



Contract number: 2004 - 026976

Advancing Traffic Efficiency and Safety through Software Technology (ATESST)

Report type	Deliverable D2.1.3
Report name	EAST-ADL Update Suggestions (for Requirements and V&V)
Dissemination level	PU
Status	Final
Version number	1.0
Date of preparation	2008-02-01

Authors**Editor**

Matthias Weber

E-mailMatthias.Weber@carmeq.com**Authors**

Lars-Olof Berntsson

Lars-Olof.Berntsson@volvo.com

Hans Blom

Hans.Blom@volvo.com

Philippe Cuenot

philippe.cuenot@continental-corporation.com

Rolf Johansson

Rolf_Johansson@mentor.com

Henrik Lönn

Henrik.Lonn@volvo.com

Mark-Oliver Reiser

Mark-Oliver.Reiser@web.de

Matthias Weber

Matthias.Weber@carmeq.com**The Consortium**

Volvo(S)

DaimlerChrysler (D)

Siemens (F)

ETAS (D)

Mentor Graphics (Hu)

CEA (F)

The Royal Institute of Technology (S) Technische Universität Berlin (D)

Carmeq GmbH (D)

Volvo Cars (S)

Mecel (S)

Revision chart and history log

Version	Date	Reason
0.1	2007-10-29	Initial Version
0.2	2007-11-21	new version of V&V update suggestions (following WP2 TelCo of Nov 19)
0.3	2007-12-17	
0.4	2008-01-02	Version for Review
1.0	2008-01-18	Final Version (various additions, clarifications, and fixes).

List of abbreviations

FDA	Functional Design Architecture
RE	Requirements Engineering
RIF	Requirements Interchange Format
RM	Requirements Management

Table of contents

List of Figures..... 7

1 Introduction 8

2 User Attributes 9

 2.1 Attaching User Attributes to an ADLEntity 9

 2.2 Defining User Attributes 10

 2.3 Appropriate Tool Support..... 12

3 Requirements in EAST-ADL2 13

 3.1 Generic Requirements 13

 3.2 Specialized Requirements 14

 3.3 Comparison to EAST-ADL 1.0 14

4 Verification & Validation in EAST-ADL2..... 16

 4.1 Basic Verification and Validation Entities..... 16

 4.2 Abstract vs. Concrete Entities..... 16

5 Plugin for Requirements Support..... 18

 5.1 Main Plugin Capabilities..... 18

 5.2 RIF Import / Export..... 18

 5.2.1 *The Requirements Interchange Format (RIF)* 18

 5.2.2 *Rationale for Using RIF* 20

 5.2.3 *Basic Approach towards RIF Import and Export* 21

 5.2.4 *Import*..... 22

 5.2.5 *Export* 26

 5.3 Further Topics..... 27

6 Update Suggestions for EAST-ADL2 – Part I: User Attributes..... 28

 6.1 ADLEntity 29

 6.2 AttributeValue (from RIF) 30

 6.3 DatatypeDefinition (from RIF) 30

 6.4 UADatatype..... 30

 6.5 UAElementType..... 30

 6.6 UATemplate 31

 6.7 UserAttributeableElement 32

 6.8 UserAttributeDefinition 33

 6.9 UserAttributeValue 34

7	Update Suggestions for EAST-ADL2 – Part II: Requirements	35
7.1	ADLRequirement	35
7.2	ADLRequirementContainer.....	36
7.3	ADLTraceableSpecification.....	37
7.4	ReqGroup	37
7.5	ReqRelation	38
7.6	ReqRelationGroup	39
7.7	UAElementType.....	39
7.8	UATemplate	39
8	Update Suggestions for EAST-ADL2 – Part III: Verification & Validation	40
8.1	AbstractVVCASE.....	41
8.2	AbstractVVProcedure	42
8.3	ADLContext.....	43
8.4	ADLEntity	43
8.5	ADLRelationship	43
8.6	ADLRequirement	44
8.7	ADLTraceableSpecification.....	44
8.8	ADLVerify.....	44
8.9	ConcreteVVCASE	45
8.10	ConcreteVVProcedure	46
8.11	VVActualOutcome	47
8.12	VVIntendedOutcome	47
8.13	VVLog.....	48
8.14	VVStimulus.....	48
8.15	VVTarget	49
9	Contribution to overall ATESST objectives	50
10	Conclusions	51
11	References	52

List of Figures

Figure 1. Attaching user attribute values to an ADLEntity.....	10
Figure 2. Defining names and data types for user attributes.....	11
Figure 3. Core structure of a RIF specification.....	19
Figure 4. Defining attributes of objects with SpecTypes and AttributeDefinitions.	20
Figure 5. Important objectives of the plugin beyond basic Import and Export.....	21
Figure 6. Mapping RIF SpecTypes and their attributes to EAST-ADL2 entities.....	23
Figure 7. Mismatch between containers of an attribute definition.	24
Figure 8. Using standard tools to view and edit EAST-ADL2 information.	26
Figure 9 Domain model for user attribute values.	28
Figure 10 Domain model for user attribute definitions.....	29
Figure 11 Domain model for generic requirements.	35
Figure 12 Domain model for verification and validation.	40
Figure 13 Relating requirements to verification & validation entities.....	41

1 Introduction

Requirements are the input information for any system design effort, and contains the constraints and rules during the entire design effort. To improve dependability and in particular safety, rigorous treatment of requirements is critical. An architecture description language captures the elements that compose the modeled system architecture in a comprehensive way. It is therefore desirable to also investigate requirements, and how they can be associated to the system elements.

In this deliverable, detailed update suggestions to EAST-ADL1 are given in order to provide a solid basic for requirements management in EAST-ADL2. Since this is closely related to the concept of user attributes as well as verification and validation support – as will be explained in the coming section – these two topics are also covered in this deliverable and related update suggestions are given below.

2 User Attributes

User attributes in EAST-ADL2 are primarily intended to provide a mechanism for augmenting the elements of an EAST-ADL2 model with customized meta-information. All instances of metaclass ADLEntity 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-ADL2 elements within UATemplates.

Since EAST-ADL2 requirements are in their most general form simple objects with all information contained in user-customized, project-specific attributes, the concept of user attributes is also perfectly suitable to define those attributes of requirements. In that sense, basic requirements in EAST-ADL2 can be seen as “empty” elements which only provide a node to which user attributes can be attached in order to supply the requirement with all necessary information, including its main textual description. However, in case of requirements the context in which the available user attributes are defined is different: here the container of the requirements is the point where user attribute definitions are store and these are then applicable only within this container.

The role of user attributes within the overall EAST-ADL2 is thus twofold: they (1) provide a means to customize the language to specific company and project needs and (2) constitute an important part of the requirements support of the language.

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

We can therefore look at user attributes in two separate steps: we first examine how user attributes can be attached to an ADLEntity and secondly we investigate attribute definitions.

2.1 Attaching User Attributes to an ADLEntity

The core of user attribute support in EAST-ADL2 is the abstract metaclass UserAttributeable-Element. It represents entities that may be “attributed”, i.e. have a customized attribute attached to them. To do so, an instance of UserAttributeValue is created and added to this entity’s “uaValues” association (see Figure 1). Each such value has a string attribute “key” which uniquely identifies the user attribute for which a value is provided. The actual value is supplied in form of an instance of class AttributeValue which is inherited from RIF. The EAST-ADL2 user attribute mechanism makes use of RIF’s value and datatype concept, but not of the other structured of RIF.

Since ADLEntity inherits from UserAttributeableElement, all instances of ADLEntity can be supplied with user attributes.

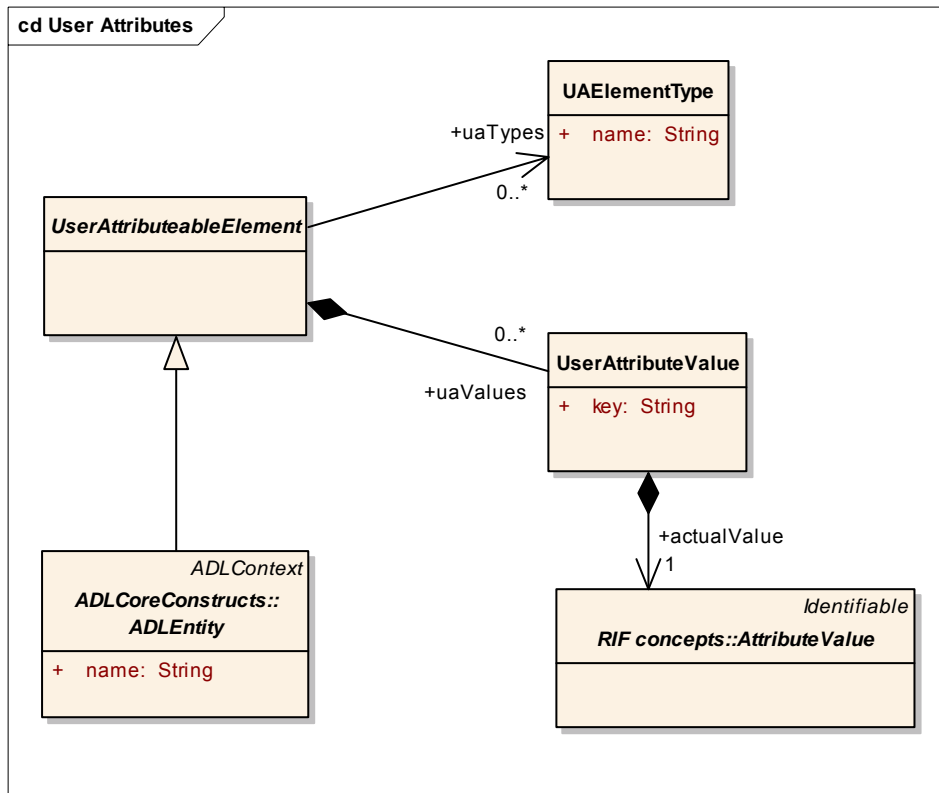


Figure 1. Attaching user attribute values to an ADLEntity.

It is important to again emphasize at this point that on the level of the language definition the attribute values are treated completely independent from the attribute definitions and the types defined for the attributes. This means it is perfectly legal to ...

1. provide an attribute value with a key for which no applicable attribute definition is available (i.e. attribute value for an undefined attribute).
2. provide an attribute value with a key for which an attribute definition is applicable but with an actual value that violates the data type specified in the attribute definition (i.e. attribute value with an invalid type).

