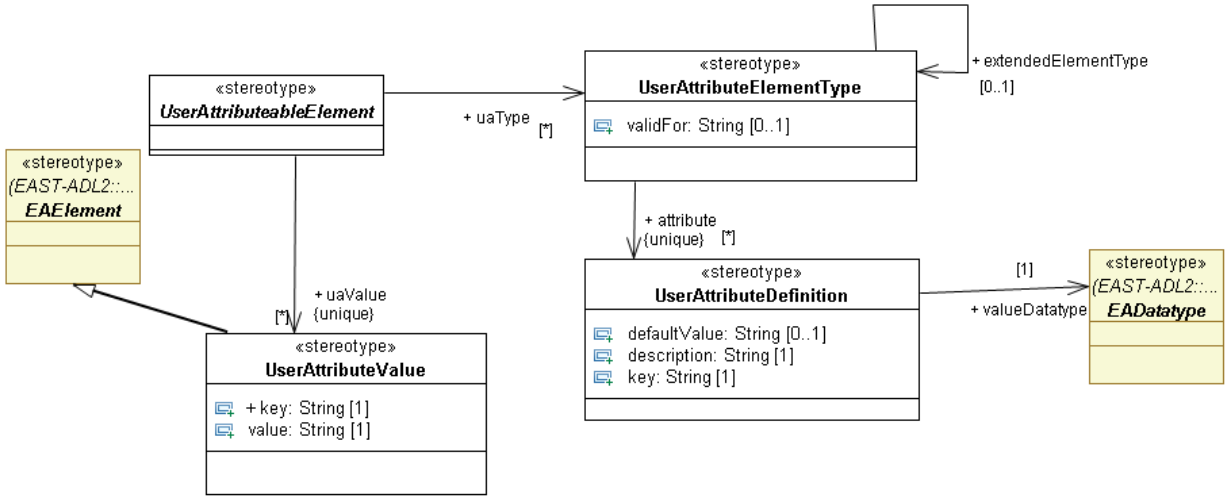

Figure 27. UserAttributes

26.3 Detailed description of UML profile elements

26.3.1 «UserAttributeableElement» (from UserAttributes) ~~{abstract}~~

Generalizations

None

Description

UserAttributableElement represents an element to which user attributes can be attached. This is done by way of UserAttributeValues (see association 'uaValues'). What user attributes a certain element should be supplied with can be defined beforehand with UserAttributeDefinitions which are organized in UserAttributeElementTypes (see association 'uaTypes').

IMPORTANT: It is technically possible and legal to attach any key/value pair, even if this is in conflict with the attribute definitions of the UserAttributeElementTypes of this UserAttributeableElement (as defined by association 'uaTypes'). All implementations of this information model must expect such attribute definition violations. The reason for this is that (1) the attribute definitions and the types they define for the attributes are only meant as a guideline for working with user attributes on the modeling level, not as an implementation level type system and (2) this convention avoids a multitude of intricate problems when editing a model's user attribute definitions or values, which significantly simplifies implementation.

Extensions

- NamedElement (from UML)
- NamedElement (from UML)

~~No direct extensions~~

Properties

- attributedElement : NamedElement (from UML) [1]
- uaType : UserAttributeElementType [0..*]
  The UAElementTypes of this user attirbuteable element.

It is possible to provide more than one type. In that case, the UserAttributeDefinitions of all UAElementTypes apply. If there are several attribute definitions with an identical 'key', then the corresponding user attribute will be applied only once.

- uaValue : UserAttributeValue [0..*]
  The user attribute values, i.e. key-value pairs, which are attached to this element.

Semantics

No additional semantics

Constraints

No additional constraints

26.3.2 «UserAttributeDefinition» (from UserAttributes)

Generalizations

- EAElement

Description

UserAttributeDefinition represents a user attribute, i.e. it states that all UserAttributeableElements of a certain UserAttributeElementType are to be attached with an attribute identified by 'key'. For example, it can be specified that certain elements should be amended with an attribute "Status".

Extensions

- Class (from UML)
- Property (from UML)

Properties

- defaultValue : String [0..1]
- ~~description : String [1]~~
  ~~A description statement.~~
- ~~key : String [1]~~
  ~~A unique identifier for the user attribute.~~

  ~~Whenever interoperability with third parties is required a URL naming scheme should be used, similar as for packages in the Java programming language. For example, a company with a home page URL of www.example.com could use the key com.example.Status for a status attribute.~~
- ~~valueDatatype~~ type : EADatatype [1]

Semantics

No additional semantics

Constraints

No additional constraints

26.3.3 «UserAttributeElementType» (from UserAttributes)

Generalizations

- EAElement

Description

UserAttributeElementType represents a certain, user-defined type of user attributeable elements. With such a type, one or more user attributes can be defined for all user attributeable elements of that type.

