EAST-ADL

EAST-ADL
Domain Model Specification

Version M.2.1.9.1
2011-06-30

Revision History

Version	Date	Reason
1.02	2004-06-30	EAST-ADL developed in the ITEA EAST-EEA project.
2.0	2008-03-20	EAST-ADL2 developed in the EC FP6 project ATESST. http://www.atesst.org/home/liblocal/docs/EAST-ADL-2.0-Specification_2008-02-29.pdf
2.1	2010-06-30	Updated version from the EC FP7 project ATESST2 with Timing Concepts from ITEA TIMMO.
M2.1.9	2011-01-30	Intermediate version from the EC FP7 project MAENAD
M2.1.9.1	2011-06-30	Intermediate version from the EC FP7 project MAENAD

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

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

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

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

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

To import EAST-ADL into an AUTOSAR metamodel:

1) Open the AUTOSAR metamodel in Enterprise Architect.

2) Import the EAST-ADL metamodel as an XMI-file.
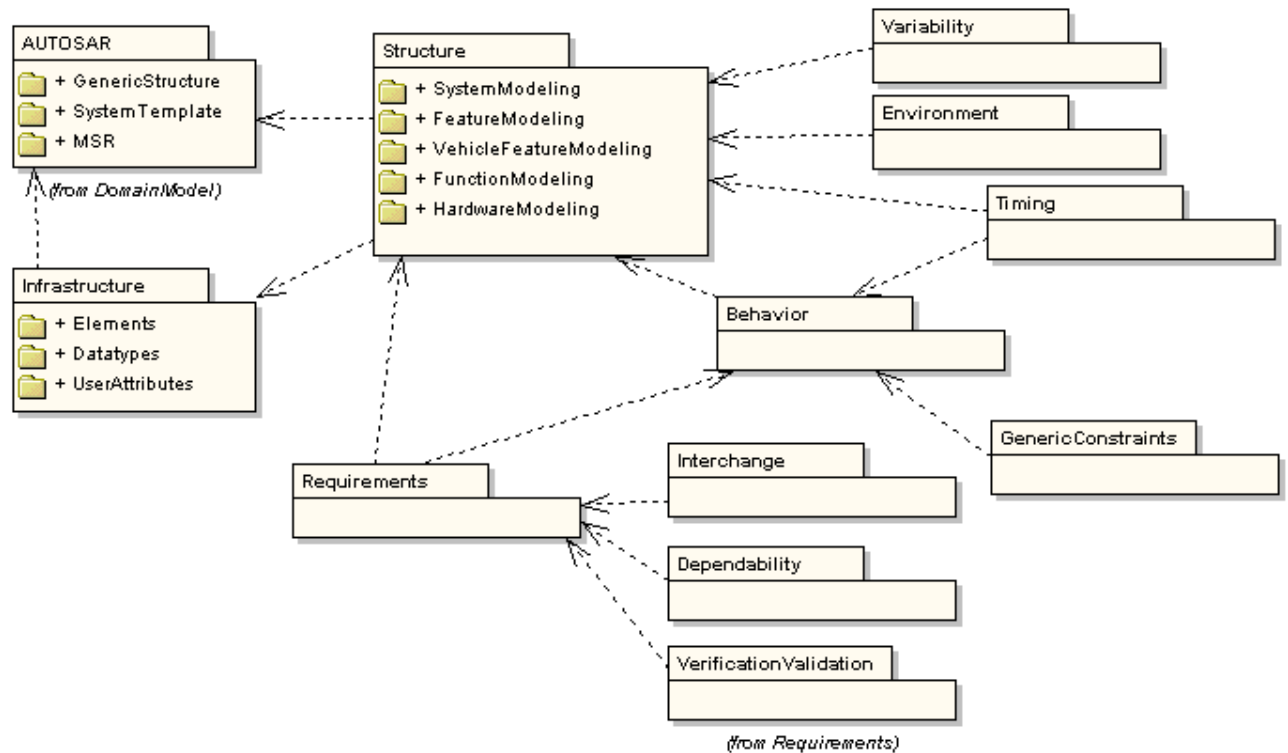
Figure 1. This diagram shows dependencies between packages in the domain model. All packages except the AUTOSAR package depend on the EAST-ADL Infrastructure package. The AUTOSAR package contains some concepts that EAST-ADL elements in the Infrastructure and Structure packages depend on.
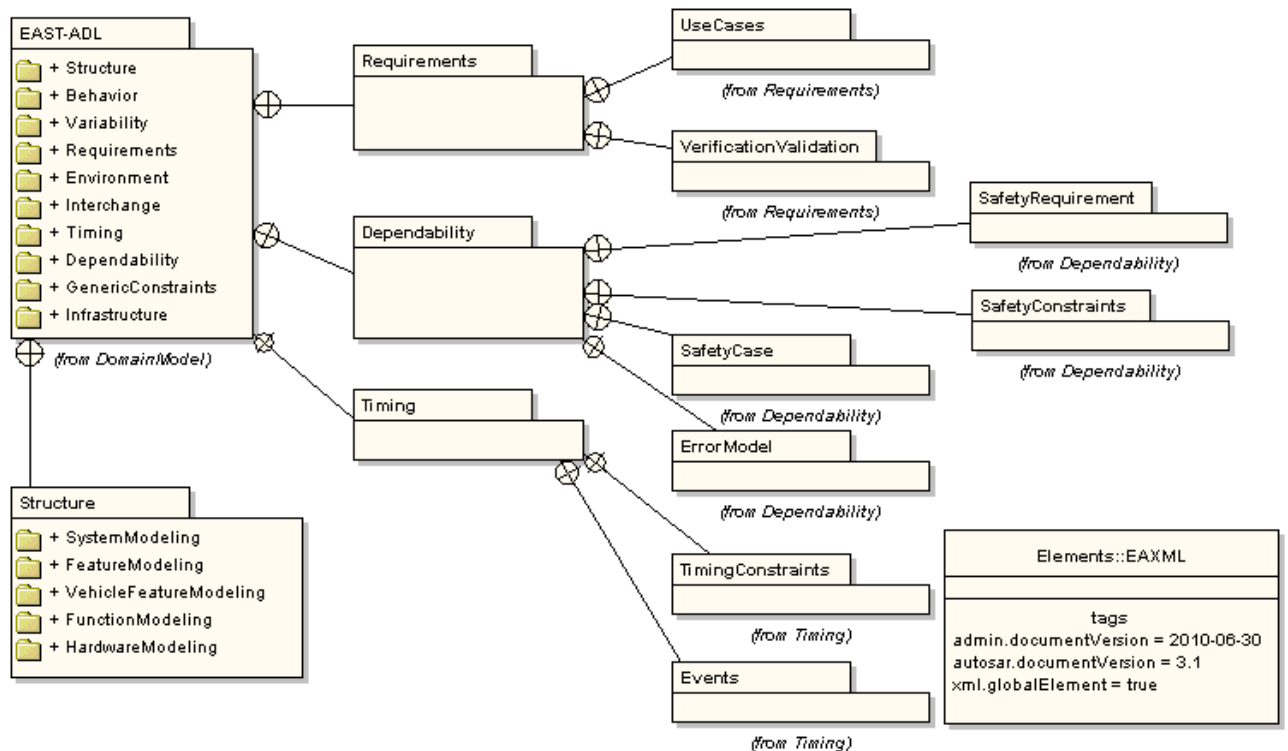
Figure 2. Packages in the EAST-ADL domain model.

1 Language Formalism

1.1 Levels of Formalism

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

1.2 Specification Structure

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

Part I includes a general introduction to the specification.

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

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

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

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

1.2.1 Overview

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

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

1.2.2 Element Descriptions

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

<Element (from subpackage)>

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

Generalizations

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

Description

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

Attributes

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

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

Associations

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

Dependencies

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

Constraints

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

Semantics

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

2   Abbreviations

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

Part II Structural Constructs

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

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

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

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

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

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

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

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

- The Environment contains Environment functions, which are encapsulations of plant models, i.e., models of the behavior of the vehicle and its non-electronic systems. Environment models are needed for validation and verification, from early analysis models to the implemented embedded system.

3 SystemModeling

3.1 Overview

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

For the design of electrical/electronic systems of arbitrary size and complexity, the possibility of hierarchical structuring of the instances is provided, so these models contain further elements in a hierarchy. Relations between these elements across the boundaries of the abstraction levels are contained in a SystemModel. This is possible because the SystemModel is a Context, and is thus able to contain relations.
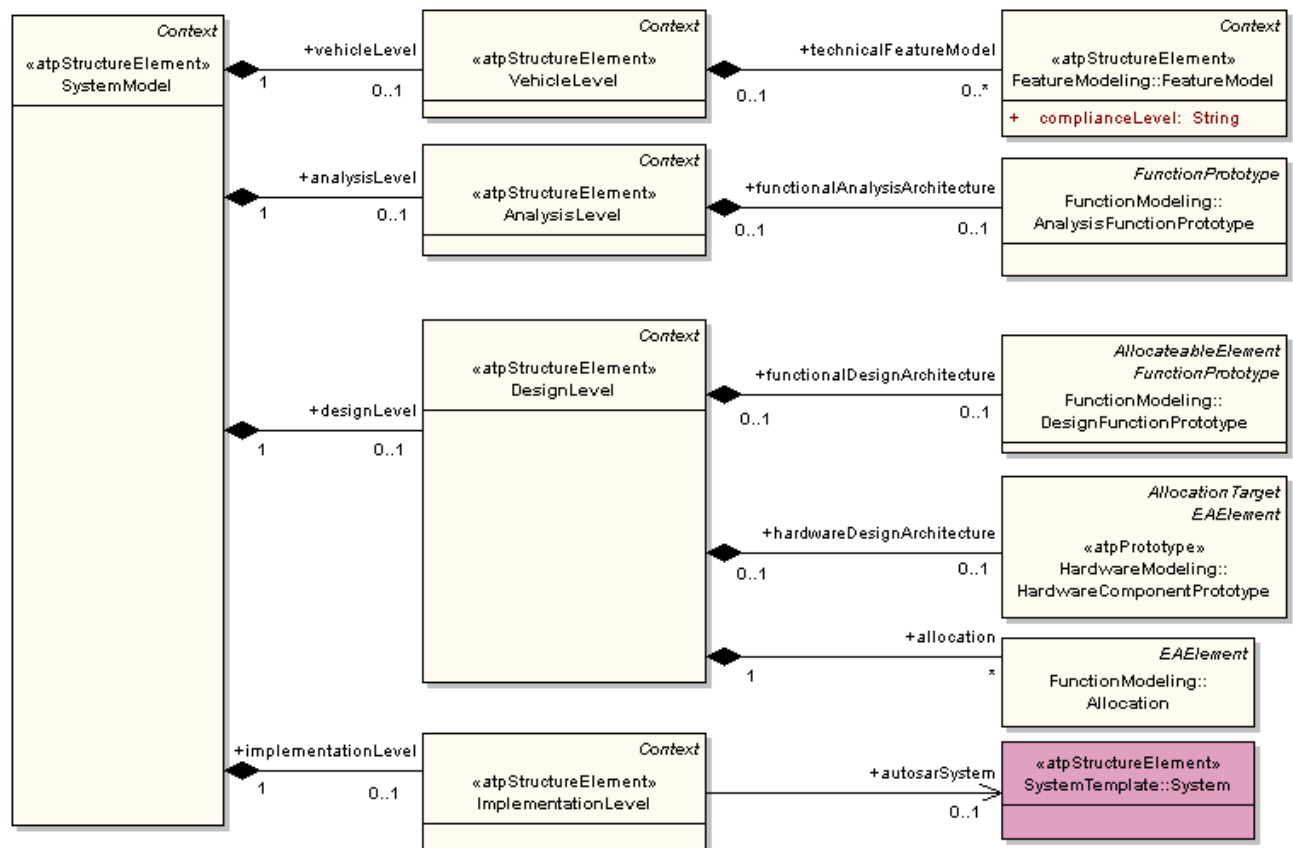
Figure 3. SystemModeling. Diagram for SystemModel. Note how the ImplementationLevel refers to the System from the AUTOSAR SystemTemplate.

3.2 Element Descriptions

3.2.1 AnalysisLevel (from SystemModeling) «atpStructureElement»

Generalizations
- Context (from Elements)

Description

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

Attributes

No additional attributes

Associations

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

Constraints

No additional constraints

Semantics

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

3.2.2 DesignLevel (from SystemModeling) «atpStructureElement»

Generalizations

- Context (from Elements)

Description

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

FDA represents a top level Function. It is supposed to implement all the functionalities of a vehicle, as specified by a 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 electrical/electronic 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 level and the ones on the analysis level.

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

The 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.

Attributes
No additional attributes

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

  The FunctionalDesignArchitecture represents the elementary design function that is used to describe the leaves of the functional hierarchy. The composition of these leaves makes up the implementation behavior of the entire functional hierarchy.

- hardwareDesignArchitecture : HardwareComponentPrototype [0..1] (from HardwareModeling)
  The included Hardware Design Architecture models the resources to which the functional design architecture parts may be allocated.

Constraints
No additional constraints

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

3.2.3   ImplementationLevel (from SystemModeling) «atpStructureElement»

Generalizations
- Context (from Elements)

Description
ImplementationLevel represents the software architecture and the hardware architecture of the electrical/electronic system in the vehicle. The ImplementationLevel is defined by the AUTOSAR SystemArchitecture and SoftwareArchitecture. For example, functions of the 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.

Attributes
No additional attributes

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

Constraints
No additional constraints

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

3.2.4   SystemModel (from SystemModeling) «atpStructureElement»

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

Associations
- vehicleLevel : VehicleLevel [0..1] (from SystemModeling)
  The included vehicle abstraction level.

- designLevel : DesignLevel [0..1] (from SystemModeling)
  The included analysis abstraction level.

- analysisLevel : AnalysisLevel [0..1] (from SystemModeling)
  The included design abstraction level.

- implementationLevel : ImplementationLevel [0..1] (from SystemModeling)
  The included implementation abstraction level.

Constraints
No additional constraints

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

3.2.5   VehicleLevel (from SystemModeling) «atpStructureElement»

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

Associations
- technicalFeatureModel : FeatureModel [0..*] (from FeatureModeling)
  This association identifies the core technical feature model of the complete system. This has a special role as it defines all the features of the complete system on vehicle level. In addition to this feature model, there may be one or more so-called product feature models (cf. association productFeatureModel in meta-class Variability in the variability extension).

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

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

Semantics

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

4	FeatureModeling

4.1 Overview

This package describes the basic feature modeling that is employed on the vehicle level as well as on the artifact levels, i.e., on AnalysisLevel and below. Details of feature modeling that are specific to the vehicle level are factored out and documented separately in the package VehicleFeatureModeling.
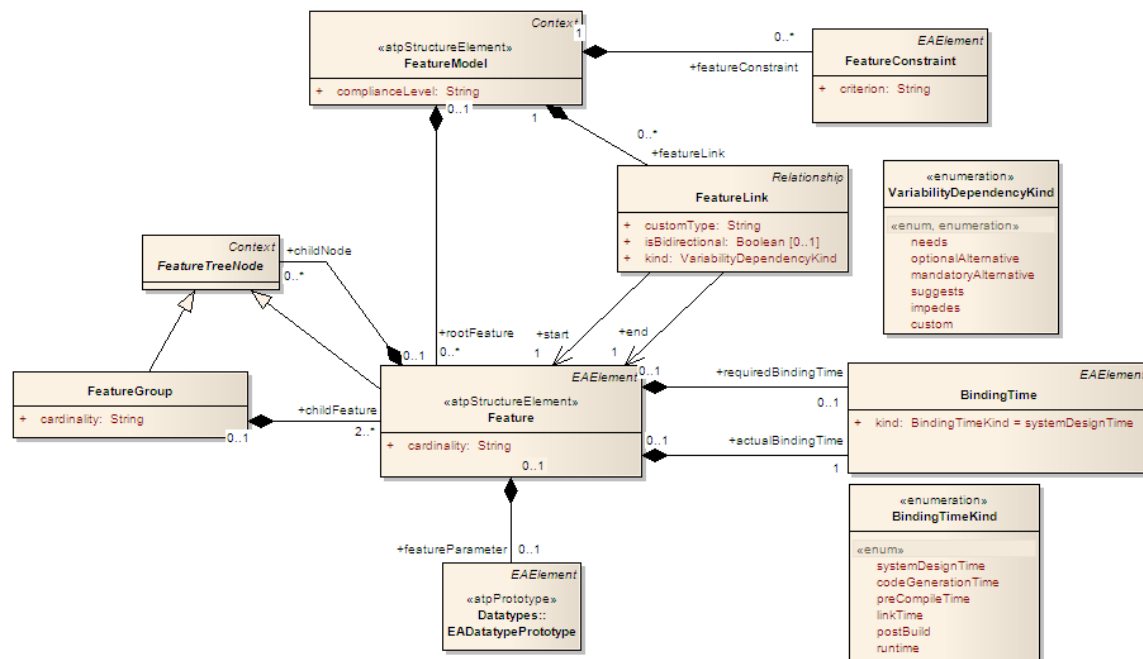
Figure 4. FeatureModeling. Diagram for FeatureModeling.

4.2 Element Descriptions

4.2.1 BindingTime (from FeatureModeling)

Generalizations
- EAElement (from Elements)

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

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

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

- SystemDesignTime

- CodeGenerationTime

- PreCompileTime

- LinkTime

- PostBuild

- Runtime

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

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

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

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

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

4.2.2   BindingTimeKind (from FeatureModeling) «enumeration»

Generalizations
None

Description
BindingTimeKind represents the set of possible binding times.

Enumeration Literals
- codeGenerationTime
  Variability will be bound during code generation.

  From AUTOSAR:

  * Coding by hand, based on requirements document.

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

  * The model may contain variants.

  * Only code for the selected variant(s) is actually generated.

- linkTime
  Variability will be bound during linking.

From AUTOSAR:

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

Based on which variant(s) are selected

E.g. for modules that are delivered as object code (as opposed to those that are delivered as source code)

- postBuild
  Variability will be bound at certain occasions after shipment, for example when the vehicle is in a workshop.

- preCompileTime
  Variability will be bound during or immediately prior to code compilation.

  From AUTOSAR:

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

- runtime
  Variability will be bound by the customer after shipment by way of vehicle configuration.

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

- systemDesignTime
  Variability will be bound during development of the electrical/electronic system.

  From AUTOSAR:

  * Designing the VFB.

  * Software Component types (portinterfaces).

  * SWC Prototypes and the Connections between SWCprototypes.

  * Designing the Topology

  * ECUs and interconnecting Networks

  * Designing the Communication Matrix and Data Mapping

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

4.2.3 Feature (from FeatureModeling) «atpStructureElement»

Generalizations
- FeatureTreeNode (from FeatureModeling)
- EAElement (from Elements)

Description

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

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

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

Attributes

- cardinality : String [1]
  Specifies the Feature's cardinality stating how often this feature may be selected during configuration.

  Typical cardinalities include:

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

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

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

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

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

Associations

- actualBindingTime : BindingTime [1] (from FeatureModeling)
  The actual binding time, independent of the required binding time.

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

  In the rationales it must be described what the reasons are for a (different) actual binding time.

- requiredBindingTime : BindingTime [0..1] (from FeatureModeling)
  The required binding time could possibly deviate from the actual binding time.

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

  Each feature/variation point must have a required binding time attribute.

- childNode : FeatureTreeNode [0..*] (from FeatureModeling)
  Features may have any number of Features or FeatureGroups as their children or none at all.

- featureParameter : EADatatypePrototype [0..1] (from Datatypes)
  For parameterized features, this specifies the type of the feature's parameter.

  Parameterized features are special features that can not only be selected or deselected during configuration but, if selected, they can also be supplied with a value of a particular type (e.g. an integer). Note that only the type but not the actual value of a feature parameter is defined within the feature model; the actual value is defined as part of the configuration of this feature model.

Constraints
No additional constraints

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

4.2.4   FeatureConstraint (from FeatureModeling)

Generalizations
- EAElement (from Elements)

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

Attributes
- criterion : String [1]
  The actual constraint. This is a logic expression in VSL like the criterion of a ConfigurationDecision. For the constraint to be met this expression always has to evaluate to true.

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

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

4.2.5 FeatureGroup (from FeatureModeling)

Generalizations
- FeatureTreeNode (from FeatureModeling)

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, and with a cardinality of [1], means that A and B are alternatives.

Attributes
- cardinality : String [1]

  The cardinality of the FeatureGroup. It states how many of its child features can be selected in a valid configuration. Mandatory features among the child features count as 1 and for cloned features all instances created in the configuration count.

Associations
- childFeature : Feature [2..*] (from FeatureModeling)

  FeatureGroups may only have Features as their children and must always have at least two children.

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

Constraints

No additional constraints

Semantics

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

4.2.6 FeatureLink (from FeatureModeling)

Generalizations
- Relationship (from Elements)

Description

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

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

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

Attributes
- customType : String [1]

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

  Each company or project can decide to use additional link types by defining unique key-words for them. In cases where FeatureModels are shared with third parties (other departments, companies, etc.) a globally unique type string must be used. Follow the

instructions for finding globally unique keys for user attributes (cf. documentation of metaclass UserAttributeValue).

- isBidirectional : Boolean [0..1]
  Tells whether the FeatureLink is bidirectional or unidirectional. For predefined kinds, such as "needs", "mandatoryAlternative", etc., this attribute will be ignored and the kind determines whether the link is bidirectional or not (as defined in the documentation of attribute 'type', below). For custom kinds, this attribute may be provided to explicitly state the link's direction. If this attribute is not provided in case of a custom link type, then the link is assumed to be unidirectional.

- kind : VariabilityDependencyKind [1] (from FeatureModeling)
  The kind determines the precise semantics of the relation between the FeatureLink's start and end feature. There are 5 predefined kinds as defined by enumeration VariabilityDependencyKind and in the case of kind 'custom' the attribute customType can be used to define a custom feature link type.

Associations
- start : Feature [1] (from FeatureModeling)
  The source [supplier] Feature of the relationship.

- end : Feature [1] (from FeatureModeling)
  The target [client] Feature of the dependency.

Constraints
No additional constraints

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

4.2.7   FeatureModel (from FeatureModeling) «atpStructureElement»

Generalizations
- Context (from Elements)

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

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

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

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

Attributes
- complianceLevel : String [1]
  This attribute specifies that the FeatureModel should comply with a certain, established feature modeling and diagramming technique (such as FODA, pure::variants).

Associations
- rootFeature : Feature [0..*] (from FeatureModeling)
  The root Features owned by the FeatureModel. Note that only root Features are directly contained in the model; non-root Features are contained in their parent Feature or parent FeatureGroup.

- featureLink : FeatureLink [0..*] (from FeatureModeling)
  The FeatureLinks owned by the FeatureModel.

- featureConstraint : FeatureConstraint [0..*] (from FeatureModeling)
  FeatureConstraints owned by the FeatureModel.

Constraints
No additional constraints

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

4.2.8   FeatureTreeNode (from FeatureModeling) {abstract}

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

4.2.9   VariabilityDependencyKind (from FeatureModeling) «enumeration»

Generalizations
None

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

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

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

- impedes
  Weak from of "excludes".

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

When used in a VariationGroup: accordingly as above.

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

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

- needs
  When used in a FeatureLink: if the FeatureLink's start feature S is selected, then also its end feature E must be selected: not (S and not E). Always unidirectional.

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

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

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

- suggests
  Weak form of "needs".

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

  When used in a VariationGroup: accordingly as above.

Associations
No additional Associations

Constraints
No additional constraints

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

-

5 VehicleFeatureModeling

5.1 Overview

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

The specification of the features themselves, together with their forms of realization, the dependencies between them, and the requirements to be respected for their realization is performed at the vehicle level and it should be done independently of any product line. This would be the basis for a consistent reuse of features in different product lines and projects. At this level, a feature represents particular high level requirements to be realized in all product line members that respect some conditions, e.g., US cars with elegance trim and engine size higher than 2.4.
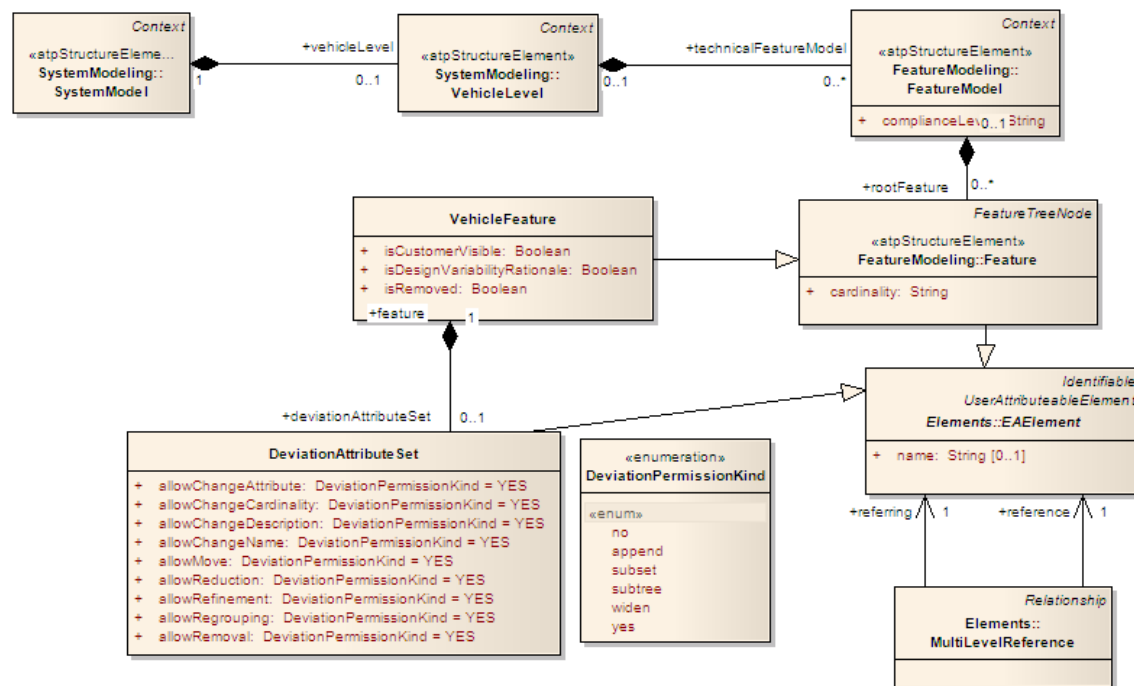
Figure 5. VehicleFeatureModeling. Diagram for VehicleFeatureModeling.

5.2 Element Descriptions

5.2.1 DeviationAttributeSet (from VehicleFeatureModeling)

Generalizations
- EAElement (from Elements)

Description
DeviationAttributeSet specifies the set of rules of allowed deviations from the reference model in a referring model. These rules are important, because they make sure that the different

FeatureModels, referring to one reference model, follow specific rules for deviation, so a later integration into one FeatureModel might be possible.

Attributes

- allowChangeAttribute : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets whether and how the VehicleFeature attributes may be changed. Allowed values: no, append, yes.

- allowChangeCardinality : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets whether and how the VehicleFeature cardinality (i.e. variability of the VehicleFeature) may be changed. Allowed values: no, subset, yes.

- allowChangeDescription : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets whether and how the VehicleFeature description may be changed. Allowed values: no, append, yes.

- allowChangeName : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets whether and how the VehicleFeature name may be changed. Allowed values: no, append, yes.

- allowMove : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets whether and how the VehicleFeature may be moved to another place in the feature diagram. Allowed values: no, subtree, yes.

- allowReduction : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets if the reference feature may have a child without a corresponding referring feature among the children of the referring feature. Allowed values: no, subtree, yes.

- allowRefinement : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets whether and how adding may be done of a child feature (without a corresponding feature in the reference model). Allowed values: no, yes.

- allowRegrouping : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets whether and how the immediate child features of the VehicleFeature are allowed to be regrouped (i.e. creation or deletion of FeatureGroups below the respective VehicleFeature). Allowed values: no, widen, yes.

- allowRemoval : DeviationPermissionKind = YES [1] (from VehicleFeatureModeling)
  This rule sets if the feature in the referring model (compared to the reference model) may be deleted. Allowed values: no, yes.

Associations

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

Constraints
No additional constraints

Semantics
-

5.2.2 DeviationPermissionKind (from VehicleFeatureModeling) «enumeration»

Generalizations
None

Description

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

Enumeration Literals

- append
  The name, description or other attribute may only be changed by appending text without changing the original text. This kind is only applicable to deviation attributes "allowChangeName", "allowChangeDescription" and "allowChangeAttribute".

- no
  The deviation is not allowed.

- subset
  The cardinality may only be changed such that the new cardinality is a subset of the original cardinality. This kind is only applicable to deviation attribute "allowChangeCardinality".

- subtree
  In case of deviation attribute "allowMove": the parent of the VehicleFeature may be changed, but the original parent must remain a predecessor (i.e. moving the VehicleFeature itself is allowed but it may only be moved further down within the same subtree).

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

  This kind is only applicable to deviation attributes "allowMove" and "allowReduction".

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

- yes
  The deviation is allowed.

Associations

No additional Associations

Constraints

No additional constraints

Semantics

-

5.2.3   VehicleFeature (from VehicleFeatureModeling)

Generalizations

- Feature (from FeatureModeling)

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.

Attributes

- isCustomerVisible : Boolean [1]

This attribute states whether the VehicleFeature is customer visible (in contrast to a VehicleFeature that is e.g. technically driven).