As stated above, this is meant to facilitate everyday work with user attributes (e.g. evolution of attribute definitions) and reduce the complexity of implementations of EAST-ADL2.

2.2 Defining User Attributes

In addition to providing values for user attributes, it is possible to optionally define what user attributes should be provided for certain elements and what actual values are allowed in each case. This is done with a **UserAttributeDefinition** (see Figure 2).

The attribute definitions are not directly provided for individual elements in the EAST-ADL2. They are rather provided for **UAElementTypes**, which define certain types of **UserAttributeableElements**. As above for values, RIF data types are reused here for to specify the data type of user attributes. The attribute definitions of a **UAElementType** are then applicable to all entities of this type, more precisely: they are applicable to all entities that have this **UAElementType** in their “uaTypes” association.

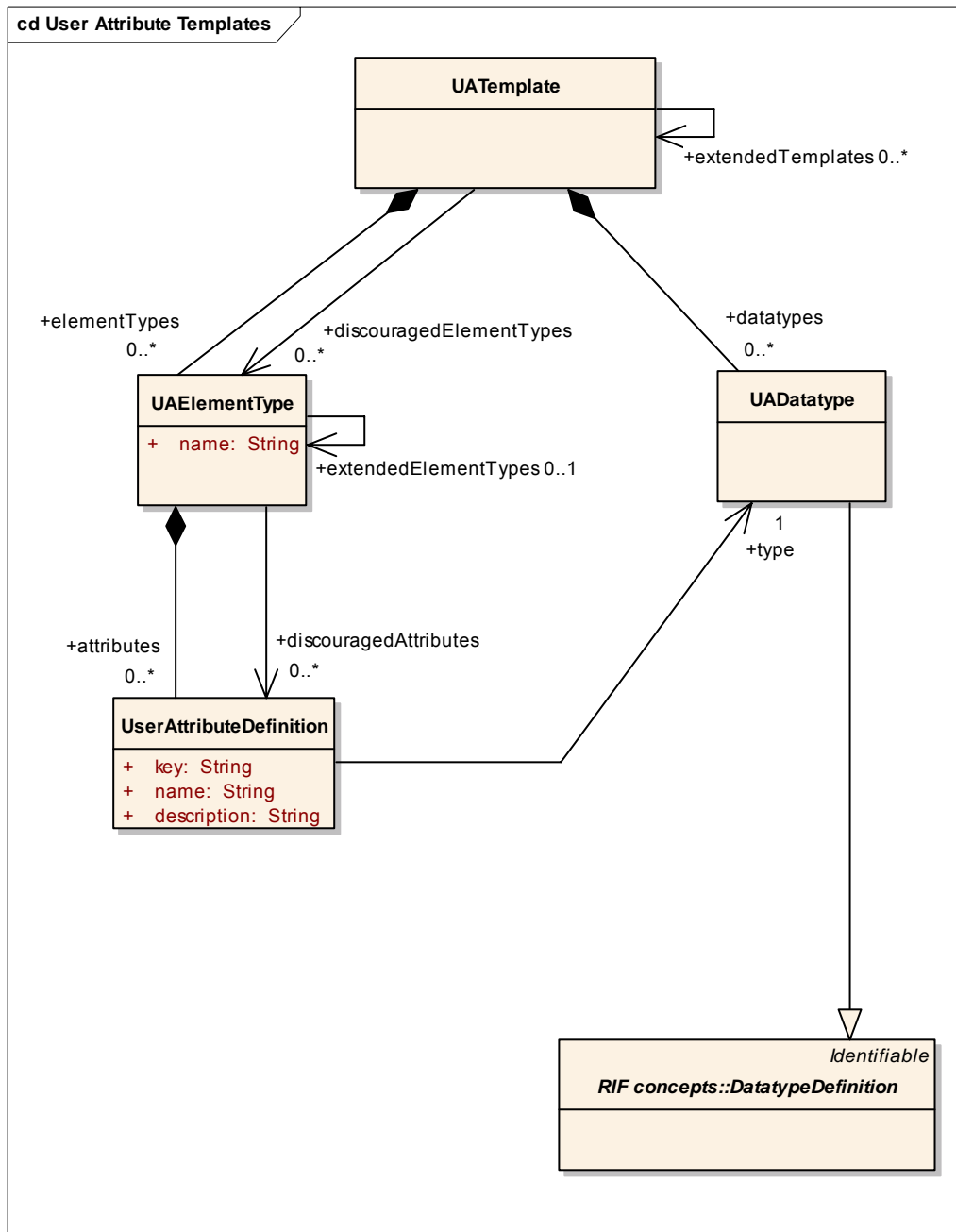


Figure 2. Defining names and data types for user attributes.

For example, the car manufacturer Audi could introduce a UAElement called “AudiFunctionType” with a UserAttributeDefinition having its attribute “key” set to “com.audi.functionType.status” and an appropriate data type specification. We can then set this UAElement in the “uaTypes” association of all ADLFunctionTypes that should be amended with this status attribute. EAST-ADL2 modeling tools should provide appropriate support for aiding the engineer in setting the uaTypes of entities, for example it should be possible to define default types for certain metaclasses which are automatically set for instances of these classes (see also below).

Another core concept of user attribute definition is that of a **UATemplate**. It is used to organize several user attribute definitions which share a certain purpose, origin or scope. This way it is also possible to easily reuse attribute definitions across projects, departments or companies or to

define, for example, the attributes specifically used in a particular manufacturer-supplier relationship separately from the manufacturers and supplier “private” attributes. A simple inheritance concept is available to hierarchically structure the definition of UATemplates and to reuse one template within another (also with the possibility of modifying it to some extent).

More details can be found below in the section describing the EAST-ADL2 update suggestions for user attributes.

2.3 Appropriate Tool Support

Both levels of the concept of user attributes – plain attribute values as well as the optional attribute definitions in UAElementTypes and UATemplates – rely on an appropriate support by EAST-ADL2 tools. This is particularly true for modeling tool.

Beyond the basic functionalities such as adding, changing and removing user attribute value or creating and editing attribute definition and templates, tool should especially provide means to

- define default UAElementTypes for certain metaclasses of the EAST-ADL2 which are then automatically set for all instances of these classes.
- extract all or selected user attribute values from a model for use cases in which only the user attributes are of interest.
- remove all or selected user attribute values from a model for use cases in which user attribute must not be disclosed to someone that the model is given to (e.g. a supplier).
- check if all attribute values conform to the applicable attribute definition / highlight violations
- many more ...

Tools that do not provide any editing functionality (e.g. code generators) can probably safely ignore most of this additional features and simply support plain attribute values. This is another strong argument for strictly separating the concept of attribute values and attribute definitions as described above.

3 Requirements in EAST-ADL2

Requirements do not constitute an independent view in EAST-ADL2. In a general ADL architecture, requirement entities will be parts of already existing artifacts. The present framework will show several constructs, which could be used, in principle, in any artifact (VFM, FAA, FDA..). Specific constraints on this will be shown in the related sections.

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

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

EAST-ADL2 has constructs that deal with these problems. Some of these constructs are constructs that deals with general issues in software development and have been already addressed in the past by general processes, like IBM RUP (IBM Rational Unified Process). Here, RUP will not be intended as a guideline; concepts and definitions from RUP will though be used when useful whereas new ones will be introduced when required.

As done for the structure part of EAST-ADL2, the requirements part will be compliant with UML2. The EAST-ADL2 adapts existing concepts whenever possible and develops new ones otherwise. And important source on the level of requirements is the Requirements Interchange Format (RIF).

The purpose of the metaclasses in the Requirements domain meta-model package is to specify rigorously ("formally") the Requirements concepts for the specific domain. Section 7 contains a specification for each individual metaclass in the Requirements package.

Support for requirement modeling is provided by the EAST-ADL2 on two levels: a generic level and a specialized level where specialized requirement entities are provided which are intended for certain special uses.

3.1 Generic Requirements

On the basic, generic level, requirements are entities for which all information is provided in user-customized, project-specific attributes. This follows the common view in requirements engineering that the detailed structure of a requirements specification cannot be prescribed for all conceivable application cases. Instead it must be possible to define the structure of the requirements, in particular their types and attributes, for each specification document separately. This is also the approach followed in most standard requirements management tools (e.g. Telelogic DOORS) and in RIF.

Generic requirements reuse the concept of user attributes for the project-specific attributes of requirements.

Generic requirements in EAST-ADL2 can also be used to capture other information which does not represent requirements in a strict sense. Consider the following example:

Example: In order to describe four use cases, we create four instances of ADLRequirement which each describe a use case (use case is just an example for some information which is not natively covered by the EAST-ADL2). We have a single UAElementType “UseCase” which defines several attributes. All 4 ADLRequirements have this UAElementType defined as their uaType (association “uaTypes” of UserAttributeableElement) and have appropriate attribute values defined with entity UserAttributeValue.

We obtain:

<pre><<ADLRequirement>> MyUseCase#1 with uaType = <<UAElementType>> UseCase and uaValues = <<UserAttributeValue>> MyValue</pre>
<p><i>similarly for use cases #2 to #4 ...</i></p>
<pre><<UAElementType>> UseCase with attributes = <<UserAttributeDefinition>> MyAttributeDefinition</pre>

Usually a use case description will consist of more than one attribute, so the UAElementType called “UseCase” will have several attribute definitions.

3.2 Specialized Requirements

Since the EAST-ADL2 language is also aimed at fostering a standardization of software development information as far as this can be deemed desirable and feasible, the language provides, in addition to the generic requirements, a number of specialized requirements for certain purposes (e.g. timing requirements). These can be seen as standard structures for requirements specifications, similar as the project-specific ones described in the previous section.

Another reason for introducing certain specialized requirements is that they can be inter-related more tightly and rigidly with the rest of the EAST-ADL2 syntax and their meaning can be defined as part of the EAST-ADL2 semantic. Examples for such specialized requirements are point-to-point timing requirements and precedence constraints.

The specialized requirements inherit from the generic requirement entity. It is thus possible to also further customize these specialized requirements, for example by defining additional attributes.

3.3 Comparison to EAST-ADL 1.0

The previous version of the EAST-ADL (as defined in the EAST-EEA project) did not make the distinction between generic and specialized requirements. Some special constructs from EAST-ADL1 are no longer supported as dedicated entities in the domain model, because it is preferable to simply model the corresponding information in form of generic requirements.