For example, engineers at Volkswagen could create a UserAttributeElementType called "VWFunction" with a single user attribute definition. That way, all FunctionTypes for which "VWFunction" is defined as the UserAttributeElementType via association uaType will have the corresponding user attribute.

User attribute element types can be compared to stereotypes in UML2, but are less rigidly defined.

Extensions

- Class (from UML)

Properties

- attribute : UserAttributeDefinition [0..*]
  The attributes defined for this type.

  Note that also inherited attribute definitions need to be taken into account.
- extendedElementType : UserAttributeElementType [0..1]
  The UAElementTypes this type is inheriting from.

  When UAElementType ET2 inherits from type ET1, then this means that all attributes defined for ET1 by way of UserAttributeDefinitions are available whenever ET2 is specified as the type of a user attributeable element (in addition to those directly defined in ET2). This includes UserAttributeDefinitions which ET1 itself may inherit from other types.
- validFor : String [0..1]

Semantics

No additional semantics

Constraints

No additional constraints

26.3.4 «UserAttributeValue» (from UserAttributes)

Generalizations

- EAElement

Description

UserAttributeValue represents a specific value for a certain user attribute. User attributes are simple key/value pairs which can be attached to all UserAttributeableElements. Each user attribute is identified by a globally unique key.

In principle, there is no restriction which user attributes, i.e. keys, may be attached to a particular element and what strings may be used as value (cf. class UserAttributeableElement). However, user attribute definitions can be used to define a set of legal values for a particular key (see class UserAttributeDefinition) and user attribute element types can be used to state what attributes, i.e. keys, may or should be attached to elements of certain types (cf. class UserAttributeElementType).

The actual value is captured in attribute 'value' and is always represented as a string.

Extensions

- Class (from UML)
- Property (from UML)

Properties

- ~~key : String [1]~~
  ~~The unique identifier of the attribute for which this UserAttributeValue provides a value for.~~
- definition : UserAttributeDefinition [0..1]
- value : String [1]

Semantics

No additional semantics

Constraints

No additional constraints

Part X Annexes

27 EAST-ADL extensions for Needs

This section contains the UML-profile specification, specifying stereotypes in the UML-profile, defined from the domain model classes in the Needs package. It includes specification details for each stereotype. If the stereotype has properties, which may be referred to as tag definitions, or if the stereotype has constraints, this section also includes specification details for these properties and constraints.