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

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

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

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

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

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

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

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

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

Semantics
-

6 FunctionModeling

6.1 Overview

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

The main modeling concept applied here is functional component modeling: Functions interact with one another via ports that are connected by connectors owned by the composing function. Occurrences of functions are modeled by typed prototypes in the composing function. These occurrences are typed by types. This naming convention of the type-prototype pattern is from AUTOSAR, however the concept of types and typed elements is also available in e.g. UML2.
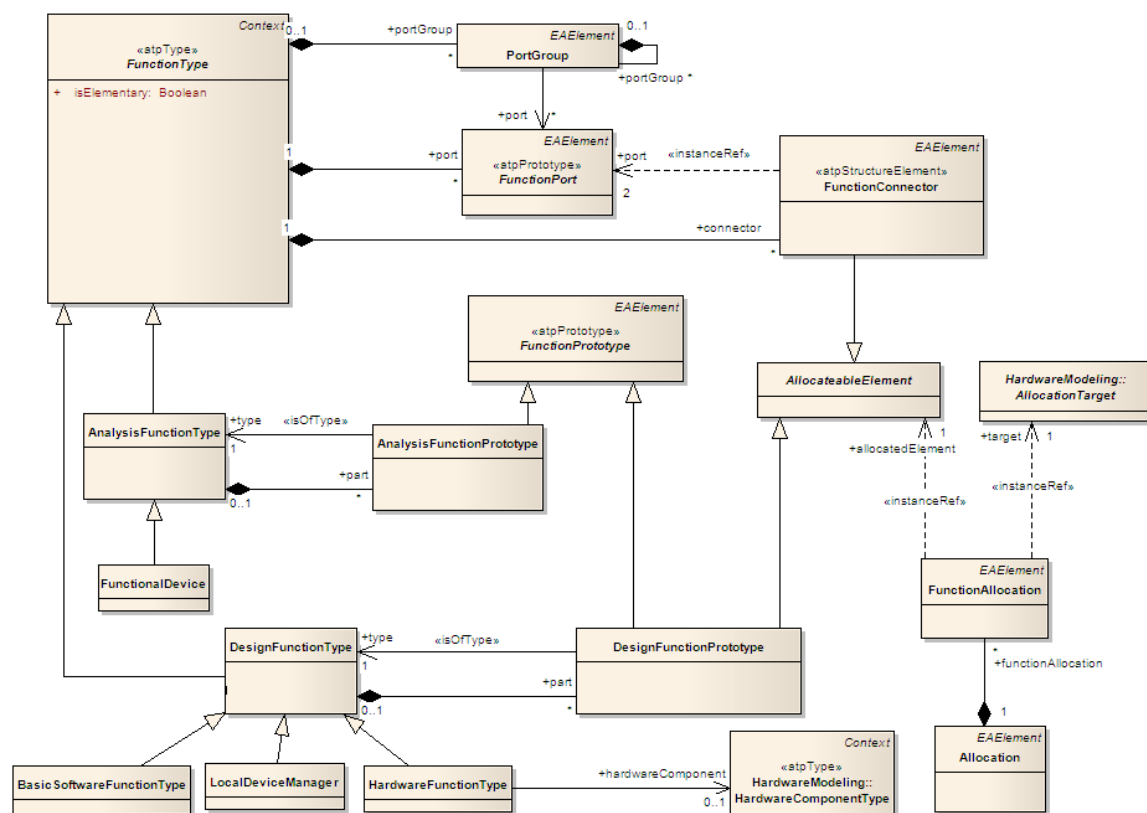
Figure 6. FunctionModeling. Diagram for FunctionModeling.

Figure 7. FunctionPort. Diagram for FunctionPorts.

6.2 Element Descriptions

6.2.1 AllocateableElement (from FunctionModeling) {abstract}

Generalizations
None

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
The AllocateableElement abstracts all elements that are allocateable.

Subclasses of the abstract class AllocateableElement add their own semantics.

6.2.2 Allocation (from FunctionModeling)

Generalizations
- EAElement (from Elements)

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

Attributes
No additional attributes

Associations
- functionAllocation : FunctionAllocation [*] (from FunctionModeling)
  The owned FunctionAllocations.

Constraints
No additional constraints

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

6.2.3 AnalysisFunctionPrototype (from FunctionModeling)

Generalizations
- FunctionPrototype (from FunctionModeling)

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

The AnalysisFunctionPrototype is typed by an AnalysisFunctionType.

Attributes
No additional attributes

Associations
- type : AnalysisFunctionType [1] (from FunctionModeling)
«isOfType»

    The type that defines this AnalysisFunctionPrototype.

Constraints
No additional constraints

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

6.2.4 AnalysisFunctionType (from FunctionModeling)

Generalizations
- FunctionType (from FunctionModeling)

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

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

Furthermore, an AnalysisFunction may be decomposed into (sub-)AnalysisFunctions. This allows the functionalities provided by the parent AnalysisFunction to be broken up hierarchically into subfunctionalities.

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

Attributes
No additional attributes

Associations
- part : AnalysisFunctionPrototype [*] (from FunctionModeling)
  The parts contained in this AnalysisFunctionType.

Constraints
No additional constraints

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

6.2.5   BasicSoftwareFunctionType (from FunctionModeling)

Generalizations
- DesignFunctionType (from FunctionModeling)

Description
The BasicSoftwareFunctionType is an abstraction of middleware functionality.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
The BasicSoftwareFunctionType is an abstraction of the middleware.

6.2.6   ClientServerKind (from FunctionModeling) «enumeration»

Generalizations
None

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

Enumeration Literals
- client
- server

Associations

No additional Associations

Constraints

No additional constraints

Semantics

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

6.2.7 DesignFunctionPrototype (from FunctionModeling)

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

Description

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

The DesignFunctionPrototype is typed by a DesignFunctionType .

Attributes

No additional attributes

Associations
- type : DesignFunctionType [1] (from FunctionModeling)

«isOfType»

> The type that defines this DesignFunctionPrototype.

Constraints

No additional constraints

Semantics

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

6.2.8 DesignFunctionType (from FunctionModeling)

Generalizations
- FunctionType (from FunctionModeling)

Description

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

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

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

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

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

Attributes
No additional attributes

Associations
- part : DesignFunctionPrototype [*] (from FunctionModeling)
  The parts contained in this DesignFunctionType.

Constraints
No additional constraints

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

6.2.9   EADirectionKind (from FunctionModeling) «enumeration»

Generalizations
None

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

Enumeration Literals
- in
- inout
- out

Associations
No additional Associations

Constraints
No additional constraints

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

6.2.10   FunctionAllocation (from FunctionModeling)

Generalizations
- EAElement (from Elements)

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

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

Attributes
No additional attributes

Associations
- target : FunctionAllocation_target [1] (from _instanceRef)

- allocatedElement : FunctionAllocation_allocatedElement [1] (from _instanceRef)

Dependencies
- allocatedElement: AllocateableElement [1] (from FunctionModeling)

«instanceRef»

- target: AllocationTarget [1] (from HardwareModeling)

«instanceRef»

Constraints
No additional constraints

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

6.2.11   FunctionClientServerInterface (from FunctionModeling) «atpType»

Generalizations
- EAPackageableElement (from Elements)

Description
The FunctionClientServerInterface is used to specify the operations in FunctionClientServerPorts.

Attributes
No additional attributes

Associations
- operation : Operation [*] (from FunctionModeling)
   The owned Operation.

Constraints
No additional constraints

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

6.2.12   FunctionClientServerPort (from FunctionModeling)

Generalizations
- FunctionPort (from FunctionModeling)

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

Attributes
- clientServerType : ClientServerKind [1] (from FunctionModeling)

Associations
- type : FunctionClientServerInterface [1] (from FunctionModeling)

«isOfType»

   The interface of this FunctionClientServerPort.

Constraints
No additional constraints

Semantics

The FunctionClientServerPort is a FunctionPort for client-server interaction.

FunctionClientServerPorts are single buffer overwrite and nonconsumable.

6.2.13   FunctionConnector (from FunctionModeling) «atpStructureElement»

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

Description

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

Attributes

No additional attributes

Associations
- port : FunctionConnector_port [2] (from _instanceRef)

Dependencies
- port: FunctionPort [2] (from FunctionModeling)

«instanceRef»

Constraints

No additional constraints

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.

6.2.14   FunctionFlowPort (from FunctionModeling)

Generalizations
- FunctionPort (from FunctionModeling)

Description

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

Attributes
- direction : EADirectionKind [1] (from FunctionModeling)

Associations
- type : EADatatype [1] (from Datatypes)

«isOfType»

> The single EADatatype for this port.

Constraints

No additional constraints

Semantics

FunctionFlowPorts are single buffer overwrite and nonconsumable.

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

EADatatypes that are ValueTypes are compatible if

* They have the same "dimension".

* They have the same "unit".

EADatatypes that are RangeableValueTypes are compatible if

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

* They have the same baseRangeable.

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

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

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

* They have the same "significantDigits".

EADatatypes that are EnumerationValueTypes are compatible if

* They have the same baseEnumeration.

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

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

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

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

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

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

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

Generalizations
  • EAElement (from Elements)
Description
The ports conserve variables for component interaction.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

6.2.16   FunctionPowerPort (from FunctionModeling)

Generalizations
- FunctionPort (from FunctionModeling)

Description

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

Attributes

No additional attributes

Associations
- type : CompositeDatatype [1] (from Datatypes)

«isOfType»

>   The Datatype for the flow physical variables of this FunctionPowerPort, specifying the Across and Through variables with two separate datatypePrototypes.

Constraints

No additional constraints

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.

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

Generalizations
- EAElement (from Elements)

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 with a FunctionPrototype.

Attributes

No additional attributes

Associations

No additional Associations

Constraints

No additional constraints

Semantics

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

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

Generalizations
- Context (from Elements)

Description

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

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

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

Associations
- port : FunctionPort [*] (from FunctionModeling)
  Owned ports.

- connector : FunctionConnector [*] (from FunctionModeling)
  The connectors that connect ports of parts as assembly connectors or ports of this type and ports of parts as delegation connectors.

- portGroup : PortGroup [*] (from FunctionModeling)
  Grouping of ports owned by this element.

Constraints

No additional constraints

Semantics

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

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

Elementary Functions have synchronous execution semantics:

1. Read inputs

2. Execute (duration: Execution time)

3. Write outputs

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

Subclasses of the abstract class FunctionType add their own semantics.

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

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

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

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

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

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

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

6.2.19   FunctionalDevice (from FunctionModeling)

Generalizations
- AnalysisFunctionType (from FunctionModeling)

Description

The FunctionalDevice represents an abstract sensor or actuator that encapsulates sensor/actuator dynamics and the interfacing software. The FunctionalDevice is the interface between the electronic architecture and the environment (connected by ClampConnectors). 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.

Attributes

No additional attributes

Associations

No additional Associations

Constraints

No additional constraints

Semantics

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

6.2.20   HardwareFunctionType (from FunctionModeling)

Generalizations
- DesignFunctionType (from FunctionModeling)

Description

The HardwareFunctionType is the transfer function for the identified HardwareComponentType or a specification of an intended transfer function. HardwareFunctionType types DesignFunctionPrototypes in the FunctionalDesignArchitecture. The Such DesignFunctionPrototypes are typically at the end of the ClampConnectors on DesignLevel.

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

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

Attributes

No additional attributes

Associations
- hardwareComponent : HardwareComponentType [0..1] (from HardwareModeling)
  The HardwareComponentType with the specified HardwareFunction.

Constraints

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

Semantics

-

6.2.21   LocalDeviceManager (from FunctionModeling)

Generalizations
- DesignFunctionType (from FunctionModeling)

Description

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

Attributes

No additional attributes

Associations

No additional Associations

Constraints

No additional constraints

Semantics

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

6.2.22   Operation (from FunctionModeling)

Generalizations
- EAElement (from Elements)

Description

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

Attributes

No additional attributes

Associations
- argument : EADatatypePrototype [*] {ordered} (from Datatypes)
  The argument value of the Operation.

- return : EADatatypePrototype [0..1] (from Datatypes)
  The return value of the Operation.

Constraints

No additional constraints

Semantics

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

6.2.23 PortGroup (from FunctionModeling)

Generalizations
- EAElement (from Elements)

Description

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

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

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

Attributes

No additional attributes

Associations
- port : FunctionPort [*] (from FunctionModeling)
  The grouped FunctionPorts.

- portGroup : PortGroup [*] (from FunctionModeling)
  Grouping of ports owned by this element.

Constraints

No additional constraints

Semantics

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

7   HardwareModeling

7.1   Overview

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

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

Conversely, the Functional Analysis Architecture and the Functional Design Architecture may be revised based on analysis using information from the Hardware Design Architecture. An example is control law design, where algorithms may be modified for expected computational and communication delays. Thus, the Hardware Design Architecture contains information about properties in order to support, e.g., timing analysis and performance in these respects.
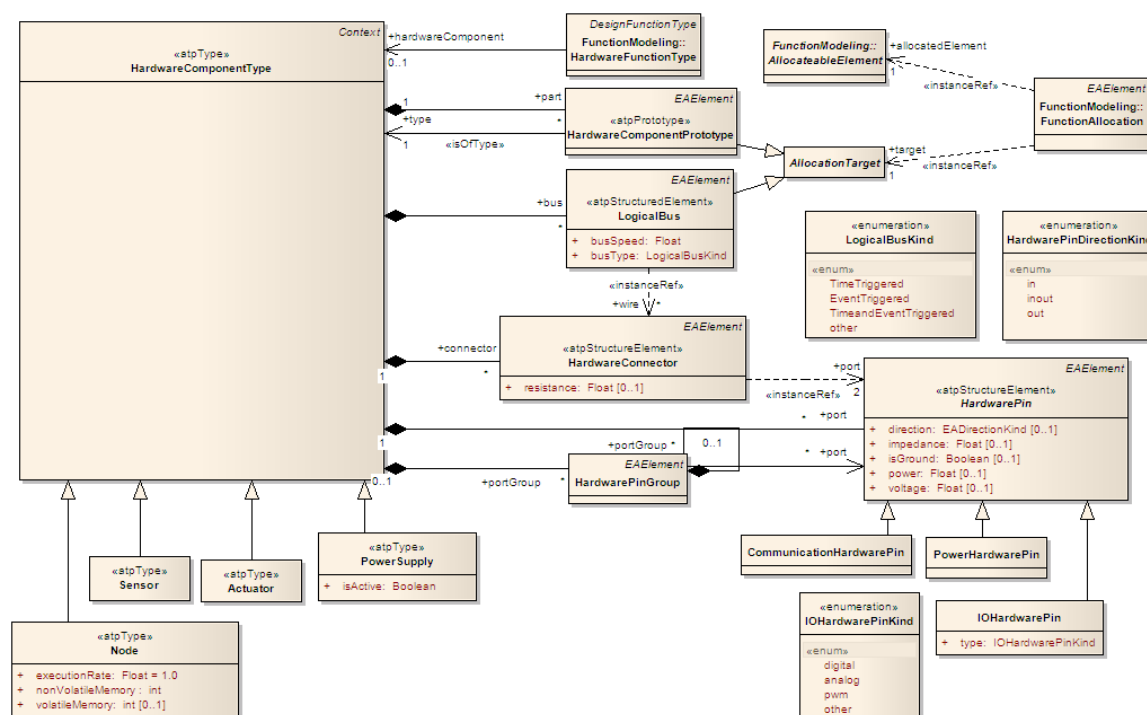
Figure 8. HardwareModeling. Diagram for HardwareModeling.

7.2   Element Descriptions

7.2.1   Actuator (from HardwareModeling)

Generalizations
• HardwareComponentType (from HardwareModeling)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

7.2.2 AllocationTarget (from HardwareModeling) {abstract}

Generalizations
None

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

7.2.3 CommunicationHardwarePin (from HardwareModeling)

Generalizations
- HardwarePin (from HardwareModeling)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

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

7.2.4   HardwareComponentPrototype (from HardwareModeling) «atpPrototype»

Generalizations
- EAElement (from Elements)
- AllocationTarget (from HardwareModeling)

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

Attributes
No additional attributes

Associations
- type : HardwareComponentType [1] (from HardwareModeling)
«isOfType»

Constraints
No additional constraints

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

7.2.5   HardwareComponentType (from HardwareModeling) «atpType»

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

Associations
- connector : HardwareConnector [*] (from HardwareModeling)
  Connectors owned by this element.

- part : HardwareComponentPrototype [*] (from HardwareModeling)
  Parts owned by this element.

- portGroup : HardwarePinGroup [*] (from HardwareModeling)
  PortGroups of owned by this element.

- port : HardwarePin [*] (from HardwareModeling)
  Hardware ports owned by this type.

- bus : LogicalBus [*] (from HardwareModeling)
  The LogicalBus contained in the HardwareComponent

Constraints
No additional constraints

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 HardwareFunctionType 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.

7.2.6	HardwareConnector (from HardwareModeling) «atpStructureElement»

Generalizations
*   EAElement (from Elements)

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

Attributes
*   resistance : Float [0..1]
    The resistance of the HardwareConnector in Ohms.

Associations
*   port : HardwareConnector_port [2] (from _instanceRef)

Dependencies
*   port: HardwarePin [2] (from HardwareModeling)
«instanceRef»

Constraints
No additional constraints

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

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

Generalizations
*   EAElement (from Elements)

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

Attributes
*   direction : EADirectionKind [0..1] (from FunctionModeling)
    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.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
Hardware pin represents an electrical connection point.

7.2.8 HardwarePinDirectionKind (from HardwareModeling) «enumeration»

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

Associations
No additional Associations

Constraints
No additional constraints

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

7.2.9 HardwarePinGroup (from HardwareModeling)

Generalizations
- EAElement (from Elements)

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, and join connectors that go between pins in pin groups to a single line.

Attributes
No additional attributes

Associations
- portGroup : HardwarePinGroup [*] (from HardwareModeling)
- port : HardwarePin [*] (from HardwareModeling)

Constraints
No additional constraints

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

7.2.10 IOHardwarePin (from HardwareModeling)

Generalizations
- HardwarePin (from HardwareModeling)

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

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

Associations
No additional Associations

Constraints
No additional constraints

Semantics
The IOHardwarePin represents an electrical pin or connection point.

7.2.11   IOHardwarePinKind (from HardwareModeling) «enumeration»

Generalizations
None

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

Enumeration Literals
- analog
  I/O with varying amplitude.

- digital
  I/O with fixed amplitude.

- other
  Another type of I/O port.

- pwm
  PWM (Pulse Width Modulated) modulated I/O, i.e. a signal with fixed frequency and amplitude but varying duty cycle.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

7.2.12   LogicalBus (from HardwareModeling) «atpStructuredElement»

Generalizations
- AllocationTarget (from HardwareModeling)
- EAElement (from Elements)

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.

Attributes
- busSpeed : Float [1]
  The net bus speed in bits per second. Used to assess communication delay and schedulability on the bus. Note that scheduling details are not represented in the model.

- busType : LogicalBusKind [1] (from HardwareModeling)
  The type of bus scheduling assumed.

Associations
- wire : LogicalBus_wire [*] (from _instanceRef)

Dependencies
- wire: HardwareConnector [*] (from HardwareModeling)

«instanceRef»

Constraints
No additional constraints

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.

7.2.13   LogicalBusKind (from HardwareModeling) «enumeration»

Generalizations
None

Description
LogicalBusKind is an enumeration type representing different kinds of busses.

Enumeration Literals
- EventTriggered
  Bus is event-triggered

- TimeTriggered
  Bus is time-triggered

- TimeandEventTriggered
  Bus is both time and event-triggered

- other
  Another type of bus communication

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

7.2.14   Node (from HardwareModeling)

Generalizations
- HardwareComponentType (from HardwareModeling)

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

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

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

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

- volatileMemory : int [0..1]
  The size in Bytes of the Node's Volatile memory (RAM)

Associations
No additional Associations

Constraints
No additional constraints

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

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

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

7.2.15 PowerHardwarePin (from HardwareModeling)

Generalizations
- HardwarePin (from HardwareModeling)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

7.2.16   PowerSupply (from HardwareModeling)

Generalizations
- HardwareComponentType (from HardwareModeling)

Description
PowerSupply represents a hardware element that supplies power.

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

Associations
No additional Associations

Constraints
No additional constraints

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

7.2.17   Sensor (from HardwareModeling)

Generalizations
- HardwareComponentType (from HardwareModeling)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

8　Environment

8.1　Overview

The Environment model is used to describe the environment of the vehicle electric and electronic architecture. It is modeled by continuous functions representing the system environment.
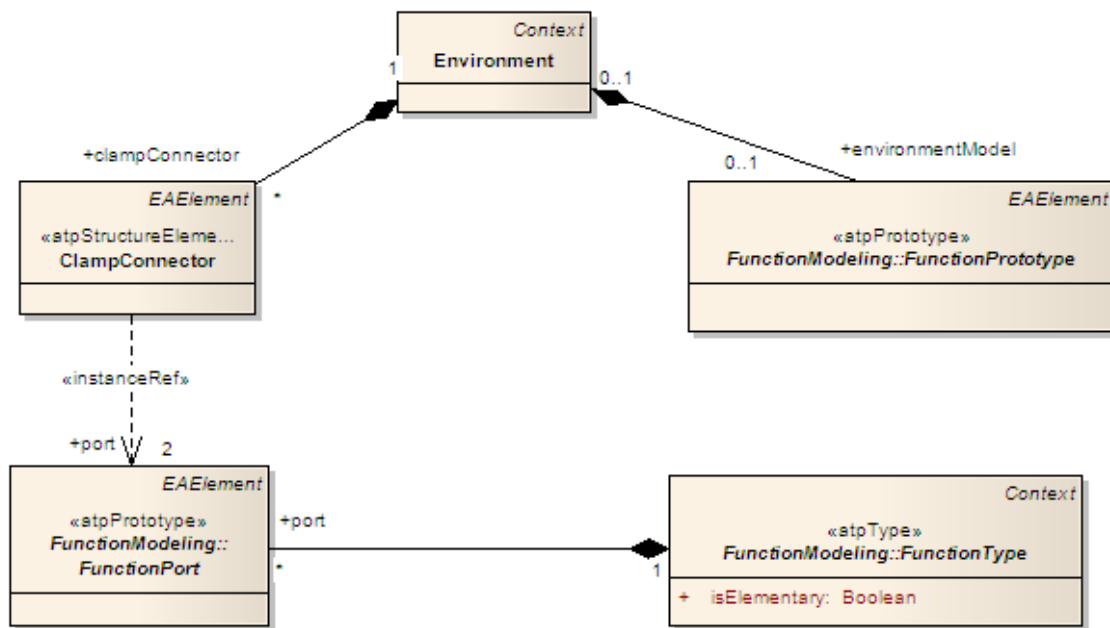
Figure 9. EnvironmentModelOrganization. Diagram for Environment. The EnvironmentModel is a packageable element, but note that it is not a part of the SystemModel.

8.2　Element Descriptions

8.2.1　ClampConnector (from Environment) «atpStructureElement»

Generalizations
- EAElement (from Elements)

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

Attributes
No additional attributes

Associations
- port : ClampConnector_port [2] (from _instanceRef)

Dependencies
- port: FunctionPort [2] (from FunctionModeling)

«instanceRef»

Constraints
[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.

Semantics
-

8.2.2   Environment (from Environment)

Generalizations
- Context (from Elements)

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

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

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

Attributes
No additional attributes

Associations
- environmentModel : FunctionPrototype [0..1] (from FunctionModeling)
- clampConnector : ClampConnector [*] (from Environment)

Constraints
No additional constraints

Semantics
-

Part III Behavioral Constructs

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

Part III Behavioral Constructs

9   Behavior

9.1   Overview

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

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

A FunctionBehavior in EAST-ADL is mainly a reference point to some description provided elsewhere in a tool-dependent format, as depicted in the diagram for the behavior of a function below. This enables the re-use of 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 formats is provided as an enumeration, FunctionBehaviorKind. Depending on the EAST-ADL language implementation, such a behavior description can be provided in the model itself; for instance, when using a UML-implementation of the EAST-ADL, UML behaviors can be used. Yet it should be noted that the behavior described shall be compliant with the execution semantics of an EAST-ADL function.

The rest of the behavioral constructs (see the following diagram of the behavior model organization) 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 constraints are defined in the Timing, Events, and TimingConstraints sections.
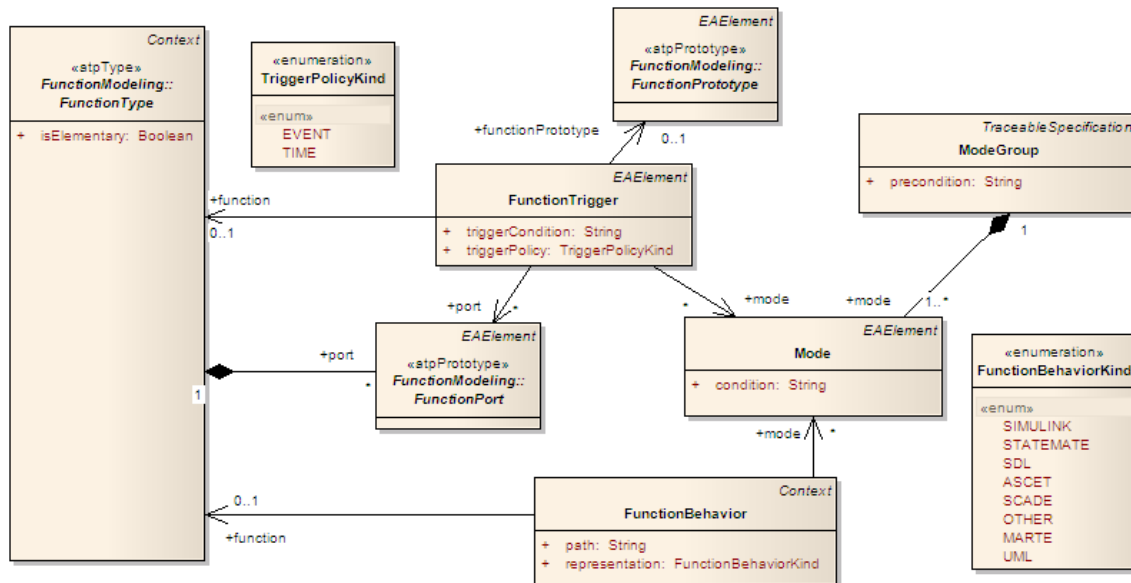
Figure 10. FunctionBehavior. Diagram for the behavior of a function.
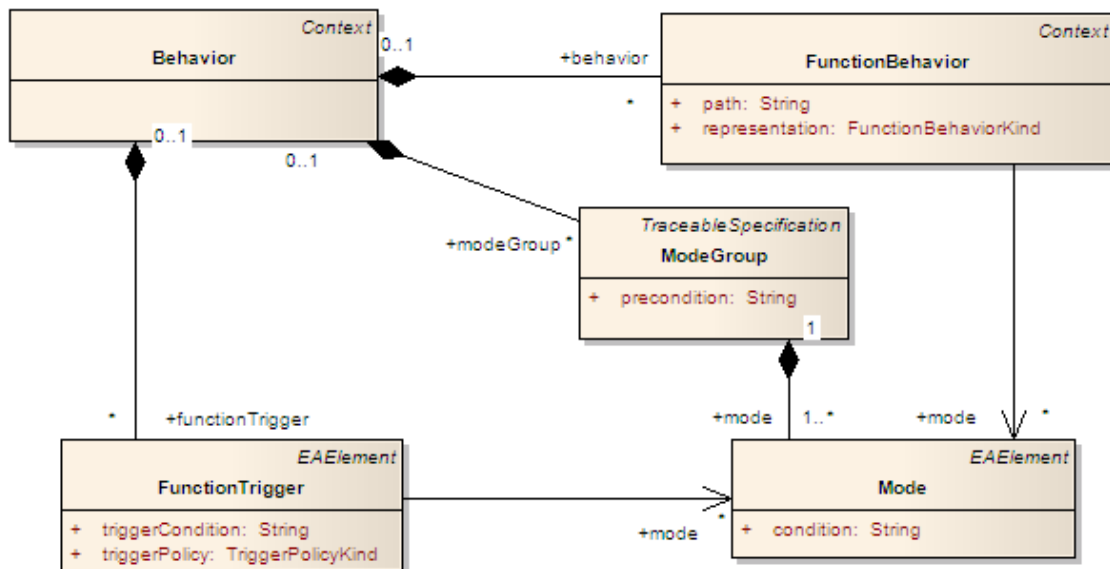
Figure 11. BehaviorModelOrganization. Diagram for behavior model organization.

9.2   Element Descriptions

9.2.1   Behavior (from Behavior)

Generalizations
- Context (from Elements)

Description

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

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

Attributes
No additional attributes

Associations
- behavior : FunctionBehavior [*] (from Behavior)
  This is the set of FunctionBehaviors managed by the container.

- modeGroup : ModeGroup [*] (from Behavior)
  The contained mode groups.

- functionTrigger : FunctionTrigger [*] (from Behavior)

Constraints
No additional constraints

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

9.2.2   FunctionBehavior (from Behavior)

Generalizations
- Context (from Elements)

Description
FunctionBehavior represents the behavior of a particular FunctionType - referred to by the association to FunctionType. What is meant by behavior is a transfer function performing some data computation (in case of FlowPort interaction) or an operation that can be called by another function (in case of ClientServer interaction). The representation property indicates the kind of representation used to describe the behavior (see FunctionBehaviorKind). The representation itself (e.g., defined in an external model file) is identified by a URL String in the path property. If the representation is provided in the same model file as the system itself, the path property is not used. It is merely a placeholder 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 refer to the same FunctionType. In this case, when a FunctionType has several behaviors, only one behavior shall be active at any given time instant, i.e., no concurrent behaviors are allowed in EAST-ADL functions. For instance we cannot have one active behavior in Simulink and one in Modelica. Both can be referenced in the same function, but at any given time, only one is executable. Conditions such as modes and variability must prevent two behaviors being potentially active.