For example, the previous version of the EAST-ADL contained a dedicated use-case concept. This is no longer the case, as use cases are used in practice for many different purposes and consequently come with heavy company- or project-specific customizations. Specific use-cases can be modeled in EAST-ADL2 using the concept of generic requirements in connection with appropriate user attributes.

Furthermore, EAST-ADL1 contained – in addition to the feature concept – a concept of functional interactions. The feature interpretation of EAST-ADL 1.0 was semantically constrained, serving only as an abstract model of individual vehicle functions. This constrained interpretation gave rise

to an additional concept of “interaction” in order to model interactions between two or more individual functions. The feature interpretation of EAST-ADL2 is more general and closer to the common interpretation of this term. It therefore can be used to model both individual functions as well as interactions between functions. Consequently, the concept of interaction was dropped from EAST-ADL2.

For more details on requirements support, refer to the corresponding EAST-ADL2 update suggestions as defined later in this deliverable.

4 Verification & Validation in EAST-ADL2

A multitude of different verification and validation (V&V) techniques, methods and tools are applied during the design of embedded software. Different techniques are applicable at different abstraction levels. Also, the technique of choice depends on the properties to validate and/or verify. Furthermore, each partner in a project may develop and employ his own V&V processes and activities. Hence it is impossible to introduce in the EAST-ADL2 a way to model all the objects that can be required by all the possible V&V techniques. As a consequence, EAST-ADL2 furnishes just the means for planning, organizing and describing V&V activities on a fairly abstract level and for defining the links between those V&V activities, the requirements that are checked by them and the objects modeling the system (Functional Analysis Architecture, Functional components, Logical Tasks, etc.). The common parts of all V&V techniques are described by the EAST-ADL2, which includes: the results expected from the V&V activities, the actual results which were obtained when applying the V&V techniques, how the V&V activities are constrained. Information that is specific to an individual V&V technique is not described in EAST-ADL2, but a place for storing this information is provided.

Single V&V techniques may be used only once or at several stages during an overall V&V effort. Some of them are specific to one model of the ADL, others can be applied at various design stages. In any case, a set of V&V techniques and activities is necessary in order to achieve a complete verification and validation of a given system. Often these techniques and activities are employed and performed by many different teams and departments, frequently even by different companies. This raises the demand for an overall planning and organization of all V&V related information.

4.1 Basic Verification and Validation Entities

Support for V&V relies on the following main entities:

- **VVCases:**
A VVCase is the core concept of verification and validation support in EAST-ADL2 and represents a certain verification and validation effort of varying scope and intention.
- **VVProcedures:**
A VVProcedure represents an individual task in the context of an overall verification and validation effort (represented by a VVCase), which has to be performed in order to achieve that effort's overall objective.
- **VVTargets:**
A concrete testing environment in which or on which a particular verification and validation 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.
- **VVLogs:**
Captures the outcome of an actual execution of a verification and validation effort.

There are numerous other, subordinate entities of course. For a description of these please refer to the section with the detailed EAST-ADL2 update suggestions below.

4.2 Abstract vs. Concrete Entities

A very important notion of V&V support in EAST-ADL2 is the distinction of abstract and concrete V&V information:

- On the **abstract level**, verification and validation information is defined without referring to a concrete testing environment and without specifying stimuli and the expected outcome of a particular VVProcedure on a detailed technical level.
- On the **concrete level**, verification and validation information specifies a concrete testing environment and provides all necessary details for testing, e.g. stimuli and expected outcomes, on a concrete technical level applicable to that testing environment.

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

5 Plugin for Requirements Support

One part of the ATESSST effort was concerned with tool support for requirements management and verification & validation within the EAST-ADL2. Since such a tool would address a certain cross-cutting aspect of system development, it would need to be closely integrated with the EAST-ADL2 modeling tool. Similarly as in the case of other extensions such as variability or analysis, it must therefore be designed as an additional plugin of the ATESSST platform. The ATESSST platform was based on the Eclipse framework and the tool support for RE and V&V can thus be implemented as an Eclipse Plugin. In the following, we therefore refer to the EAST-ADL2 tool support for RE and V&V as “the plugin for requirements support” or simply “the plugin”.

5.1 Main Plugin Capabilities

The investigation of tool support in WP2.1 was not aimed at reinventing the wheel by providing yet another tool for requirements engineering. There exist several well-established tool suites for that purpose, for example Telelogic DOORS or IBM’s RequisitePro, and they are widespread in industry already, having been introduced and being applied in many companies. In the automotive domain the tool DOORS is particularly common, thanks to its flexibility and extensibility. With such a situation at hand, it would not make sense to introduce an all new RE tool.

For this reason, the work in WP2.1 on the plugin for requirements support was aimed at investigating how requirements management in the context of the EAST-ADL2 can be achieved by making use of legacy RE tools. This is mainly achieved by way of a very flexible import and export filter that makes available EAST-ADL2 models in some common RE tool. This is not done by a direct export from EAST-ADL2 to the RE tool. Instead, the requirements interchange format (RIF) can be used as an intermediate representation which may then serve as a link to many different legacy RE tools.

5.2 RIF Import / Export

In this section we discuss the import and export functionality of the plugin. As stated above, this is of great relevance, because we assume that editing of requirements will take place in some external standard tool for requirements management, such as Telelogic DOORS.

5.2.1 The Requirements Interchange Format (RIF)

The import and export of development information to and from the EAST-ADL2 is not accomplished by a direct import or export between EAST-ADL2 and a standard RE tool. Instead, the *Requirements Interchange Format* (RIF) is employed as an intermediate representation which may serve as a uniform link to many different legacy RE tools¹.

RIF was introduced by the “Herstellerinitiative Softwaretechnik” (HIS) [1], which is a joint effort of the German automotive manufacturers Audi, BMW, Daimler, Porsche, and Volkswagen to achieve

¹ The fact that RIF as an intermediate representation is actually not perfectly uniform and independent of the target tool is discussed in detail below.

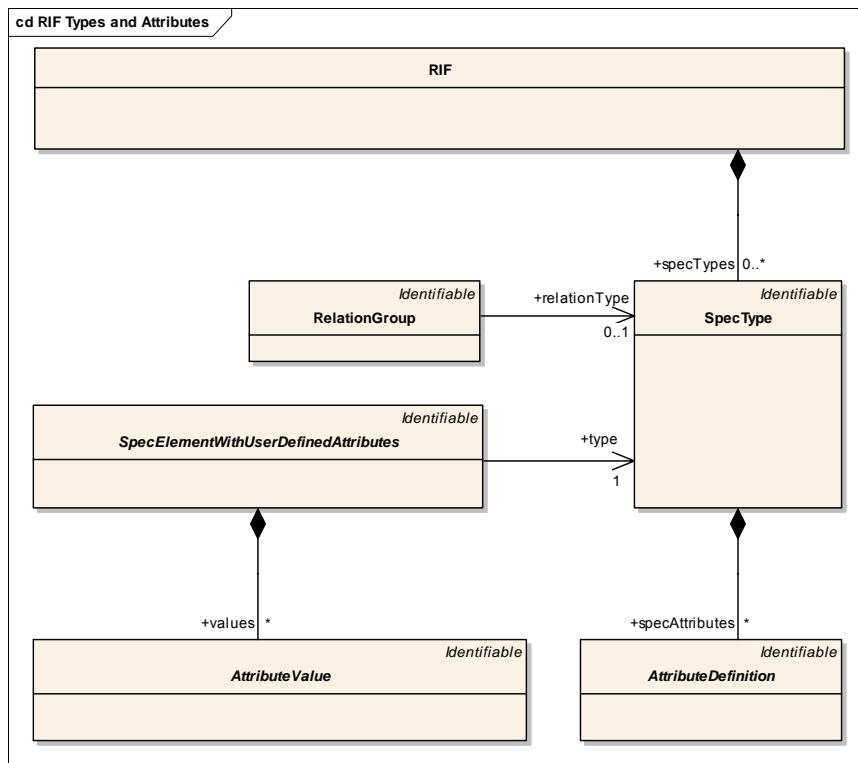


Figure 4. Defining attributes of objects with SpecTypes and AttributeDefinitions.

5.2.2 Rationale for Using RIF

Before going into the details of importing and exporting, let us take a brief look at the strength of the RIF format which are of particular interest in the context of the ATESSST project:

- *Flexibility*
As shown above, the basic structure of a RIF specification is extremely flexible: it mainly consists of – optionally inter-related or grouped – SpecObjects of various SpecTypes, which are each provided with a number of AttributeDefinitions.
- *Open Standard.*
The specification of RIF is freely available over the web and is further maintained under the auspices of the “Herstellerinitiative Softwaretechnik” (HIS) [1].
- *RIF is becoming increasingly popular.*
More and more tool vendors provide an import and export from/to RIF for their tools. In addition, the RIF format has drawn during the last two years increasing attention from practitioners and researchers in the field of requirements engineering.
- *Fits well to important conceptions in EAST-ADL2.*
In particular, a requirement in EAST-ADL2 is a very flexible element which mainly consists of project-specific attributes. There are special cases of requirements with predefined attributes and relations, but they can be seen as a special case of a flexible requirement. This matches well the flexible structure of RIF with SpecTypes and their AttributeDefinitions.

Other important features and characteristics of RIF, such as its support for access rights management with AccessPolicy objects, can be found in the RIF specification document [2].

5.2.3 Basic Approach towards RIF Import and Export

The evident purpose of a RIF import and export filter for the EAST-ADL2 is to import pure requirement specifications, i.e. RIF documents that only contain requirements information, to only requirement related entities in the EAST-ADL2. Correspondingly, the most evident form of an export is to export only requirement related EAST-ADL2 entities to a RIF document. These two trivial cases of a pure requirement related import and export is depicted in Figure 5 with the solid, bidirectional arrow from left to right.

