Grant Agreement 260057

Model-based Analysis & Engineering of Novel Architectures for Dependable Electric Vehicles

Report type	**Deliverable D3.1.1**
Report name	**Language Concepts Supporting Engineering Scenarios**

Dissemination level	**PU**
Status	**Intermediate**
Version number	**1.1.0**
Date of preparation	**2011-12-20**

Authors

Editor	**E-mail**
Mark-Oliver Reiser	moreiser@cs.tu-berlin.de

Authors	**E-mail**
Anders Sandberg	anders.sandberg@mecel.se
David Parker	D.J.Parker@hull.ac.uk
DeJiu Chen	chen@md.kth.se
Fulvio Tagliabò	fulvio.tagliabo@crf.it
Henrik Lönn	Henrik.Lonn@volvo.com
Juha-Pekka Tolvanen	jpt@metacase.com
Peter Lindqvist	peter.lindqvist@systemite.se
Renato Librino	renato.librino@4sgroup.it
Sandra Torchiaro	sandra.torchiaro@crf.it

The Consortium

Revision chart and history log

Version	Date	Reason
0.1	2010-11-26	Initial outline.
		Various contributions by authors of individual chapters.
0.9	2011-02-07	Integration of author contributions (i.e. accepted all changes in Word's "track changes" mode). Some layout fixes.
	mid Feb 11	Review.
1.0	2011-03-02	Intermediate (Final version for 1st delivery at MS3)
1.0.1	2011-08-30	Intermediate (Work-in-progress version at end of Y1) Next delivery due at Dec 1, 2011.
1.1.0	2011-12-20	Intermediate (Final version for 2nd delivery at MS4.5)

Table of contents

1 Introduction

Welcome to MAENAD deliverable D3.1.1! In its final version, later in the project at month 24, this deliverable will contain detailed descriptions of all new language elements and concepts that will have been added to EAST-ADL during MAENAD in order to support ISO 26262, optimization, analysis, and the other MAENAD objectives.

For this intermediate, work-in-progress delivery after Y1 at the end of November 2011, D3.1.1 contains descriptions of what new concepts and modifications of existing concepts *are required and expected* by project partners together with *a proposal* of how to realize these concepts in the language. All concepts presented here are work in progress and have, in some cases, not been discussed with and accepted by all project partners.

Also, the aim is to give a brief summary and overview of the current support of ISO 26262, optimization, analysis, etc. (depending on the chapter) that is already present in EAST-ADL.

The purpose of the document at this stage is to document the ongoing conceptual work and help project partners working on a particular topic in planning their future work in MAENAD and help others in the project to catch up with the current status of the topic and join discussions.

Structure

The document is structured based on the cross-work-package work groups identified in MAENAD. Each of these work groups is focused on a particular project objective – as defined in the MAENAD Description of Work – plus an additional work group on language consolidation, which deals with an overall refinement of all parts the language (consistency, etc.). Each cross-work-package group has its own chapter. Consequently, you could say there is a chapter for each MAENAD objective plus one on consolidation. In addition, Chapters 7 and 8 go into detail on selected topics; these might be integrated into the other chapters in the final release of this deliverable.

Scope

This deliverable differs from deliverable D4.1.1 (the EAST-ADL language specification), which is also focused on language concepts, in that we here provide more background, more motivation and document discussions that were taking place while working out the concepts. Also, some alternatives might be documented that did not make their way into the final language for some reason. In contrast, D4.1.1 will only document the final outcome of the work on the language concepts.

And now, enjoy reading D3.1.1 !

The MAENAD Consortium

2 Modeling Concepts for Supporting ISO 26262

This section discusses language refinements related to MAENAD Objective 1, "Develop capabilities for modeling and analysis support, following ISO 26262". In this section we will describe which ISO 26262 concepts are supported by EAST-ADL at the moment. So we will discuss on how the current support has to be improved and which are the ISO 26262 concepts not yet covered by EAST-ADL.

The ISO/FDIS 26262 requires that the application of the "functional safety approach", starts from the preliminary vehicle development phases and continuing along the complete life-cycle of the product. This approach ensures the design of a safe automotive system. Furthermore it provides an automotive specific risk-based approach for determining the risk classes, called ASILs (Automotive Safety Integrity Levels). The new standard uses the ASILs for specifying necessary safety requirements on each corresponding item for achieving an acceptable residual risk. ISO 26262 also provides requirements for validation and confirmation measures to ensure a sufficient and acceptable level of safety being achieved.

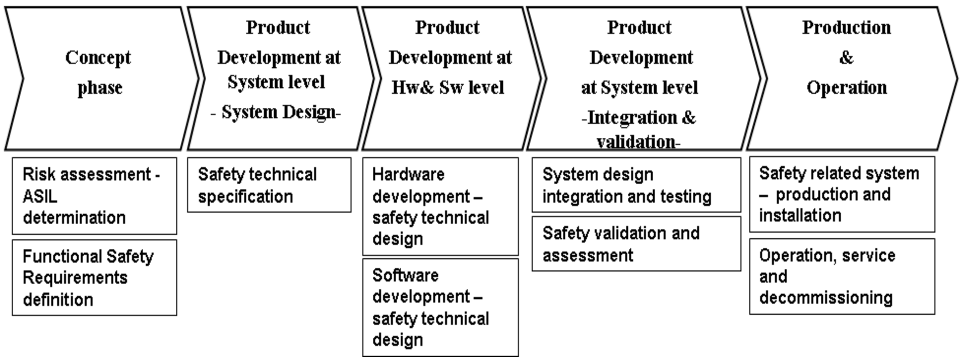

Concept phase
Product Development at System level - System Design-
Product Development at Hw& Sw level
Product Development at System level -Integration & validation-
Production & Operation
Risk assessment - ASIL determination
Functional Safety Requirements definition
Safety technical specification
Hardware development – safety technical design
Software development – safety technical design
System design integration and testing
Safety validation and assessment
Safety related system – production and installation
Operation, service and decommissioning

Figure 1. ISO 26262 Safety life-cycle

The ISO 26262 safety life-cycle includes the following phases:

- Concept phase, (Part 3)
- System level development – specification, (Part 4)
- Hardware level development, (Part 5)
- Software level development, (Part 6)
- System level development – integration and validation (Part 4)

The EAST-ADL supports several of the safety life-cycle phases defined in ISO 26262. EAST-ADL provides support for the safety design flow and related safety design concepts such as item, hazard, and safety concept according to ISO 26262.