Note also that FunctionBehaviors are assigned to FunctionTypes and not to FunctionPrototypes. This means that among a set of FunctionPrototypes, which share the same type, behaviors are also shared. However when a FunctionBehavior refer to Modes, which are referred to by different

FunctionTriggers, different triggering conditions can be provided among a set of FunctionPrototypes for the same set of behaviors - see FunctionTrigger.

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

Attributes
- path : String [1]
  The path to the file or model entity containing the behavior.

- representation : FunctionBehaviorKind [1] (from Behavior)
  The type of representation used to describe the behavior.

Associations
- function : FunctionType [0..1] (from FunctionModeling)
  The FunctionType to which the behavior is assigned.

- mode : Mode [*] (from Behavior)
  The execution Modes in which the behavior can be potentially executed.

Constraints
No additional constraints

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

1. Read inputs from input ports

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

3. Provide outputs to output ports

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

9.2.3   FunctionBehaviorKind (from Behavior) «enumeration»

Generalizations
None

Description
FunctionBehaviorKind is an enumeration which lists the various 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

Associations
No additional Associations

Constraints
No additional constraints

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

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

9.2.4	FunctionTrigger (from Behavior)

Generalizations
- EAElement (from Elements)

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

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

Attributes
- triggerCondition : String [1]
  A Boolean expression that must evaluate to true for this Function to execute. This value is used both for time and event triggered elementary functions.

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

Associations
- port : FunctionPort [*] (from FunctionModeling)
  The FunctionPorts that are referred to in the FunctionTrigger.

  The FunctionPorts that act as triggers individually or as specified in the triggerCondition

- function : FunctionType [0..1] (from FunctionModeling)
  The FunctionType that the FunctionTrigger refers to.

- functionPrototype : FunctionPrototype [0..1] (from FunctionModeling)
  The FunctionPrototype that the FunctionTrigger refers to

- mode : Mode [*] (from Behavior)
  The execution Modes in which the FunctionTrigger is active.

Constraints
No additional constraints

Semantics
Association Mode defines in which modes the trigger is active

The FunctionBehavior referenced by the FunctionTrigger is invoked when the FunctionTrigger is active. If multiple ports are referenced, it is sufficient if one is active before triggering. If AND semantics is desired, this is stated in the trigger condition.

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

9.2.5   Mode (from Behavior)

Generalizations
- EAElement (from Elements)

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

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

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

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

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

Associations
No additional Associations

Constraints
No additional constraints

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

9.2.6   ModeGroup (from Behavior)

Generalizations
- TraceableSpecification (from Elements)

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

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

Associations
- mode : Mode [1..*] (from Behavior)
  The modes in this group.

Constraints
No additional constraints

Semantics
-

9.2.7   TriggerPolicyKind (from Behavior) «enumeration»

Generalizations
None

Description
TriggerPolicyKind represents an enumeration for triggering policies.

Enumeration Literals
- EVENT
  Triggering by event.

- TIME
  Triggering by time.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
The TriggerPolicyKind contains EVENT and TIME as possible triggering policies.

Part IV Variability

This part covers variability extension to EAST-ADL.

10   Variability

10.1   Overview

This package contains elements to express variability in the analysis architecture, design architecture and implementation architecture. These abstraction levels in EAST-ADL will sometimes be called the artifact levels.
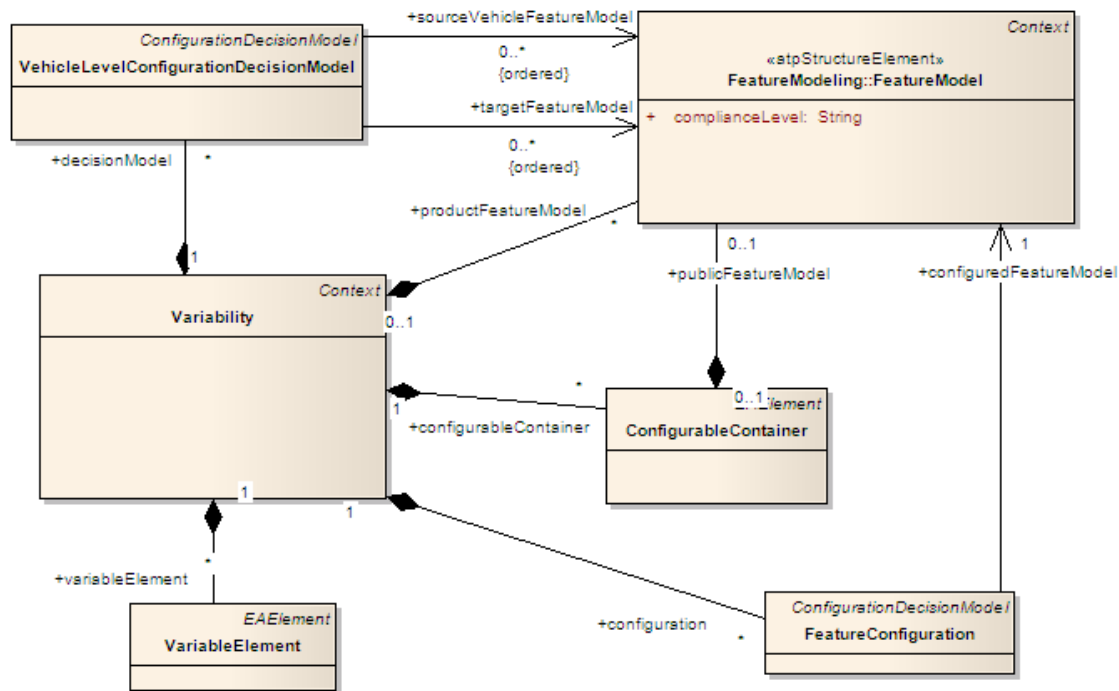
Figure 12. VariabilityElementsOrganization. Diagram depicting the organization of variability modeling elements.
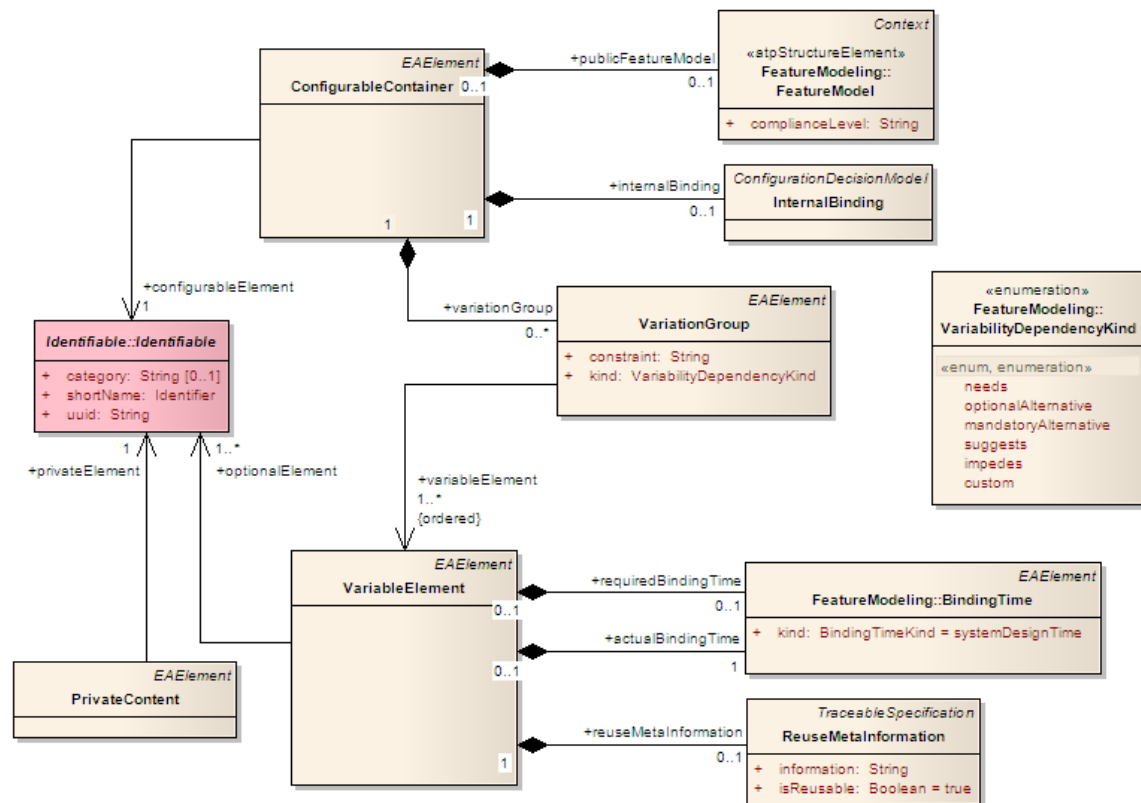
Figure 13. ArtifactLevelVariationManagement. Diagram depicting the elements involved in artifact-level variation management.
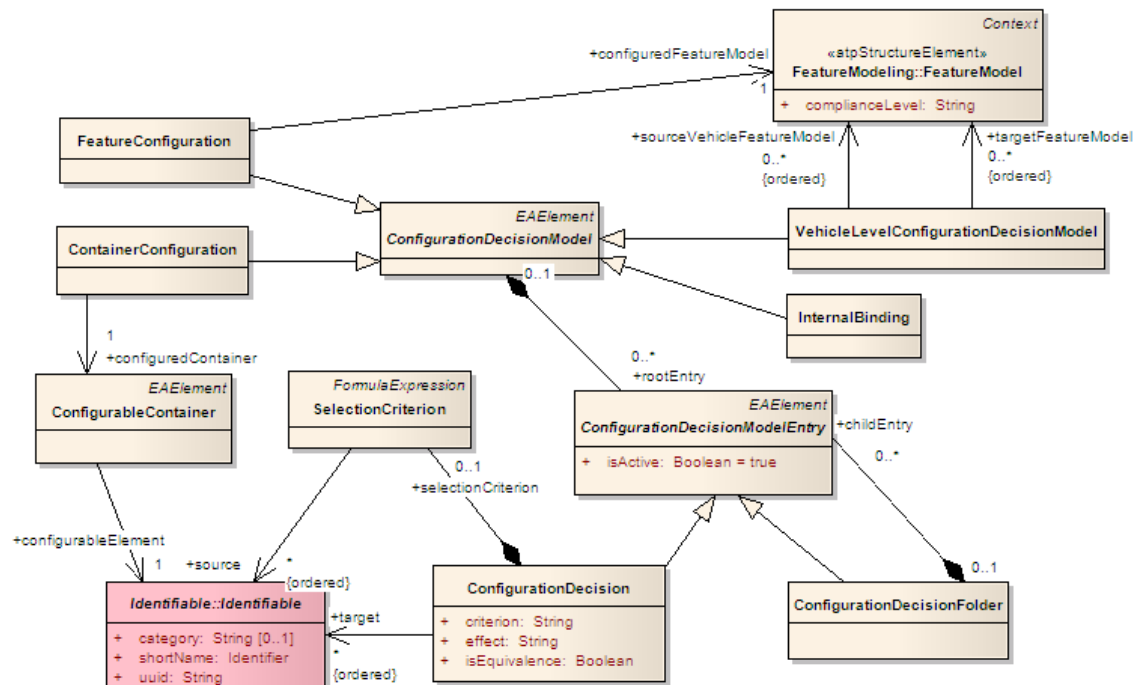
Figure 14. ConfigurationModeling. Diagram depicting the elements for configuration modeling.

10.2 Element Descriptions

10.2.1   ConfigurableContainer (from Variability)

Generalizations
- EAElement (from Elements)

Description

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

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

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

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

Attributes

No additional attributes

Associations
- publicFeatureModel : FeatureModel [0..1] (from FeatureModeling)
  The local feature model of the ConfigurableContainer.

  PublicFeatureModel represents internal variability of a ConfigurableContainer. Thus it can be seen as being part of the public interface of a ConfigurableContainer.

- internalBinding : InternalBinding [0..1] (from Variability)
  The ConfigurationDecisionModel of the ConfigurableContainer.

- variationGroup : VariationGroup [0..*] (from Variability)
  The variation groups that define certain dependencies and constraints between this ConfigurableContainer's variable elements.

- configurableElement : [1]
  This association points to the actual element in the core model that is marked as a configurable container of some variable content by this ConfigurableContainer. The

ConfigurableContainer in the variability extension can thus be seen as merely a marker element (this marker mechanism follows the global guideline for relating the EAST-ADL extensions to the core and is not specific to the variability extension).

Constraints

[1] Identifies one FunctionType or one HardwareComponentType. [2] The publicFeatureModel is only allowed to contain Features (no VehicleFeatures).

Semantics

-

10.2.2   ConfigurationDecision (from Variability)

Generalizations

- ConfigurationDecisionModelEntry (from Variability)

Description

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

Example: Lets assume we have two FeatureModels: FM1 and FM2. FM1 has possible end-customer decisions like USA, Canada, EU, Japan and A-Class, C-Class, etc. FM2 has another possible end-customer decision such as CruiseControl, AdaptiveCruiseControl, RearWiper, RainSensor. End-customer decisions in FM2 describe possible technical features of the delivered products. By way of a set of ConfigurationDecisions it is now possible to define the configuration of FM2 (i.e. if there is a RainSensor, etc.) 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[+]".

Attributes

- criterion : String [1]
  The criterion is a logical expression on the source configuration(s) and states under which condition the 'effect' will be applied to the target configuration(s). This attribute adheres to the syntax and semantics of the VSL language.

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

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

Configuring a cloned feature does not mean selecting or deselecting it but instead instances of the cloned feature are created. Each such instance is provided with a name, which thus becomes a part of the configuration (not the feature model). If several instances are created for a single cloned feature, then the name is used to identify these instances. For example, consider a cloned feature Wiper with cardinality [0..*]. A first configuration decision might create an instance called "front" and a second might create another named "rear"; a third configuration decision creating or otherwise referring to an instance called "front" will denote the same instance as the first configuration decision. The name space for these instance names is a particular feature configuration.

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

FMa#Wiper.RainSensor

or simply write:

RainSensor

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

FMa#ClimateControl.Advanced

or simply:

ClimateControl.Advanced

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

FMb#Wiper

- isEquivalence : Boolean [1]
  Setting the attribute isEquivalence to true means that the features referred to in the ConfigurationDecision's effect are exclusively configured by this ConfigurationDecision (i.e. no other ConfigurationDecision in the same ConfigurationDecisionModel may refer to these features). This means that this ConfigurationDecision is the ONLY way in which these features can be selected and therefore the usual logical implication that a ConfigurationDecision represents is turned into a logical equivalence, hence the name: the effect is applied to the target configurations if and only(!) if the specified criterion holds.

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

Associations
- selectionCriterion : SelectionCriterion [0..1] (from Variability)
  The mixed string expression.

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

Constraints
No additional constraints

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

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

10.2.3   ConfigurationDecisionFolder (from Variability)

Generalizations
- ConfigurationDecisionModelEntry (from Variability)

Description
ConfigurationDecisionFolder represents a grouping for ConfigurationDecisions.

Attributes
No additional attributes

Associations
- childEntry : ConfigurationDecisionModelEntry [0..*] (from Variability)
  The child entries of the ConfigurationDecisionFolder.

Constraints
No additional constraints

Semantics
ConfigurationDecisionFolder is a grouping entity for ConfigurationDecisions.

10.2.4   ConfigurationDecisionModel (from Variability) {abstract}

Generalizations
- EAElement (from Elements)

Description
A ConfigurationDecisionModel defines how to configure m 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, they are only defined explicitly for those used on vehicle level; for ConfigurationDecisionModels used as an internal binding on FunctionTypes, the source and target feature models are defined implicitly (cf. metaclass InternalBinding). In addition, in the special case of FeatureConfiguration there is by definition no source and only a single target feature model, which is defined explicitly (cf. metaclass FeatureConfiguration).

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

Attributes
No additional attributes

Associations
- rootEntry : ConfigurationDecisionModelEntry [0..*] (from Variability)

The root entries of the ConfigurationDecisionModel.

Constraints

No additional constraints

Semantics

-

10.2.5   ConfigurationDecisionModelEntry (from Variability) {abstract}

Generalizations
- EAElement (from Elements)

Description

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

Attributes
- isActive : Boolean = true [1]

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

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

Associations

No additional Associations

Constraints

No additional constraints

Semantics

-

10.2.6   ContainerConfiguration (from Variability)

Generalizations
- ConfigurationDecisionModel (from Variability)

Description

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

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

Attributes

No additional attributes

Associations
- configuredContainer : ConfigurableContainer [1] (from Variability)
  The ConfiguredContainer being configured by this ContainerConfiguration.

Constraints
No additional constraints

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

10.2.7   FeatureConfiguration (from Variability)

Generalizations
- ConfigurationDecisionModel (from Variability)

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

Note that configurations of feature models are realized as a specialization of metaclass ConfigurationDecisionModel. This is possible because a ConfigurationDecisionModel also captures 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 is no source feature model to which the criterion can refer, all ConfigurationDecisions in a FeatureConfiguration must have "true" as their criterion.

Attributes
No additional attributes

Associations
- configuredFeatureModel : FeatureModel [1] (from FeatureModeling)

Constraints
No additional constraints

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

10.2.8   InternalBinding (from Variability)

Generalizations
- ConfigurationDecisionModel (from Variability)

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

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

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

10.2.9   PrivateContent (from Variability)

Generalizations
  • EAElement (from Elements)

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

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

Attributes
No additional attributes

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

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

Semantics
-

10.2.10   ReuseMetaInformation (from Variability)

Generalizations
  • TraceableSpecification (from Elements)

Description
ReuseMetaInformation represents the description information needed in the context of reuse. For example a specific entity is only a short-time solution that is not intended to be reused. Also a

specific entity can only be reused for specific model ranges (that are not reflected in the product model). This kind of information can be stored in this information.

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

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

Associations
No additional Associations

Constraints
No additional constraints

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

10.2.11   SelectionCriterion (from Variability)

Generalizations
- FormulaExpression (from Elements)

Description
A mixed string description, identifying the source elements.

Attributes
No additional attributes

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

Constraints
No additional constraints

Semantics
-

10.2.12   Variability (from Variability)

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

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

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

- decisionModel : VehicleLevelConfigurationDecisionModel [*] (from Variability)
- configuration : FeatureConfiguration [*] (from Variability)
- variableElement : VariableElement [*] (from Variability)
- configurableContainer : ConfigurableContainer [*] (from Variability)

Constraints
No additional constraints

Semantics
-

10.2.13 VariableElement (from Variability)

Generalizations
- EAElement (from Elements)

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

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

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

Attributes
No additional attributes

Associations
- actualBindingTime : BindingTime [1] (from FeatureModeling)
  Actual binding time attribute. Due to technical conditions it may occur that the actually realized binding time of the feature/variation point differs from the originally intended binding time. In this case one has to provide information about the actual binding time. In the rationales it must be described what the reasons are for a (different) actual binding time.

- requiredBindingTime : BindingTime [0..1] (from FeatureModeling)
  Required binding time attribute. Each feature/variation point must have a required binding time attribute. The required binding time describes the binding time that the feature is intended to have.

- reuseMetaInformation : ReuseMetaInformation [0..1] (from Variability)
  Reuse-relevant meta-information for the element.

- optionalElement : [1..*]
  This association points to the actual element in the core model that is marked optional by this VariableElement. The VariableElement in the variability extension can thus be seen as merely a marker element (this marker mechanism follows the global guideline for relating the EAST-ADL extensions to the core and is not specific to the variability extension).

Constraints

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

Semantics

-

10.2.14   VariationGroup (from Variability)

Generalizations
- EAElement (from Elements)

Description

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

Attributes
- constraint : String [1]
  Only defined iff kind=="custom". An OCL constraint specifying how the VariableElements in the variation group can be combined.

- kind : VariabilityDependencyKind [1] (from FeatureModeling)
  The kind of the variation group (see enumeration VariationGroupKind).

Associations
- variableElement : VariableElement [1..*] {ordered} (from Variability)
  Associated variable elements.

Constraints

No additional constraints

Semantics

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

10.2.15   VehicleLevelConfigurationDecisionModel (from Variability)

Generalizations
- ConfigurationDecisionModel (from Variability)

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).

Attributes

No additional attributes

Associations
- sourceVehicleFeatureModel : FeatureModel [0..*] {ordered} (from FeatureModeling)
- targetFeatureModel : FeatureModel [0..*] {ordered} (from FeatureModeling)

Constraints

[1] The sourceVehicleFeatureModels shall only contain VehicleFeatures. [2] The sourceVehicleFeatureModels shall be different from the targetFeatureModels

Semantics

-

Part V Requirements

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

11   Requirements

11.1   Overview

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

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

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

Elements inspired by SysML are Requirement, Satisfy, Refine, DeriveRequirement, and Verify.
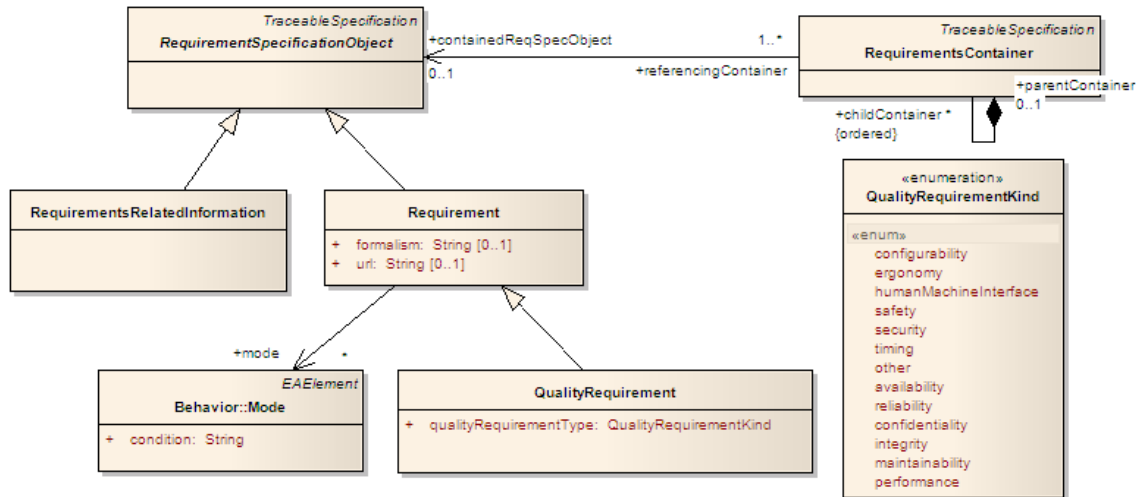
Figure 15. RequirementsOverview. Diagram for Requirements overview.

Figure 16. RequirementsRelationships. Diagram for Relationships including Requirement.

Figure 17. RequirementsOrganization. Diagram for Requirements organization.

11.2   Element Descriptions

11.2.1   DeriveRequirement (from Requirements)

Generalizations
- RequirementsRelationship (from Requirements)

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

Attributes
No additional attributes

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

- derived : Requirement [1..*] (from Requirements)
  The set of requirements derived from the supplier requirement.

Constraints
No additional constraints

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

11.2.2   OperationalSituation (from Requirements)

Generalizations
- TraceableSpecification (from Elements)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

11.2.3   QualityRequirement (from Requirements)

Generalizations
- Requirement (from Requirements)

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

The attribute qualityRequirementType allows a more specific classification.

Attributes
- qualityRequirementType : QualityRequirementKind [1] (from Requirements)

Associations
No additional Associations

Constraints
No additional constraints

Semantics
QualityRequirement element represent a requirement which is non-functional.

11.2.4   QualityRequirementKind (from Requirements) «enumeration»

Generalizations
None

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

Enumeration Literals
- availability
  The requirement is related to availability, the readiness for correct service.

- confidentiality
  The requirement is related to confidentiality.

- configurability
  The requirement is related to the ability to configure the functionality.

- ergonomy
  The requirement is related to the ergonomy of the functionality.

- humanMachineInterface
  The requirement is related to the human-machine interface.

- integrity
  The requirement is related to integrity, absence of improper system alteration.

- maintainability
  The requirement is related to maintainability, the ability to undergo modifications and repairs.

- other
  The requirement is a quality requirement with a general classification.

- performance
  The requirement is related to performance in general.

- reliability
  The requirement is related to reliability, the continuity of correct service.

- safety
  The requirement is related to safety, the absence of catastrophic consequences on the user(s) and the environment.

- security
  The requirement is related to security.

- timing
  The requirement is related to timing.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

11.2.5   Refine (from Requirements)

Generalizations
- RequirementsRelationship (from Requirements)

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

Attributes
No additional attributes

Associations
- refinedRequirement : Requirement [1..*] (from Requirements)
  List of refined Requirements.

- refinedBy : Refine_refinedBy [1..*] (from _instanceRef)

Dependencies
- refinedBy: EAElement [1..*] (from Elements)
«instanceRef»

Constraints
No additional constraints

Semantics
The Refine metaclass signifies a refined requirement/refined by relationship between a Requirement and an EAElement, where the modification of the supplier Requirement may impact the refining client EAElement. The Refine metaclass implies the semantics that the refining client EAElement is not complete, without the supplier Requirement.

11.2.6   Requirement (from Requirements)

Generalizations
- RequirementSpecificationObject (from 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 with 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.

Attributes
- formalism : String [0..1]
  Specifies the language used for the requirement statement.

- url : String [0..1]

Reference to possible external file containing the requirement statement.

Associations
- mode : Mode [*] (from Behavior)
  The mode where this requirement is valid.

Constraints
No additional constraints

Semantics
The string defined in the Text attribute of the Requirement metaclass applies to the EAElement that is associated to the Requirement through the Satisfy relation.

11.2.7   RequirementSpecificationObject (from Requirements) {abstract}

Generalizations
- TraceableSpecification (from Elements)

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).

Attributes
No additional attributes

Associations
- referencingContainer : RequirementsContainer [1..*] (from Requirements)
  Several containers may have a reference to one requirement specification object. But at least one container shall stay in reference with a requirement specification object. 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.

Constraints
No additional constraints

Semantics
-

11.2.8   RequirementsContainer (from Requirements)

Generalizations
- TraceableSpecification (from Elements)

Description
RequirementContainer represents a larger unit or module of specification information. It is used to bundle 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.

Furthermore, the RequirementContainer allows the introduction of 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 RequirementContainers (see parent child relationship). Only non-root RequirementContainers that have a parentContainer are allowed to reference a RequirementSpecificationObject.

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

Attributes
No additional attributes

Associations
- containedReqSpecObject : RequirementSpecificationObject [0..1] (from Requirements)
  E.g. a pure requirement or a rational etc.

  This relationship couples requirement specification object and requirement container. Such coupling is only allowed when the requirement container is not a root requirement container.

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

- parentContainer : RequirementsContainer [0..1] (from Requirements)
  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.

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

Semantics
-

11.2.9   RequirementsLink (from Requirements)

Generalizations
- RequirementsRelationship (from Requirements)

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

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

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

Associations
- target : Requirement [1..*] (from Requirements)
  The requirement(s) at which this relation ends.

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

Constraints
No additional constraints

Semantics
-

11.2.10   RequirementsModel (from Requirements)

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

Associations
- relationship : RequirementsRelationship [*] (from Requirements)
- requirementContainer : RequirementsContainer [*] (from Requirements)
- requirement : RequirementSpecificationObject [*] (from Requirements)
- operationalSituation : OperationalSituation [*] (from Requirements)
- useCase : UseCase [*] (from UseCases)

Constraints
No additional constraints

Semantics
-

11.2.11   RequirementsRelatedInformation (from Requirements)

Generalizations
- RequirementSpecificationObject (from Requirements)

Description
This is a placeholder for all objects which are not Requirements (such as Rational, Explanations, Related Material etc.). For example, an element of type RequirementsRelatedInformation, which is a rational of an element of type Requirement, will directly succeed this requirement as a sibling element (see structuring of requirement elements via RequirementContainer).

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
This metaclass can be used to represent 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.

11.2.12   RequirementsRelationGroup (from Requirements)

Generalizations
- TraceableSpecification (from Elements)

Description
RequirementsRelationGroup represents a group of relations between Requirements.

Attributes
No additional attributes

Associations
- relation : RequirementsLink [1..*] (from Requirements)
  The relations that are grouped by this relation group. Note that this is not a containment association, i.e., a single relation may be grouped by several RequirementRelationGroups.

Constraints
No additional constraints

Semantics
-

11.2.13   RequirementsRelationship (from Requirements) {abstract}

Generalizations
- Relationship (from Elements)

Description
Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

11.2.14   Satisfy (from Requirements)

Generalizations
- RequirementsRelationship (from Requirements)

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

Attributes
No additional attributes

Associations
- satisfiedRequirement : Requirement [*] (from Requirements)
  List of Requirements that are satisfied by the client ADLElement or satisfied by the client AUTOSAR element.

- satisfiedUseCase : UseCase [*] (from UseCases)
  List of satisfied UseCases that are satisfied by the client EAElements or satisfied by the client AUTOSAR elements.