27.1 Overview

No overview

27.2 Profile diagrams

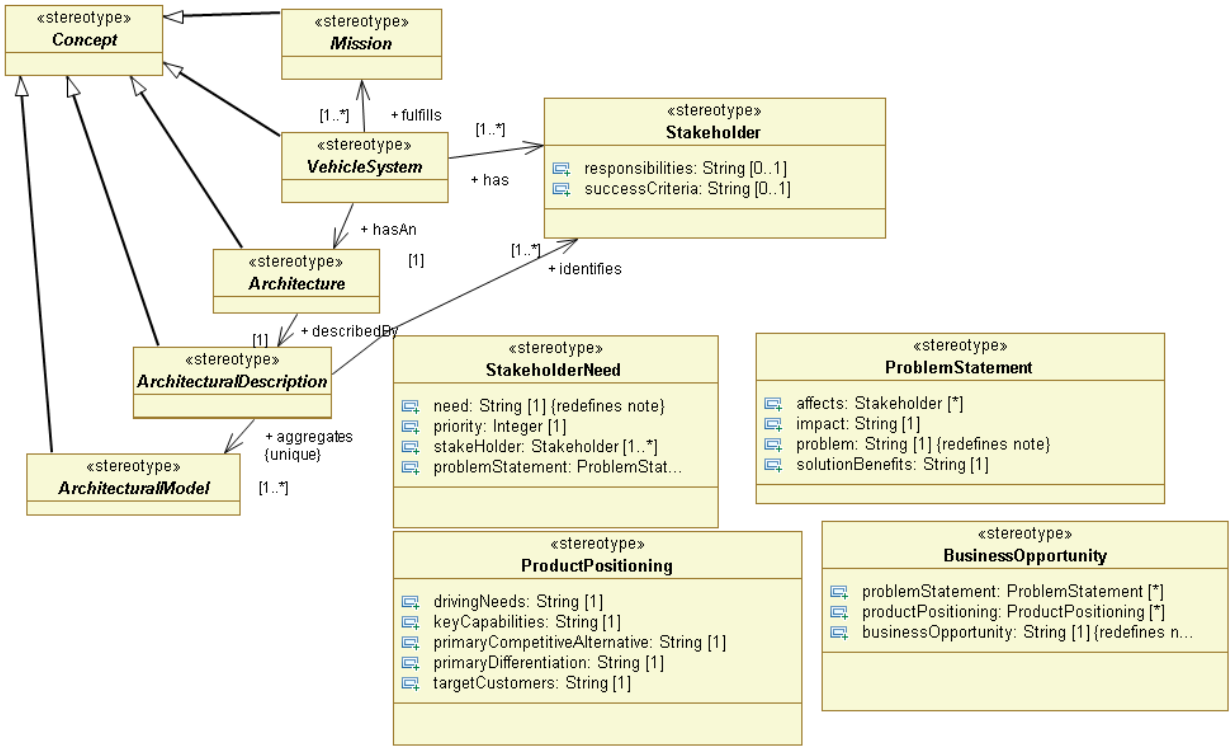

Figure 28. Needs

27.3 Detailed description of UML profile elements

27.3.1 «ArchitecturalDescription» (from Needs) ~~{abstract}~~

Generalizations

- Concept

Description

A collection of products to document an architecture. [IEEE 1471]

Extensions

No direct extensions

Properties

- aggregates : ArchitecturalModel [1..*]
- identifies : Stakeholder [1..*]

Semantics

No additional semantics

Constraints

No additional constraints

27.3.2 «ArchitecturalModel» (from Needs) ~~{abstract}~~

Generalizations

- Concept

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]

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

27.3.3 «Architecture» (from Needs) ~~{abstract}~~

Generalizations

- Concept

Description

The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution. [IEEE 1471]

Extensions

No direct extensions

Properties

- describedBy : ArchitecturalDescription [1]

Semantics

No additional semantics

Constraints

No additional constraints

27.3.4 «BusinessOpportunity» (from Needs)

Generalizations

- TraceableSpecification

Description

The business opportunity represents a brief description of the business opportunity being met by developing the EE-System which establishes traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

Extensions

- Class (from UML)

Properties

- businessOpportunity : String [1]
- problemStatement : ProblemStatement [0..*]
- productPositioning : ProductPositioning [0..*]

Semantics

No additional semantics

Constraints

No additional constraints

27.3.5 «Concept» (from Needs) {abstract}

Generalizations

- EAElement

~~None~~

Description

An abstract or general idea inferred or derived from specific instances. [Webster]

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

27.3.6 «Mission» (from Needs) ~~{abstract}~~

Generalizations

- Concept

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]

Extensions

No direct extensions

Properties

No additional properties

Semantics

No additional semantics

Constraints

No additional constraints

27.3.7 «ProblemStatement» (from Needs)

Generalizations

- TraceableSpecification

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

Extensions

- Class (from UML)

Properties

- affects : Stakeholder [0..*]
- impact : String [1]
- problem : String [1]
- solutionBenefits : String [1]

Semantics

No additional semantics

Constraints

No additional constraints

27.3.8 «ProductPositioning» (from Needs)

Generalizations

- TraceableSpecification

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.

Extensions

- Class (from UML)

Properties

- drivingNeeds : String [1]
- keyCapabilities : String [1]
- primaryCompetitiveAlternative : String [1]
- primaryDifferentiation : String [1]
- targetCustomers : String [1]

Semantics

No additional semantics

Constraints

No additional constraints

27.3.9 «Stakeholder» (from Needs)

Generalizations

- TraceableSpecification

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]

Extensions

- Class (from UML)

Properties

- responsibilities : String [0..1]
- successCriteria : String [0..1]

Semantics

No additional semantics

Constraints

No additional constraints

27.3.10 «StakeholderNeed» (from Needs)

Generalizations

- TraceableSpecification

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.