This information corresponds to the Dependability extension in EAST-ADL. Following a top-down approach, the safety analysis can start at the Vehicle level, beginning from the item's "target feature" definition (the feature description in terms of the vehicle's output(s) behaviour), and the feature flaws definition, as anomalies of the item's outputs on Vehicle Level. Therefore, on Vehicle level, it is already possible to perform a Hazard analysis and Risk assessment to preliminarily evaluate the "safety relevance" of the Item under safety analysis. For this purpose, the hazards should be evaluated in different scenarios for assessing Severity, Controllability and Exposure. The hazard under analysis, when applied to the various operational situations (operative &

environmental conditions), results in the so called “hazardous events”, as you can see in the diagram in Figure 1.

Each Hazardous Event has to be classified in terms of associated risk defined by its Automotive Safety Integrity Level (ASIL). The ASIL level is captured as an attribute in the Safety Goal element and the safety goal itself is defined by a referenced requirement. Another referenced requirement defines the Safe State.

To verify the correctness and completeness of the preliminary Hazard analysis and risk assessment performed on VehicleLevel, a complementary analysis can be performed by looking at the architectural level. Therefore the target function (the function description in terms of its output(s) behaviour) on AnalysisLevel should be defined by deriving it from the target feature introduced at the upper abstraction level. At this point it is possible to define the malfunction as anomalies of the item's outputs. These anomalies may be foreseen by the engineer or found by analyses such as Failure Modes and Effects Analysis of the architectural solution.

This serves as a more concrete basis for hazard identification and risk assessment, and therefore offers an opportunity for validation. Note that this process may be iterative and parallel: hazards and risks may be identified and assessed at any abstraction level, but the information is solution independent and Hazards, the Safety Goals and the Safe States are managed as Vehicle level information.

The top-down approach described is intended to be applicable whether or not the item/function is a new development. In the case of a modification of an already existing item, an impact analysis is required and a tailored safety lifecycle is advisable. Therefore, with the hypothesis that the safety analysis on VehicleLevel is already available (inherited from original item), the most convenient approach is the bottom-up one, i.e. by entering directly on the AnalysisLevel and by verifying the impact in terms of differences in hazard list and risk assessment outcomes. The vehicle level compared to the analysis level abstracts away all implementation details of a function. This means that even if you have a rough architecture on analysis level to start with, it is easy to present this on VehicleLevel where you express the Item. For each safety goal resulting from the preliminary hazard analysis, at least one, but also 2 or more, functional safety requirement must/can be specified. The definition of functional safety requirements is appropriate at the EAST-ADL AnalysisLevel. Note that what is expressed in the ISO 26262 standard as “preliminary architectural assumptions” is exactly the purpose of analysis architecture in the EAST-ADL language. At this level, the goal is to verify that the functional safety concept realizes all safety goals defined at VehicleLevel. Note that more than one safety requirement could be associated with the same functional safety requirement.

Once the functional safety concept is specified, the item can be developed with a system perspective that includes detailed functional solutions and hardware platform on the EAST-ADL Design Level. This corresponds to the “system design specification” according to ISO 26262. The functional safety requirements are refined to technical safety requirements allocated to the architectural elements on the Design Level.

2.1 ASILs (Automotive Safety Integrity Levels)

Safety Integrity Levels (SILs) are abstract classification levels that can be used to indicate the level of safety required of safety-critical systems (or elements thereof). SILs have been adopted as part of safety standards such as IEC 61508 and - in the automotive domain - ISO/FDIS 26262. In the context of the upcoming ISO 26262, SILs are known as ASILs - Automotive Safety Integrity Levels - and form a major part of the standard: ASILs are used to specify the necessary safety requirements for achieving an acceptable residual risk, as well as providing requirements for validation and confirmation to ensure the required levels of safety are being achieved.

Safety requirements in these standards are intended to ensure the system being designed is free from unacceptable risk (assuming the requirements are met) and are derived through a process of analysis and risk assessment. The aim of the process is to determine the critical system functions - those which have the potential to be hazardous in the instance of failure - and the requirements necessary to mitigate the effects or reduce the likelihood of those hazards. These safety requirements are often associated with integrity requirements that apply to those critical functions to indicate, in essence, what level of contribution they have towards the overall system safety and thus what level of safety they should implement to avoid system failures. A low ASIL therefore indicates that the element is not a major contributor to severe system failures, while a high ASIL indicates that it is potentially is a major contributor, and this allows a means of verifying that system safety requirements are being achieved by ensuring that the ASILs allocated to system elements are also being met.

Therefore ASILs play a dual role in the development of safety-critical systems: they allow for top-down allocation of safety requirements to different elements of the system according to their contribution to risk, and they allow for bottom-up verification to show that the safety requirements are being met by the developed system.