- satisfiedBy : Satisfy_satisfiedBy [*] (from _instanceRef)

Dependencies
- satisfiedBy: [1..*]

«instanceRef»

Constraints
No additional constraints

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

12   UseCases

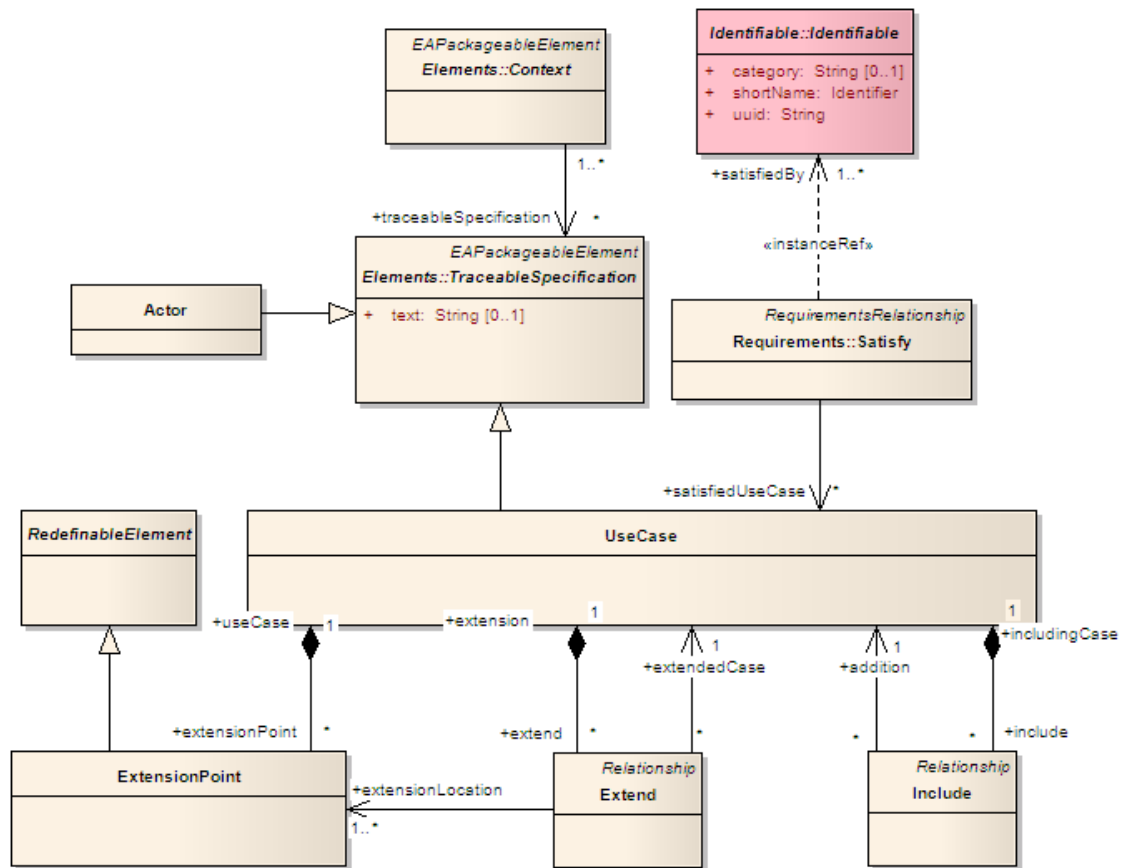The use case package conteins elements for defining the required usage of a system.

Figure 18. UseCase. Diagram for UseCase.

12.1   Element Descriptions

12.1.1   Actor (from UseCases)

Generalizations
- TraceableSpecification (from Elements)

Description

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

Attributes

No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

12.1.2   Extend (from UseCases)

Generalizations
- Relationship (from Elements)

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

Attributes
No additional attributes

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

- extension : UseCase [1] (from UseCases)
  The (usually supplementary) UseCase that extend the UseCase.

- extendedCase : UseCase [1] (from UseCases)
  The UseCase that is extended.

Constraints
No additional constraints

Semantics
-

12.1.3   ExtensionPoint (from UseCases)

Generalizations
- RedefinableElement (from UseCases)

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

Attributes
No additional attributes

Associations
- useCase : UseCase [1] (from UseCases)
  The UseCase that can be augmented with elements of another (extending) UseCase.

Constraints
No additional constraints

Semantics
-

12.1.4 Include (from UseCases)

Generalizations
- Relationship (from Elements)

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

Attributes
No additional attributes

Associations
- addition : UseCase [1] (from UseCases)
  UseCase providing behavior to include.

- includingCase : UseCase [1] (from UseCases)
  Including UseCase.

Constraints
No additional constraints

Semantics
-

12.1.5 RedefinableElement (from UseCases) {abstract}

Generalizations
None

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

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

The RedefinableElement is an abstract metaclass.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

12.1.6   UseCase (from UseCases)

Generalizations
- TraceableSpecification (from Elements)

Description
A UseCase specifies the required usage of a system. Typically, they are used to capture the functional requirements of a system, that is, what a system is supposed to do. Strictly speaking, the term "use case" refers to a use case type. An instance of a UseCase refers to an occurrence of the emergent behavior that conforms to the corresponding use case type. Such instances are often described by interaction specifications.

Attributes
No additional attributes

Associations
- extensionPoint : ExtensionPoint [*] (from UseCases)
  An ExtensionPoint identifies a point where the behavior of a UseCase can be augmented with elements of another (extending) UseCase.

- include : Include [*] (from UseCases)
  Include is a Relationship between two UseCases; the behavior of the included UseCase is inserted into the behavior of the including UseCase.

- extend : Extend [*] (from UseCases)
  This Relationship specifies that the behavior of a UseCase may be extended by the behavior of another (usually supplementary) UseCase.

Constraints
No additional constraints

Semantics
-

13 VerificationValidation

13.1 Overview

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

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

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

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

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

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

Using a "what vs. how" definition of requirements one could say: the abstract level defines what needs to be done to verify and validate a certain system, but not precisely how this is done. Conversely, the concrete level defines the precise technical details of particular testing environments. The abstract VVCases and VVProcedures for a particular system form a "to-do"-list, which describes what needs to be done when actually testing the system with a concrete testing environment, but in a form applicable to all conceivable testing environments.
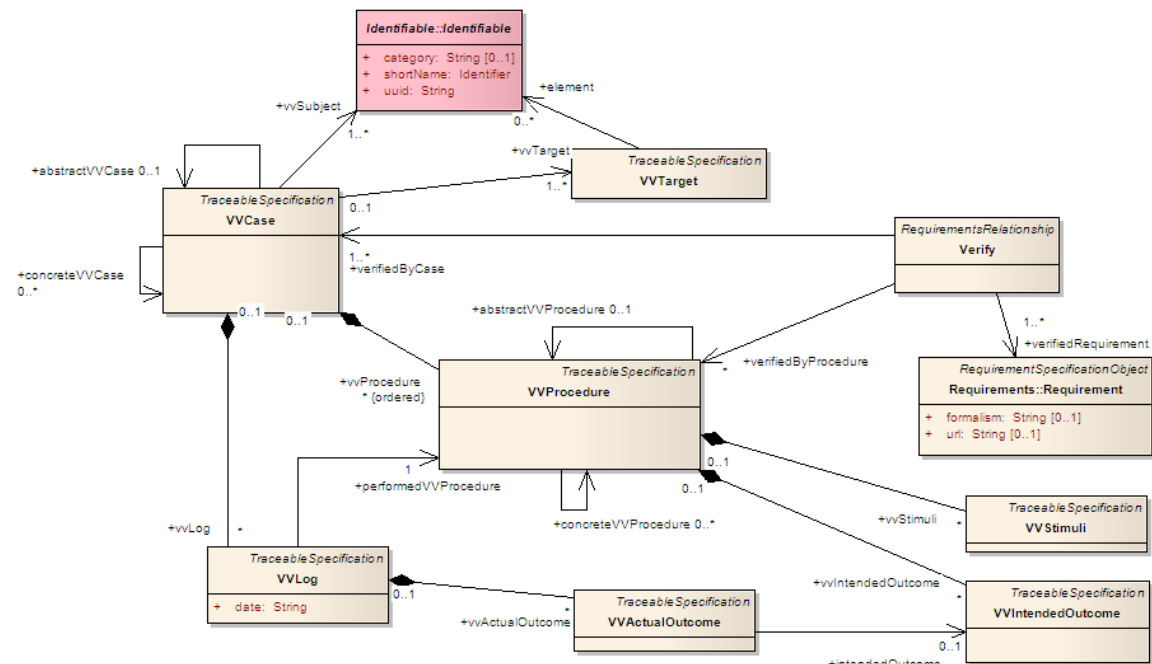
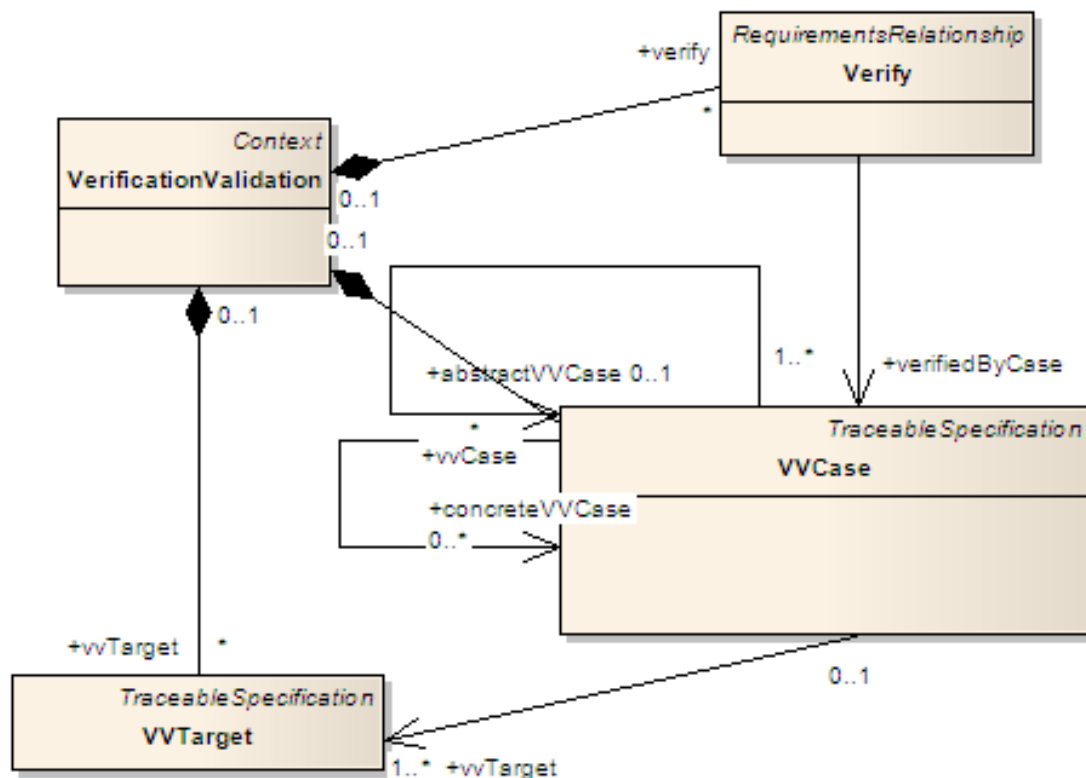**Figure 19. VerificationValidation. Diagram for Verification & Validation.**

Figure 20. VerificationValidationOrganization. Diagram for Verification and Validation Organization.

13.2 Element Descriptions

13.2.1   VVActualOutcome (from VerificationValidation)

Generalizations
- TraceableSpecification (from Elements)

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

Attributes
No additional attributes

Associations
- intendedOutcome : VVIntendedOutcome [0..1] (from VerificationValidation)
  Denotes the VVIntendedOutcome that this actual outcome must match.

Constraints
No additional constraints

Semantics
-

13.2.2   VVCase (from VerificationValidation)

Generalizations
- TraceableSpecification (from Elements)

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

Attributes
No additional attributes

Associations
- vvProcedure : VVProcedure [*] {ordered} (from VerificationValidation)
  The VVProcedures for this VVCase.

- vvTarget : VVTarget [1..*] (from VerificationValidation)
  The VVTargets for this VVCase. See association 'vvSubjects' for more information.

- vvLog : VVLog [*] (from VerificationValidation)
  The VVLogs captured while executing this ConcreteVVCase.

- vvSubject : [1..*]
  The elements that are being verified and validated by this VVCase.

  Usually this will be a subset of those elements which are realized by the VVTarget(s) of the VVCase; but this need not always be the case.

  The difference between the vvSubjects and the entities which are realized by the case's VVTarget(s), is that the vvSubjects are related to the primary, overall objective of the ConcreteVVCase, while the realized entities can comprise more than these. For example:

  (a) For technical reasons additional entities need to be realized only to permit the testing of the entities of actual interest or

  (b) If a VVTarget is reused among many ConcreteVVCases and therefore realizes more entities than are actually being tested by any single ConcreteVVCase.

- concreteVVCase : VVCase [0..*] (from VerificationValidation)
  A concreteVVCase not only describes "what" needs to be done for a particular verification and validation effort, but also the necessary details of "how" this is done.

- abstractVVCase : VVCase [0..1] (from VerificationValidation)
  An abstractVVCase describes "what" needs to be done.

Constraints
No additional constraints

Semantics
-

13.2.3 VVIntendedOutcome (from VerificationValidation)

Generalizations
- TraceableSpecification (from Elements)

Description
VVIntendedOutcome represents the expected output of the testing environment represented by VVTarget when triggered by the corresponding VVStimuli of the containing 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 that can be directly compared to the output of the VVTarget(s) defined for the containing ConcreteVVCase.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

13.2.4 VVLog (from VerificationValidation)

Generalizations
- TraceableSpecification (from Elements)

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 an 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 results from Friday and Monday respectively.

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

Associations
- performedVVProcedure : VVProcedure [1] (from VerificationValidation)
  Associated procedure.

- vvActualOutcome : VVActualOutcome [*] (from VerificationValidation)
  Set of outcome results.

Constraints
No additional constraints

Semantics
-

13.2.5   VVProcedure (from VerificationValidation)

Generalizations
- TraceableSpecification (from Elements)

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

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

Attributes
No additional attributes

Associations
- abstractVVProcedure : VVProcedure [0..1] (from VerificationValidation)
- vvStimuli : VVStimuli [*] (from VerificationValidation)
  Set of involved stimuli.

- concreteVVProcedure : VVProcedure [0..*] (from VerificationValidation)
- vvIntendedOutcome : VVIntendedOutcome [*] (from VerificationValidation)
  Set of intended outcomes.

Constraints
No additional constraints

Semantics
-

13.2.6   VVStimuli (from VerificationValidation)

Generalizations
- TraceableSpecification (from Elements)

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

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

13.2.7   VVTarget (from VerificationValidation)

Generalizations
  * TraceableSpecification (from Elements)

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

Usually, a VVTarget will 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.

Attributes
No additional attributes

Associations
  * element : [0..*]
    Alternative would be to remove this association and use Realize for the dependency between VVTarget and EAElement.

Association Documentation:

Constraints
No additional constraints

Semantics
-

13.2.8   VerificationValidation (from VerificationValidation)

Generalizations
  * Context (from Elements)

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

Attributes
No additional attributes

Associations
  * vvTarget : VVTarget [*] (from VerificationValidation)
  * vvCase : VVCase [*] (from VerificationValidation)
  * verify : Verify [*] (from VerificationValidation)

Constraints
No additional constraints

Semantics
-

13.2.9   Verify (from VerificationValidation)

Generalizations
- RequirementsRelationship (from Requirements)

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

Attributes
No additional attributes

Associations
- verifiedRequirement : Requirement [1..*] (from Requirements)
  The set of Requirements which the client VVCase verify.

- verifiedByProcedure : VVProcedure [*] (from VerificationValidation)
  The AbstractVVProcedures used to verify the Requirement.

- verifiedByCase : VVCase [1..*] (from VerificationValidation)
  TheVVCase that verifies the supplier Requirement

Constraints
No additional constraints

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

-

14   Interchange

14.1   Overview

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 and exporting requirements specifications into and out of an EAST-ADL system model.
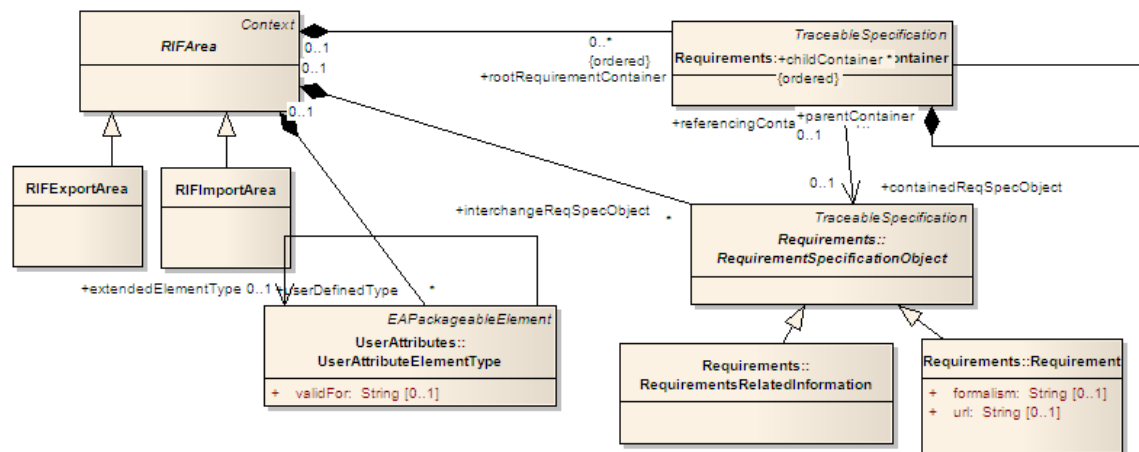
Figure 21. RIFArea. Diagram for RIF Area.

14.2   Element Descriptions

14.2.1   RIFArea (from Interchange) {abstract}

Generalizations
- Context (from Elements)

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

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

Requirement data to be imported/exported will be put into an RIFArea. Requirement data elements which are not inside an RIFArea but which 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.

Attributes
No additional attributes

Associations
- rootRequirementContainer : RequirementsContainer [0..*] {ordered} (from Requirements)
- interchangeReqSpecObject : RequirementSpecificationObject [*] (from Requirements)
- userDefinedType : UserAttributeElementType [*] (from UserAttributes)

Contained defined types.

Constraints
No additional constraints

Semantics
-

14.2.2   RIFExportArea (from Interchange)

Generalizations
- RIFArea (from Interchange)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

14.2.3   RIFImportArea (from Interchange)

Generalizations
- RIFArea (from Interchange)

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

When an element is imported from an external source the uuid will be taken from the given external exchange data file, because the identifier is globally unique and should not be changed anywhere.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

Part VI Timing

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

15   Timing

15.1   Overview

The timing package contains constructs for defining timing constraints.
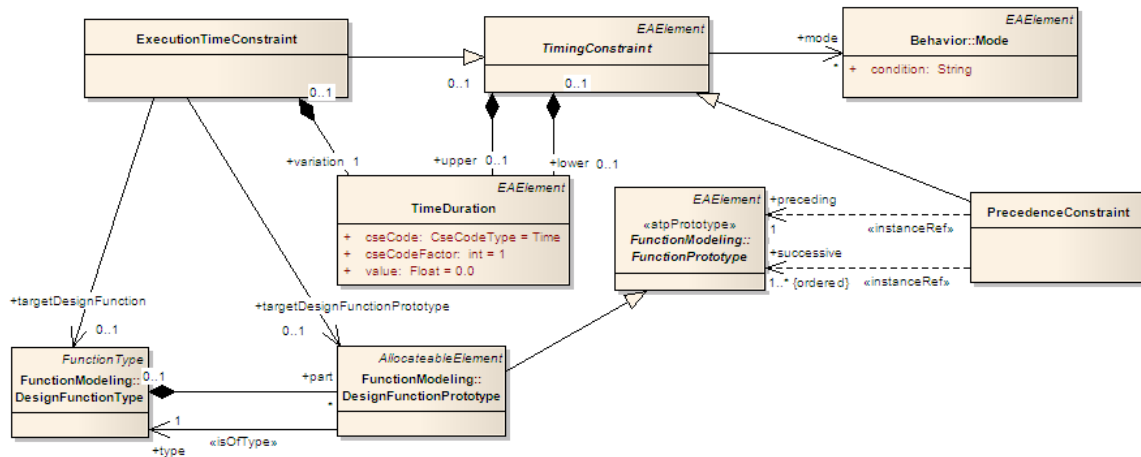
Figure 22. ExecutionTime. Diagram for Execution time.
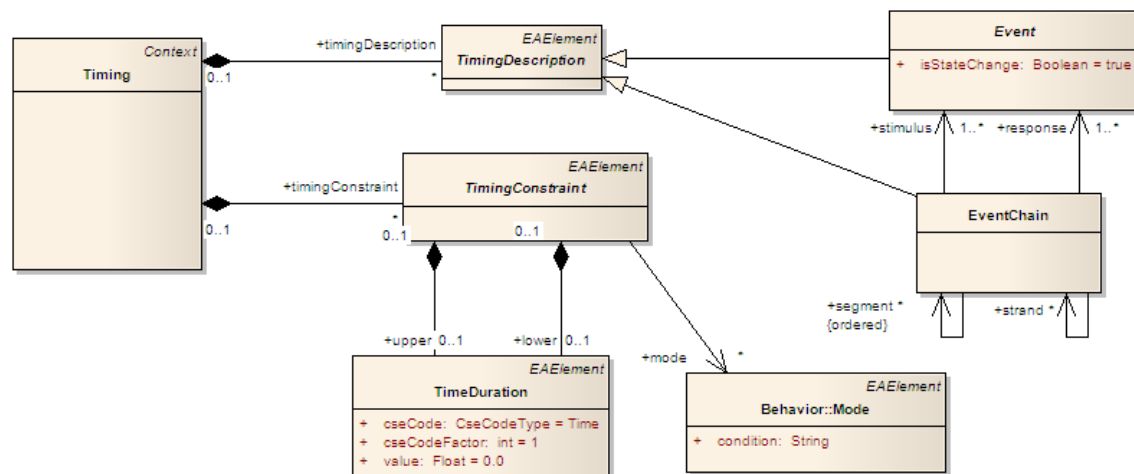
Figure 23. TimingElementsOrganization. Diagram for the extensions of EAST-ADL abstraction layers with timing information. Timing description and timing constraints are contained in elements dedicated to the abstraction level.

15.2   Element Descriptions

15.2.1   Event (from Timing) {abstract}

Generalizations
- TimingDescription (from Timing)

Description

An Event (E) denotes 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 always succeed stimuli.

An event occurs instantaneously, which means that an event occurs at an instant of time 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.

Attributes

- isStateChange : Boolean = 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.

Associations

No additional Associations

Constraints

[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.

Semantics

-

15.2.2   EventChain (from Timing)

Generalizations

- TimingDescription (from Timing)

Description

Event chains describe the temporal behavior of a number of steps to be taken to respond to one or more events. [An event chain is also used to express that a temporal requirement/constraint is imposed on these steps (-> 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 and 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.

Attributes
No additional attributes

Associations
- segment : EventChain [*] {ordered} (from Timing)
  Referred EventChains that are not parallel and in sequence refine this EventChain.

- response : Event [1..*] (from Timing)
  The Response element is the entity to describe an event that is a response to a stimulus that occurred before.

- stimulus : Event [1..*] (from Timing)
  The Stimulus element is the entity to describe an event that stimulates the steps to be taken to respond to this event.

- strand : EventChain [*] (from Timing)
  Parallel EventChains refining this EventChain.

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

Semantics
-

15.2.3   ExecutionTimeConstraint (from Timing)

Generalizations
- TimingConstraint (from Timing)

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/or 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).

Attributes
No additional attributes

Associations
- targetDesignFunction : DesignFunctionType [0..1] (from FunctionModeling)
  Identifies the DesignFunction with this execution time.

- targetDesignFunctionPrototype : DesignFunctionPrototype [0..1] (from FunctionModeling)
  Identifies the DesignFunctionPrototype with this execution time, and may be used when the constraint applies to a prototype.

- variation : TimeDuration [1] (from Timing)
  Denotes the allowed variation in execution time, i.e. between minimal and maximal execution time.

Constraints

[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.

Semantics

-

15.2.4 PrecedenceConstraint (from Timing)

Generalizations
- TimingConstraint (from Timing)

Description

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

Attributes

No additional attributes

Associations
- preceding : PrecedenceConstraint_preceding [1] (from _instanceRef)
- successive : PrecedenceConstraint_successive [1] {ordered} (from _instanceRef)

Dependencies
- successive: FunctionPrototype [1..*] (from FunctionModeling)

«instanceRef»

- preceding: FunctionPrototype [1] (from FunctionModeling)

«instanceRef»

Constraints

No additional constraints

Semantics

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

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

15.2.5 TimeDuration (from Timing)

Generalizations
- EAElement (from Elements)

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).

Attributes
- cseCode : CseCodeType = Time [1]
  This is normally time, note that when it is expressed as angle it can be converted to time.

- cseCodeFactor : int = 1 [1]
  Is normally equal to 1.

- value : Float = 0.0 [1]
  The actual value complemented with the cseCode.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

15.2.6   Timing (from Timing)

Generalizations
- Context (from Elements)

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.

Attributes
No additional attributes

Associations
- timingConstraint : TimingConstraint [*] (from Timing)
- timingDescription : TimingDescription [*] (from Timing)

Constraints
No additional constraints

Semantics
-

15.2.7   TimingConstraint (from Timing) {abstract}

Generalizations
- EAElement (from Elements)

Description
TimingConstraint is an abstract entity that identifies a mode.

Attributes
No additional attributes

Associations
- mode : Mode [*] (from Behavior)
  The mode where the TimingConstraint is valid.

- upper : TimeDuration [0..1] (from Timing)
  denotes a maximal value (e.g. worst case execution time)

- lower : TimeDuration [0..1] (from Timing)
  denotes a minimal value (e.g. best case execution time)

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

Semantics
-

15.2.8   TimingDescription (from Timing) {abstract}

Generalizations
- EAElement (from Elements)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

16   Events

16.1   Overview

This section describes the concept of events for EAST-ADL.
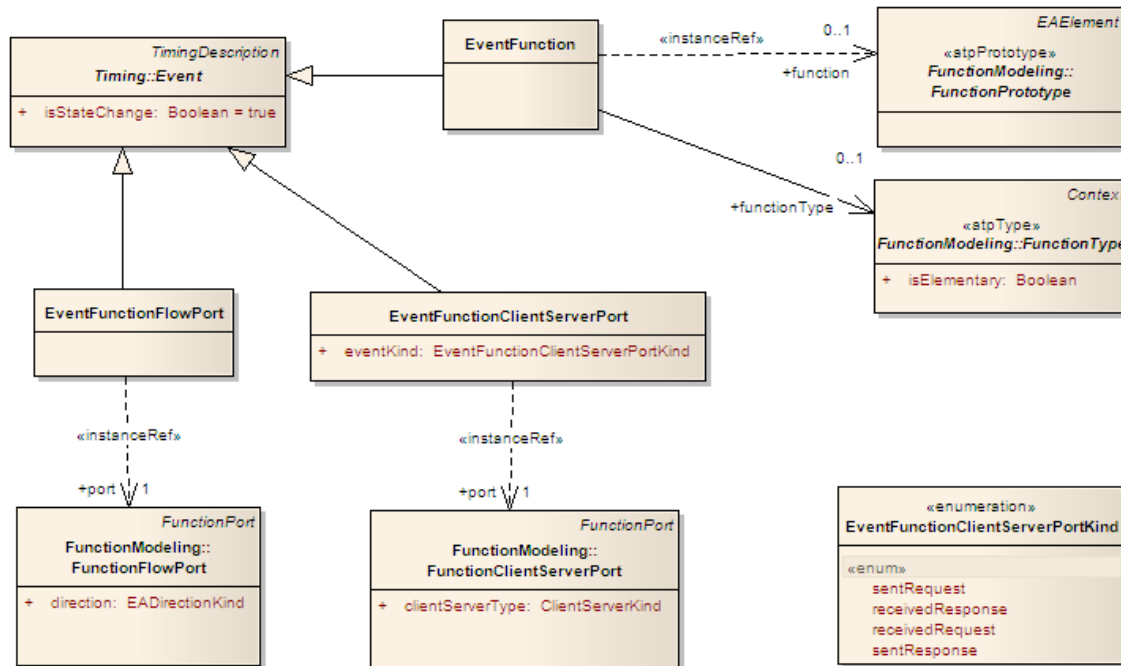
Figure 24. Events. The events for EAST-ADL functional modeling.

16.2   Element Descriptions

16.2.1   EventFunction (from Events)

Generalizations
- Event (from Timing)

Description

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

Compare categories of AUTOSAR runnables:

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

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

Attributes
No additional attributes

Associations
- functionType : FunctionType [0..1] (from FunctionModeling)
  The event is valid for all occurences of this function.

- function : EventFunction_function [1] (from _instanceRef)

Dependencies
- function: FunctionPrototype [0..1] (from FunctionModeling)
«instanceRef»

Constraints
No additional constraints

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

16.2.2   EventFunctionClientServerPort (from Events)

Generalizations
- Event (from Timing)

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

Attributes
- eventKind : EventFunctionClientServerPortKind [1] (from Events)

Associations
- port : EventFunctionClientServerPort_port [1] (from _instanceRef)

Dependencies
- port: FunctionClientServerPort [1] (from FunctionModeling)
«instanceRef»

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

Semantics
-

16.2.3   EventFunctionClientServerPortKind (from Events) «enumeration»

Generalizations
None

Description
Possible values of eventKind.