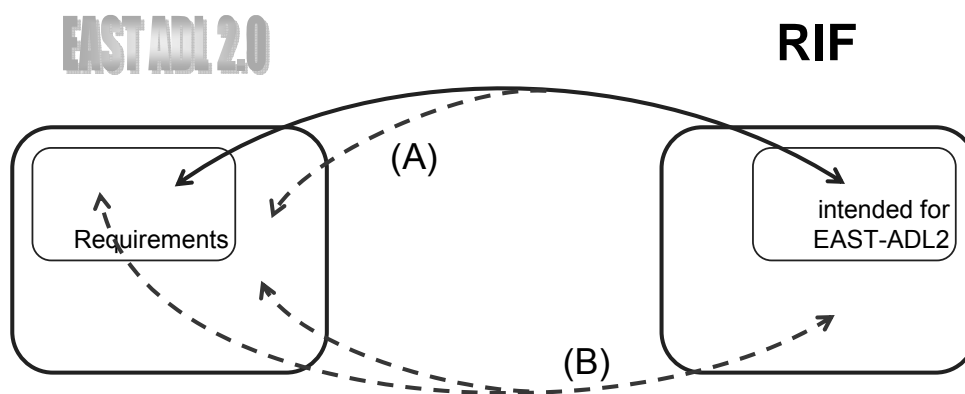


Figure 5. Important objectives of the plugin beyond basic Import and Export.

In addition to such a “simple” import and export of requirements, we also want to support two other, advanced forms of importing and exporting information:

1. all EAST-ADL2 elements can be target of an import and all can be exported (not only requirements)
2. support for import of RIF data which was not initially intended for EAST-ADL2

The first item might seem fairly obvious at first sight, but in general RIF is primarily intended for requirements information and it is not common to represent information on, for example, the software design as captured in the EAST-ADL2 functional design architecture in form of requirements. However, for many use cases of and operations on such non-requirement related data this is feasible and desirable. This additional aim of the import/export filter is depicted in Figure 5 with the dashed arrow labeled with (A).

The second advanced aim for our RIF import/export is related to the scope and structure of the data on the RIF side. Since the RIF format is extremely flexible, the input might be any type of information of arbitrary structure. In such cases, import filters often impose certain assumptions on the data to be imported. In our case however, we do not want to introduce such limitations; in particular, we want to be able to cope with the following mismatches:

- The data on the RIF side contains information for which there is no appropriate entity or attribute in the EAST-ADL2. In this case we want to be able to capture that excessive information within the EAST-ADL2.
- The data on the RIF side corresponds well to EAST-ADL2 entities and attributes but it is conversely structured. For example, information that is split across multiple elements in the EAST-ADL2 is joined in a single SpecObject on the RIF side.

The import and export of such “orthogonal” data which was not initially intended for use in the EAST-ADL2 is shown in Figure 5 as a dashed arrow labeled with (B).

In both approaches, EAST-ADL2 and RIF, requirements are entities with project-specific, customizable attributes. Therefore the import of plain requirements data into EAST-ADL2 requirements is a standard RIF import and does not introduce any additional, ATESSST specific challenges². Therefore, the following sections focus on the two advanced cases above.

5.2.4 Import

To organize the following discussion in this section and to deal with the multitude of different problem cases that must be considered during import, especially when taking into account the advanced objectives described in the previous section, we have to identify a few main problem cases. These are:

1. Perfect match
(i.e. for all SpecTypes and AttributeDefinitions in RIF, there is a corresponding EAST-ADL2 entity or attribute)
2. Missing attribute in EAST-ADL2 for a RIF AttributeDefinition
3. Missing entity in EAST-ADL2 for a RIF SpecType
4. Lack of or inappropriate information on RIF side
5. Fundamental semantic mismatch
6. Combinations of the above cases

In the remainder of this section we will look at each of these cases in detail.

Case 1: Perfect Match

Here we assume that the information on the RIF side perfectly corresponds to EAST-ADL2 entities, but we have to provide a mapping between the SpecTypes and AttributeDefinitions in RIF and the EAST-ADL2 entities and their attributes.

² However, even this “trivial” case is considerable complex and is involved with several intricacies. Since they are to be seen as ‘classic’ RIF issues they are deemed outside the scope of this deliverable. Some discussions on these issues can be found in the aforementioned RIF specification and accompanying documentations.

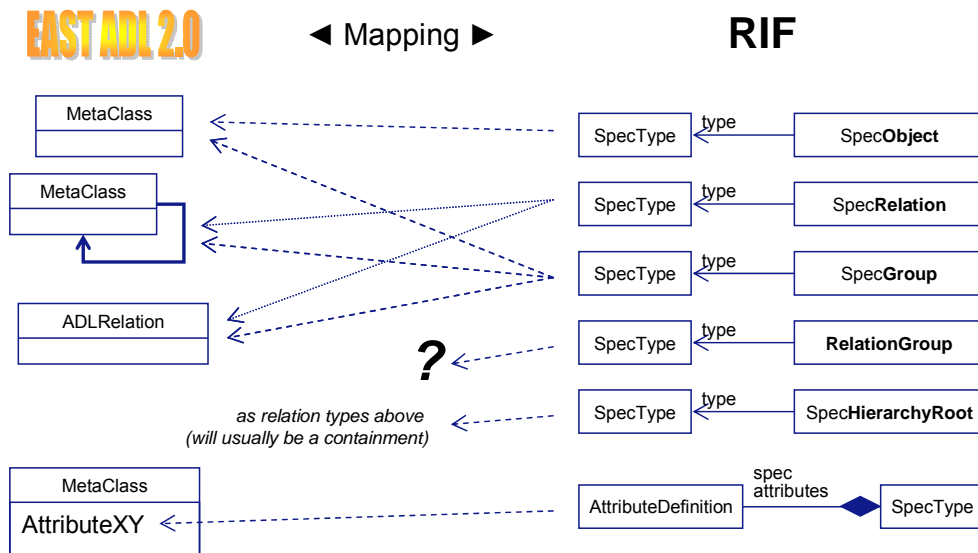


Figure 6. Mapping RIF SpecTypes and their attributes to EAST-ADL2 entities.

The right side of Figure 6 shows several cases of type and attribute definition in RIF. The cases of SpecType definition differ in that each time the SpecType is used for different sub-types of SpecElementWithUserDefinedAttributes, i.e. SpecObject, SpecRelation, SpecGroup, and SpecHierarchyRoot (highlighted in the figure with bold face font). To simplify the following discussion, we call a SpecType which is used as the type of a SpecObject an “object type”, one that is used as the type of a SpecRelation a “relation type”, and so on.

In practice, a single SpecType may be used to type objects of more than one of these classes, i.e. it is possible to use a single SpecType as the type of one or more SpecObjects and at the same time one or more SpecRelations. For the import filter this means that it must be possible to distinguish these cases, i.e. it must be possible to provide a different mapping to EAST-ADL2 entities for the same SpecType depending on whether it is used as the type of a SpecObject or SpecRelation, for example. With the above abbreviation we can say that a single SpecType may at the same time be an object type and a relation type.

Object types are relatively easy in that they are always mapped to a meta-class in the EAST-ADL2 domain model (remember that we are here assuming the “perfect match” case).

Relation types are slightly more intricate because two different techniques are used in the EAST-ADL2 domain model to allow to relate entities in the ADL:

1. Plain Associations
2. subclasses of the metaclass ADLRelation

When mapping RIF SpecTypes to the EAST-ADL2 these two cases can occur and the import filter has to be able to cope with them.

Group types are also a bit intricate because outside the scope of requirements entities, no generic grouping mechanism is provided in the EAST-ADL2. Since we are still assuming the perfect match case, the information captured within a SpecGroup of that SpecType must have a corresponding entity or attribute in the EAST-ADL2. All kinds of entities, associations and attributes are conceivable as a target for this information. Therefore the import filter must allow to map groups accordingly.

Relation-group types are particularly problematic. Not only do they lack a corresponding generic concept in the EAST-ADL2 but there are also not really any concrete entities in the ADL that may be a reasonable target for importing RIF relation groups. In addition, they are most often used in a very RIF or target-tool specific manner (for example, relation groups are used to mimic DOORS

link modules in RIF). Therefore, there usually won't be a "perfect match" case with respect to relation-group but instead a "fundamental semantic match" (see below).

Hierarchy-root types can be mapped quite easily to EAST-ADL2. At first sight this concepts seem quite odd and incompatible to the EAST-ADL2. However, a SpecHierarchyRoot, together with its child SpecHierarchy objects, simply defines a containment hierarchy on the SpecObjects (with the specialty that a single SpecObject may be part of several orthogonal containment hierarchies, which is forbidden with the usual interpretation of containment). And containment is nothing else but a special kind of association or, in the terms of RIF, a relation. Consequently, the above remarks on relation types equally apply here. The only exception is that in the case of hierarchy-root types the target will most often be a plain containment association. A difficulty that may arise here is that containment hierarchies are often made up of objects of various types; in that case, the containment relation will need to be mapped to different associations on the EAST-ADL2 side, which must be supported by the mapping definition and the import filter.

Attribute definitions are mapped to attributes of EAST-ADL2 entities.

Note that an AttributeDefinition and its corresponding attribute on the EAST-ADL2 side to which it is mapped need not have containers that correspond (and are mapped) to each other, as illustrated in Figure 7. In other words, the containing SpecType of an AttributeDefinition may be mapped to an other class than the AttributeDefinition's corresponding attributes containing metaclass.

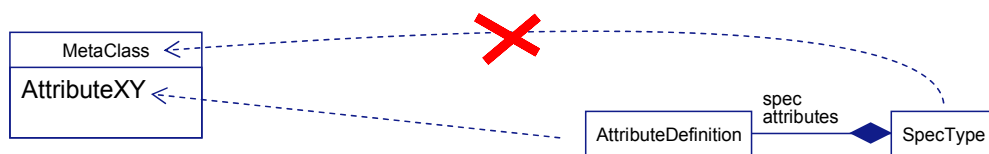


Figure 7. Mismatch between containers of an attribute definition.

This becomes significant when mapping the AttributeDefinitions of relation types to EAST-ADL2. The problem at this point is that in RIF attributes can be defined for all objects of type SpecElementWithUserDefinedAttributes. This becomes problematic if, for example, a relation type is mapped to a plain association and this relation type has AttributeDefinitions, because a plain association cannot have attributes. In this case it is important that the attribute from the RIF side can be mapped to an attribute of some other EAST-ADL2 entity.

So far we assumed that the information on the RIF side directly matches and entity, association or attribute on the EAST-ADL2 side. Let us now examine what happens we are less lucky and one of the problem cases occurs.

Case 2: Missing Attribute

First of all, when mapping an AttributeDefinition from RIF we may run into the problem of not finding an attribute of an EAST-ADL2 entity which constitutes a reasonable match. We thus have some information on the RIF side for which there is no predefined slot in the EAST-ADL2, which may occur especially in the case of project- or company-specific meta-information.

Fortunately, we above already introduced a concept to solve this problem: user attributes. In fact, they were added into the EAST-ADL2 as a global concept, applicable beyond requirements entities, precisely for this reason.