ASILs are divided into one of four classes, see table below. These four classes specify the item's necessary safety requirements for achieving an acceptable residual risk, with D representing the highest and A the lowest class. QM (Quality Management) can be applied to non-safety critical elements to indicate that there are no specific safety requirements in place. The ASIL-Level shall be determined for each hazardous event using the estimation parameters severity (S), probability of exposure (E) and controllability (C)

		C1	**C2**	**C3**
S1	**E1**	QM	QM	QM
	E2	QM	QM	QM
	E3	QM	QM	A
	E4	QM	A	B
S2	**E1**	QM	QM	QM
	E2	QM	QM	A
	E3	QM	A	B
	E4	A	B	C
S3	**E1**	QM	QM	A
	E2	QM	A	B
	E3	A	B	C
	E4	B	C	D

Table 1 - Determining ASILs

ASIL = QM (Quality Management) → the function has no impact on safety - it is not necessary to define any safety requirement

ASIL = A → the function has a minimal impact on safety

ASIL = B → the function has an impact on safety – considerable damage

ASIL = C → the function has impact on safety – relatively high damage associated with medium-high probability of being in a situation of risk

ASIL = D → the function is critical - very significant damage associated with a high probability of being in a situation of risk

Top-level safety requirements

During the concept phase a *safety goal* shall be defined for each hazardous event. This is a fundamental task, since the safety goal is the top level safety requirement, and it will be the base on which the functional and technical safety requirements are defined. The safety goal leads to item characteristics needed to avert the hazard or to reduce risk associated with the hazard to an acceptable level. Each safety goal is assigned an ASIL value to indicate the required integrity level according to which the goal shall be fulfilled. For every safety goal a *Safe state*, if applicable, shall be identified in order to declare a system state to be maintained or to be reached when the failure is detected, so to allow a failure mitigation action without any violation of the associated safety goal. For each safety goal and safe state (if applicable) that are the results of the risk assessment, at least one safety requirement shall be specified.

2.1.1 Language Support for ASILs

ASIL decomposition and allocation is an important objective for MAENAD and a major requirement in order to be able to fully support ISO 26262-compatible safety-driven design. D3.2.1 details a newly developed algorithm that enables the automatic decomposition and allocation of ASILs across independent elements of the system by building upon earlier work on FTA; ASILs assigned to hazards can then be decomposed to the minimal cut sets that cause those hazards. By enumerating the different permutations of those ASILs assigned to multi-event cut sets in a recursive process, it is possible to determine all possible valid ASIL allocations for the basic events of a system while ensuring that the resulting allocations are still capable of meeting the original safety requirements. EAST-ADL language support is relatively mature and language elements for both hazard analysis and ASILs are present. However, it may be that these need tweaking or extending to streamline the process in response to practical experience gained during the project.

At present, much of the infrastructure required to support ASIL decomposition and analysis is already present in both EAST-ADL and HiP-HOPS. In particular, EAST-ADL supports:

- Hazard analysis and definition of Hazards, HazardousEvents, and SafetyGoals. HazardousEvents and SafetyGoals can both store ASIL values.
- SafetyConstraints can be used to assign ASILs to elements of the error model.
- SafetyConstraints may also provide a mechanism for linking the resulting ASIL allocations back to the faults & failures of the error model after decomposition.

EAST-ADL therefore has sufficient language support to enable ASIL decomposition and the next step is to ensure nothing is missing by developing the algorithms and performing tests on case studies. The main obstacle at present is the link between the two: namely, the Papyrus plugin, which needs extending to enable the output of hazards and ASIL information to HiP-HOPS and allow the starting of the decomposition process. This will enable us to begin testing the decomposition of actual EAST-ADL models with the algorithm, which will highlight bugs to be fixed and other areas for further work (e.g. any streamlining or clarification of the existing language elements, any additions necessary to the language, what sort of bugs exist in the tools, and how efficient and scalable the algorithm is).

A secondary issue is that of storing the results back in the model. This is a general unsolved issue not specific to ASILs (e.g. many other analysis results are currently completely external as well) but it is something to be investigated further.

2.2 Modeling concepts for ISO26262 - gaps analysis

In the following table a description of what language concepts (i.e. modeling elements, attributes, associations, etc.) are already present in EAST-ADL and what are required to cover ISO 26262 has been provided.

ISO26262 ref.	Requirement of the standard	Requirement to system description and modeling already covered in EAST-ADL	Requirement to system description and modeling to be added in EAST-ADL
Part 3 - Clause 5	**Item definition**		
Part 3 -Clause 5.4.1	Description of the item's purpose and functionality, including operating modes and states	Item references Features Realizing Artifacts and define SystemBoundaries. Features describe purposes and functionality, including operating modes and states on user level	No additional requirements
Part 3 -Clause 5.4.1	Description of the interactions with other items or elements	Features (its use cases, requirements and refined requirements) describe interactions with other items or elements on user level. Realizing Artifacts describe interactions with other items or elements of solution	No additional requirements
Part 3 -Clause 5.4.1	Applicable laws and regulations, national and international standards	Features (its requirements) define Applicable laws and regulations, both national and international standards.	No additional requirements
Part 3 -Clause 5.4.1	The operating scenarios which impact the functionality of the item. Expected or required environmental conditions	OperatingScenario on Item describes operating scenarios which impact the functionality of the item. Requirements on Features define expected or required environmental conditions that	No additional requirements