Enumeration Literals
- receivedRequest
  Request arrived at server.

- receivedResponse
  Response arrived at client.

- sentRequest
  Request sent from client.

- sentResponse
  Response sent from server.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
See each literal.

16.2.4   EventFunctionFlowPort (from Events)

Generalizations
- Event (from Timing)

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

Attributes
No additional attributes

Associations
- port : EventFunctionFlowPort_port [1] (from _instanceRef)

Dependencies
- port: FunctionFlowPort [1] (from FunctionModeling)

«instanceRef»

Constraints
No additional constraints

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

- -

17 TimingConstraints

17.1 Overview

This section describes the timing constraints.
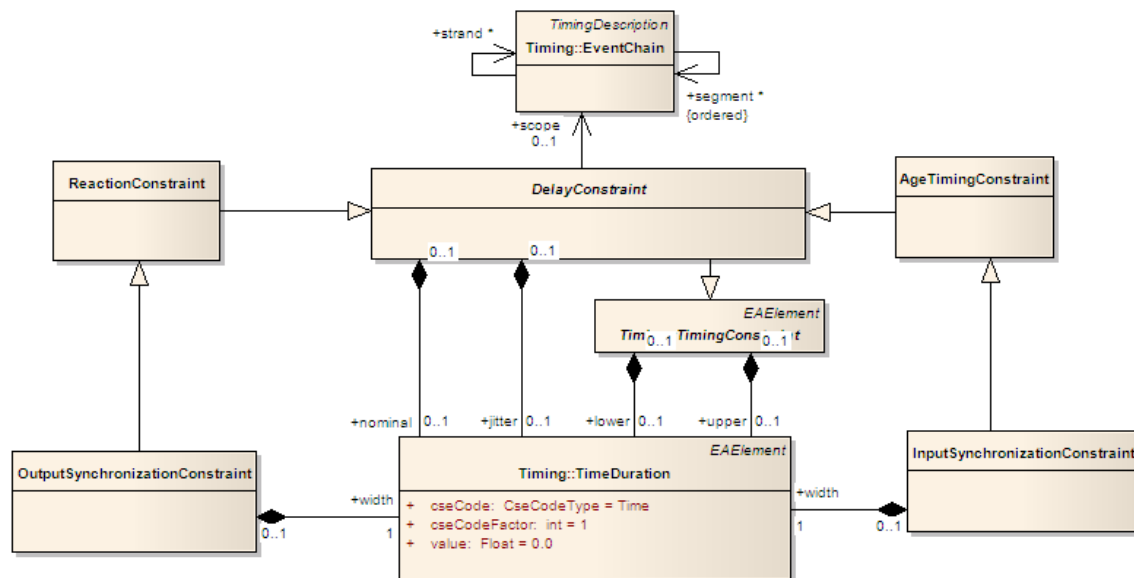
Figure 25. DelayConstraints. The constraints shown here imposes constraints on Event Chain.
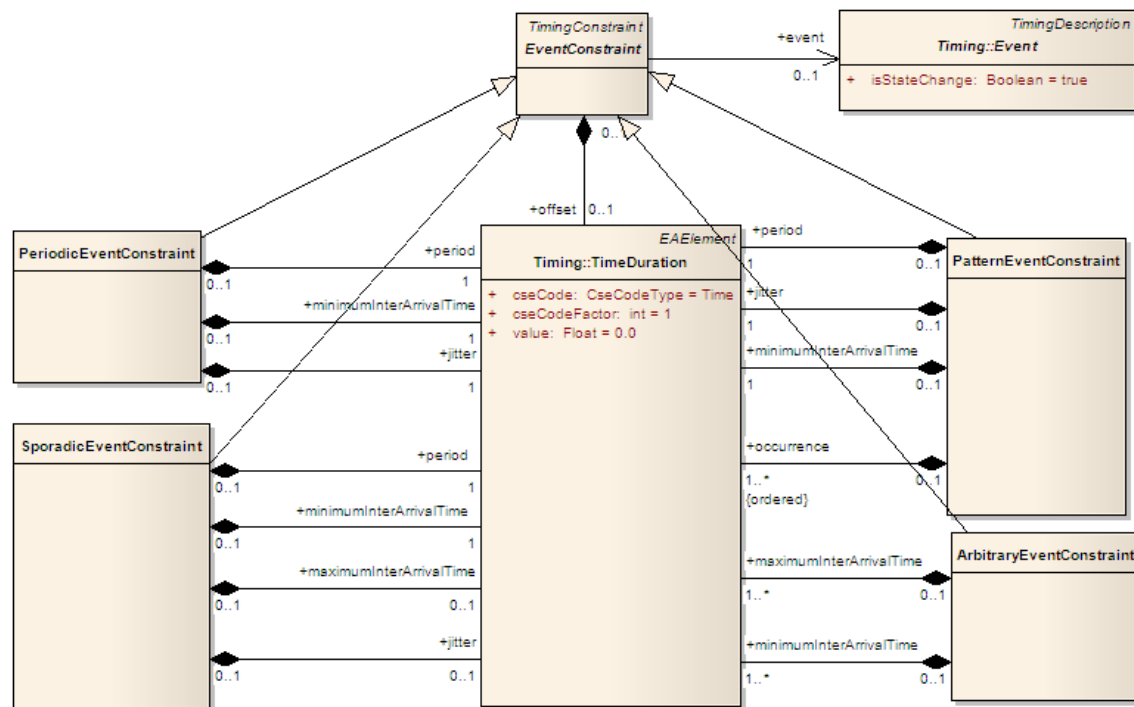
Figure 26. EventConstraint. The constraints shown here imposes constraints on Event.

17.2 Element Descriptions

17.2.1 AgeTimingConstraint (from TimingConstraints)

Generalizations
- DelayConstraint (from TimingConstraints)

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, a one-to-one relation is not 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.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

17.2.2 ArbitraryEventConstraint (from TimingConstraints)

Generalizations
- EventConstraint (from TimingConstraints)

Description

The Arbitrary Event Model describes whether an event occurs occasionally, singly, irregularly 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.

Attributes
No additional attributes

Associations
- minimumInterArrivalTime : TimeDuration [1..*] (from Timing)
  The set of minimum inter-arrival times specifies the minimum inter-arrival time between two or 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 on.

- maximumInterArrivalTime : TimeDuration [1..*] (from Timing)

The set of maximum inter-arrival times specifies the maximum inter-arrival time between two or 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 on.

Constraints
[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.

Semantics
-

17.2.3 DelayConstraint (from TimingConstraints) {abstract}

Generalizations
- TimingConstraint (from Timing)

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. For example, [lower=10, upper=20, nominal=15] means a nominal value of 15 +/- 5. This is equivalent to [nominal=15, jitter=10], i.e. the nominal value varies by +/- 5 around 15. Note that the nominal value may also vary asymmetrically, e.g. [lower=10, nominal=12, upper=20]. Defining [nominal=15], without upper/lower or jitter, denotes an exact value of 15 without variations.

The bound will be measured in a given unit, see TimeDuration.

Attributes
No additional attributes

Associations
- jitter : TimeDuration [0..1] (from Timing)
  Jitter is the range within which a value varies.

- nominal : TimeDuration [0..1] (from Timing)
  The recurring distance between each occurrence.

- scope : EventChain [0..1] (from Timing)
  The EventChain on which this constraint is applied.

Constraints
[1] Exactly one of the following combinations of upper, lower, jitter, and nominal shall be specified: {upper, lower}, {upper, lower, jitter}, {upper}, {lower}, {nominal, jitter}. Any combination may in addition have a nominal parameter. If nominal is defined, it shall be in the range [lower ... upper]. Rationale: At least one value is necessary to describe a reasonable DelayConstraint, and the given combinations are sufficient to describe all possible variations.

Semantics
-

17.2.4 EventConstraint (from TimingConstraints) {abstract}

Generalizations
- TimingConstraint (from Timing)

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

Attributes
No additional attributes

Associations
- offset : TimeDuration [0..1] (from Timing)
  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.

- event : Event [0..1] (from Timing)

Constraints
No additional constraints

Semantics
-

17.2.5   InputSynchronizationConstraint (from TimingConstraints)

Generalizations
- AgeTimingConstraint (from TimingConstraints)

Description
InputSynchronizationConstraint is a language entity that expresses a timing constraint on the input synchronization among the set of stimulus events. Basically, the InputSynchronizationConstraint looks from the response event(s) into the past to the stimuli events. All stimulus events must occur within a given sliding window. The sliding window itself may occur within a time interval specified by means of a minimum and maximum distance from the response event(s).

Attributes
No additional attributes

Associations
- width : TimeDuration [1] (from Timing)
  The width of the sliding window.

Constraints
No additional constraints

Semantics
The parameters of InputSynchronizationConstraint, see TimingConstraint, constrain the time from the first stimulus until last stimulus (i.e., maximum skew between these stimuli). Parameter width defines the sliding window, i.e. the maximum distance between the first and the last stimulus event shall be smaller or equal to width. Furthermore, the minimum and maximum distances of the sliding window to the response event(s) is defined by the parameters upper and lower (from TimingConstraint). In this case, upper denotes the maximal allowed distance from the last response event to the first stimulus event (looking backwards in time), and lower denotes the minimal allowed distance from the first response event to the last stimulus event (looking backwards in time).

A join point is identified by the response event in the scope EventChain.

17.2.6   OutputSynchronizationConstraint (from TimingConstraints)

Generalizations
- ReactionConstraint (from TimingConstraints)

Description
OutputSynchronizationConstraint is a language entity that expresses a timing constraint on the output synchronization among the set of response events. Basically, the OutputSynchronizationConstraint looks from the stimulus event(s) into the future to the response

events. All response events must occur within a given sliding window. The sliding window itself may occur within a time interval specified by means of a minimum and maximum distance from the stimulus event(s).

Attributes
No additional attributes

Associations
- width : TimeDuration [1] (from Timing)
  The width of the sliding window.

Constraints
No additional constraints

Semantics
The parameters of OutputSynchronizationConstraint, see TimingConstraints, constrain the time from the first response until last response (i.e., maximum skew between these responses). Parameter width defines the sliding window, i.e. the maximum distance between the first and the last response event shall be smaller or equal to width. Furthermore, the minimum and maximum distances of the sliding window to the stimulus event(s) is defined by the parameters upper and lower (from TimingConstraint). In this case, upper denotes the maximal allowed distance from the first stimulus event to the first response event, and lower denotes the minimal allowed distance from the last stimulus event to the first response event.

A fork point is identified by the stimulus event in the scope EventChain.

17.2.7   PatternEventConstraint (from TimingConstraints)

Generalizations
- EventConstraint (from TimingConstraints)

Description
The [Concrete] PatternEventConstraint describes that an event occurs following a known pattern.

Attributes
No additional attributes

Associations
- period : TimeDuration [1] (from Timing)
  The period specifies the time interval within which the event occurs any number of times following a pattern.

- minimumInterArrivalTime : TimeDuration [1] (from Timing)
  The minimum inter-arrival time specifies the minimal possible time interval between two consecutive occurrences of the event within the given period.

- occurrence : TimeDuration [1..*] {ordered} (from Timing)
  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

- jitter : TimeDuration [1] (from Timing)
  The jitter specifies maximal possible time interval the occurrence of the events within the given period can vary (formerly: can be delayed).

Constraints
No additional constraints

Semantics
-

17.2.8   PeriodicEventConstraint (from TimingConstraints)

Generalizations
- EventConstraint (from TimingConstraints)

Description
The PeriodicEventConstraint describes that an event occurs periodically.

Attributes
No additional attributes

Associations
- jitter : TimeDuration [1] (from Timing)
  The jitter specifies the maximal possible time interval the occurrence of an event can vary (formerly: be delayed).

- period : TimeDuration [1] (from Timing)
  The period specifies the [ideal] time interval between two subsequent occurrences of the event.

- minimumInterArrivalTime : TimeDuration [1] (from Timing)
  The minimum inter-arrival time specifies the minimal possible time interval between two consecutive occurrences of an event.

Constraints
No additional constraints

Semantics
-

17.2.9   ReactionConstraint (from TimingConstraints)

Generalizations
- DelayConstraint (from TimingConstraints)

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.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

17.2.10   SporadicEventConstraint (from TimingConstraints)

Generalizations
- EventConstraint (from TimingConstraints)

Description

The SporadicEventConstraint describes that an event occurs occasionally. In general it is supposed that the event eventually occurs. However, 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.

Attributes

No additional attributes

Associations

- jitter : TimeDuration [0..1] (from Timing)
  The optional parameter jitter specifies the maximal possible time interval the occurrence of an event can vary (formerly: be delayed). By its nature, a sporadic event can occur at any time, thus the occurrence is characteristically jittery.

- period : TimeDuration [1] (from Timing)
  The period specifies the [ideal] time interval between two subsequent occurrences of the event.

- maximumInterArrivalTime : TimeDuration [0..1] (from Timing)
  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] (from Timing)
  The minimum inter-arrival time specifies is the minimal possible time interval between two consecutive occurrences of an event.

Constraints

No additional constraints

Semantics

-

Part VII Dependability

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

Part VII Dependability

18   Dependability

18.1   Overview

Dependability of a system is the system's ability to ensure service failures are at a level of frequency and severity that is acceptable. Dependability includes several aspects, namely Availability, Reliability, Safety, Integrity and Maintainability. The Dependability package includes support for defining and classifying safety requirements through preliminary Hazard Analysis Risk Assessment, tracing and categorizing safety requirements according to 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.
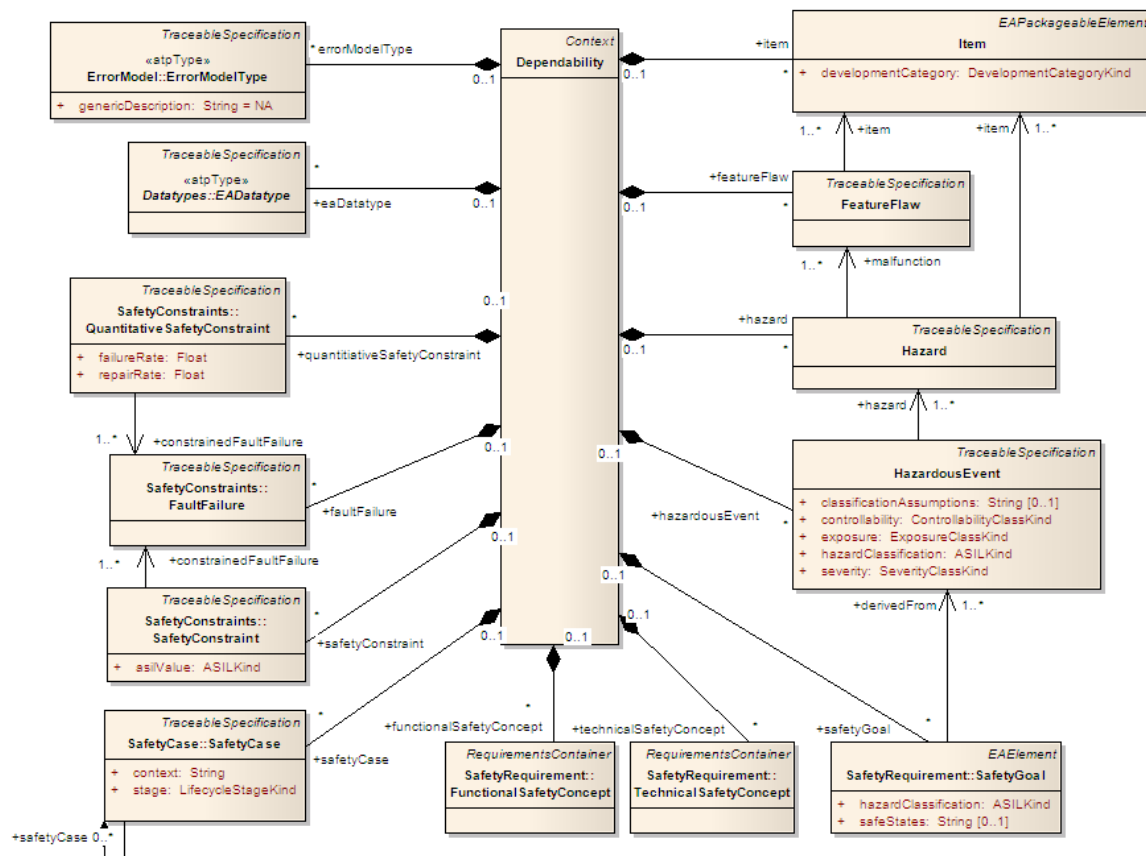
Figure 27. DependabilityOrganization. Diagram for organization of dependability related information.
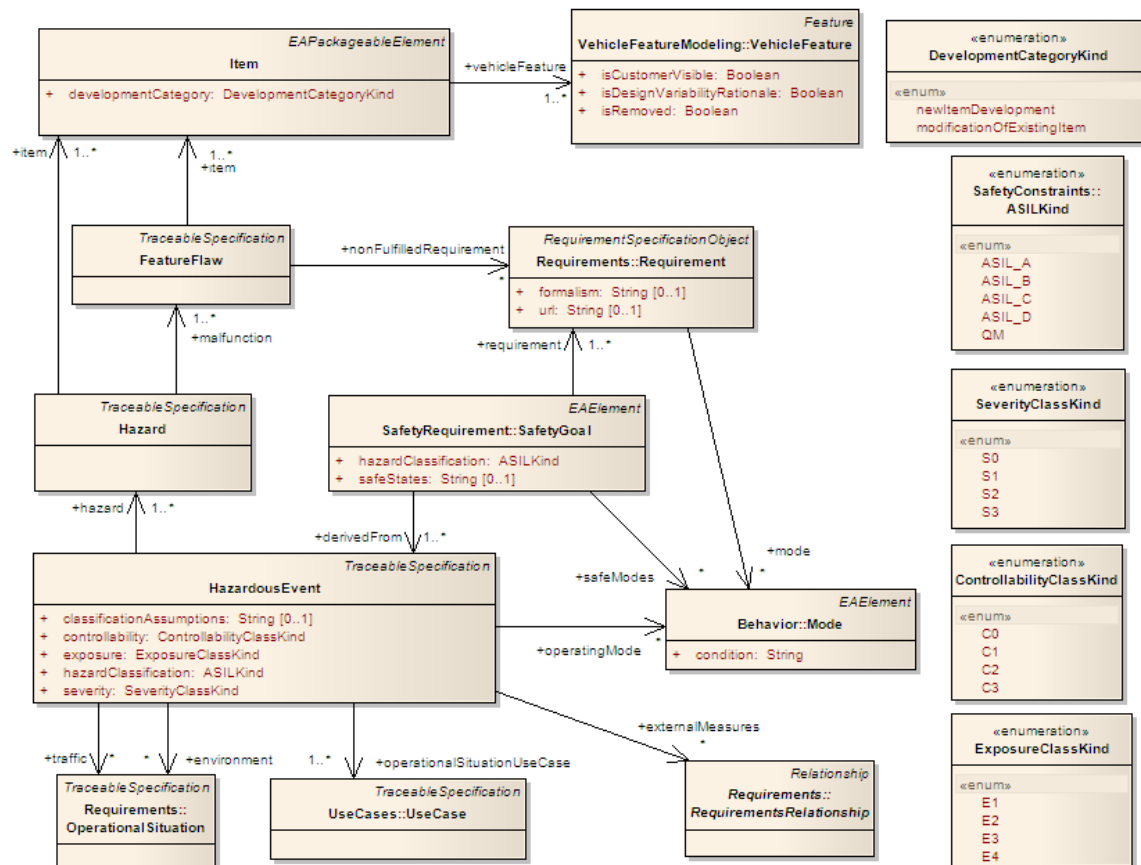
Figure 28. Dependability. Diagram for Dependability.

18.2 Element Descriptions

18.2.1 ControllabilityClassKind (from Dependability) «enumeration»

Generalizations
None

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

Enumeration Literals
- C0
  Controllable in general.

- C1
  Simply controllable.

- C2
  Normally controllable.

- C3
  Difficult to control or uncontrollable.

Associations
No additional Associations

Constraints
No additional constraints

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

18.2.2 Dependability (from Dependability)

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

Associations
- technicalSafetyConcept : TechnicalSafetyConcept [*] (from SafetyRequirement)
- eaDatatype : EADatatype [*] (from Datatypes)
  Datatypes defined in this context.

- safetyCase : SafetyCase [*] (from SafetyCase)
- quantitiativeSafetyConstraint : QuantitativeSafetyConstraint [*] (from SafetyConstraints)
- hazard : Hazard [*] (from Dependability)
- functionalSafetyConcept : FunctionalSafetyConcept [*] (from SafetyRequirement)
- faultFailure : FaultFailure [*] (from SafetyConstraints)
- errorModelType : ErrorModelType [*] (from ErrorModel)
- featureFlaw : FeatureFlaw [*] (from Dependability)
- item : Item [*] (from Dependability)
- safetyGoal : SafetyGoal [*] (from SafetyRequirement)
- safetyConstraint : SafetyConstraint [*] (from SafetyConstraints)
- hazardousEvent : HazardousEvent [*] (from Dependability)

Constraints
No additional constraints

Semantics
-

18.2.3 DevelopmentCategoryKind (from Dependability) «enumeration»

Generalizations
None

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

Enumeration Literals
- modificationOfExistingItem
  In case of a modification the relevant lifecycle sub-phases and activities shall be determined.

- newItemDevelopment
  In case of a new development, the entire lifecycle shall be passed through.

Associations
No additional Associations

Constraints
No additional constraints

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

18.2.4   ExposureClassKind (from Dependability) «enumeration»

Generalizations
None

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

Enumeration Literals
- E1
  Rare events. Situations that occur less often than once a year for the great majority of drivers

- E2
  Sometimes. Situations that occur a few times a year for the great majority of drivers

- E3
  Quite often. Situations that occur once a month or more often for an average driver

- E4
  Often. All situations that occur during almost every drive on average

Associations
No additional Associations

Constraints
No additional constraints

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

18.2.5   FeatureFlaw (from Dependability)

Generalizations
- TraceableSpecification (from Elements)

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

Attributes
No additional attributes

Associations
- nonFulfilledRequirement : Requirement [*] (from Requirements)
  Identifies the requirements that are not fulfilled.

- item : Item [1..*] (from Dependability)

> The item for which the FeatureFlaw is identified

Constraints
No additional constraints

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

18.2.6 Hazard (from Dependability)

Generalizations
- TraceableSpecification (from Elements)

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

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

Attributes
No additional attributes

Associations
- item : Item [1..*] (from Dependability)
  The item for which the Hazard is identified

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

Constraints
No additional constraints

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

18.2.7 HazardousEvent (from Dependability)

Generalizations
- TraceableSpecification (from Elements)

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

Attributes
- classificationAssumptions : String [0..1]
  The classificationAssumptions attribute denotes assumptions concerning the classification of the Hazard.

- controllability : ControllabilityClassKind [1] (from Dependability)
  The controllability by the driver or other traffic participants defined by the enumeration C0, C1, C2 or C3 in accordance with ISO26262.

- exposure : ExposureClassKind [1] (from Dependability)

132 (209)

The probability of exposure of the operational situations defined by the probability attributes E1, E2, E3 or E4 in accordance with ISO26262.

- hazardClassification : ASILKind [1] (from SafetyConstraints)
  The ASIL-Level shall be determined for each hazardous event using the estimation parameters in accordance with ISO26262.

- severity : SeverityClassKind [1] (from Dependability)
  The severity of potential harm defined by the severity attributes S0, S1, S2 or S3 in accordance with ISO26262.

Associations
- operatingMode : Mode [*] (from Behavior)
  OperatingMode denotes the Operating mode of the item.

- externalMeasures : RequirementsRelationship [*] (from Requirements)
- traffic : OperationalSituation [*] (from Requirements)
  A definition of the traffic situation in terms of adjacent vehicles, pedestrians and other dynamic aspects. Represents the external and dynamic aspects of the vehicle operating situation .

- environment : OperationalSituation [*] (from Requirements)
  A definition of the road environment in terms of road conditions, lanes, geometry, etc. Represents the external and static aspects of the vehicle operating situation .

- operationalSituationUseCase : UseCase [1..*] (from UseCases)
  Operational situation with respect to the activities of actors, typically the driver.

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

Constraints
No additional constraints

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

18.2.8	Item (from Dependability)

Generalizations
- EAPackageableElement (from Elements)

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

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

Associations
- vehicleFeature : VehicleFeature [1..*] (from VehicleFeatureModeling)

Constraints
No additional constraints

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

18.2.9   SeverityClassKind (from Dependability) «enumeration»

Generalizations
None

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

Enumeration Literals
- S0
  No injuries.

- S1
  Light and moderate injuries.

- S2
  Severe and life-threatening injuries (survival probable).

- S3
  Life-threatening injuries (survival uncertain), fatal injuries.

Associations
No additional Associations

Constraints
No additional constraints

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

-

19   SafetyConstraints

19.1   Overview

The SafetyConstraints package contains constructs for defining safety constraints.
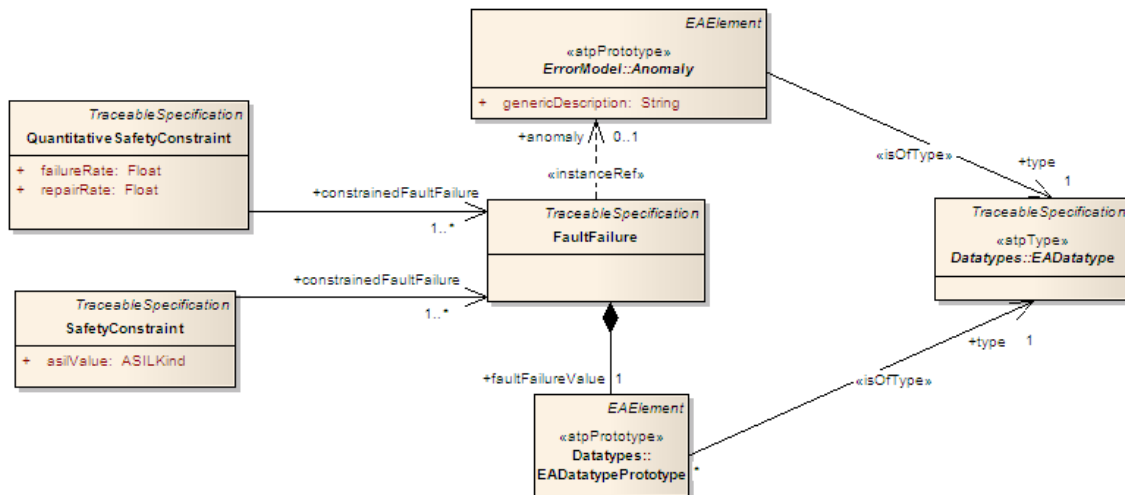
Figure 29. SafetyConstraints. Diagram for SafetyConstraints.

19.2   Element Descriptions

19.2.1   ASILKind (from SafetyConstraints) «enumeration»

Generalizations
None

Description
The ASILKind is an enumeration metaclass with enumeration literals indicating the level of safety integrity in accordance with ISO26262.

Enumeration Literals
- ASIL_A
  ASIL A, Lowest Safety Integrity Level.

- ASIL_B
  ASIL B, second lowest Safety Integrity Level.

- ASIL_C
  ASIL C, second highest Safety Integrity Level.

- ASIL_D
  ASIL D, Highest Safety Integrity Level.

- QM
  Quality Management only, no requirement according to ISO 26262.

Associations
No additional Associations

Constraints
No additional constraints

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

19.2.2   FaultFailure (from SafetyConstraints)

Generalizations
- TraceableSpecification (from Elements)

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

Attributes
No additional attributes

Associations
- anomaly : FaultFailure_anomaly [0..1] (from _instanceRef)
- faultFailureValue : EADatatypePrototype [1] (from Datatypes)
  The faultFailureValue defines the specific value among the possible faults or failures that the FaultFailure represents.

Dependencies
- anomaly: Anomaly [0..1] (from ErrorModel)
«instanceRef»

Constraints
No additional constraints

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

19.2.3   QuantitativeSafetyConstraint (from SafetyConstraints)

Generalizations
- TraceableSpecification (from Elements)

Description
The QuantitativeSafetyConstraint metaclass represents the quantitative integrity constraints on a fault or failure. Thus, the system has the same or better performance with respect to the constrained fault or failure, and depending on the role this is either a requirement or a property.

Attributes
- failureRate : Float [1]
  failureRate denotes the number of failures per unit time, i.e. the density of probability of failure divided by probability of survival for a hardware element (ISO26262 definition). For exponential failure distributions it is often denoted by lambda.

- repairRate : Float [1]
  repairRate denotes the number of repairs per unit time. For exponential repair distributions it is often denoted by mu.

Associations
- constrainedFaultFailure : FaultFailure [1..*] (from SafetyConstraints)
  A QuantitativeSafetyConstraint defines quantitative bounds on the constrainedFaultFailure in terms of the failure and repair rates, failureRate and repairRate. The rates are exponentially distributed (user defined attributes may be used to specify alternative distributions and additional quantitative parameters).

Constraints
No additional constraints

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

19.2.4   SafetyConstraint (from SafetyConstraints)

Generalizations
- TraceableSpecification (from Elements)

Description
The SafetyConstraint metaclass represents the qualitative integrity constraints on a fault or failure. Thus, the system has the same or better performance with respect to the constrained fault or failure, and depending on the role this is either a requirement or a property.