Extensions

- Class (from UML)

Properties

- need : String [1]
- priority : Integer [1]
- problemStatement : ProblemStatement [1..*]
- stakeHolder : Stakeholder [1..*]

Semantics

No additional semantics

Constraints

No additional constraints

27.3.11 «VehicleSystem» (from Needs) ~~{abstract}~~

Generalizations

- Concept

Description

A collection of components organized to accomplish a specific function or set of functions. [IEEE 1471]

Extensions

No direct extensions

Properties

- fulfills : Mission [1..*]
- has : Stakeholder [1..*]
- hasAn : Architecture [1]

Semantics

No additional semantics

Constraints

No additional constraints

28 Direct extensions used in this profile

- Actor (from UML)
- AssociationClass (from UML)
- Behavior (from UML)
- Class (from UML)
- Comment (from UML)
- Connector (from UML)
- Constraint (from UML)
- DataType (from UML)
- Dependency (from UML)
- Enumeration (from UML)
- Event (from UML)
- Extend (from UML)
- ExtensionPoint (from UML)
- Include (from UML)
- Interface (from UML)
- NamedElement (from UML)
- Operation (from UML)
- Package (from UML)
- PackageableElement (from UML)
- Parameter (from UML)
- Port (from UML)
- Property (from UML)
- Realization (from UML)
- RedefinableElement (from UML)
- UseCase (from UML)

29 Imported concepts specialized in this profile

- Block (from SysML::Blocks)
- DeriveReqt (from SysML::Requirements)
- Verify (from SysML::Requirements)
- Satisfy (from SysML::Requirements)
- FlowPort (from SysML::PortAndFlows)
- Rationale (from SysML::ModelElements)
- Requirement (from SysML::Requirements)
- Refine (from Standard)

30 Documentation of external concepts specialized in this profile

30.1.1 «Block» (from SysML::Blocks)

Generalizations

None

Description

No additional description

Extensions

- Class (from UML)

Properties

- isEncapsulated : Boolean [0..1]

Constraints

No additional constraints

Notation

- Icon: Serialized

30.1.2 «DeriveReqt» (from SysML::Requirements)

Generalizations

- Trace (from Standard)

Description

No additional description

Extensions

No direct extensions

Properties

No additional properties

Constraints

No additional constraints

Notation

- Icon: Serialized

30.1.3 «Verify» (from SysML::Requirements)

Generalizations

- Trace (from Standard)

Description

No additional description

Extensions

No direct extensions

Properties

No additional properties

Constraints

No additional constraints

Notation

- Icon: Serialized

30.1.4 «Satisfy» (from SysML::Requirements)

Generalizations

- Trace (from Standard)

Description

No additional description

Extensions

No direct extensions

Properties

No additional properties

Constraints

No additional constraints

Notation

- Icon: Serialized

30.1.5 «FlowPort» (from SysML::PortAndFlows)

Generalizations

None

Description

No additional description

Extensions

- Port (from UML)

Properties

- direction : FlowDirection = inout [1]
- /isAtomic : Boolean [1] {readOnly}
- isConjugated : Boolean [0..1]

Constraints

No additional constraints

Notation

- Icon: Serialized
- Icon: Serialized
- Icon: Serialized

- Icon: Serialized
- Icon: Serialized
- Icon: Serialized
- Icon: Serialized
- Icon: Serialized
- Icon: Serialized

30.1.6 «Rationale» (from SysML::ModelElements)

Generalizations

None

Description

No additional description

Extensions

- Comment (from UML)

Properties

No additional properties

Constraints

No additional constraints

Notation

- Icon: Serialized

~~30.1.6~~30.1.7 Requirement» (from SysML::Requirements)

Generalizations

None

Description

No additional description

Extensions

- Class (from UML)

Properties

- /derived : Requirement [0..*] {readOnly}
- /derivedFrom : Requirement [0..*] {readOnly}
- id : String [1]
- /master : Requirement [0..1] {readOnly}
- /refinedBy : NamedElement (from UML) [0..*] {readOnly}
- /satisfiedBy : NamedElement (from UML) [0..*] {readOnly}
- text : String [1]
- /tracedTo : NamedElement (from UML) [0..*] {readOnly}
- /verifiedBy : TestCase [0..*] {readOnly}

Constraints

No additional constraints

Notation

- Icon: Serialized 

~~30.1.7~~30.1.8 «Refine» (from Standard)

Generalizations

None

Description

No additional description

Extensions

- Abstraction (from UML)

Properties

No additional properties

Constraints

No additional constraints

31 External concepts typing properties in this profile

- FlowDirection (from SysML::PortAndFlows)
- Boolean
- Integer
- String
- Boolean (from UML)
- Comment (from UML)
- Integer (from UML)
- NamedElement (from UML)
- String (from UML)
- UseCase (from UML)

32 Derived and read-only properties

32.1.1 «AnalysisFunctionPrototype» (from FunctionModeling)

- /type : AnalysisFunctionType [1] {readOnly}

32.1.2 «AnalysisFunctionType» (from FunctionModeling)

- /part : AnalysisFunctionPrototype [0..*] {readOnly}

32.1.3 «ClampConnector» (from Environment)

- /port : FunctionPort [2] {readOnly}

32.1.4 «DeriveRequirement» (from Requirements)

- /derived : Requirement [1..*] {readOnly}
  The set of requirements derived from the supplier requirement.