ISO26262 ref.	Requirement of the standard	Requirement to system description and modeling already covered in EAST-ADL	Requirement to system description and modeling to be added in EAST-ADL
		are independent of solution, Requirements on Artifacts define expected or required environmental conditions that are dependent of solution	
	Known failures and hazards	ErrorModels linked to artifacts identify known failures Hazards identify known Hazards	No additional requirements (?)
Part 3 -Clause 5.4.1	Behavior achieved by similar functions, items or elements, if any. Pre-trials information	Behaviour achieved by similar functions, items or elements, if any are defined on the respective element. (Identification of "similar functions, items or elements" TBD)	To be further investigated
Part 3 – Clause 6.4.2.1	Impact Analysis: - Definition and description of the intended modifications, in terms of design modifications and/or implementation modifications - Identification of the areas affected by the intended modifications - Implications of the intended modifications with regard to functional safety - Identification and description of the affected work products	Not covered	It shall be possible to categorize the elements (components, interfaces) included in the item as: - new development; - already existing with modification; Already existing without modification. To add the “proven in use” concept
Part 3 – Clause 7	**Hazard Analysis and Risk Assessment**		
Part 3 – Clause 7.4.2.1.1	The operational situations and operating modes in which an item's malfunctioning behaviour will result in a hazardous event shall be described, both for cases when the vehicle is correctly used and when it is incorrectly used in a foreseeable way.	Operating Mode; Operational Situation – traffic, environment; Operational Situation – Use Case.	No additional requirements

ISO26262 ref.	Requirement of the standard	Requirement to system description and modeling already covered in EAST-ADL	Requirement to system description and modeling to be added in EAST-ADL
Part 3 – Clause 7.4.2.2.1	- The hazards shall be determined systematically by using adequate techniques; - Hazards shall be defined in terms of the conditions or behavior that can be observed at the vehicle level.	FeatureFlaw, Hazard	No additional requirements
Part 3 – Clause 7.4.2.2.3	The hazardous events shall be determined for relevant combinations of operational situations and hazards.	HazardousEvent metaclass	No additional requirements
Part 3 – Clause 7.4.3	- All hazardous events identified shall be classified, except those that are outside the scope of ISO 26262; - The severity of potential harm shall be estimated based on a defined rationale for each hazardous event; - The probability of exposure of each operational situation shall be estimated based on a defined rationale for each hazardous event; - The controllability of each hazardous event, by the driver or other traffic participants, shall be estimated based on a defined rationale for each hazardous event	SeverityClassKind, ControllabilityClassKind, ExposureClassKind (Enumeration Metaclass)	To check the possibility to add analysis information related to the estimation of these parameters
Part 3 – Clause 7.4.4.1	An ASIL shall be determined for each hazardous event using the parameters "severity","probability of exposure" and "controllability"	ASILClassKind	To check the possibility to add analysis information related to estimation of the level of risk
Part 3 – Clause 7.4.4.3	Safety goal shall be determined for each hazardous event. Possible grouping of safety goals	SafetyGoal (EAElement)	No additional requirements
Part 3 – Clause 7.4.4.5	Safe state shall be defined, if possible	For every Safety Goal, a safe state should be defined as attribute of SafetyGoal (safeStates : String [0..1])	No additional requirements
Part 3 – Clause 8	**Functional Safety Concept**		
Part 3 – Clause 8.4.2.3, 8.4.2.4, 8.4.2.5, 8.4.2.6	Safety requirements, including: fault tolerant time, warning and degradation concept, driver's actions	Requirements in a FunctionalSafetyConcept related with Satisfy links to FAA without redundancy or safety measures OR Requirements in a FunctionalSafetyConcept related with Satisfy links to FAA with	“fault tolerant time”, “warning and degradation concept” and “driver’s actions” has to be included.
	Functional architecture, including: functional redundancies, safe states, emergency operation, driver actions, external measures (if any), ASILs		Add the “external measure” concept, “emergency operation” concept.

ISO26262 ref.	Requirement of the standard	Requirement to system description and modeling already covered in EAST-ADL	Requirement to system description and modeling to be added in EAST-ADL
	Preliminary physical architecture, in which functionality is allocated	redundancy and safety measures (In case safety solutions are modelled as a modified FAA, this structure may also linked with a refine relation to a Functional Safety Requirement. The original FAA stays non-redundant in that case) ("Preliminary physical architecture, in which functionality is allocated" appears to be too early. ISO26262 does not mention preliminary hardware, it only mentions architectural elements which can stay purely functional in concept phase)	To be investigated
Part 4 – Clause 6	**Technical Safety Requirements**		
Part 4 – Clause 6.4.1.1	Interfaces including communication and HMI (if applicable)	FDA for Interfaces including communication and HMI (if applicable)	No additional requirements (?)
Part 4 – Clause 6.4.1.1	Environmental and functional constraints	EnvironmentModel; FDA for functional constraints	No additional requirements (?)
Part 4 – Clause 6.4.1.1	Configuration requirements	Requirements on FDA, variability mechanisms	No additional requirements (?)
Part 4 – Clause 6.4.1.1	Response to stimuli	FDA behavior or Requirements on FDA	No additional requirements (?)
Part 4 – Clause 6.4.2.3	Safety mechanisms (fault detection and control): - detection, indication and control of faults of the item - detection, indication and control of faults in external devices that interact with the system - measures that enable the system to achieve or maintain a safe state	Requirements on FDA and FDA elements represent Safety mechanisms (fault detection and control);	To investigate the possibility to enrich the modeling concept related to safety measures (mechanism of transition to safe state, warning and degradation).