Attributes
- asilValue : ASILKind [1] (from SafetyConstraints)
  The ASIL level of the target fault or failure.

Associations
- constrainedFaultFailure : FaultFailure [1..*] (from SafetyConstraints)
  The constrained fault or failure.

Constraints
No additional constraints

Semantics
A SafetyConstraint defines qualitative bounds on the constrainedFaultFailure in terms of safety integrity level, asilValue.

Depending on role, the SafetyConstraint may define a required or an actual safety integrity level.

- -

20	**ErrorModel**

20.1 Overview

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

Abnormal behaviors of architectural elements as well as their instantiations in a particular product context can be represented. This forms a basis for safety analysis through external techniques and tools. Through the integration with other language constructs, definitions of error behaviors and hazards can be traced to the specifications of safety requirements, and further to the subsequent functional and non-functional requirements on error 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 the 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).
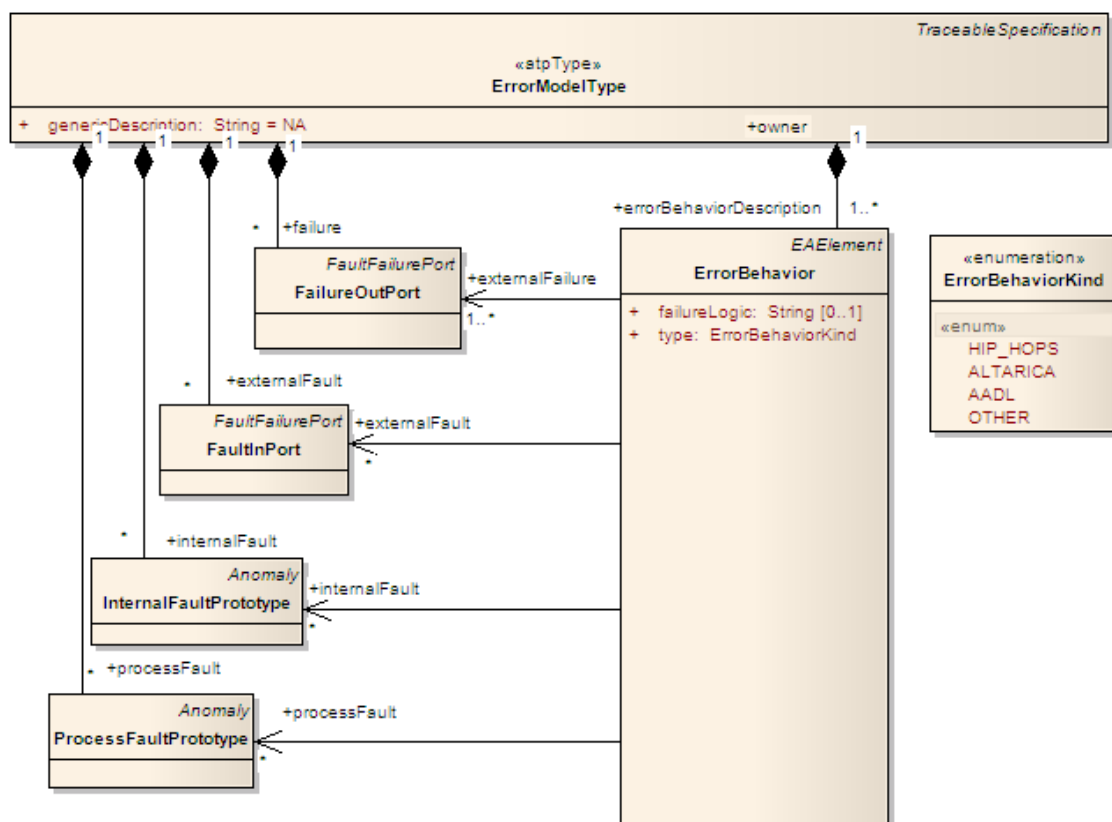
Figure 30. ErrorBehavior. Diagram for ErrorBehavior.
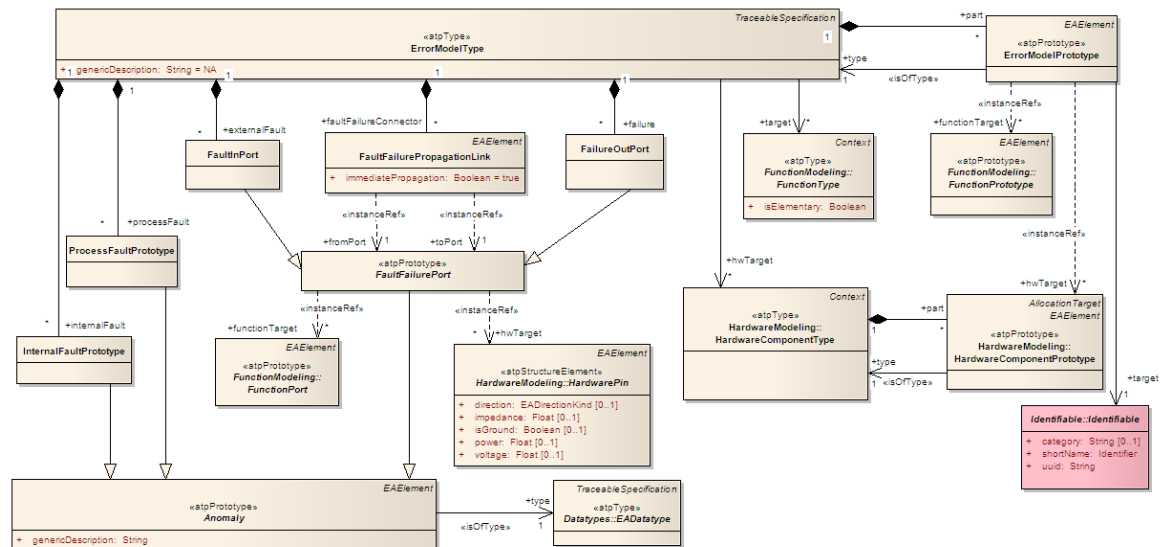
Figure 31. ErrorModel. The EAST-ADL metaclasses for defining the error model structure.

20.2 Element Descriptions

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

Generalizations
- EAElement (from Elements)

Description

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

BrakeAnomaly:

- BrakePressureTooLow

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

- Omission

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

- Comission

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

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

Attributes
- genericDescription : String [1]
  A description of the Anomaly

Associations
- type : EADatatype [1] (from Datatypes)
«isOfType»

The declaration of port type.

Constraints
No additional constraints

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

20.2.2   ErrorBehavior (from ErrorModel)

Generalizations
- EAElement (from Elements)

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

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

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

- type : ErrorBehaviorKind [1] (from ErrorModel)
  The type of formalism applied for the error behavior description.

Associations
- internalFault : InternalFaultPrototype [*] (from ErrorModel)
  internalFaults that influence the errorBehavior

- processFault : ProcessFaultPrototype [*] (from ErrorModel)
  processFaults that may affect the errorBehavior

- externalFailure : FailureOutPort [1..*] (from ErrorModel)
  Failures that may result from the ErrorBehavior

- externalFault : FaultInPort [*] (from ErrorModel)
  external(incoming) faults that influence the errorBehavior.

- owner : ErrorModelType [1] (from ErrorModel)
  the container ErrorModelType for the error behavior description.

Constraints
No additional constraints

Semantics

ErrorBehavior defines the error propagation logic of its containing ErrorModelType.

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

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

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

20.2.3   ErrorBehaviorKind (from ErrorModel) «enumeration»

Generalizations
None

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

Enumeration Literals
- AADL
  A specification of error behavior according to the external formalism AADL.

- ALTARICA
  A specification of error behavior according to the external formalism ALTARICA.

- HIP_HOPS
  A specification of error behavior according to the external formalism HiP-HOPS.

- OTHER
  A specification of error behavior according to other user defined formalism.

Associations
No additional Associations

Constraints
No additional constraints

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

20.2.4   ErrorModelPrototype (from ErrorModel) «atpPrototype»

Generalizations
- EAElement (from Elements)

Description
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.

Attributes
No additional attributes

Associations
- type : ErrorModelType [1] (from ErrorModel)
«isOfType»

>   The ErrorModelType that types the ErrorModelPrototype.

- target : [1]
  The target element (i.e., a system, a function, a component, or hardware device) owning the anomalies.

  ARElement can also be the target or ErrorModelType.

- hwTarget : ErrorModelPrototype_hwTarget [*] (from _instanceRef)
- functionTarget : ErrorModelPrototype_functionTarget [*] (from _instanceRef)

Dependencies
- functionTarget: FunctionPrototype [*] (from FunctionModeling)

«instanceRef»

- hwTarget: HardwareComponentPrototype [*] (from HardwareModeling)

«instanceRef»

Constraints
No additional constraints

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

20.2.5 ErrorModelType (from ErrorModel) «atpType»

Generalizations
- TraceableSpecification (from Elements)

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

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

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

ErrorModelType inherits the abstract metaclass TraceableSpecification, allowing the ErrorModelType to be referenced from its design context in a similar way as requirements, test cases and other specifications.

Attributes
- genericDescription : String = NA [1]

Associations
- target : FunctionType [*] (from FunctionModeling)
  The nominal FunctionType whose ErrorModel is defined by the ErrorModelType

- hwTarget : HardwareComponentType [*] (from HardwareModeling)
- internalFault : InternalFaultPrototype [*] (from ErrorModel)
  An internal fault that the ErrorModelType may propagate or mask

- faultFailureConnector : FaultFailurePropagationLink [*] (from ErrorModel)
  The contained links for internal propagations of faults/failures between the subordinate error models.

- processFault : ProcessFaultPrototype [*] (from ErrorModel)
  A processFault that affects the ErrorModelType. Process faults cannot be masked, and propagate to all defined externalFailures.

- part : ErrorModelPrototype [*] (from ErrorModel)

The contained error models forming a hierarchy.

- failure : FailureOutPort [*] (from ErrorModel)
  A failureOutPort represent a propagated Failure

- externalFault : FaultInPort [*] (from ErrorModel)
  An external fault that the ErrorModelType may propagate or mask

- errorBehaviorDescription : ErrorBehavior [1..*] (from ErrorModel)
  The description of failure logic of the target element.

Constraints
An ErrorModelType without part shall have one errorBehaviorDescription

Semantics
-

20.2.6   FailureOutPort (from ErrorModel)

Generalizations
- FaultFailurePort (from ErrorModel)

Description
The FailureOutPort represents a propagation point for failures that propagate out from the containing ErrorModelType.The EADatatype of the FailureOutPort defines the range of valid failures.

Attributes
No additional attributes

Associations
No additional Associations

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

Semantics
-

20.2.7   FaultFailurePort (from ErrorModel) {abstract} «atpPrototype»

Generalizations
- Anomaly (from ErrorModel)

Description
Abstract port for Faults and Failures.

Attributes
No additional attributes

Associations
- hwTarget : FaultFailurePort_hwTarget [*] (from _instanceRef)
- functionTarget : FaultFailurePort_functionTarget [*] (from _instanceRef)

Dependencies
- functionTarget: FunctionPort [*] (from FunctionModeling)
«instanceRef»

- hwTarget: HardwarePin [*] (from HardwareModeling)
«instanceRef»

Constraints
No additional constraints

Semantics
-

20.2.8   FaultFailurePropagationLink (from ErrorModel)

Generalizations
- EAElement (from Elements)

Description
The FaultFailurePropagationLink metaclass represents the links for the propagations of faults/failures across system elements. In particular, it defines that one error model provides the faults/failures that another error model receives.

A fault/failure link can only be applied to compatible ports, either for fault/failure delegation within an error model or for fault/failure transmission across two error models. A FaultFailurePropagationLink can only connect fault/failure ports that have compatible types.

Attributes
- immediatePropagation : Boolean = true [1]

Associations
- fromPort : FaultFailurePropagationLink_fromPort [1] (from _instanceRef)
- toPort : FaultFailurePropagationLink_toPort [1] (from _instanceRef)

Dependencies
- fromPort: FaultFailurePort [1] (from ErrorModel)

«instanceRef»

- toPort: FaultFailurePort [1] (from ErrorModel)

«instanceRef»

Constraints
- **OCL:**otherRelatedPropagation
  inv: not self.otherRelatedPropagations = self

- **OCL:**propagationMeans
  inv: (self.throughADLEntity->isEmpty   xor   self.throughARElement->isEmpty)   xor
  self.throughADLRealization->isEmpty

  inv: (self.throughARElement->notEmpty)   implies   (self.from.portOwner.targetARType->notEmpty and self.to.portOwner.targetARType->notEmpty) and (self.throughADLEntity <> self.to.portOwner.targetARType   and   self.throughARElement   <>
  self.from.portOwner.targetARElement)

  inv: (self.throughADLEntity->notEmpty)   implies   (self.from.portOwner.targetADLType->notEmpty and self.to.portOwner.targetADLType->notEmpty) and (self.throughADLEntity <>   self.to.portOwner.targetADLType   and   self.throughADLEntity   <>
  self.from.portOwner.targetADLType)   and
  (self.throughADLEntity.oclIsKindOf(ADLFunctionPrototype)   or
  self.throughADLEntity.oclIsKindOf(ADLConnectorPrototype))

Semantics
-

20.2.9 FaultInPort (from ErrorModel)

Generalizations
- FaultFailurePort (from ErrorModel)

Description
The FaultInPort represents a propagation point for faults that propagate to the containing ErrorModelType. The EADatatype of the FaultInPort defines the range of valid failures.

Attributes
No additional attributes

Associations
No additional Associations

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

Semantics
-

20.2.10 InternalFaultPrototype (from ErrorModel)

Generalizations
- Anomaly (from ErrorModel)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

20.2.11 ProcessFaultPrototype (from ErrorModel)

Generalizations
- Anomaly (from ErrorModel)

Description
The ProcessFaultPrototype metaclass represents the anomalies that the target component/system can have due to design or implementation flaws (e.g., incorrect requirements, buffer size configuration, scheduling, etc.).

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

145 (209)

Semantics

- -

21   SafetyRequirement

21.1   Overview

The SafetyRequirement package contains constructs for organizing ISO 26262 safety requirements.
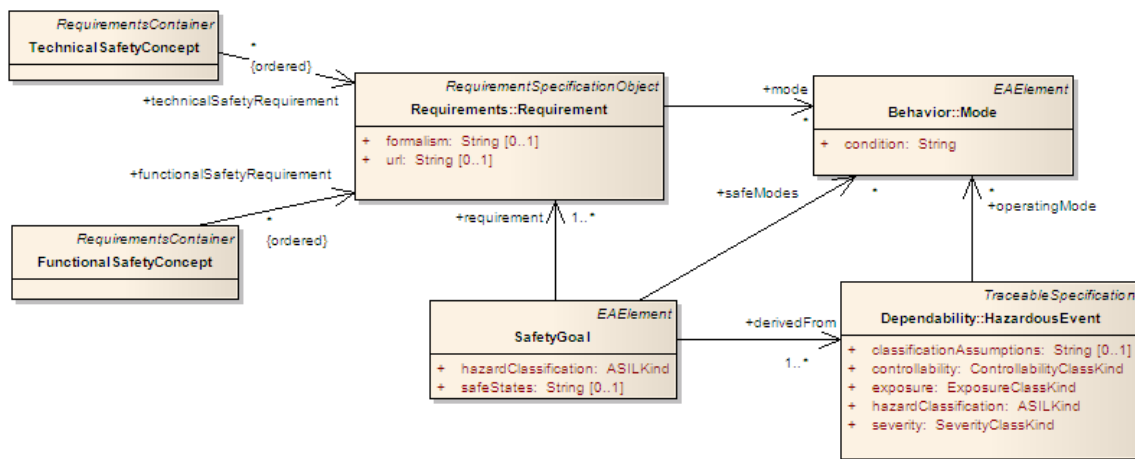
Figure 32. SafetyConcept. Diagram for Safety Concepts.

21.2   Element Descriptions

21.2.1   FunctionalSafetyConcept (from SafetyRequirement)

Generalizations
- RequirementsContainer (from Requirements)

Description

FunctionalSafetyConcept represents the set of functional safety requirements that together fulfils a SafetyGoal in accordance with ISO 26262.

To comply with the SafetyGoals, the FunctionalSafetyConcept specifies the basic safety mechanisms and safety measures in the form of functional safety requirements.

Attributes

No additional attributes

Associations
- functionalSafetyRequirement : Requirement [*] {ordered} (from Requirements)
  Represents a functional safety requirement that describes the measures for complying with the safety goals and the corresponding ASIL.

Constraints

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

Semantics

-

21.2.2   SafetyGoal (from SafetyRequirement)

Generalizations
- EAElement (from Elements)

Description

SafetyGoal represents the top-level safety requirement defined in ISO26262. Its purpose is to define how to avoid its associated HazardousEvents, or reduce the risk associated with the hazardous event to an acceptable level.

The SafetyGoal is defined through one or several associated requirement elements.

An ASIL shall be assigned to each SafetyGoal, to represent the integrity level at which the SafetyGoal must be met.

Similar SafetyGoals can be combined into one SafetyGoal. If different ASILs are assigned to similar SafetyGoals, the highest ASIL shall be assigned to the combined SafetyGoal.

For every SafetyGoal, a safe state should be defined, either textually or by referencing a specific mode. The safe state is a system state to be maintained or to be reached when a potential source of its hazardous event is detected.

Attributes
- hazardClassification : ASILKind [1] (from SafetyConstraints)
- safeStates : String [0..1]
  For every SafetyGoal, a safe state should be defined, in order to declare a system state to be maintained or to be reached when the failure is detected and so to allow a failure mitigation action without any violation of the associated SafetyGoal.

Associations
- safeModes : Mode [*] (from Behavior)
  The safe modes identified for the SafetyGoal

- requirement : Requirement [1..*] (from Requirements)
- derivedFrom : HazardousEvent [1..*] (from Dependability)
  The HazardousEvent which the SafetyGoal shall address

Constraints

No additional constraints

Semantics

SafetyGoal represents a safety Goal according to ISO26262. Requirements define the SafetyGoal, and HazardousEvents identify the responsibility of each SafetyGoal. hazardClassification defines the integrity classification of the SafetyGoal, and safeStates may be defined by a string or formalized through associated Modes.

21.2.3   TechnicalSafetyConcept (from SafetyRequirement)

Generalizations
- RequirementsContainer (from Requirements)

Description

TechnicalSafetyConcept represents the set of technical safety requirements that together fulfils a FunctionalSafetyConcept and SafetyGoal in accordance with ISO 26262.

These are derived from FunctionalSafetyConcepts i.e. TechnicalSafetyRequirements are derived from FunctionalSafetyRequirements.

Attributes

No additional attributes

Associations
- technicalSafetyRequirement : Requirement [*] {ordered} (from Requirements)

Constraints
No additional constraints

Semantics
The TechnicalSafetyConcept consists of the technical safety requirements and details the functional safety concept considering the functional concept and the preliminary architectural design. It corresponds to the Technical Safety Concept of ISO26262.

-

22 SafetyCase

22.1 Overview

Safety is a property of a system that is difficult to verify quantitatively since no clear measurement method exists that can be applied during the development. Not even exhaustive testing is feasible, as faults in electronics can have an intensity of 10^-9 faults/hour and still pose an unacceptable risk. Hence, it is only when sufficient field data have been collected from a system used in a particular context that it can be said to be safe enough. Nonetheless, safety must be addressed and assessed during development; restricted to qualitative reasoning about the safety of a product. A structured engineering method is thus needed to approach this problem. One such method is the so called safety case, which came originally from the nuclear industry.
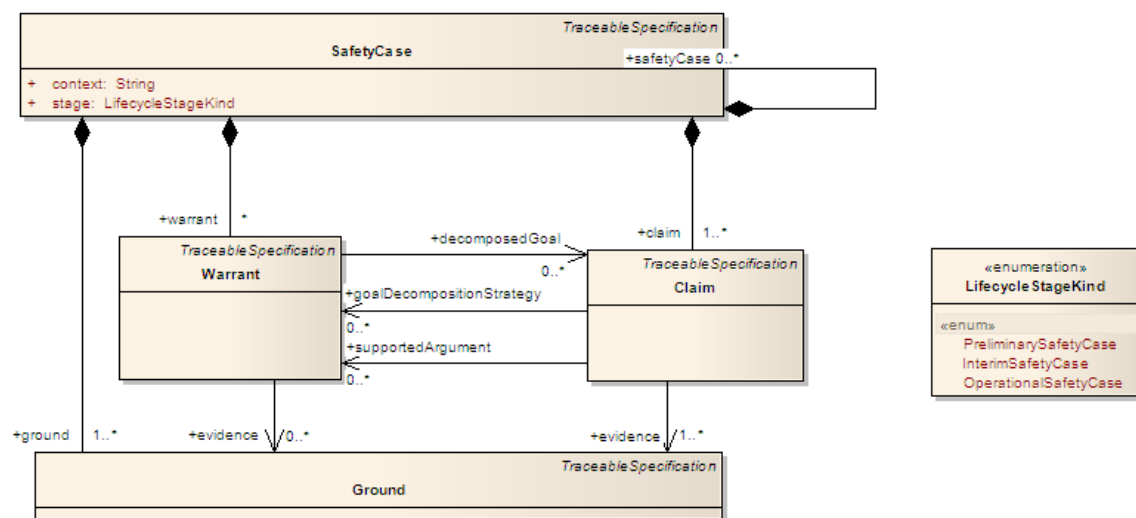
Figure 33. SafetyCase. Diagram for SafetyCase.

Figure 34. ClaimWarrantGround. Diagram for Ground, Warrant, and Claim.

22.2   Element Descriptions

22.2.1   Claim (from SafetyCase)

Generalizations
- TraceableSpecification (from Elements)

Description

Claim represents a statement, the truth of which needs to be confirmed.
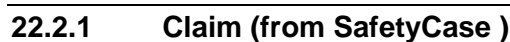
Claim has associations to the strategy for goal decomposition and to supported arguments. It also holds associations to the evidences for the SafetyCase.

Attributes

No additional attributes

Associations
- goalDecompositionStrategy : Warrant [0..*] (from SafetyCase)
  Strategies can be used to add further detail to a goal decomposition.

- supportedArgument : Warrant [0..*] (from SafetyCase)
  Supported argument for the Claim.

- evidence : Ground [1..*] (from SafetyCase)
  An evidence provides the backing for stating that a requirement (Claim) has been meet.

- justification : Rationale [*] (from Elements)
  Justification can be used wherever it is felt to be valuable to provide the rationale behind the Claim.

- safetyRequirement : TraceableSpecification [0..*] (from Elements)
  Safety requirements and objectives in the SystemModel.

Constraints
No additional constraints

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

Goal is what the argument must show to be true.

22.2.2   Ground (from SafetyCase)

Generalizations
- TraceableSpecification (from Elements)

Description
Claim is based on Grounds (evidences) - specific facts about a precise situation that clarify and make good the Claim.

Ground represents statements that explain how the SafetyCase Ground clarifies and make good the Claim.

Ground has associations to the entities that are the evidences in the SafetyCase.

Attributes
No additional attributes

Associations
- justification : Rationale [*] (from Elements)
  Justification can be used wherever it is considered valuable to provide the rationale behind the Ground.

- safetyEvidence : [0..*]
  Safety evidence in the SystemModel. May also refer to elements in the AUTOSAR model.

Constraints
No additional constraints

Semantics
Ground (evidence) is information that supports the Claim that the safety requirements and objectives are met i.e. used as the basis of the safety argument.

Solution is evidence that the sub-goals have been met. This can be achieved by decomposing all goal claims to a level where direct reference to evidences was considered possible.

The evidences address different aspects of the goal. It always has to be ensured that each of them is defensible enough to confirm the underlying statement.

22.2.3   LifecycleStageKind (from SafetyCase) «enumeration»

Generalizations
None

Description
The SafetyCase should be initiated at the earliest possible stage in the safety program so that hazards are identified and dealt with while the opportunities for their exclusion exist.

The LifecycleStageKind is an enumeration metaclass with enumeration literals indicating safety case life cycle stage.

Enumeration Literals
- InterimSafetyCase
  The interim safety case is situated after the first system design and tests

- OperationalSafetyCase
  The operational safety case is prior to in-service use

- PreliminarySafetyCase
  The preliminary safety case is started when development of the system is started.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
The safety case is one incremental safety case, rather than several complete new ones. The safety case lifecycle stage has the following meanings:

- The preliminary safety case is started when development of the system is started. After this stage discussions with the customer can commence about possible safety issues (hazards).

- The interim safety case is situated after the first system design and tests.

- The operational safety case is prior to in-service use.

22.2.4   SafetyCase (from SafetyCase)

Generalizations
- TraceableSpecification (from Elements)

Description
SafetyCase represents a safety case that communicates a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a given context.

Safety Cases are used in safety related systems, where failures can lead to catastrophic or at least dangerous consequences.

Attributes
- context : String [1]
  Description of how the SafetyCase Warrant (argument) relates to, and depends upon, information from other viewpoints.

- stage : LifecycleStageKind [1] (from SafetyCase)
  Safety case life cycle stage (preliminary, interim or operational)

Associations
- safetyCase : SafetyCase [0..*] (from SafetyCase)
  Sub SafetyCase

- warrant : Warrant [*] (from SafetyCase)
  Argumentation of the facts to the Claim in general ways.

- ground : Ground [1..*] (from SafetyCase)
  Explains how the SafetyCase Ground clarifies and make good the Claim.

- claim : Claim [1..*] (from SafetyCase)
  A statement the truth of which needs to be confirmed.

Constraints
No additional constraints

Semantics
-

22.2.5 Warrant (from SafetyCase)

Generalizations
- TraceableSpecification (from Elements)

Description

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

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

Attributes

No additional attributes

Associations
- evidence : Ground [0..*] (from SafetyCase)
  Explains how the SafetyCase Ground clarifies and make good the Claim.

- decomposedGoal : Claim [0..*] (from SafetyCase)
  A statement the truth of which needs to be confirmed

- justification : Rationale [*] (from Elements)
  Justification can be used wherever it is felt to be valuable to provide the rationale behind the Warrant.

Constraints

No additional constraints

Semantics

The overall objective of an argument is to lead the evidence to the claim.

Arguments are actions of inferring a conclusion from premised propositions. An argument is considered valid if the conclusion can be logically derived from its premises. An argument is considered sound if it is valid and all premises are true.

A goal decomposition strategy breaks down a goal into a number of sub-goals. It is recommended that the strategies are of specific form.

Part VIII Generic Constraints

This part contains support for GenericConstraints, i.e. those that do not belong to the predefined timing and safety constraints.

23	GenericConstraints

23.1 Overview

The main concept in this package is GenericConstraint which denotes a property, requirement, or a validation result for the identified element of the model. The kind of GenericConstraint is described as one of the predefined GenericConstraintKind literals.

Figure 35. GenericConstraintsOrganization. Diagram of GenericConstraint.

23.2 Element Descriptions

23.2.1 GenericConstraint (from GenericConstraints)

Generalizations
- TraceableSpecification (from Elements)

Description
The GenericConstraint denotes a property, requirement, or a validation result for the identified element of the model. The kind of GenericConstraint is described as one of the GenericConstraintKind literals.

Example: If the attribute genericConstraintType is cableLength, the genericConstraintValue could be "5 meters" (value of a numerical datatype with unit "meters").

Attributes
- genericConstraintType : GenericConstraintKind [1] (from GenericConstraints)
  The type of the GenericConstraint, see GenericConstraintKind.

- genericConstraintValue : String [1]
  The genericConstraintValue is the concrete value of the GenericConstraint according to the semantics of the genericConstraintType.

Associations
- mode : Mode [*] (from Behavior)
  The mode where this GenericConstraint is valid.

- target : [*]

The subject of the GenericConstraint.

Constraints
No additional constraints

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

23.2.2 GenericConstraintKind (from GenericConstraints) «enumeration»

Generalizations
None

Description
Enumeration for different type of constraints.

Enumeration Literals
- cableLength
  The length of the cable.

- developmentCost
  The overall development cost.

- functionAllocationDifferentNodes
  The targets (the DesignFunctions) shall be allocated on different Nodes.

  This constraint needs to be implemented by appropriate FunctionAllocations in the DesignLevel.

- functionAllocationSameNode
  The targets (the DesignFunctions) shall be allocated on the same Node.

  This constraint needs to be implemented by appropriate FunctionAllocations in the DesignLevel.

- other
- pieceCost
  The costs per piece.

- powerConsumption
  The power consumption of the unit.

- powerSupplyIndependent
  The targets (the DesignFunctions) shall be allocated to Nodes with independent power supplies.

  This constraint needs to be implemented by appropriate FunctionAllocations in the DesignLevel.

- spaceRedundancy
  The targets are replicated for redundancy, genericConstraintValue times.

- standard
  The standard (e.g., ISO26262) that is the basis for development of the target.

- timeRedundancy
  The targets are executed with time redundancy, genericConstraintValue times.

- weight
  The physical weight of the unit.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
The semantics is defined on each literal.

23.2.3   GenericConstraintSet (from GenericConstraints)

Generalizations
- Context (from Elements)

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

Attributes
No additional attributes

Associations
- genericConstraint : GenericConstraint [*] (from GenericConstraints)

Constraints
No additional constraints

Semantics
GenericConstraintSet is container element for GenericConstraints and has no specific semantics.

23.2.4   TakeRateConstraint (from GenericConstraints)

Generalizations
- GenericConstraint (from GenericConstraints)

Description
The TakeRateConstraint defines the ratio between the number of configurations that includes the target elements and the number of configurations that include the source. If several source elements are referenced, it would be the configurations in which all these exist.