  {derived from UML::DirectedRelationship::target}
- /derivedFrom : Requirement [1..*] {readOnly}
  The set of requirements that the client requirement are derived from.

  {derived from UML::DirectedRelationship::source}

32.1.5 «DesignFunctionPrototype» (from FunctionModeling)

- /type : DesignFunctionType [1] {readOnly}

32.1.6 «DesignFunctionType» (from FunctionModeling)

- /part : DesignFunctionPrototype [0..*] {readOnly}

32.1.7 «ErrorModelPrototype» (from ErrorModel)

- /type : ErrorModelType [1] {readOnly}
  {derived from UML::TypedElement::type}

32.1.8 «ErrorModelType» (from ErrorModel)

- /externalFault : FaultInPort [0..*] {readOnly}
- /failure : FailureOutPort [0..*] {readOnly}
- /faultFailureConnector : FaultFailurePropagationLink [0..*] {readOnly}
  The links for the error propagations between subordinate error models. {derived from UML::StructuredClassifier::ownedConnector}
- /internalFault : InternalFaultPrototype [0..*] {readOnly}
- /part : ErrorModelPrototype [0..*] {readOnly}
  {derived from UML::Classifier::attribute}

- /processFault : ProcessFaultPrototype [0..*] {readOnly}

32.1.9 «FaultFailurePropagationLink» (from ErrorModel)

- /fromPort : FaultFailurePort [1] {readOnly}
- /toPort : FaultFailurePort [1] {readOnly}

32.1.10 «FunctionAllocation» (from FunctionModeling)

- /allocatedElement : AllocateableElement [1] {readOnly}
- /target : AllocationTarget [1] {readOnly}
  The ECU where the functionality must be allocated.

32.1.11 «FunctionClientServerInterface» (from FunctionModeling)

- /operation : Operation [0..*] {readOnly}

32.1.12 «FunctionClientServerPort» (from FunctionModeling)

- /type : FunctionClientServerInterface [1] {readOnly}
  The interface of this port.

  {derived from UML::TypedElement::type}

32.1.13 «FunctionConnector» (from FunctionModeling)

- /port : FunctionPort [0..2] {readOnly}
  The ports that are connected by this connector.

  {derived from UML::Connector::end}

32.1.14 «FunctionFlowPort» (from FunctionModeling)

- /type : EADatatype [1] {readOnly}

32.1.15 «FunctionPowerPort» (from FunctionModeling)

- /type : CompositeDatatype [1] {readOnly}

32.1.16 «FunctionType» (from FunctionModeling) {abstract}

- /connector : FunctionConnector [0..*] {readOnly}
- /isElementary : Boolean = false [1] {readOnly}
  True, when this type does not have any parts.

  Derived from size of UML::StructuredClassifier::ownedConnector and UML::EncapsulatedClassifier::ownedPort
- /port : FunctionPort [0..*] {readOnly}
  Owned in- and out-flow ports.

  {derived from UML::EncapsulatedClassifier::ownedPort}
- /portGroup : PortGroup [0..*] {readOnly}

Grouping of ports owned by this element.

{derived from UML::Class::nestedClassifier}

32.1.17 «HardwareComponentPrototype» (from HardwareModeling)

- /type : HardwareComponentType [1] {readOnly}
  The type of the HWElement.

  {derived from UML::TypedElement::type}

32.1.18 «HardwareComponentType» (from HardwareModeling)

- /connector : HardwareConnector [0..*] {readOnly}
  The HWConnectors.

  {derived from UML::StructuredClassifier::ownedConnector}
- /part : HardwareComponentPrototype [0..*] {readOnly}
  The HWElementPrototypes.

  {derived from UML::Classifier::attribute}
- /port : HardwarePin [0..*] {readOnly}
  The Ports.