ISO26262 ref.	Requirement of the standard	Requirement to system description and modeling already covered in EAST-ADL	Requirement to system description and modeling to be added in EAST-ADL
	- measures to detail and implement the warning and degradation concept - measures to detail and implement the warning and degradation concept		
Part 4 – Clause 6.4.2.3	For each safety mechanism that enables an item to achieve or maintain a safe state the following shall be specified: the transition to the safe state, including the requirements to control the actuators - the fault tolerant time interval - the emergency operation interval, if the safe state cannot be reached immediately - the measures to maintain the safe state.	Not covered	To add: - fault tolerant time interval; - emergency operational interval
Part 4 – Clause 6.4.4	measures which prevent faults from being latent shall be defined	Not covered	Add the “latent fault” concept.
Part 4 – Clause 7.4.1; 7.4.5	**Technical Safety Concept**		
Part 4 – Clause 7.4.1.1	The system design shall be based on the functional concept, the preliminary architectural assumptions and the technical safety requirements	Requirements on FDA, refined by SafetyConstraints represent Safety requirements	No additional requirements
Part 4 – Clause 7.4.1.5	The technical safety requirements shall be allocated to hardware and software elements (ASIL Allocation)	SafetyConstraint represent ASIL allocation FDA and Requirements on FDA and HDA represent	No additional requirements
Part 4 – Clause 7.4.1-7.4.4	**System design specification**		
Part 4 – Clause 7.4.4	'- Measures for control of random hardware failures: - Specifications of the measures to detect, control or mitigate the random failures - Target values for metrics - Evaluation procedures of violation of the safety goals - Diagnostics and coverage targets at element level	- FDA and Requirements on FDA and HDA represent; - Measures for control of random hardware failures. Instruction for Engineers instruct that "Target values for metrics", etc. are covered; - Requirements and related VVCase define how to "evaluate	To investigate the possibility to enrich the concepts related to the measures for control of random failures

ISO26262 ref.	Requirement of the standard	Requirement to system description and modeling already covered in EAST-ADL	Requirement to system description and modeling to be added in EAST-ADL
		procedures of violation of the safety goals";	
Part 4 – Clause 7.4.3	Measures to eliminate or to mitigate the effects of internal and external systematic failures	Requirements define the "Diagnostics and coverage targets at element level" FDA and HDA requirements specify some "Measures to eliminate or to mitigate the effects of internal and external systematic failures"	To be further investigated
Part 4 – Clause 7.4.6	Hardware software interface specifications: - the relevant operating modes of hardware devices and the relevant configuration parameters	- HWFunction, BSWFunction and LocalDeviceManager specify "Hardware software interface"; - A ModeGroup can be owned by HW element to define "relevant operating modes of hardware devices" while variability mechanisms may define "relevant configuration parameters"	To be further investigated
Part 4 – Clause 7.4.6	Hardware software interface specifications: - the hardware features that ensure the independence between elements and that support software partitioning	Requirements on HDA define "hardware features that ensure the independence between elements and that support software partitioning"	To be further investigated
Part 4 – Clause 7.4.6	Hardware software interface specifications: - shared and exclusive use of hardware resources - the access mechanism to hardware devices	FDA and requirements on FDA define "shared and exclusive use of hardware resources" and "the access mechanism to hardware devices"	To be further investigated
Part 4 – Clause 7.4.6	Hardware software interface specifications: - the timing constraints defined for each service involved in the technical safety concept	Resources for software (memory, I/O, etc) are treated on Implementation level. Timing constraints on FDA are	To be further investigated

ISO26262 ref.	Requirement of the standard	Requirement to system description and modeling already covered in EAST-ADL	Requirement to system description and modeling to be added in EAST-ADL
		used to "Timing constraints defined for each service involved in the technical safety concept"	
Part 4 – Clause 7.4.6	Hardware software interface specifications: - the hardware diagnostic features - the diagnostic features concerning the hardware, to be implemented in software	FDA and requirements on FDA and HDA specify "the hardware diagnostic features " and "the diagnostic features concerning the hardware, to be implemented in software"	To be further investigated
Part 4 – Clause 7.4.7	Specification of requirements for production, operation, service and decommissioning: - Assembly instructions requirements - Safety-related special characteristics - Requirements dedicated to ensure proper identification of systems or elements -Verification methods and measure for production - Service requirements including diagnostic data and service notes - Decommissioning requirements	Not covered	To be added (Requirements on relevant elements, possible organized in a RequirementContainer structure according to the concerns "production, operation, service and decommissioning")

3 Modeling Concepts for Supporting the Analyses of Behavior-Centric Properties

The reasoning and analysis of dependability & performance involve many aspects in a system's lifecycle. In system development, this requires not only information about the system's topologies, but also an understanding of system behaviors in reacting environmental stimuli and in managing the deployment of internal communication and computation resources. While providing necessary modeling support for capturing important performance and dependability constraints (e.g. end-to-end timing, reliability and safety constraints), current EAST-ADL provides only a rather limited support for capturing the behaviors underlying the generations of such analytical models.

This chapter presents the proposals that have been developed in MAENAD to enhance EAST-ADL for allowing advanced analysis of dependability & performance. An overview of potential EAST-ADL support for analysis and a review of the previous EAST-ADL behavior annex proposal can be found in D3.2.1. In this chapter, we focus on the recent advances towards a final language upgrade. The proposed EAST-ADL enhancement on native behavior modeling can bring in many important benefits. Besides the decisions underlying the assignments of time budgets and error behaviors, such an enhancement will also improve the EAST-ADL support for safety requirements, function/component contracts, fault injection, and test case generation. It will also constitute a necessary step towards the integrations of external mature formalisms and tools for advance prediction of dependability& performance. See Figure 1 for an illustration of the key factors that have affected the language enhancement proposal.