Thus, solving this mismatch is simple. For each attribute definition in RIF that does not have a correspondence in the EAST-ADL2 we introduce an additional UserAttributeDefinition for the EAST-ADL2 entity to which the SpecType containing the AttributeDefinition is mapped. If there is no such entity (e.g. in case of relation types which are mapped to plain associations, see above) then the user attribute is defined for some other, related entity.

Case 3: Missing entity in EAST-ADL2 for a RIF SpecType

When an entire entity is missing on the EAST-ADL2 side instead of only an attribute, then we would need the possibility to add a custom entity to the EAST-ADL2, something like a “user entity”. In principle, this would not be much different from allowing user attributes. However, if such a concept was consequently implemented into the EAST-ADL2, this would lead to several significant downsides:

- It would add a great deal of additional complexity to the language.
- With such a concept, we would in fact mimic or reuse UML2 stereotypes.
- All tools that want to fully support the EAST-ADL2 would have to provide an implementation for this extension mechanism.
- Reusing existing implementations of such an extension mechanism from standard MDA frameworks is not feasible at present (immature, many open issues, ...).
- Offering too much flexibility would conflict the overall objective of EAST-ADL to provide a standardized and unified conception of automotive software development.

Considering these negative effects, it was deemed preferable to not add such an extensive customization mechanism and instead accept this limitation.

This decision was further encouraged by the fact that in the case of missing entities some quite feasible workarounds are available. First, the surplus information on the RIF side can be stored in user attributes attached to some related element. This solution will already serve well in many cases and has the advantage of storing the additional information close to elements to which it is related. Second, if that fails, it is always possible to just do on the EAST-ADL2 side what was actually done in RIF: storing some information which is not a requirement in form of a requirement specification. Precisely speaking, we define a UAElementType for the missing entity and store the surplus information in an instance of the EAST-ADL2 Requirement entity, typed by that UAElementType.

Case 4: Lack of or inappropriate information on RIF side

When importing information from RIF into the EAST-ADL2, it is always possible that some information which is necessary from the EAST-ADL2 perspective is missing or that the information provided is in some way inappropriate when interpreted in terms of the EAST-ADL2.

At first this does not seem to pose a problem. Even without considering imports, there will always be times during intermediate stages of modeling at which at least some of the model’s entities are incompletely defined. However, when taking into account that in the case of an import the source data can be of a more or less orthogonal structure, this may lead to invalid EAST-ADL2 models, particularly due to

- incompleteness
- inconsistencies
- violations of EAST-ADL2 domain model constraints
- ...

There is no technical means to avoid such situations. Instead, a flexible import mechanism must allow the user to manually resolve the emerging violations and conflicts. As a consequence, we can draw the following conclusion: Permitting the import of data that was not originally intended for EAST-ADL2 requires that the import filter as well as the EAST-ADL2 tool used to view and edit the imported data must be able to cope with invalid EAST-ADL2 models, in order to present an invalid import to the user and allow him to manually correct the model.

Case 5: Fundamental semantic mismatch

Beyond the rather manageable mismatches covered by the previous three cases, it is of course possible that the semantic meaning or the structuring of the data on the RIF side is fundamentally irreconcilable with the semantic and/or structure of the related EAST-ADL2 elements.

To illustrate this a bit further, let us briefly consider two examples:

- The RIF file to be imported contains information on the system design which is completely incompatible with FDA entities of EAST-ADL2.
- On the RIF side only a few coarse-grained types were used to characterize the SpecObjects. Then it is likely that we would need to consider related elements or the concrete value of attributes in order to decide on the correct target entity on the EAST-ADL2 side (e.g. the correct EAST-ADL2 type depends on whether a certain SpecObject is contained in a certain SpecGroup or a SpecGroup of a certain SpecType).

Such semantic clashes are a principal problem when relating models of different approaches to each other. It cannot be averted on a conceptual or technical level and trying to do so would unnecessarily and unprofitably complicate the language. But of course it is always possible to revert to the workaround presented for case 4 above to simply import the information as EAST-ADL2 requirements.

5.2.5 Export

The export filter must provide similar means as those just described for the import filter and many considerations from above apply to it correspondingly. Instead of being able to interpret the orthogonally structured information on the RIF side and map it to EAST-ADL2 constructs, the export filter must be able to generate such information structures. For obvious practical reasons the export filter should use the same mapping description as the import filter to minimize the effort for the end-user.

However, an additional issue is to be mentioned here. It is also related to the import filter but its significance becomes more evident when exporting information. The problem comes up when the exported information is intended to be again imported in some other, standard RE tool, such as Telelogic DOORS (Figure 8).

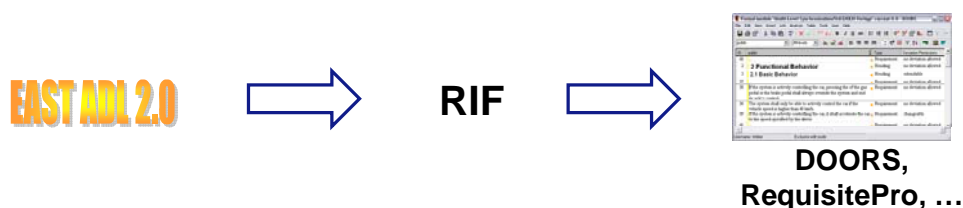


Figure 8. Using standard tools to view and edit EAST-ADL2 information.

Such a practice is encumbered with the following obstacle: it is difficult to achieve a feasible presentation (i.e. suitable for reading and editing) in the target tool, because each target tool's specific interpretation and application of RIF would have to be taken into account to precisely determine the final structure and presentation in the target tool. Unfortunately there are numerous such tool-specific peculiarities (for example, SpecGroups are often used to further characterize SpecObjects in addition to their SpecType).

As a consequence, the export filter would need to take into account which standard RE tool is being targeted for external viewing and editing of the EAST-ADL2 data. However, this is an issue pertinent to all RIF import / export filters and not specific to the EAST-ADL2 case.

5.3 Further Topics

Further topics in connection with the plugin include:

- Implementation Options: When implementing an import/export as described above, various existing methodologies and frameworks may be considered as a starting point, for example:
 - model transformation techniques and frameworks
 - the EMF Mapping library (part of the Eclipse Modeling Framework)
 - ...
- Advanced Capabilities of RIF: The RIF format has a few advanced features which may also be interesting from the EAST-ADL2 point of view. For example, RIF provides support for so called AccessPolicies which allow tagging the data in a RIF file with certain read and write permissions in order to highlight what may be changed by the person or company that the data is sent to. It would be desirable to be able (1) to generate such access permissions during export from EAST-ADL2 to RIF and (2) to import such permissions into the EAST-ADL2 and to store them for later export.

More considerations on this topic can be found in deliverable D2.1.2.

6 Update Suggestions for EAST-ADL2 – Part I: User Attributes

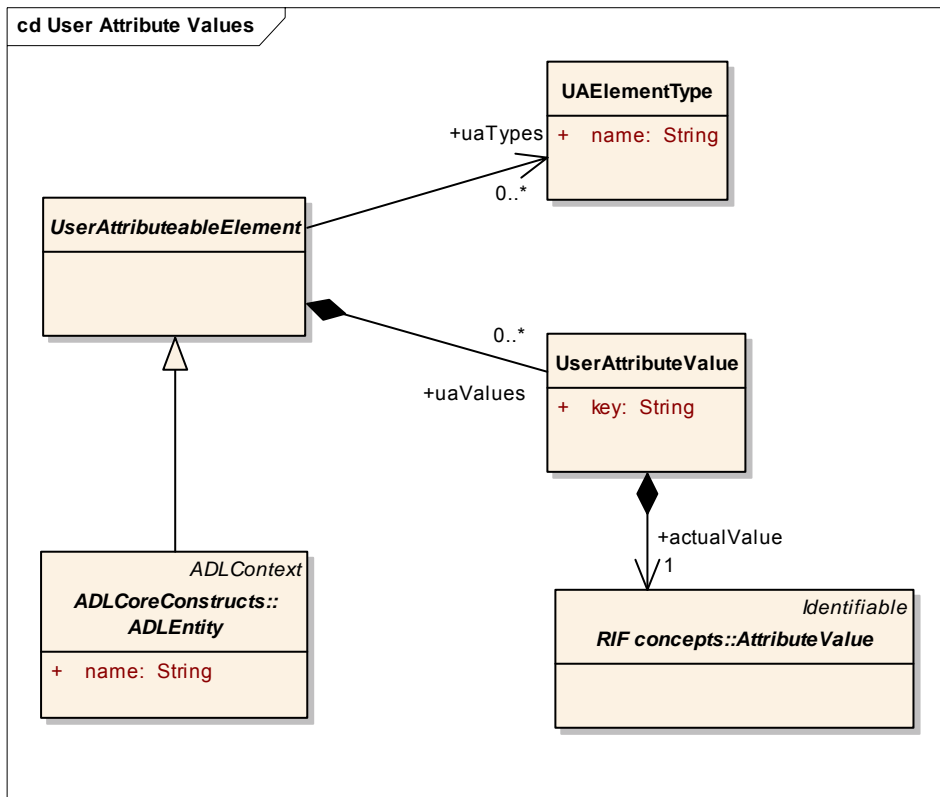


Figure 9 Domain model for user attribute values.

Changes:

New class in EAST-ADL2

Extension:

The ADLEntity stereotype is an abstract stereotype which specializes ADLContext and extends UML2 metaclass NamedElement. The ADLEntity stereotype thus includes the name property from UML2 metaclass:Named Element.

Attributes:

- **name : String**
Name of the ADLEntity.

Associations:

None.

6.2 AttributeValue (from RIF)

An instance of this class represents an individual concrete value of an attribute. As defined in the RIF Specification [2].

6.3 DatatypeDefinition (from RIF)

An instance of this class defines a data type. This can be a simple data type (e.g. integer or float), a complex type (e.g. html) or an enumeration. For details, refer to the RIF Specification document [2].

6.4 UADatatype

A data type intended to type the value of a user attribute. By inheriting from RIF's DatatypeDefinition meta-class, this actually mimics the data type system in RIF.

Notation:

There is no particular notation defined for this entity.

Extension:

DatatypeDefinition (from RIF)

6.5 UAElementType

A certain 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 `UAElementType` called "VWFunction" with a single user attribute definition with key "de.vw-ag.status". That way, all `ADLFunctionTypes` for which "VWFunction" is defined as the `UAElementType` will have such a status attribute.