  {derived from UML::EncapsulatedClassifier::ownedPort}

32.1.19 «MultiLevelReference» (from Elements)

- /reference : EAElement [1] {readOnly}
  Referencing the source element of a Multi-Level reference link.
- /referring : EAElement [1] {readOnly}
  Referencing the target element of a Multi-Level reference link.

32.1.20 «Operation» (from FunctionModeling)

- /argument : EADatatypePrototype [0..*] {ordered, readOnly}
- /return : EADatatypePrototype [0..1] {readOnly}

32.1.21 «PortGroup» (from FunctionModeling)

- /port : FunctionPort [1..*] {readOnly}
  The grouped ports.

  {derived from UML::EncapsulatedClassifier::ownedPort} when this stereotype is applied on a Class. When the stereotype is applied on a Port the value is derived from the ports in the type.

32.1.22 «PrecedenceConstraint» (from Timing)

- /preceding : FunctionPrototype [1] {readOnly}
  The function prototype that must be executed first.

  {derived from UML::DirectedRelationship::source}

- /successive : FunctionPrototype [1..*] {ordered, readOnly}
  The function prototypes that must be executed after preceding was executed.

  {derived from UML::DirectedRelationship::target}

32.1.23 «Realization» (from Elements)

- /realized : EAElement [1..*] {readOnly}
  The set of entities, which are realized by the set of client entities or realized by the set of client AUTOSAR elements.

  {derived from UML::DirectedRelationship::target}
- /realizedBy : NamedElement (from UML) [0..*] {readOnly}
  The set of client entities, realizing the set of supplier entities. {derived from UML::Dependency::client}

32.1.24 «Refine» (from Requirements)

- /refinedBy : NamedElement (from UML) [1..*] {readOnly}
  List of elements participating to the refinement of the refined requirements.

  {derived from UML::Dependency::client}
- /refinedRequirement : Requirement [1..*] {readOnly}
  List of refined requirements. {derived from UML::DirectedRelationship::target}

32.1.25 «Satisfy» (from Requirements)

- /satisfiedBy : NamedElement (from UML) [0..*] {readOnly}
  List of satisfied use cases, which are satisfied by the client entities or satisfied by the client AUTOSAR elements.

  {derived from UML::Dependency::client}
- /satisfiedRequirement : Requirement [0..*] {readOnly}
  List of satisfied requirements, which are satisfied by the client entities. {derived from UML::DirectedRelationship::target}

32.1.26 «Verify» (from VerificationValidation)

- /verifiedByCase : VVCase [1..*] {readOnly}
  The verification that verifies the supplier requirement(s).

  {derived from UML::DirectedRelationship::source}
- /verifiedRequirement : Requirement [1..*] {readOnly}
  The set of requirements which the client VV cases verify.

{derived from UML::DirectedRelationship::target}

33 Annex D: Element Icons

	Actuator	
	AnalysisFunctionPrototype	
	AnalysisFunctionType	
	AnalysisFunctionType	composition
	AnalysisFunctionType	elementary
	AnalysisLevel	
	CommunicationHardwarePin	
	DelayConstraint	
	DeriveRequirement	
	DesignFunctionPrototype	
	DesignFunctionType	
	DesignFunctionType	composition
	DesignFunctionType	elementary
	DesignLevel	
	EADatatype	
	Environment	
	ErrorBehavior	
	ErrorModelPrototype	
	ErrorModelType	
	ExecutionTimeConstraint	
	FailureOutPort	
	FaultInPort	
	Feature	
	FunctionalAnalysisArchitecture	
	FunctionalDevice	
	FunctionAllocation	
	FunctionBehavior	
	FunctionClientServerPort	
	FunctionFlowPort	

	FunctionFlowPort	In
	FunctionFlowPort	InOut
	FunctionFlowPort	Out
	FunctionPowerPort	
	FunctionPrototype	
	FunctionTrigger	
	FunctionType	composition
	FunctionType	elementary
	GenericConstraint	
	HardwareComponentPrototype	
	HardwareConnector	
	ImplementationLevel	
	InputSynchronizationConstraint	
	IOHardwarePin	
	LocalDeviceManager	
	LogicalBus	
	Node	
	OutputSynchronizationConstraint	
	PeriodicEventConstraint	
	PortGroup	
	PortGroup	In
	PortGroup	InOut
	PortGroup	Out
	PowerHardwarePin	
	PowerSupply	
	PrecedenceConstraint	
	QualityRequirement	
	Realization	
	Refine	
	Requirement	
	RequirementsRelatedInformation	


Sensor


VehicleFeature


VehicleLevel


VVCase


VVProcedure

34 Index