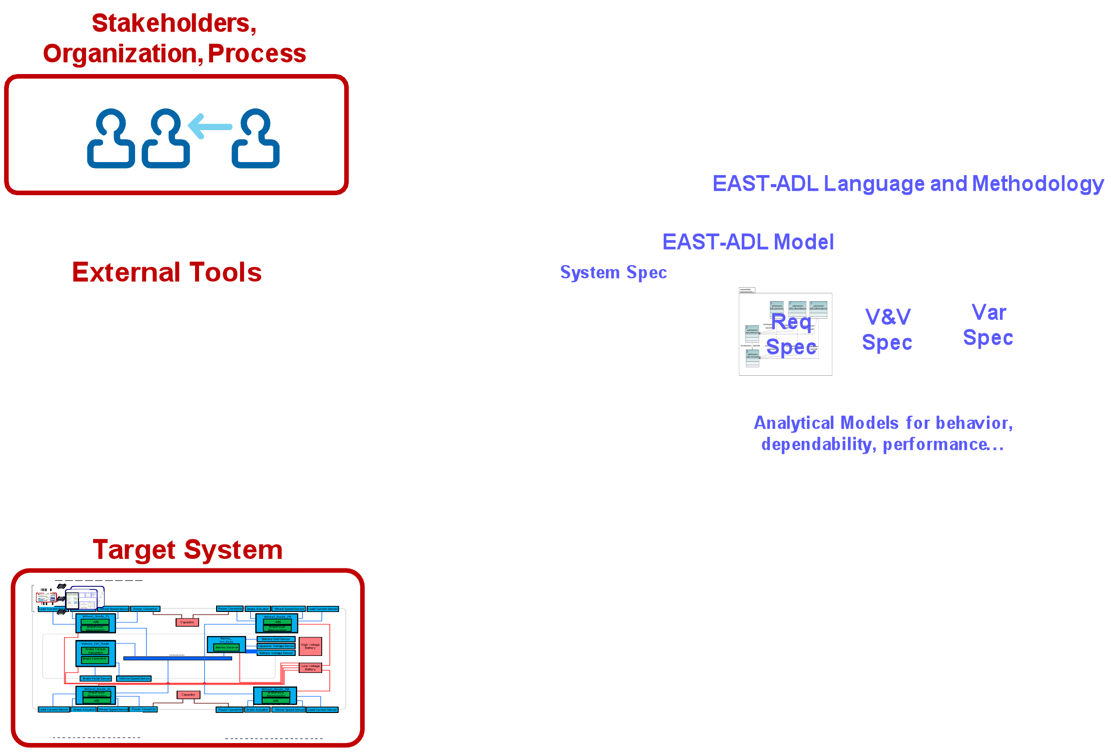

Stakeholders, Organization, Process
EAST-ADL Language and Methodology
EAST-ADL Model
System Spec
External Tools
Req Spec
V&V Spec
Var Spec
Analytical Models for behavior, dependability, performance...
Target System

Figure 1. The scope of EAST-ADL enhancement and related contextual factors.

3.1 Background

In FEV (Fully Electrical Vehicles), embedded systems play important roles in regard to advanced control and mode management and have stringent dependability and performance constraints. A specification of the expected EAST-ADL language support, together with the related FEV specific engineering scenarios, can be found in the MENAD deliverable D2.1.1. It is concluded that an enhanced language support for behavior specification is necessary for many reasons, such as unambiguous interpretation of requirements and early quality predictions. In particular, the following categories of language features are considered important for the engineering of FEV:

- To support precise definitions of temporal characteristics for the definition and analysis of safety constraints (4SG#0050, 4SG#0057, 4SG#0058, 4SG#0059)
- To support the assessment of completeness and correctness of the safety requirements (4SG#0048)
- To support the descriptions of driving profiles (CON#2001), physical dynamics (CRF#0006b, CRF#0007b), power management procedures (CRF#0010b, CRF#0011b, CRF#0013b, CRF#0014b, CRF#0015b), fault tolerance design (CRF#0017b, CRF#0018b)
- To support the generation and precise definition of test cases (4SG#0049a, 4SG#0050)
- To support the integration with external formalisms (CON#0017, CON#0018, CON#0019)

In regard to system behaviors, current EAST-ADL provides language support for specifying the executions of system functions, together with related allocations, triggering policies, and timing constraints. The specification of actual behaviors of system functions relies on external tools (e.g., Simulink/Matlab). This means that behavior models, simulation, analysis, and code generation for the final software synthesis are all maintained and carried out based on external tools. This kind of black-box approach to behavior specification is considered sufficient for implementation design, such as in regard to multitasking and final software configuration. See Figure 2 for an overview of related modeling constructs.