Notation:

There is no particular notation defined for this entity.

Extension:

Class

Attributes

- **name : String**
The name of this `UAElementType`.

Associations

- **attributes**
The attributes defined for this type. Note that also inherited attribute definitions need to be taken into account.
- **discouragedAttributes**
Marks certain inherited attributes as discouraged, i.e. they should not be used. With the inheritance mechanism defined by the two "extended..." associations it is only possible to add new `UAElementTypes` to a template or add `UserAttributeDefinitions` to a `UAElementType` (by inheriting from them). However, it is sometimes desirable to rule out certain types/attributes. An actual removal of an attribute in an inheriting type would violate the general notion of inheritance. Therefore, it is only possible to mark certain inherited attributes as discouraged. This ensures, for example, that scripts which were programmed for a certain template T will also always work for other templates which are inheriting from T.
- **extendedElementTypes**
The `UAElementTypes` this type is inheriting from. When `UAElementType` ET2 inherits from type ET1, then this means that all attributes defined for ET1 by way of `UserAttributeDefinitions` are available whenever ET2 is specified as the type of a user attributeable element (in addition to those directly defined in ET2). This includes `UserAttributeDefinitions` which ET1 itself may inherit from other types.

6.6 UATemplate

A user attribute template. This is a collection of user attribute definitions which can be reused across projects, departments or companies.

User attribute definitions (see class `UserAttributeDefinition`) are not directly contained in a template but are instead organized in user attributeable element types (`UAElementTypes`). These types gather all attribute definitions which are applicable to a certain type of user attributeable element.

With a simple inheritance mechanism it is possible to define templates which reuse the user attribute definitions of one or more other templates or to introduce modifications to a template which are specific to a certain project or supplier/manufacture relation. For details on this refer to the description of the "extended..." associations in `UATemplate` and `UAElementType`.

Notation:

There is no particular notation defined for this entity.

Extension:

Package

Attributes:

None.

Associations:

- **elementTypes**
The element types organized in this template.
- **discouragedElementTypes**
Marks certain inherited element types as discouraged, i.e. they should not be used. With the inheritance mechanism defined by the two "extended..." associations it is only possible to add new UAElementTypes to a template or add UserAttributeDefinitions to a UAElementType (by inheriting from them). However, it is sometimes desirable to rule out certain types/attributes.
An actual removal of a type in an inheriting template would violate the general notion of inheritance. Therefore, it is only possible to mark certain inherited types as discouraged. This ensures, for example, that scripts which were programmed for a certain template T will also always work for other templates which are inheriting from T.
- **datatypes**
- **extendedTemplates**
The templates this template is inheriting from. When template T2 inherits from template T1, then this means that all UAElementTypes defined for T1 are available whenever T2 is in effect (in addition to those directly defined in T2). This includes UAElementTypes which T1 itself may inherit from other templates.

6.7 UserAttributeableElement

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 UAElementTypes (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 UAElementTypes 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 simplifies implementation a lot.

Notation:

There is no particular notation defined for this entity.

Extension:

Class

Attributes:

None.

Associations:

- **uaTypes**
The UAElementTypes of this user attributeable element. It is possible to provide more than one type. In that case, the UserAttributeDefinitions of all UAElementTypes apply. If there are several attribute definitions with an identical 'key', then the corresponding user attribute will be applied only once.
- **uaValues**
The user attribute values, i.e. key/value pairs, which are attached to this element.

6.8 UserAttributeDefinition

This class defines a user attribute, i.e. it states that all UserAttributeableElements of a certain UAElementType 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".

Notation:

There is no particular notation defined for this entity.

Extension:

Class

Attributes:

- **key : String**
A unique identifier for the user attribute. Whenever interoperability with third parties is required a URL naming scheme should be used, similar as for packages in the Java programming language. For example, a company with a home page URL of "www.example.com" could use the key "com.example.Status" for a status attribute.
- **name : String**
The name of the user attribute. If this is unset, then the key will be used as the user attribute's visual name.
This is similar to 'key' but need not be a unique identifier and will only be used as a visual name of the user attribute for on-screen presentation in a tool, for example.
This should usually be a short form of the 'key'.
- **description : String**
Textual description of this user attribute definition.

Associations:

- **type**
The type that specifies the legal values for this attribute.

6.9 UserAttributeValue

This class represents a specific value for a certain user attribute. User attributes are simple key/value pairs which can be attached to all UserAttributeableElements. User attribute definitions can be used to state what attributes should be attached to elements of certain types and what the allowed values are (see class UserAttributeDefinition).

The actual value is captured in the AttributeValue class which is reused from RIF. For example, in the case of a simple datatype this class will contain a String attribute called 'theValue' which holds the actual value.

Notation:

There is no particular notation defined for this entity.

Extension:

Class

Attributes:

- **key : String**
The unique identifier of the attribute for which this UserAttributeValue provides a value for.

Associations:

- **actualValue**
The actual value. This is represented by a RIF AttributeValue.

7 Update Suggestions for EAST-ADL2 – Part II: Requirements

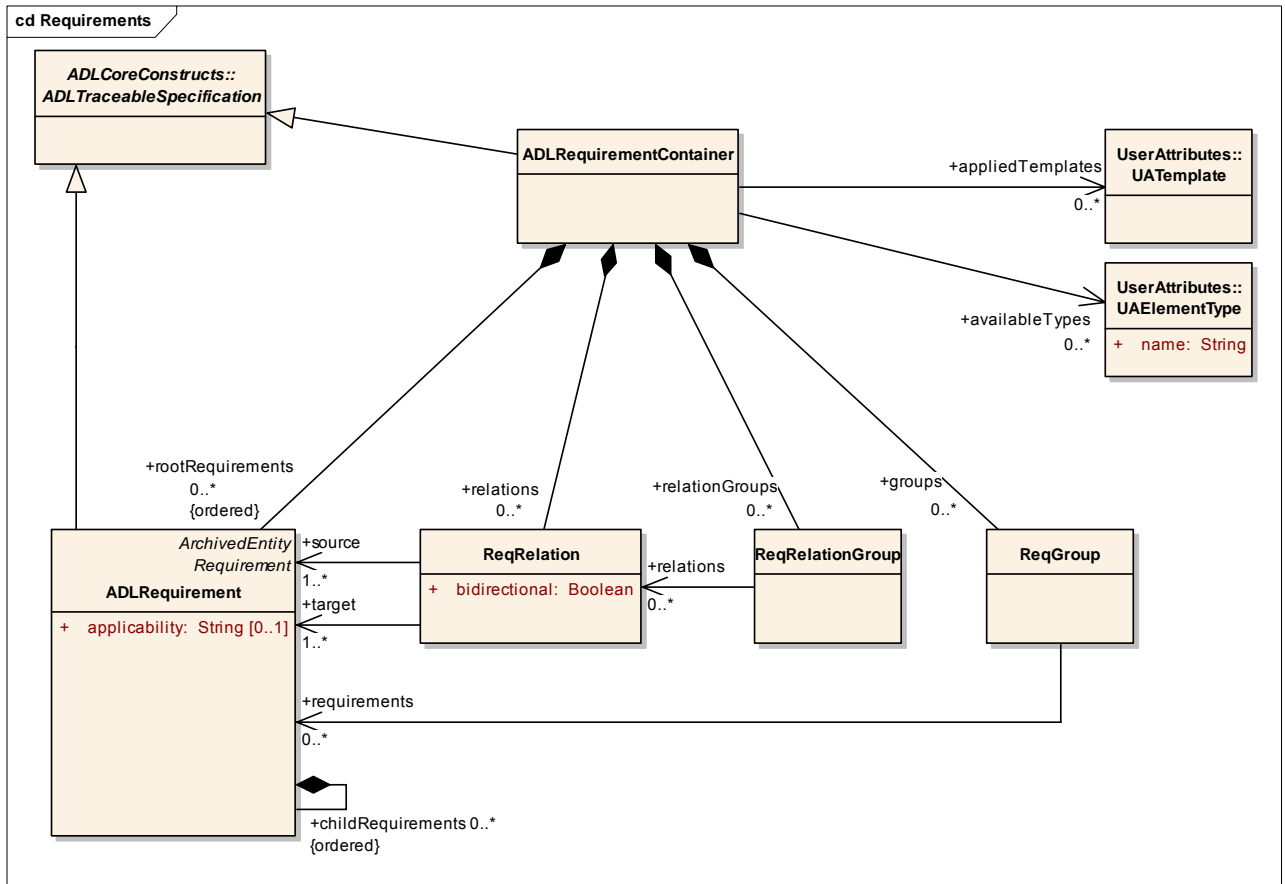


Figure 11 Domain model for generic requirements.

7.1 ADLRequirement

The ADLRequirement will be used to unite the common properties of specific requirement types.

The ArchivedEntity allows any ADLRequirement to be versioned and archived. Each version will carry information like author, creation date etc.

An ADLRequirement may either be directly contained in an ADLContext (by inheriting from ADLTraceableSpecification) or it may be contained in an ADLRequirementContainer, which represents a larger unit or module of specification information. The first possibility is used in trivial cases or often for specialized requirements.

The traceability between ADL requirement entities and other specification or design entities will be ensured by the relationship dependencies described earlier in the ADLCoreConstructs part of this specification.

Semantics:

The ADLRequirement metaclass applies to the ADLEntity that is associated to the ADLRequirement through the ADLSatisfy relation.

The optional Applicability attribute defines the conditions under which the requirement should hold.

Notation:

ADLRequirement is shown as a solid rectangle with Req top right and its name.

Extension:

To specialize SysML::Requirement, which extends Class.

Attributes:

- **applicability : String**
Defines the conditions under which the requirement should hold.

Associations:

- **childRequirements**
Used to organize requirements hierarchically.

7.2 ADLRequirementContainer

The ADLRequirementContainer metaclass is used to aggregate several ADLRequirements which are semantically related to each other. It can be seen as a larger unit or module of specification information.

By inheriting from ADLTraceableSpecification, an ADLRequirementContainer may itself be contained in an ADLContext.

In addition to aggregating related ADLRequirements, the ADLRequirementContainer allows to define relations between its contained requirements and also a grouping of them.

Furthermore, the ADLRequirementContainer allows to introduce additional user attribute definitions by way of UAElementTypes or UATemplates which are valid only locally inside this ADLRequirementContainer. These are additional in that they are used in addition to the user attribute definitions which are provided globally for the entire EAST-ADL2 repository.