TakeRateConstraint complements configuration decisions, as the latter defines the rules for actual configuration. TakeRateConstraint defines expected rates of configurations and the set of constraints should be consistent with the configuration decisions. Also, the set of TakeRateConstraints shall be consistent among themselves.

Attributes
- takeRate : Float [1]
  The rate of target compared with source configurations.

Associations
- source : [*]
  The elements that are compared with the elements identified by target (see GenericConstraint).

Constraints
[1] The cardinality of target is >0

Semantics

-

Part IX Infrastructure

This part contains the EAST-ADL infrastructure which is Datatypes, Elements and User attributes.

24	Datatypes

24.1　　Overview

The Datatypes subpackage of EAST-ADL defines EAST-ADL general-purpose datatypes that may be used to type structural constructs in several different modeling diagrams.

The purpose of the metaclasses in the Datatypes subpackage is to specify the concepts for the specific domain.

Figure 36. Datatypes. Diagram for Datatypes.

24.2　　Element Descriptions

24.2.1　　CompositeDatatype (from Datatypes)

Generalizations
- EADatatype (from Datatypes)

Description
A CompositeDatatype represents a non-scalar datatype. Take as an example a CompositeDatatype "MyCountries" that can refer, e.g., to an Enumeration "CountryEnumeration" {USA, Canada, Japan, EU} via two EADatatypePrototypes (record variables): FirstCountry and SecondCountry. Then an attribute typed by this CompositeDatatype "MyCountries" may have a value like: (EU (identified as FirstCountry), Japan (identified as SecondCountry)).

Attributes
No additional attributes

Associations
- datatypePrototype : EADatatypePrototype [1..*] {ordered} (from Datatypes)
  The record variable owned by the CompositeDatatype.

Constraints
No additional constraints

Semantics
A CompositeDatatype represents a non-scalar datatype. The contained datatypePrototypes act as record variables to identify the ordered datatype instances of the tuple (the CompositeDatatype).

24.2.2   EABoolean (from Datatypes)

Generalizations
- EADatatype (from Datatypes)

Description
A Boolean value denotes a logical condition that is either 'true' or 'false'.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
Boolean is the primitive type that holds two literals: true, false.

24.2.3   EADatatype (from Datatypes) {abstract} «atpType»

Generalizations
- TraceableSpecification (from Elements)

Description
The EADatatype is a metaclass, which signifies a type whose instances are identified only by their value. The EADatatype metaclass represents the description of the value set for some variable, parameter etc. without a description of how these possible values are represented at implementation level. The implementation representation is defined at implementation level by the AUTOSAR concept PrimitiveTypeWithSemantics, and the implemented datatype shall be associated with a Realization relationship. The realizing datatype must match the EADatatype regarding range, resolution, unit, and dimension.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
EADatatype metaclass is a special kind of classifier, similar to a class. It differs from the class in that instances of a data type are identified only by their value.

24.2.4   EADatatypePrototype (from Datatypes) «atpPrototype»

Generalizations
- EAElement (from Elements)

Description
The EADatatypePrototype represents a typed variable. An example is a composite datatype ColorValue with parts R, G, and B of type integer. ColorValue would contain three prototypes only to be able to reference the record parts by name.

Attributes
No additional attributes

Associations
- type : EADatatype [1] (from Datatypes)

«isOfType»

> The type of the EADatatypePrototype.

Constraints
No additional constraints

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

24.2.5   EAFloat (from Datatypes)

Generalizations
- RangeableDatatype (from Datatypes)

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.

Attributes
- max : Float [1]
  The maximal value of the range.

- min : Float [1]
  The minimum value of the range.

Associations
No additional Associations

Constraints
No additional constraints

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

24.2.6   EAInteger (from Datatypes)

Generalizations
- RangeableDatatype (from Datatypes)

Description
An instance of Integer is an element in the set of integer numbers (..., -2, -1, 0, 1, 2, ...).

Attributes
- max : int [1]
  The maximal value of the range.

- min : int [1]
  The minimum value of the range.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
An instance of Integer is an element in the set of integer numbers (..., -2, -1, 0, 1, 2, ...).

24.2.7   EAString (from Datatypes)

Generalizations
- EADatatype (from Datatypes)

Description
A string is a sequence of characters in some suitable character set used to display information about the model. Character sets may include non-Roman alphabets and characters. An instance of String defines a piece of text. The semantics of the string itself depends on its purpose. It can be a comment, computational language expression, OCL expression, etc. It is used for String attributes and String expressions in the metamodel.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
String is the primitive type that defines a sequence of characters in some suitable character set used to display information.

24.2.8   Enumeration (from Datatypes)

Generalizations
- EADatatype (from Datatypes)

Description
An enumeration is a datatype whose values are enumerated in the model as enumeration literals. Enumeration is a kind of datatype, whose instances may be any of a number of user-defined enumeration literals.

Attributes
No additional attributes

Associations
- literal : EnumerationLiteral [2..*] {ordered} (from Datatypes)
  The literal (value) of the enumeration.

Constraints
No additional constraints

Semantics
Enumeration is a kind of datatype, whose instances may be any number > 1 of user-defined enumeration literals. Enumerations contain at least two literals, otherwise it would be a constant). The contained literals need to be ordered.

24.2.9   EnumerationLiteral (from Datatypes)

Generalizations
- EAElement (from Elements)

Description
An enumeration literal is a user-defined data value for an enumeration.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
An EnumerationLiteral defines an element of the run-time extension of an enumeration data type. An EnumerationLiteral has a name (inherited from EAElement) that can be used to identify it within its Enumeration datatype. The EnumerationLiteral name is scoped and must therefore be unique within its Enumeration. EnumerationLiteral names are not global and must be qualified for general use. The run-time values corresponding to EnumerationLiterals can be compared for equality.

24.2.10 EnumerationValueType (from Datatypes)

Generalizations
- ValueType (from Datatypes)

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.

Attributes
- isMultiValued : Boolean [1] (from Datatypes)
  This boolean attribute is true, if multiple enumeration values can be selected. It is false, if only one enumeration value is allowed to be selected.

- literalSemantics : String [2..*]
  The specific semantics for each literal of the baseEnumeration.

Associations
- baseEnumeration : Enumeration [1] (from Datatypes)
  The enumeration that the EnumerationValueType points to.

Constraints
No additional constraints

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.

24.2.11 RangeableDatatype (from Datatypes) {abstract}

Generalizations
- EADatatype (from Datatypes)

Description
The abstract metaclass 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.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

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

24.2.12   RangeableValueType (from Datatypes)

Generalizations
- ValueType (from Datatypes)

Description
The RangeableValueType is a specific ValueType applicable for RangeableDatatypes. It describes the accuracy, resolution, and the significant digits of the baseRangeable datatypes.

Attributes
- accuracy : Float [1] (from Datatypes)
  The accuracy of the data (e.g., the FunctionFlowports input or output).

  Example: An accuracy of 0.5 of the temperature means a communicated value of 19 represents an actual temperature of 19 +/- 0.5 degrees.

- resolution : Float [1] (from Datatypes)
  The resolution of the data expressed as the size of the minimum difference between data values.

  Example: A resolution of 0.1 means that temperature may be represented in increments of 0.1 degrees.

- significantDigits : int [0..1]
  The number of significant digits, e.g., for the speed case: if the speed is a one digit number (e.g., 5 km/h), then this digit is significant, if the speed is a two digits number (e.g., 15 km/h), then the first digit is significant (here: 1), if the speed is a three digits number (e.g., 215 km/h), then the first two digits are significant (here: 21). Significant means here, that the respective digits are reliable.

Associations
- baseRangeable : RangeableDatatype [1] (from Datatypes)
  The RangeableDatatype that the RangeableValueType points to.

Constraints
No additional constraints

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

24.2.13   ValueType (from Datatypes) {abstract}

Generalizations
- EADatatype (from Datatypes)

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 measurement 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.

Attributes

- description : String [0..1]
  Description of the datatype ValueType.

- dimension : String [0..1]
  The (physical) quantity, e.g., "Speed", "Temperature".

- unit : String [0..1]
  The unit of data.

  Example: For temperature the unit may be "degree Celsius".

Associations
No additional Associations

Constraints
No additional constraints

Semantics
The abstract metaclass 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.

-

25   Elements

25.1   Overview

The Element subpackage of the Infrastructure package of the EAST-ADL specifies the most basic abstract structural constructs in EAST-ADL.
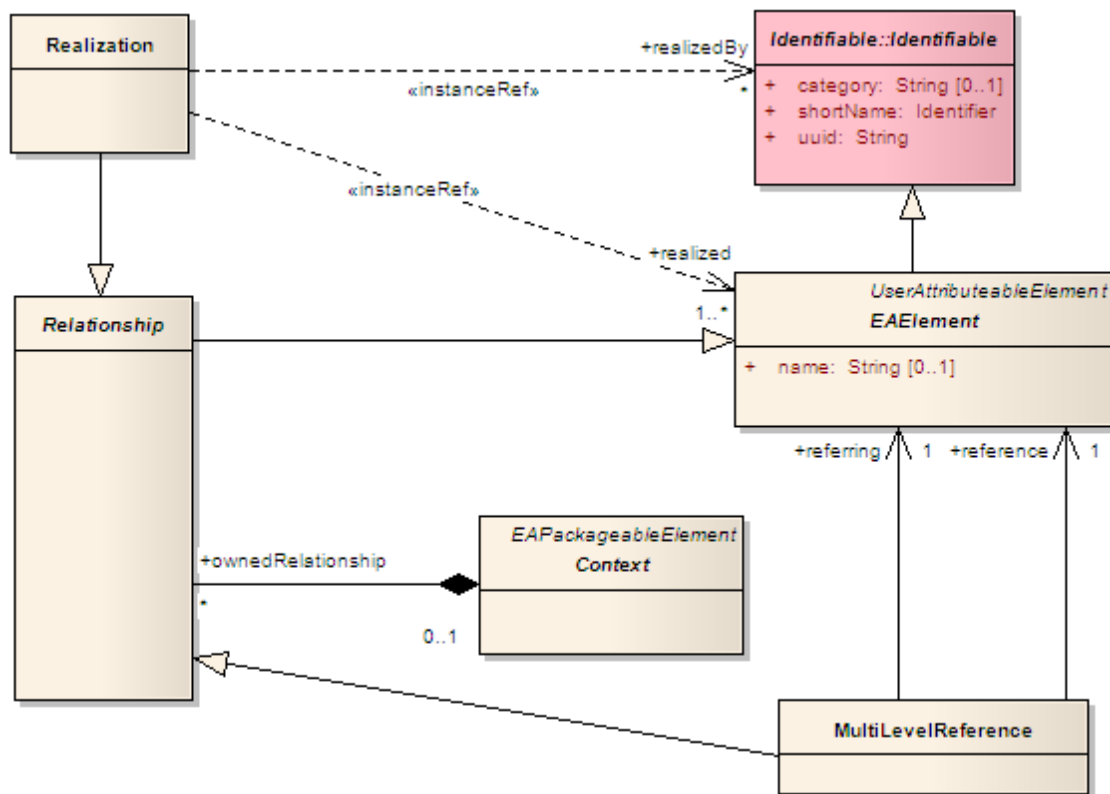
Figure 37. RelationshipModeling. Diagram for RelationshipModeling.
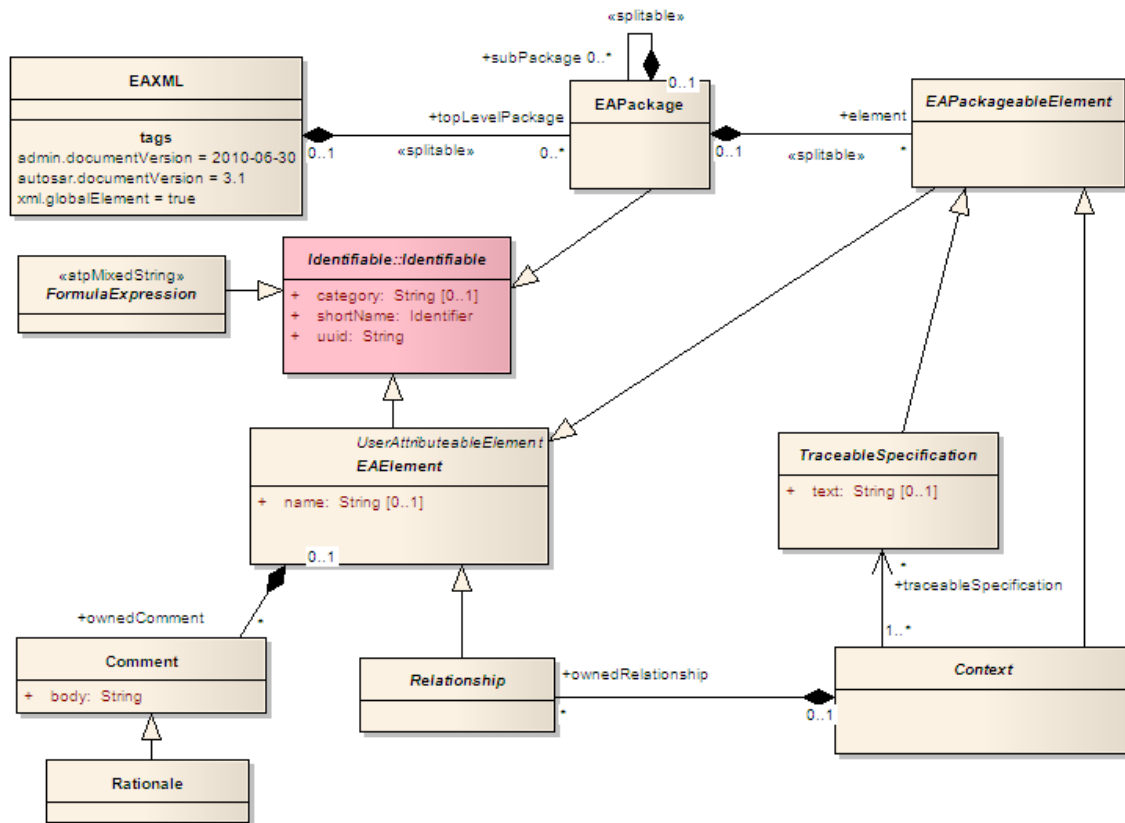
Figure 38. Elements. Diagram for Elements.

25.2 Element Descriptions

25.2.1 Comment (from Elements)

Generalizations
None

Description
Comment represents a textual annotation.

Attributes
- body : String [1]
  Specifies a string that is the comment.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

25.2.2 Context (from Elements) {abstract}

Generalizations
- EAPackageableElement (from Elements)

Description
Context represents a simple and practical way to allocate TraceableSpecifications to a specific EAST-ADL model context, and to let this specific model context own Relationships.

Attributes
No additional attributes

Associations
- ownedRelationship : Relationship [*] (from Elements)
  Relationship(s) owned by this context.

- traceableSpecification : TraceableSpecification [*] (from Elements)
  Traceable specification(s) identified by this context.

Constraints
No additional constraints

Semantics
See Relationship and TraceableSpecification.

25.2.3 EAElement (from Elements) {abstract}

Generalizations
- UserAttributeableElement (from UserAttributes)

Description
The EAElement is an abstract metaclass that represents an arbitrary named entity in the domain model. It specializes AUTOSAR Identifiable which has the shortName attribute used for identification of the element within the namespace in which it is defined.

The abbreviation EA in the name of this metaclass is short for EAST-ADL.

Attributes
- name : String [0..1]
  Optional descriptive name of the EAElement, this name does not have the length restrictions as found for the AUTOSAR Identfiable shortName.

Associations
- ownedComment : Comment [*] (from Elements)
  Comment owned by this EAElement.

Constraints
No additional constraints

Semantics
Also the EAElement can be used to extend the EAST-ADL approach to other languages and standards by adding a generalize relation from the respective (non EAST-ADL) element to the EAElement.

25.2.4 EAPackage (from Elements)

Generalizations
Description
Used for organization of the packageable elements in the model.

Attributes
No additional attributes

Associations
- element : EAPackageableElement [*] (from Elements)
«splitable»

    Contained packageable elements.

- subPackage : EAPackage [0..*] (from Elements)
«splitable»

    Contained packages.

Constraints
No additional constraints

Semantics
EAPackages can be organized hierarchically, where each level may contain a number of EAPackageableElements.

25.2.5   EAPackageableElement (from Elements) {abstract}

Generalizations
- EAElement (from Elements)

Description
Elements that are packageable may be directly contained in a package.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
Elements specializing EAPackageableElement can be created directly within an EAPackage.

25.2.6   EAXML (from Elements)

Generalizations
None

Description
The root element of an exchanged XML file which contains an EAST-ADL model.

Attributes
No additional attributes

Associations
- topLevelPackage : EAPackage [0..*] (from Elements)
«splitable»

    Contained top level packages.

Constraints
No additional constraints

Semantics

-

25.2.7 FormulaExpression (from Elements) {abstract} «atpMixedString»

Generalizations
Description
Subclasses have the capability of using the elements identified through associations in formulas. This may be realized by qualified names in a formula or in an XML implementation as an ordered mix between elements and text.

Compare implementation in AUTOSAR.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics

-

25.2.8 MultiLevelReference (from Elements)

Generalizations
- Relationship (from Elements)

Description
The metaclass MultiLevelReference gives the possibility to establish reference links between model elements. Such a reference may be established between two elements when both of them are slightly different but one element is newer and originates from the other element.

With such reference, it is possible to keep track of changes (by humans and also computational) in compare to origin elements. Moreover, it is possible to take over the changes into the original. In EAST-ADL, the Multi-Level concept will be used for the Feature Modeling and for the Requirements Interchange. More detailed informations about the Multi-Level concept in general and also about the use of this concept in the context of Feature Trees and for Requirements Exchange can be found in the ATESST2 papers "Multi Level Feature Trees" and "Manufacturer-Supplier Requirements Synchronization Using Exchange Containers and Multi-Level Systems

Attributes
No additional attributes

Associations
- referring : EAElement [1] (from Elements)
  The target elements of a MultiLevelReference link.

Association Documentation:

- reference : EAElement [1] (from Elements)
  The source element of a MultiLevelReference link.

Association Documentation:

Constraints
No additional constraints

Semantics

-

25.2.9 Rationale (from Elements)

Generalizations
- Comment (from Elements)

Description
Rationale represents a justification to any model element.

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics

-

25.2.10 Realization (from Elements)

Generalizations
- Relationship (from Elements)

Description
The Realization is a relationship which relates two or more elements across boundaries of the EAST-ADL abstraction levels.

It identifies an element that serves as a specification within this realization relationship and on the other side it identifies an element that is supposed to realize this specification on a lower abstraction level or an implementation.

Attributes
No additional attributes

Associations
- realized : Realization_realized [1..*] (from _instanceRef)
- realizedBy : Realization_realizedBy [1..*] (from _instanceRef)

Dependencies
- realized: EAElement [1..*] (from Elements)

«instanceRef»

- realizedBy: [*]

«instanceRef»

Constraints
No additional constraints

Semantics
Modification of the realized element impacts the realizing element.

25.2.11 Relationship (from Elements) {abstract}

Generalizations
- EAElement (from Elements)

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
In many cases, Contexts such as functions and sensors need to have requirements and other specification elements allocated to them. In other cases, the relationship between an element and the related specification element is specific for a certain Context: for example a Requirement on a sensor is only applicable in certain hardware architectures. These relationships are modeled by concrete specializations of Relationship.

See Context and TraceableSpecification.

25.2.12 TraceableSpecification (from Elements) {abstract}

Generalizations
- EAPackageableElement (from Elements)

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

Attributes
- text : String [0..1]
  An optional description attribute that provides textual representation, or a reference to the textual representation, of the Traceable Specification in a specific formalism.

Associations
No additional Associations

Constraints
No additional constraints

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

See Context and Relationship.

- -

26 UserAttributes

26.1 Overview

User attributes in EAST-ADL are primarily intended to provide a mechanism for augmenting the elements of an EAST-ADL model with customized meta-information. All instances of metaclass UserAttributeableElement 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 UserAttributeTemplates.

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 for defining those attributes of requirements. In that sense, basic Requirements in EAST-ADL can be seen as "empty" elements which only provide a node to which user attributes can be attached in order to supply the Requirement with all necessary information, including its main textual description. However, in the case when the Requirement is the context in which the available user attributes are defined, the container (context) of the Requirements is the point where user attribute definitions are stored and these are only applicable 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. User attributes are realized by simple key/value pairs where the globally unique key identifies the user attribute (cf. class UserAttributeValue). In principle, any key/value combination may be attached to any element, but user attribute definitions may optionally be provided to define valid keys and a set of legal values for them (cf. class UserAttributeDefinition). However, the actual attributes attached to an element and/or their values may well conflict with these attribute definitions: for example, it is perfectly legal to not provide an attribute value if an attribute definition was specified, or 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 tool support). With this concept 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 makes sure that a model, and all its user attribute values, can be safely viewed and edited even if the attribute definitions for the model are temporarily unavailable or permanently lost.

Whenever interoperability with third parties is required an internet domain naming scheme should be used, similar to 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.

User attributes in EAST-ADL serve a similar purpose to stereotypes in UML2 but are intended as a much simpler mechanism, especially with respect to tool implementation.
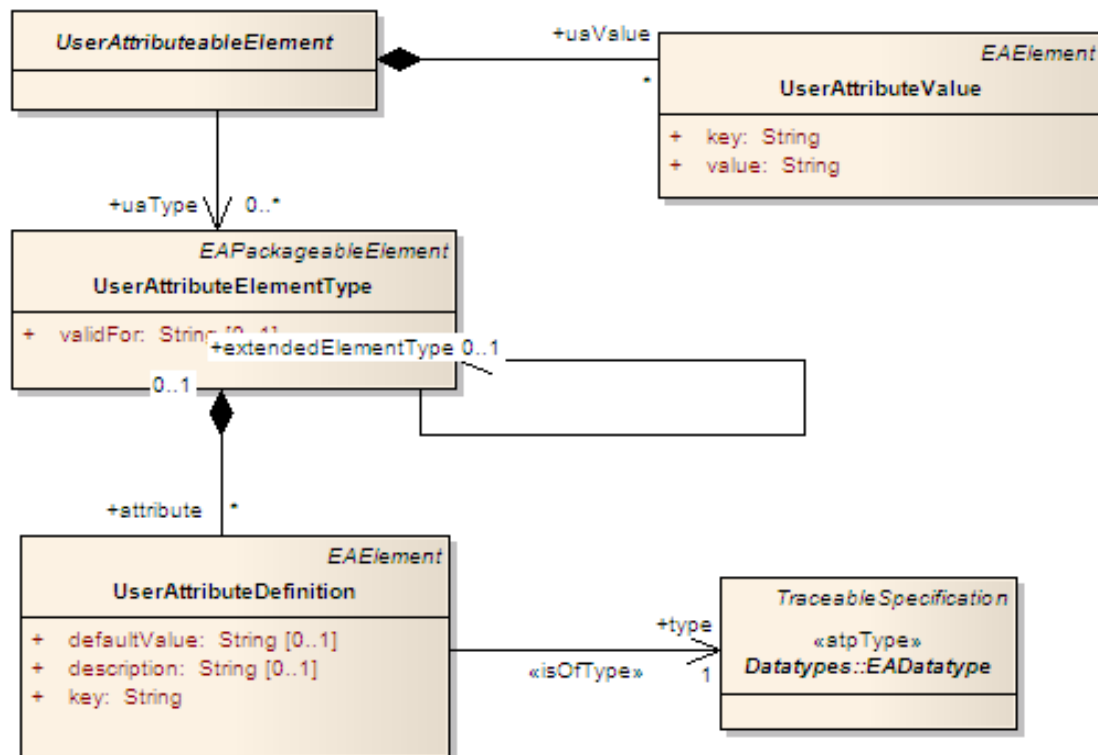
Figure 39. UserAttributes. Diagram for User Attributes.

26.2 Element Descriptions

26.2.1 UserAttributeDefinition (from UserAttributes)

Generalizations
- EAElement (from Elements)

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".

Attributes
- defaultValue : String [0..1]
  The default value. This is to be used whenever a user attributeable element has no UserAttributeValue for the key of this UserAttributeDefinition.

- description : String [0..1]
  A description statement.

- key : String [1]
  A unique identifier for the user attribute. Please refer to the description of attribute 'key' in metaclass UserAttributeValue for a detailed discussion on how to construct valid, globally unique keys.

Associations
- type : EADatatype [1] (from Datatypes)

The type of the user attribute. This type defines the set of legal values for the given user attribute, i.e. for all UserAttributeValues with the same key as this UserAttributeDefinition.

Constraints
No additional constraints

Semantics
UserAttributeDefinition represents a user attribute whose type is defined by the UserAttributeElementType that is identified by 'key'.

26.2.2 UserAttributeElementType (from UserAttributes)

Generalizations
- EAPackageableElement (from Elements)

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.

Attributes
- validFor : String [0..1]
  Comma-separated list of metaclass names this user attribute element type is applicable to. If undefined, then this user attribute element type is applicable to all UserAttributeableElements.

  Example: If UserAttributeElementType 'VWFunction' has its validFor attribute set to "FunctionalDevice,LocalDeviceManager", then element type 'VWFunction' is only applicable to functional devices and local device managers, i.e. only instances of FunctionalDevice and LocalDeviceManager may have their association uaType point to 'VWFunction'.

Associations
- extendedElementType : UserAttributeElementType [0..1] (from UserAttributes)
  The UserAttributeElementType this type is inheriting from.

  When UserAttributeElementType 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.

- attribute : UserAttributeDefinition [*] (from UserAttributes)
  The attributes defined for this type. Note that inherited attribute definitions also need to be taken into account (cf. association 'extendedElementTypes').

Constraints
No additional constraints

Semantics
UserAttributeElementType defines the type of UserAttributeDefinition that share the same key.

26.2.3 UserAttributeValue (from UserAttributes)

Generalizations
- EAElement (from Elements)

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.

Attributes
- key : String [1]
  The globally unique identifier of the user attribute for which this UserAttributeValue provides a value. Any string may be used as key as long as it is globally unique.

  However, there is a recommended procedure for building globally unique keys for user attributes, similar to packages in the Java programming language:

  (1) use an internet domain name which is sufficiently specific so that you have control over who will use it for user attribute key generation (e.g. "myDepartment.myCompany.com")

  (2) reverse it as in Java package names (e.g. "com.myCompany.myDepartment")

  (3) optionally append additional, dot-separated names for the specific context in which the user attribute is to be used (e.g. "myProject" which results in "com.myCompany.myDepartment.myProject")

  (4) add a last segment that names the user attribute and is sufficiently descriptive to explain its purpose (e.g. "ReviewStatus").

  In this example, the key of our status attribute would be "com.myCompany.myDepartment.myProject.ReviewStatus".

  In general, the last segment of the key, i.e. everything following the last dot, should be sufficient to identify the attribute in its usual, most specific context of use. Therefore, implementations may use this last segment as an abbreviated name of the user attribute, e.g. for presenting it in a GUI.

- value : String [1]
  Holds the actual value of the user attribute identified by 'key'. This value is always represented as a string. Non-string values, such as integers, are specified by their corresponding string representation.

  In cases where a UserAttributeDefinition is in effect and declares a particular Datatype for a user attribute (cf. association 'type' in UserAttributeDefinition), the 'value' attribute will hold the string representation of any valid value of this data type. The precise format depends on the data type: for numeric types the radix must be 10 and "." is to be used as decimal point ; date format is milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT ; multi-valued types are realized as a comma-separated list of individual values.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
UserAttributeValue is an annotation of the containing UserAttributableElement. It has a value and the value type is the UserAttributeType with the matching key.

26.2.4   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.

Attributes
No additional attributes

Associations
- uaValue : UserAttributeValue [*] (from UserAttributes)
  The user attribute values, i.e. key/value pairs, which are attached to this element.

- uaType : UserAttributeElementType [0..*] (from UserAttributes)
  The UAElementTypesUserAttributeElementTypes of this user attributeable element.

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

Constraints
No additional constraints

Semantics
UserAttributeableElement can be annotated with UserAttributes.

Part X Annexes

This part contains the EAST-ADL Annexes. The first annex is about notation followed by element packages that are preliminary and subject to further refinement before inclusion in the main language.

27 Annex A: Notation

This annex lists the elements with defined notations to be used when the element is shown in a diagram. For those elements that are not listed here the general notation is a solid-outline rectangle with the metaclass name at the top right. The rectangle contains the user defined name of the element.

AnalysisLevel
The Analysis Architecture is shown as a solid-outline rectangle containing the name, with its ports or port groups on the perimeter. Contained entities may be shown with their connectors (White-box view).

CompositeDatatype
The datatype CompositeDatatype is denoted using the rectangle symbol with keyword «Datatype CompositeDatatype».

DeriveRequirement
A DeriveRequirement relationship is shown as a dashed arrow between two Requirements. The Requirement at the tail of the arrow (the derived Requirement) depends on the Requirement at the arrowhead (the Requirement derived from).

DesignLevel
The DesignLevel is shown as a solid-outline rectangle containing the name, with its ports or port groups on the perimeter. Contained entities may be shown with their connectors (White-box view).

EABoolean
The datatype Boolean is denoted using the rectangle symbol with keyword «Datatype Boolean».

EAFloat
The datatype Float is denoted using the rectangle symbol with keyword «Datatype Float».

EAInteger
The datatype Integer is denoted using the rectangle symbol with keyword «Datatype Integer».

EAString
The datatype String is denoted using the rectangle symbol with keyword «Datatype String».