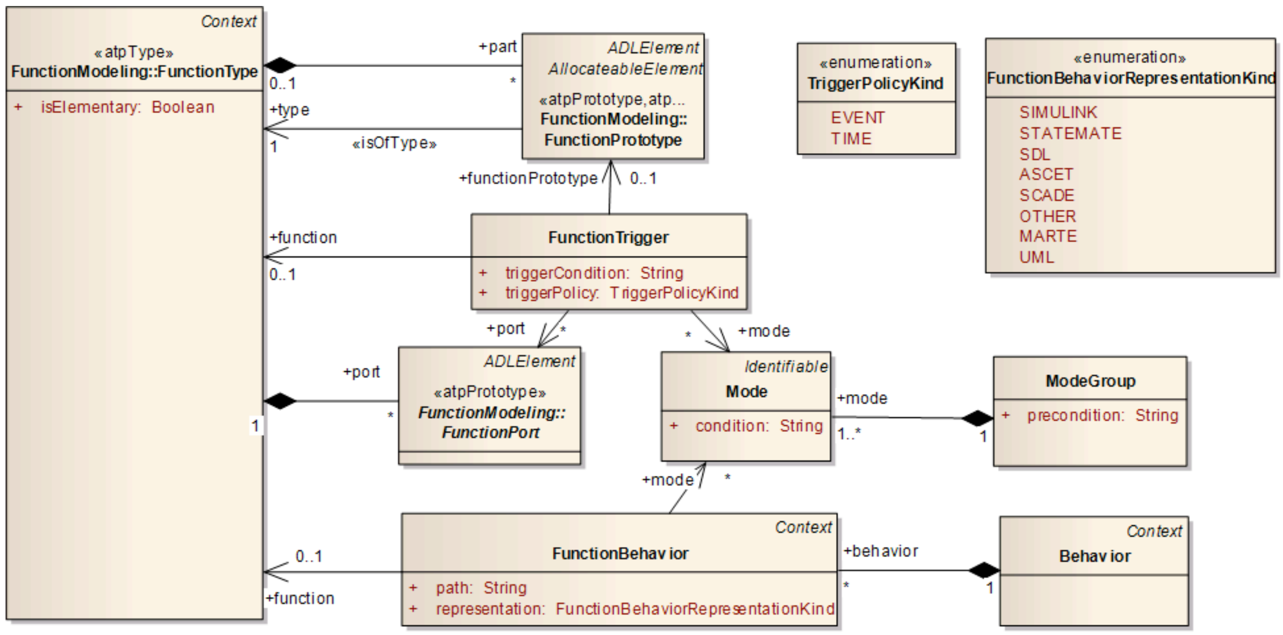


Figure 2. An overview of the meta-model definitions in current EAST-ADL2 Behavior Modeling Package.

From a system design point of view, the behavioral issues that can be of particular concern include not only the execution scheme (e.g., time- or event-triggered execution of system functions), but also the system's operational situations, the dynamics of plant under control, the nominal and erroneous behaviors of functions and components (e.g., their internal state transitions), as well as the compositions of various behaviors and related mode assignments. A precise specification of such issues is fundamental for many overall design decisions, including requirements definitions and refinements, function structuring, the synthesis of analytical models and test cases, and safety engineering, etc. It is seldom the case that a single analysis tool would cover all these. Even if the actual behaviors of system functions are captured in external tools, there is still a need of explicitly annotating related bounds (e.g., invariants of data, internal states and state transitions) and permitting the traceability of behavior concerns at different levels of abstraction, such as for the control of consistency and completeness.

To facilitate the predication of dependability and performance, it is expected that EAST-ADL as a system architecture description language would constitute the basis for consolidating various kinds of behavior information. Such behavior information can for example be associated to requirements, architectural and analytical models, or V&V cases.

Current EAST-ADL supports the annotations of error behaviors of system functions and components through error models. The aim is to provide analytical information for fault-tree analysis and safety constraint assignment. The specifications of error logics are directly based on external formalisms, such as expressions in Boolean logic. There is still a lack of support for consolidating the error logics in different external formalisms. Moreover, an enhancement of the language in regard to the temporal aspects of anomalies is necessary for allowing advanced safety analysis (e.g., model-checking and fault injection). The aim is to support precise definitions of faulty conditions in both value- and time- domain, the transitions across nominal and erroneous states, and thereby the reasoning of emergent properties due to compositions.

3.2 EAST-ADL Enhancement Proposals

This proposal further refines the EAST-ADL behavior annex proposed in the ATESST2 project. The aim of the behavior annex is to allow a more precise specification of behavioral constraints, which are implied by requirements and satisfied by functions, hardware and environmental components, etc. See the MENAD deliverable D3.2.1 for an introduction of the proposed behavior annex. Major improvements in MAEAD include:

- A harmonization with the syntax and semantics of current EAST-ADL support for specifications of architectural structures, execution behaviors (e.g., Triggers, FunctionEvents) of functions and components, and execution specific timing constraints;
- A consolidation of proposed behavior constraints in regard to their definitions and relations.
- Support for type-prototype pattern, allowing the instantiating of behavior types in particular contexts.
- An investigation of alignment with time-automata semantics and the transformation to the model-checking tool UPPAAL.

We introduce the related key concepts in the following parts of this section. See the table below for a summary of the proposed language updates for the EAST-ADL BehaviorAnnex (Annexes:: BehaviorConstraints).

Table 2. An overview of main updates for an enhanced behavior description support.

Old Definition	Update(s)	Comment
BehaviorAnnex	**removed**	This top-level container is now removed. The composition support is now given by *BehaviorConstraintType*
BehaviorConstraint	**Replaced** by *BehaviorConstraintType* and *BehaviorConstraintPrototype*	To align with the type-prototype pattern
-	**Added** *BehaviorConstraintType*	(See above)
-	**Added** *BehaviorConstraintPrototype*	(See above)
	Added *BehaviorConstraintType.part* : *BehaviorConstraintPrototype*	(See above)

	Added *BehaviorConstraintPrototype.type* : *BehaviorConstraintType*	(See above)
-	**Added** *BehaviorInstantiationParameter*	To support the parameterization and instantiations of *BehaviorConstraintPrototype*
-	**Added** *BehaviorConstraintType.parameter* : *BehaviorInstantiationParameter*	(See above)
-	**Added** *BehaviorConstraintType. partBindingParameter* : *BehaviorConstraintBindingParameter*	(See above)
-	**Added** *BehaviorConstraintBindingParameter*	(See above)
-	**Added** *BehaviorConstraintPrototype. instantiatedWithParameter* : *BehaviorInstantiationParameter*	(See above)
-	**Added** the specialization of *BehaviorInstantiationParameter* to *BehaviorConstraintBindingParameter*	(See above)
ParameterConstraint	**Renamed** to *AttributeQuantificationConstraint*	“Parameter” is an overloaded term.
Parameter	**Renamed** to *Attribute*	(See above)
ParameterCondition	**Renamed** to *Quantification*	A more exact definition of the role.
StateMachineConstraint	**Renamed to** *TemporalConstraint*	Better support for other constraints on the history of behaviors, which are not directly expressed in SM (e.g. in temporal logic)
Specialization of *BehaviorConstraint* to *ParameterConstraint*	**Replaced** with the aggregation from *BehaviorConstraintType* to *AttributeQuantificationConstraint*	Better support for the internal structuring of content of behavior constraint annotation.
Specialization of *BehaviorConstraint* to *StateMachineConstraint*	**Replaced** with the aggregation from *BehaviorConstraintType* to *TemporalConstraint*	(See above)
Specialization of *BehaviorConstraint* to *ComputationConstraint*	**Replaced** with the aggregation from *BehaviorConstraintType* to *ComputationConstraint*	(See above)
-	**Added** *LogicalEvent*	To support explicitly events related to values.
State.denote* : *ParameterCondition	**Replaced** with *State.quantificationInvariant*: quantificationInvariant	A more exact definition of the role.
-	**Added** *EventOccurrence*	A key concept introduced to integrate existing EAST-ADL constructs for the specifications of various behavior constraints.
-	**Added** the aggregation from *TemporalConstraint* to *EventOccurrence*	(See above)
-	**Added** *EventOccurrence.occurredExecutionEvent* : *Timing::Event*	(See above)
-	**Added** *EventOccurrence.occurredLogicalEvent* : *LogicalEvent*	(See above)