Notation:

ADLRequirementContainer is shown as a solid-outline rectangle containing the name. Contained entities may also be shown inside (White-box view)

Extension:

Package

Attributes:

None.

Associations:

- **appliedTemplates**
The UATemplates that provide the UAElementTypes available in the requirement specification represented by this ADLRequirementContainer. In addition, the types from association 'availableTypes' will be available in this specification.

- **availableTemplates**

The UAElementTypes that are available in the requirement specification represented by this ADLRequirementContainer. In addition, the types from the templates from association 'appliedTemplates' will be available in this specification.

- **rootRequirements**

The root requirements of this ADLRequirementContainer.

Note that unlike the other associations such as 'relations' or 'groups' this association does not link to all ADLRequirements of the ADLRequirementContainer but only to the root requirements. Other, lower-level requirements are hierarchically contained in their parent requirements (cf. association childRequirements of ADLRequirement).

- **relations**

- **relationGroups**

- **groups**

7.3 ADLTraceableSpecification

ADLTraceableSpecification is an abstract metaclass which defines some kind of traceable specification. Specializations of traceable specifications are for example ADL requirements, ADL requirement containers or ADLUseCase.

Semantics:

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

Notation:

There is no general notation for ADLTraceableSpecification. Further subclasses of ADLTraceableSpecification will define their own notation.

Changes:

New class in EAST-ADL2

Extension:

The ADLTraceableSpecification is a specification stereotype which extends UML2 metaclass Element.

7.4 ReqGroup

A group of requirements.

This can be used, for example, to define that all requirements in the group share a certain quality.

Notation:

There is no particular notation defined for this entity.

Extension:

Class

Associations:

- **requirements**
The set of requirements grouped by this ReqGroup.

7.5 ReqRelation

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

The standard case will be a relation with one source and one target requirement. However, it is possible to have several source and/or several target requirements so that general n:m relations can be expressed with instances of this class.

The semantic of a concrete requirement relation is not defined by the EAST-ADL2 and therefore needs to be provided by the modeler. In particular, three ways are conceivable:

- (1) The user attributes of the relation can be used to specify its meaning, for example with a user attribute called "relationType" which is set to values such as "needs" or "excludes".
- (2) The uaType (user attributeable element type) can be used. Certain types will be used for certain relation semantics.
- (3) ReqRelationGroups can be used, i.e. all relations with an "excludes" meaning are put in one relation group and all with a "needs" meaning are put in another.

Notation:

There is no particular notation defined for this entity.

Extension:

Class

Attributes:

- **bidirectional : Boolean**
When set to true, the semantic relation represented by this instance of ReqRelation 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 ReqGroup.

Associations:

- **source**
The ADLRequirement at this relation's source end.
- **target**
The ADLRequirement at this relation's target end.

7.6 ReqRelationGroup

A group of relations between requirements.

This can be used, for example, to define that all relations in the group share a certain quality.

Notation:

There is no particular notation defined for this entity.

Extension:

Class

Attributes:

None.

Associations:

- **relations**
The relations grouped by this ReqRelationGroup.

7.7 UAElementType

See Chapter 6.5.

7.8 UATemplate

See Chapter 6.6.

8 Update Suggestions for EAST-ADL2 – Part III: Verification & Validation

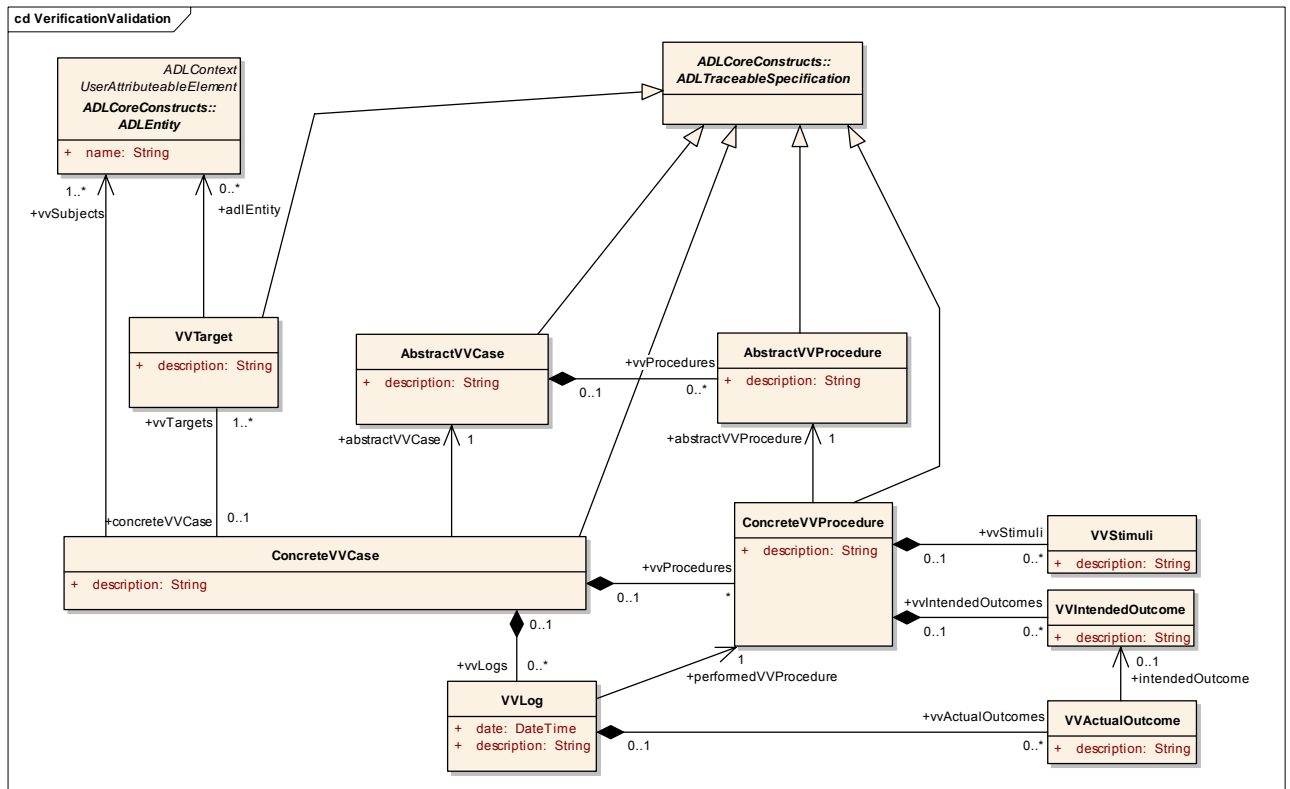


Figure 12 Domain model for verification and validation.

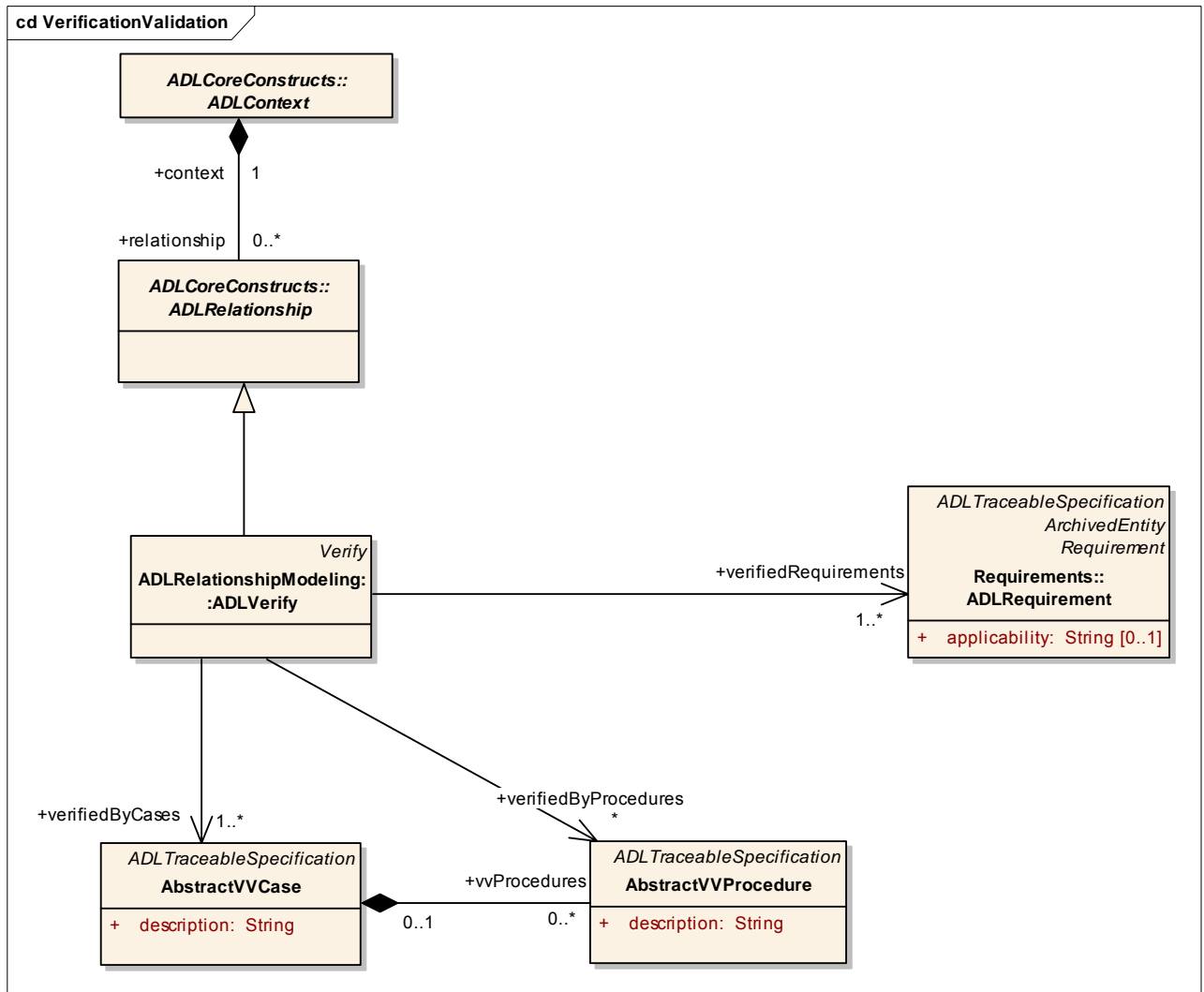


Figure 13 Relating requirements to verification & validation entities.