Enumeration
The datatype Enumeration is denoted using the rectangle symbol with keyword «Datatype Enumeration».

EnumerationLiteral
An EnumerationLiteral is typically shown as a name, one per line, in the compartment of the Enumeration notation.

EnumerationValueType
The datatype EnumerationValueType is denoted using the rectangle symbol with keyword «Datatype EnumerationValueType».

FunctionAllocation
A FunctionAllocation is shown as a dependency (dashed line) with an "allocation" keyword attached to it.

FunctionBehavior
FunctionBehavior appears as a solid-outline rectangle with "Behavior" at the top right. The rectangle contains the name.

FunctionConnector
 FunctionConnector is shown as a solid line

FunctionPrototype
Shall be shown in the same style as the class specified as type, however it shall be clear that this is a part.

FunctionType
The FunctionType is shown as a solid-outline rectangle containing the name, with its FunctionPorts or PortGroups on the perimeter. Contained entities may be shown with their FunctionConnectors (White-box view).

HardwareComponentPrototype
Shall be shown in the same style as the class specified as type, however it shall be clear that this is a part.

Hazard
The Hazard is shown as a solid-outline rectangle with "Haz" at the top right. It contains the name of the Hazard and optionally the name of the source entity.

HazardousEvent
The HazardousEvent is shown as a solid-outline rectangle with "Haz" at the top right. It contains the name of the HazardousEvent and optionally the name of the source entity.

PortGroup
FunctionConnectors connected to FunctionPorts of a PortGroup are graphically collapsed into a single line.

The PortGroup is rendered as its contained ports, but with a double outline.

PrecedenceConstraint
PrecedenceConstraint is shown as a dashed arrow with "Precedes" next to it. It points from preceeding to the successive entity.

RangeableValueType
The datatype RangeableValueType is denoted using the rectangle symbol with keyword «Datatype RangeableValueType».

Realization
A Realization relationship is shown as a dashed line with a triangular arrowhead at the end that corresponds to the realized entity. The entity at the tail of the arrow (the realizing EAElement or the realizing ARElement) depends on the entity at the arrowhead (the realized EAElement).

Refine
A Refine relationship is shown as a dashed arrow between the Requirements and EAElement. The entity at the tail of the arrow (the refining EAElement) depends on the Requirement at the arrowhead (the refined Requirement).

Requirement
Requirement is shown as a solid rectangle with Req top right and its name.

RequirementsContainer
RequirementContainer is shown as a solid-outline rectangle containing the name. Contained entities may also be shown inside (White-box view)

SafetyGoal
SafetyGoal is a box with text SafetyGoal at the top left.

Satisfy
A Satisfy relationship is shown as a dashed line with an arrowhead at the end that corresponds to the satisfied Requirement or UseCaseUseCase. The entity at the tail of the arrow (the satisfying EAElement or the satisfying ARElement) depends on the entity at the arrowhead (the satisfied Requirement or UseCaseUseCase).

SystemModel
The default notation for a SystemModel is a solid-outline rectangle containing the SystemModel's name, and with compartments separating by horizontal lines containing features or other members of the SystemModel. Contained entities may also be shown with their connectors (White-box view).

Verify
A Verify relationship is shown as a dashed arrow between the Requirements and VVCase.

28 Annex B: Needs

This annex contains preliminary extensions to EAST-ADL for the modeling of stakeholder needs and related information. It is fully aligned with the language but not yet validated and ready for inclusion in the base specification.
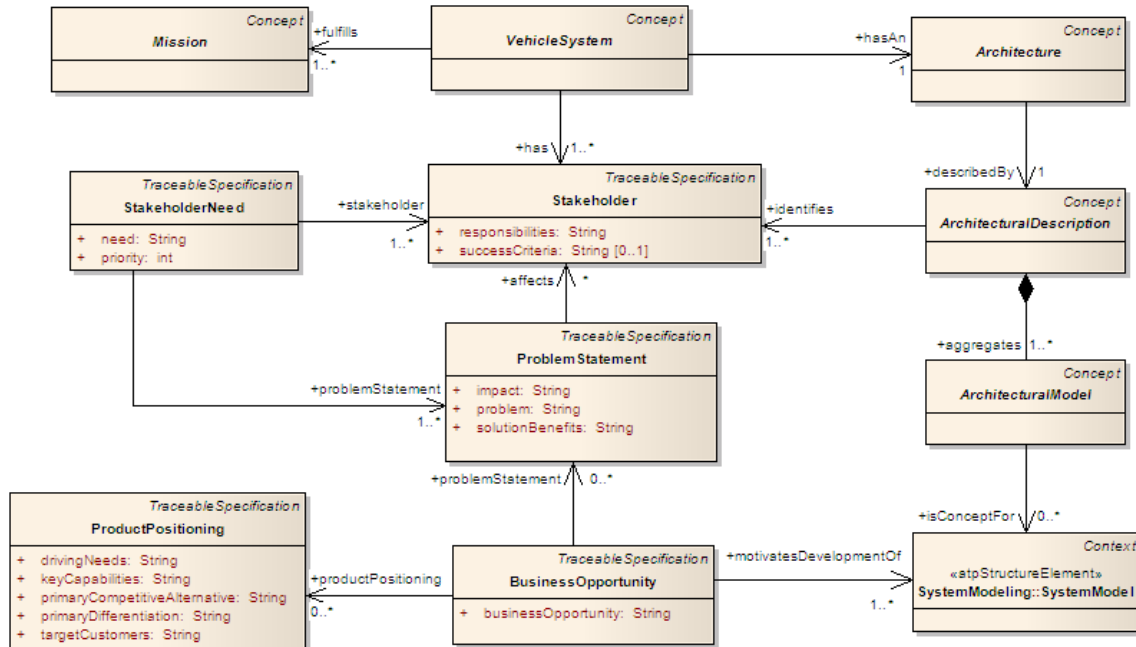
Figure 40. Needs. Diagram for Needs.

28.1 Element Descriptions

28.1.1 ArchitecturalDescription (from Needs) {abstract}

Generalizations
- Concept (from Needs)

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

Attributes
No additional attributes

Associations
- aggregates : ArchitecturalModel [1..*] (from Needs)
- identifies : Stakeholder [1..*] (from Needs)

Constraints
No additional constraints

Semantics
-

28.1.2 ArchitecturalModel (from Needs) {abstract}

Generalizations
- Concept (from Needs)

Description
A view may consist of one or more architectural models. Each such architectural model is developed using the methods established by its associated architectural viewpoint. An architectural model may participate in more than one view. [IEEE 1471]

Attributes
No additional attributes

Associations
- isConceptFor : SystemModel [0..*] (from SystemModeling)

Constraints
No additional constraints

Semantics
-

28.1.3 Architecture (from Needs) {abstract}

Generalizations
- Concept (from Needs)

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

Attributes
No additional attributes

Associations
- describedBy : ArchitecturalDescription [1] (from Needs)

Constraints
No additional constraints

Semantics
-

28.1.4 BusinessOpportunity (from Needs)

Generalizations
- TraceableSpecification (from Elements)

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

Attributes
- businessOpportunity : String [1]
  This attribute holds a brief description of the business opportunity being met by developing the electrical/electronic system. This redefines the text attribute in TraceableSpecification.

Associations
- motivatesDevelopmentOf : SystemModel [1..*] (from SystemModeling)
  The SystemModel that the BusinessOpportunity motivates development of.

- problemStatement : ProblemStatement [0..*] (from Needs)
  Optional relation to brief statements summarizing the problem being solved.

- productPositioning : ProductPositioning [0..*] (from Needs)
  The optional ProductPositioning provides an overall statement summarizing, at the highest level, the unique position the product intends to fill in the marketplace.

Constraints
No additional constraints

Semantics
-

28.1.5   Concept (from Needs) {abstract}

Generalizations
None

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

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

28.1.6   Mission (from Needs) {abstract}

Generalizations
- Concept (from Needs)

Description
A mission is a use or operation for which a system is intended by one or more stakeholders to meet some set of objectives. [IEEE 1471]

Attributes
No additional attributes

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

28.1.7   ProblemStatement (from Needs)

Generalizations
- TraceableSpecification (from Elements)

Description

The problem statement represents a brief statement summarizing the problem being solved which gives the opportunity to establish traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

The problem statement could be extended with further modeling of dependencies between different problems and deduction of root problems

Attributes
- impact : String [1]
  The impact of the problem

- problem : String [1]
  The brief problem statement. This redefines the text attribute in TraceableSpecification.

- solutionBenefits : String [1]
  Lists some key benefits of a successful solution.

Associations
- affects : Stakeholder [*] (from Needs)
  The Stakeholders affected by the problem.

Constraints
No additional constraints

Semantics
-

28.1.8   ProductPositioning (from Needs)

Generalizations
- TraceableSpecification (from Elements)

Description

The problem positioning represents an overall brief statement summarizing, at the highest level, the unique position the product intends to fill in the marketplace which gives the opportunity to establish traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

Positioning is assumed to belong to a particular context, typically a system, but also for a smaller part of a system.

Attributes
- drivingNeeds : String [1]
  Brief statement of key benefit; that is, the compelling need for the product.

- keyCapabilities : String [1]
  Brief statement of the key capabilities

- primaryCompetitiveAlternative : String [1]
  Brief statement of primary competitive alternative

- primaryDifferentiation : String [1]
  Brief statement of primary differentiation

- targetCustomers : String [1]
  Brief statement of target customers.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

28.1.9   Stakeholder (from Needs)

Generalizations
- TraceableSpecification (from Elements)

Description
The stakeholder represents various roles with regard to the creation and use of architectural descriptions. Stakeholders include clients, users, the architect, developers, and evaluators. [IEEE 1471]

Attributes
- responsibilities : String [1]
  Summarize the Stakeholder's key responsibilities with regard to the electrical/electronic system being developed; that is, their interest as a Stakeholder.

- successCriteria : String [0..1]
  Describes how the Stakeholder defines success.

Associations
No additional Associations

Constraints
No additional constraints

Semantics
-

28.1.10   StakeholderNeed (from Needs)

Generalizations
- TraceableSpecification (from Elements)

Description
Stakeholder needs represent a list of the key problems as perceived by the stakeholder, and it gives the opportunity to establish traceability from artifacts created later, for example to provide rationales to design decisions or trade-off analysis.

Attributes
- need : String [1]
  The brief need statement. Redefines text.

- priority : int [1]
  The priority of the need.

Associations
- problemStatement : ProblemStatement [1..*] (from Needs)
  The ProblemStatement that provide statements summarizing the problem being solved.

- stakeholder : Stakeholder [1..*] (from Needs)
  Role with regard to the creation and use of architectural description.

Constraints
No additional constraints

Semantics

-

28.1.11   VehicleSystem (from Needs) {abstract}

Generalizations
- Concept (from Needs)

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

Attributes
No additional attributes

Associations
- hasAn : Architecture [1] (from Needs)
- fulfills : Mission [1..*] (from Needs)
- has : Stakeholder [1..*] (from Needs)

Constraints
No additional constraints

Semantics

-

29   Annex C: BehaviorConstraints

This annex contains preliminary extensions to EAST-ADL for the modeling of behavior constraints. It is fully aligned with the language but not yet validated and ready for inclusion in the base specification.

The Behavior Constraints Annex extends EAST-ADL with additional constructs for a more fine-grained specification of key behavior attributes. It targets system requirements, vehicle features, system functions, and error models. It can be used to constrain behavior definitions in external models and formalisms (e.g., Simulink, UML, etc). The provision of such behavioral declarations in EAST-ADL is also considered necessary to support the formalization of textual requirements as well as for the reasoning of operational behaviors of vehicle features and environmental situations in early development stages.

Constructs contained in this annex are integrated into the EAST-ADL meta-model in a similar way as timing, dependability, and other non-functional constraints. This means that the behavior constraints target directly Behavior::Mode, Behavior::FunctionTrigger, Behavior::FunctionBehavior, and thereby any external behavior models linked by these EAST-ADL definitions.

Depending on the system artifacts under constraint, a behavior constraint can be applied either for FAA (Functional Analysis Architecture), or for FDA (Functional Design Architecture), or for environment specification.
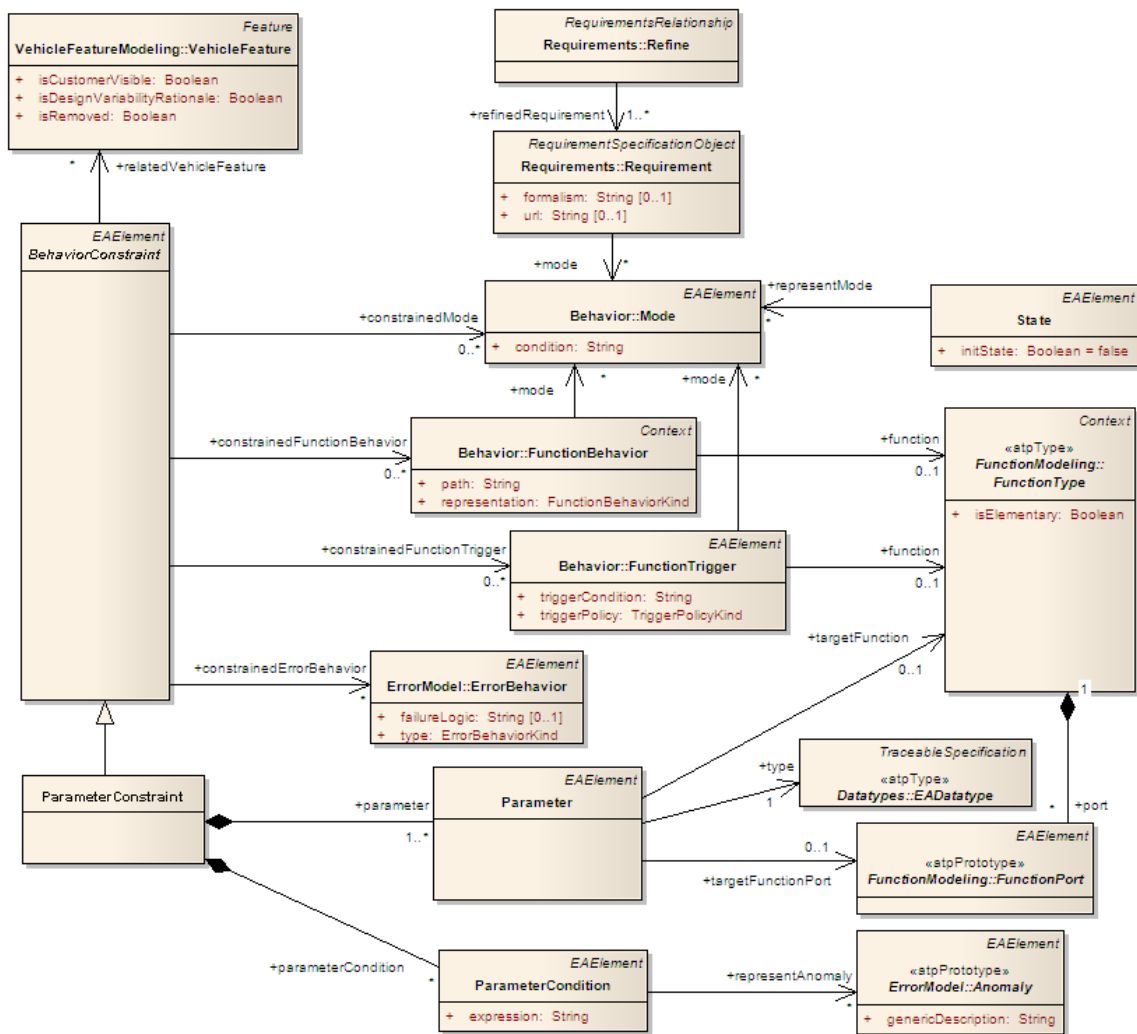
Figure 41. BehaviorContraintsMappingDependability. Diagram for dependencies of BehaviorConstraints.
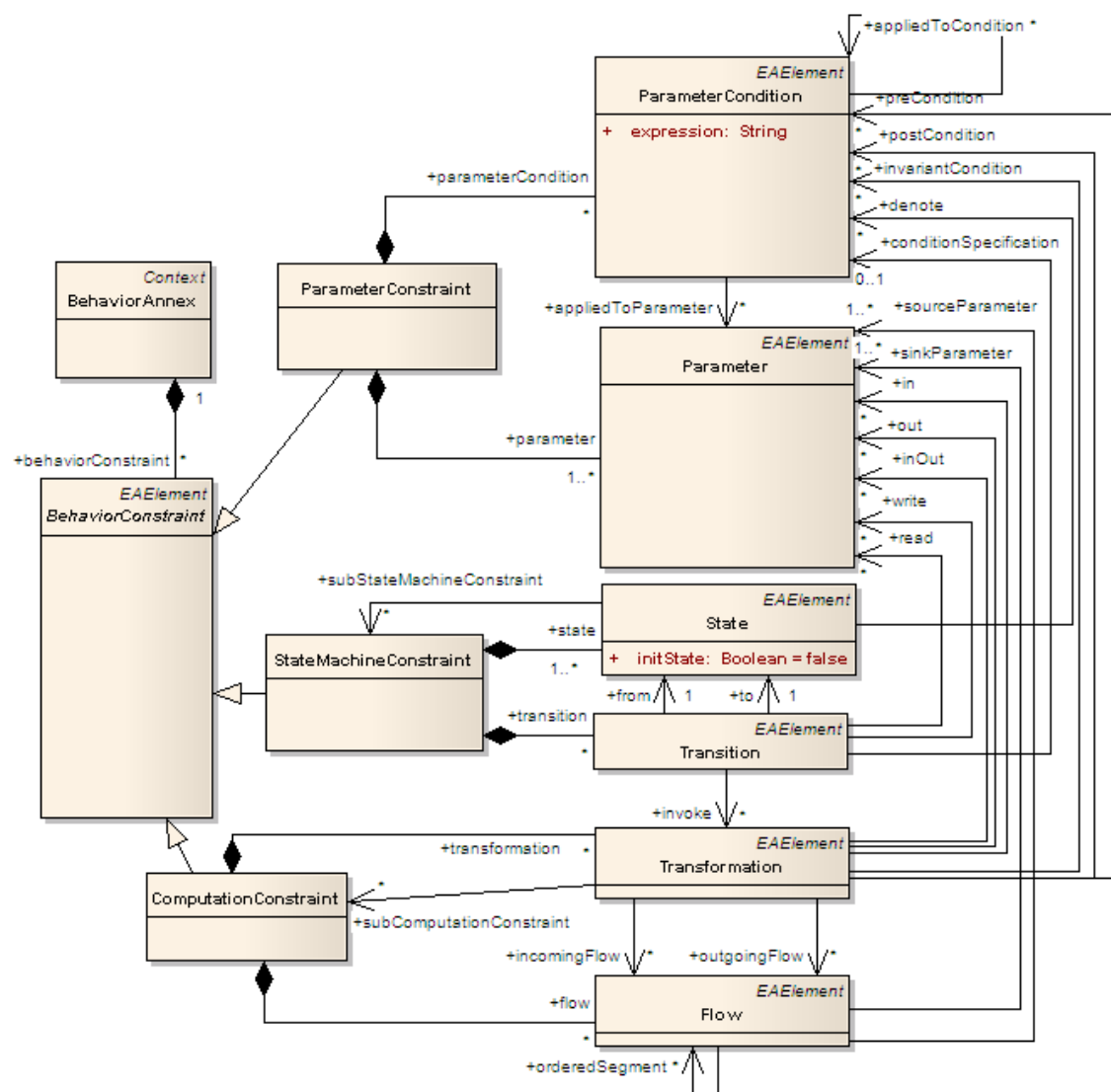
Figure 42. BehaviorConstraintsOrganization. Diagram for organization in BehaviorConstraints.

29.1 Element Descriptions

29.1.1 BehaviorAnnex (from BehaviorConstraints)

Generalizations
- Context (from Elements)

Description

Attributes

No additional attributes

Associations
- behaviorConstraint : BehaviorConstraint [*] (from BehaviorConstraints)

Constraints

No additional constraints

Semantics

-

29.1.2   BehaviorConstraint (from BehaviorConstraints) {abstract}

Generalizations
- EAElement (from Elements)

Description

Behavior constraints specify the behaviors to be fulfilled by a vehicle feature, a system artifact, or an environment entity. In particular, for system artifacts and environment entities, the behaviors under constraint include their function behaviors and function triggers.

For functional requirements and operation situations, the introduction of behavior constraints allows the related textual descriptions in the requirement model to be refined through the existing requirement refine relationship. The refinements in terms of behavior constraints provide declarations of behavior elements, including the expected parameters, states and transitions, of one or multiple functional requirements. This facilitates the analysis and validation of requirements in regards to the semantics and consistency of textual descriptions, and thereby the reuse and management of requirements and their implied behaviors.

BehaviorConstraint is an EAElement. It is further specialized by ParameterConstraint, StateMachineConstraint, and ComputationConstraint.

Attributes

No additional attributes

Associations
- relatedVehicleFeature : VehicleFeature [*] (from VehicleFeatureModeling)
  The related vehicle features under constraint. Such vehicle features satisfy the requirements that are refined by the behavior constraint.

- constrainedMode : Mode [0..*] (from Behavior)
  The constrained system mode.

- constrainedFunctionBehavior : FunctionBehavior [0..*] (from Behavior)
  The constrained function behavior.

- constrainedFunctionTrigger : FunctionTrigger [0..*] (from Behavior)
  The constrained function trigger.

- constrainedErrorBehavior : ErrorBehavior [*] (from ErrorModel)
  The error behavior that is defined by a behavior constraint.

Constraints

A behavior constraint references at least one vehicle feature, mode, function behavior, function trigger, or error behavior definition.

Semantics

-

29.1.3   ComputationConstraint (from BehaviorConstraints)

Generalizations
- BehaviorConstraint (from BehaviorConstraints)

Description

Computation constraints define required computation activities and the paths of quantities across such activities.

Attributes

No additional attributes

Associations
- transformation : Transformation [*] (from BehaviorConstraints)
  The required computation activities.

- flow : Flow [*] (from BehaviorConstraints)
  The paths of quantities across the required computation activities.

Constraints
A computation constraint contains at least one transformation or one flow definition.

Semantics
-

29.1.4   Flow (from BehaviorConstraints)

Generalizations
- EAElement (from Elements)

Description
Statement of the paths of quantities in a computation. A flow connecting the input and output parameters of a system represents an end-to-end flow of the system.

Attributes
No additional attributes

Associations
- sourceParameter : Parameter [1..*] (from BehaviorConstraints)
  The beginning of a flow.

- orderedSegment : Flow [*] (from BehaviorConstraints)
  The composed subordinate flows with an order of precedence.

- sinkParameter : Parameter [1..*] (from BehaviorConstraints)
  The end of a flow.

Constraints
A flow has at least one source and one sink parameter.

Semantics
-

29.1.5   Parameter (from BehaviorConstraints)

Generalizations
- EAElement (from Elements)

Description
Statement of quantities (e.g., temperature) in the behaviors to be fulfilled by a vehicle feature, a system artifact, or an environment entity. While input/output parameters target the I/O ports of system functions, internal parameters target directly system functions.

Attributes
No additional attributes

Associations
- targetFunctionPort : FunctionPort [0..1] (from FunctionModeling)
  The corresponding port having the parameter as its input/output parameter.

- targetFunction : FunctionType [0..1] (from FunctionModeling)
  The system function having the parameter as its internal parameter.

- type : EADatatype [1] (from Datatypes)
  The type of a parameter.

Constraints

Each parameter in the parameter constraints of function behaviors references either one function type owning such function behaviors or one function port of the same function type.

Semantics

-

29.1.6   ParameterCondition (from BehaviorConstraints)

Generalizations
- EAElement (from Elements)

Description

Statements of the conditions of individual parameters in relation to the operations of behaviors to be fulfilled by a vehicle feature, a system artifact, or an environment entity.

Attributes
- expression : String [1]
  The description of parameter condition.

Associations
- representAnomaly : Anomaly [*] (from ErrorModel)
  The fault(s)/failure(failures) represented by the parameter condition.

- appliedToParameter : Parameter [*] (from BehaviorConstraints)
  The quantity characterized by the parameter condition.

- appliedToCondition : ParameterCondition [*] (from BehaviorConstraints)
  The subordinate parameter conditions characterized by the parameter condition.

Constraints

A parameter condition is applied to at least one parameter or one parameter condition.

Semantics

-

29.1.7   ParameterConstraint (from BehaviorConstraints)

Generalizations
- BehaviorConstraint (from BehaviorConstraints)

Description

Statement of the expected parameters and parameter conditions in the operations of behaviors to be fulfilled by a vehicle feature, a system artifact, or an environment entity.

Attributes

No additional attributes

Associations
- parameterCondition : ParameterCondition [*] (from BehaviorConstraints)
  The description of parameter conditions.

- parameter : Parameter [1..*] (from BehaviorConstraints)
  The expected parameters.

Constraints

No additional constraints

Semantics

Computation constraints refine textual requirements and provide detailed specifications about the quantities and their particular conditions in the operation of behaviors.

29.1.8   State (from BehaviorConstraints)

Generalizations
- EAElement (from Elements)

Description

Statement of state elements in state-machine constraints.

Attributes
- initState : Boolean = false [1]
  Indicating an initial state when the value is true.

Associations
- representMode : Mode [*] (from Behavior)
  The Mode(s) that is represented by the state.

- subStateMachineConstraint : StateMachineConstraint [*] (from BehaviorConstraints)
  The subordinate state machine in the state.

- denote : ParameterCondition [*] (from BehaviorConstraints)
  The parameter conditions represented by the state.

Constraints

No additional constraints

Semantics

A state is an element in state-machine description. Each state represents a set of parameter conditions that are of particular concern in the operations of behaviors to be fulfilled by a vehicle feature, a system artifact, or an environment entity. A state can also represent a vehicle mode and thereby provides detailed information about the related parameters and mode transitions.

Within each state, there can be subordinate state-machines. A subordinate state-machine becomes active if its composite state is active and terminates when the composite state is exited.

29.1.9   StateMachineConstraint (from BehaviorConstraints)

Generalizations
- BehaviorConstraint (from BehaviorConstraints)

Description

Statement of state-machine constraints of behaviors to be fulfilled by a vehicle feature, a system artifact, or an environment entity.

Attributes

No additional attributes

Associations
- transition : Transition [*] (from BehaviorConstraints)
  Owned transitions in the state-machine.

- state : State [1..*] (from BehaviorConstraints)
  Owned states in the state-machine.

Constraints

No additional constraints

Semantics

State-machine constraints refine textual requirements and provide detailed specifications about the states of quantities and the state transitions in the operation of behaviors.

The definition of state-machine constraint follows a generic definition of automata: In one state, read certain parameter, upon certain parameter condition(s), do certain transformation(s), then go to another state.

Each state-machine description normally has a set of states and transitions. A state-machine has a single initial state. Only one state is active during the operation.

29.1.10   Transformation (from BehaviorConstraints)

Generalizations
- EAElement (from Elements)

Description

A transformation defines an expected computation activity on two sets of quantities in terms of parameters. It describes one set of parameters as a function of other parameters, under the constraint of pre-, post-, and invariant parameter conditions.

Attributes

No additional attributes

Associations

- invariantCondition : ParameterCondition [*] (from BehaviorConstraints)
  The parameter conditions that must remain unchanged by the execution of the transformation.

- subComputationConstraint : ComputationConstraint [*] (from BehaviorConstraints)
  The subordinate computation constraint in the transformation.

- inOut : Parameter [*] (from BehaviorConstraints)
  The parameters that are used both as inputs and as outputs of the transformation.

- outgoingFlow : Flow [*] (from BehaviorConstraints)
  The related outgoing flows from the transformation. The definitions are derived according to the declarations of flows connected to the output parameters of the transformation.

- out : Parameter [*] (from BehaviorConstraints)
  The output parameters of the transformation.

- in : Parameter [*] (from BehaviorConstraints)
  The input parameters of the transformation.

- preCondition : ParameterCondition [*] (from BehaviorConstraints)
  The parameter conditions that must hold before the transformation can start its execution.

- postCondition : ParameterCondition [*] (from BehaviorConstraints)
  The parameter conditions that must hold after the execution of the transformation.

- incomingFlow : Flow [*] (from BehaviorConstraints)
  The related incoming flows to the transformation. The definitions are derived according to the declarations of flows connected to the input parameters of the transformation.

Constraints

[1] A transformation has at least one out or one inOut parameter.

Semantics

-

29.1.11 Transition (from BehaviorConstraints)

Generalizations
- EAElement (from Elements)

Description
Statement of state transtion elements in state-machine constraints.

Attributes
No additional attributes

Associations
- invoke : Transformation [*] (from BehaviorConstraints)
  The transformations to be invoked when the transition is fired.

- to : State [1] (from BehaviorConstraints)
  The target state of the transition.

- conditionSpecification : ParameterCondition [0..1] (from BehaviorConstraints)
  The parameter condition(s)that must hold to fire the transition.

- write : Parameter [*] (from BehaviorConstraints)
  The parameters that will be available when the transition is fired.

- read : Parameter [*] (from BehaviorConstraints)
  The parameters that must be available to fire the transition.

- from : State [1] (from BehaviorConstraints)
  The source state of the transition.

Constraints
No additional constraints

Semantics
A transition goes from a source state to a target state. A transition can only fire if its source state is active, its read parameter is available, and the related parameter condition(s) holds. When it is fired, a transition can invoke certain transformations and write certain parameters (i.e., making such parameters available).

30   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

31 Index