8.1 AbstractVVCASE

A VVCASE is the core concept of verification and validation support in EAST-ADL2 and represents a certain verification and validation effort of varying scope and intention. The definition of VVCASEs is separated in two levels: an abstract and a concrete level, represented by the entities AbstractVVCASE and ConcreteVVCASE.

An AbstractVVCASE describes a verification and validation effort on an abstract level, i.e. without referring to a concrete testing environment and without specifying stimuli and the expected outcome on a concrete technical level.

Slightly simplified one could say: An AbstractVVCASE describes "what" needs to be done in the context of a certain verification and validation effort, but not precisely "how" this is done. Together, all AbstractVVCASEs for a certain system form sort of a "to-do"-list, which describes what needs to be done when actually testing the system with a concrete testing environment, but in a form applicable to all conceivable testing environments.

Notation:

AbstractVVCASE is shown as a solid rectangle with "AbstractVVCASE" top right and its name.

Extension:

Class

Attributes:

- **description : String**
A textual description in natural language.

Associations:

- **vvProcedures**
The abstract VV procedures for this AbstractVVCASE.

8.2 AbstractVVProcedure

A VVProcedure represents an individual task in the context of an overall verification and validation effort (represented by a VVCASE), which has to be performed in order to achieve that effort's overall objective. Just as is the case for VVCASEs, the definition of VVProcedures is separated in two levels: an abstract and a concrete level, represented by the entities AbstractVVProcedure and ConcreteVVProcedure.

An AbstractVVProcedure defines such a task on an abstract level, i.e. without referring to a concrete testing environment and without specifying stimuli and an expected outcome on a concrete technical level.

Notation:

AbstractVVProcedure is shown as a solid rectangle with "AbstractVVProcedure" top right and its name.

Extension:

Class

Attributes:

- **description : String**
A textual description in natural language.

Associations:

None.

8.3 ADLContext

ADLContext is an abstract metaclass with no superclass. It is used as the common superclass for all structural metaclasses in the EAST-ADL2 domain model. It is defined to achieve a simple and practical way to associate traceable specifications and relationships to a specific context.

Semantics:

Subclasses of ADLContext provide semantics appropriate to the concepts they represent. The traceable specifications and/or relationships add no semantics, but may represent information useful for the reader of the model or for tools extracting information from the model.

Notation:

There is no general notation for an ADLContext. Further subclasses of ADLContext will define their own notation.

Changes:

New class in EAST-ADL2

Extension:

The ADLContext stereotype is an abstract stereotype which extends UML2 metaclass Element.

Attributes:

None.

Associations:

- relationship
- traceableSpecification

8.4 ADLEntity

See Chapter 6.1.

8.5 ADLRelationship

ADLRelationship is an abstract metaclass which defines a relationship, which references elements.

Semantics:

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

Notation:

There is no general notation for ADLRelationship. The various subclasses of ADLRelationship will define their own notation.

Changes:

New class in EAST-ADL2

Extension:

The ADLRelationship stereotype is a relationship stereotype that specializes UML2 stereotype Relationship, which extends UML2 metaclass Dependency

Attributes:

None.

Associations:

None.

8.6 ADLRequirement

See Chapter 7.1.

8.7 ADLTraceableSpecification

See Chapter 7.3.

8.8 ADLVerify

ADLVerify is a relationship metaclass, which signifies a dependency relationship in-between an ADL requirement and a VV Case, showing the relationship when a client VV Case verifies the supplier ADL requirement.

Semantics:

ADLVerify metaclass signifies a refined requirement/verified by relationship between an ADLRequirement and a VV case, where the modification of the supplierADLRequirement may impact the verifying client VV case. The ADLVerify metaclass implies the semantics that the verifying client VV case is not complete, without the supplier ADLRequirement.

Notation:

An ADLVerify relationship is shown as a dashed arrow between the ADLRequirements and AbstractVVCASE. The AbstractVVCASE at the tail of the arrow (the verifying AbstractVVCASE) depends on the ADLRequirement at the arrowhead (the verified ADLRequirement)

Extension:

To specializes SysML::Verify, which specializes the UML stereotype Trace, which extends Dependency.

Attributes:

None.

Associations:

- **verifiedRequirements**
One or more requirements which are verified by the related VV cases and procedures.
- **verifiedByCases**
One or more abstract VV cases that verify the requirement(s) denoted by association 'verifiedRequirements'.
- **verifiedByProcedures**
Optionally certain abstract VV procedures can be specified which actually verify the requirement(s) denoted by association 'verifiedRequirements'. These VV procedures must be contained in the abstract VV case(es) specified by association 'verifiedByCases'.

8.9 ConcreteVVCASE

A VVCASE is the core concept of verification and validation support in EAST-ADL2 and represents a certain verification and validation effort of varying scope and intention. The definition of VVCASEs is separated in two levels: an abstract and a concrete level, represented by the entities AbstractVVCASE and ConcreteVVCASE.

A ConcreteVVCASE describes a verification and validation effort on a concrete level, i.e. it specifies concrete test subjects and targets and provides stimuli and the expected outcome on a concrete technical level.

Slightly simplified one could say: A ConcreteVVCASE not only describes "what" needs to be done in the context of a certain verification and validation effort, but also the necessary details of "how" this is done.

Notation:

ConcreteVVCASE is shown as a solid rectangle with "ConcreteVVCASE" top right and its name.

Extension:

Class.

Attributes:

- **description : String**
A textual description in natural language.

Associations:

- **abstractVVCASE**
The abstract VV case that this ConcreteVVCASE provides a concrete interpretation of.

- **vvProcedures**
The concrete VV procedures for this ConcreteVVCASE.
- **vvTargets**
The VVTargets for this ConcreteVVCASE. See association 'vvSubjects' for more information.
- **vvSubjects**
The ADLEntities that are being verified and validated by this ConcreteVVCASE. Usually this will be a subset of those ADLEntities which are realized by the VVTarget(s) of the ConcreteVVCASE; but this need not always be the case. The difference between these two sets of ADLEntities, i.e. 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) if 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 of these ConcreteVVCASEs.
- **vvLogs**
The VVLogs stored for this ConcreteVVCASE.

8.10 ConcreteVVProcedure

A VVProcedure represents an individual task in the context of an overall verification and validation effort (represented by a VVCASE), which has to be performed in order to achieve that effort's overall objective. Just as is the case for VVCASEs, the definition of VVProcedures is separated in two levels: an abstract and a concrete level, represented by the entities AbstractVVProcedure and ConcreteVVProcedure.

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

Notation:

ConcreteVVProcedure is shown as a solid rectangle with "ConcreteVVProcedure" top right and its name.

Extension:

Class

Attributes:

- **description : String**
A textual description in natural language.

Associations:

- **abstractVVProcedure**
The abstract VV procedure that this ConcreteVVProcedure provides a concrete interpretation of.

- **vvStimuli**
- **vvIntendedOutcomes**

8.11 VVActualOutcome

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

Notation:

VVActualOutcome is shown as a solid rectangle with "VVActualOutcome" top right and its name.

Extension:

Class

Attributes:

- **description : String**
A textual description in natural language.

Associations:

- **intendedOutcome**
Denotes the VVIntendedOutcome that must be matched by this actual outcome.

8.12 VVIntendedOutcome

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

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

Notation:

VVIntendedOutcome is shown as a solid rectangle with "VVIntendedOutcome" top right and its name.

Extension:

Class

Attributes:

- **description : String**
A textual description in natural language.

Associations:

None.

8.13 VVLog

A ConcreteVVCASE precisely describes a verification and validation 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 represents a certain execution of a ConcreteVVCASE.

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

Notation:

VVLog is shown as a solid rectangle with "VVLog" top right and its name.

Extension:

Class

Attributes:

- **date : DateTime**
- **description : String**
A textual description in natural language.

Associations:

- performedVVProcedure
- vvActualOutcome

8.14 VVStimulus

Input values to the testing environment represented by VVTarget in order to perform the corresponding VVProcedure.

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

Notation:

VVStimulus is shown as a solid rectangle with "VVStimulus" top right and its name.

Extension:

Class

Attributes:

- **description : String**
A textual description in natural language.

Associations:

None.

8.15 VVTarget

A concrete testing environment in which or on which a particular verification and validation 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 ADLEntities. 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 ADLEntity has a minimum cardinality of 0.

VVTargets can be reused across several ConcreteVVCases.

Notation:

VVTarget is shown as a solid rectangle with "VVTarget" top right and its name.

Extension:

Class

Attributes:

- **description : String**
A textual description in natural language.

Associations:

- **adlEntity**
- **concreteVVCASE**
The VVTargets for this ConcreteVVCASE. See association 'vvSubjects' for more information.

9 Contribution to overall ATESSST objectives

User attributes are used by the EAST-ADL2 Requirement metaclass to provide the requirement with all its necessary information; they therefore constitute an integral part of the basic structures for requirements modeling. But in addition they are applicable beyond requirements modeling to all entities of the EAST-ADL2 to attach customized, project-specific attributes with meta-information to them. Their role for EAST-ADL2 is thus twofold: they are (1) an important part of the requirements support of the language and (2) they provide a means to customize the language to specific company and project needs.

Requirements engineering and management has been identified as a vital activity in software development for more than a decade now. It not only serves to clearly define what is expected from the final system but it also provides techniques to identify these needs in the first place, usually referred to as requirement elicitation. The requirement support in EAST-ADL2 has two levels: (1) generic requirement modeling entities and (2) a set of specialized requirements in order to capture certain aspects of the system in a standardized form, for example timing requirements. Comprehensive support for requirements modeling is key to delivering a high-quality system.

Verification and validation activities are critical elements of system development. The information they rely on, and the information that is produced are well-defined among the system development artifacts. It is also important to keep track of the exact verification setting that is used. Test and verification artifacts should therefore be part of the ADL and are an important contribution to the correctness of the developed system.

10 Conclusions

With the updates described above, EAST-ADL2 will provide solid support for requirements management, verification and validation activities and tracking of relations between these two aspects of system development and the remainder of the system description as captured, for example, in the FDA entities.

11 References

- [1] Herstellerinitiative Software, <http://www.automotive-his.de>
- [2] Requirements Interchange Format – Specification, available at: <http://www.automotive-his.de>