-	**Added** *EventOccurrence. occurredFeatureFlaw*: *FeatureFlaw*	(See above)
-	**Added** *EventOccurrence. occurredAnomaly*: *Anomaly*	(See above)
-	**Added** *EventOccurrence. occurredHazardousEvent* : *HazardousEvent*	(See above)
Transition.read*:*Parameter	**Replaced** by *Transition.readEventOccurrence?* : *EventOccurrence*	(See above)
Transition.write*:*Parameter	**Replaced** by *Transition.writeEventOccurrence?* : *EventOccurrence*	(See above)
-	**Added** *LogicalTimeCondition*	A key concept introduced to allow fine-grained specification of timing constraints for behaviors, while reusing the support of execution timing for the semantics.
-	**Added** the aggregation from *TemporalConstraint* to *LogicalTimeCondition*	(See above)
-	**Added** *Quantification.timeCondition* : *LogicalTimeCondition*	(See above)
-	**Added** *State.timeInvariant* : *LogicalTimeCondition*	(See above)
-	**Added** *Transition.timeGuard* : *LogicalTimeCondition*	(See above)
-	**Added** *LogicalTransformation.timeInvariant* : *LogicalTimeCondition*	(See above)
-	**Added** *TransformationOccurrance.timeCondition*: *LogicalTimeCondition*	(See above)
-	**Added** *LogicalTimeCondition.upper*: *Timing*::*TimeDuration*	(See above)
-	**Added** *LogicalTimeCondition.lower*: *Timing*::*TimeDuration*	(See above)
-	**Added** *LogicalTimeCondition.width*: *Timing*::*TimeDuration*	(See above)
-	**Added** *LogicalTimeCondition. startPointReference*: *EventOccurrence*	(See above)
-	**Added** *LogicalTimeCondition. endPointReference*: *EventOccurrence*	(See above)
Transformation	**Renamed** to *LogicalTransformation*	A more exact definition of the role.
-	**Added** *LogicalTransformation. clientServerInterfaceOperation* : *Operation*	To merge with existing related constructs.
Transformation.incomingFlow : Flow	**removed**	Unnecessary (due to the new *TransformationOccurrance*).
Transformation. outgoingFlow: Flow	**removed**	(See above)
Flow	**Renamed** to *LogicalPath*	A more exact definition of the role.
-	**Added** *TransformationOccurrance*	Concept introduced to support the invocations of logical transformation.

-	**Added** *TransformationOccurrance. invokedLogicalTransformation*: *LogicalTransformation*	(See above)
-	**Added** *Transition.effect* :*LogicalTransformation*	(See above)
-	**Added** *LogicalPath. transformationOccurrance*:*LogicalTransformation*	(See above)
Flow.sinkParameter* : *Parameter	**removed**	Unnecessary (due to the new *TransformationOccurrance*).
Flow.sourceParameter* : *Parameter	**removed**	Unnecessary (due to the new *TransformationOccurrance*).
Flow.orderedSegment* : *Flow	**Replaced** by: *LogicalPath.segment*{ordered} : *LogicalPath*	A more exact definition.
-	**Added**: *LogicalPath.strand* : *LogicalPath*	(See above)
-	**Added**: *LogicalPath.correspondingExecutionEventChain*: *Timing::EventChain*	To allow the merge of control flows and timing chains.
-	**Added**: *LogicalPath. precedingExecutionEventChain*:*Timing::EventChain*	(See above)
-	**Added**: *LogicalPath. succeedingExecutionEventChain*:*Timing::EventChain*	(See above)

3.2.1 Behavior Constraint Types and Their Targets

The proposed behavior extension provides a language basis for allowing a more precise declaration of various behavior concerns, such as assumed or implied by requirements and quality constraints, assigned to system environment, functions and components, or test procedures. To capture those concerns, three categories of behavior constraints are proposed. It is up to the users of EAST-ADL, in their particular design and analysis contexts, to decide the exact types and degree of constraints to be applied. These categories of behavior constraints are:

- **Attribute Quantification Constraint** – relating to the declarations of value attributes and the related acausal quantifications (e.g., U=I*R).
- **Temporal Constraint** – relating to the declarations of behavior constraints where the history of behaviors on a timeline is taken into consideration.
- **Computation Constraint** – relating to the declarations of cause-effect dependencies of data in terms of logical transformations (for data assignments) and logical paths.

As shown in Figure 3, we distinguish the types of behavior constraints from their prototypes. The latter represent the instantiations of the types in particular context (BehaviorContraintPrototype). The language extension for behavior constraints are currently managed in the BehaviorAnnex. The meta-model integration is done in a modular way such that no existing EAST-ADL constructs are modified by the extension. Also shown in Figure 3, the proposed language extension for behavior constraints can be applied to address a variety of behavioral concerns in a system. The advantages are introduced in the following sub-sections.

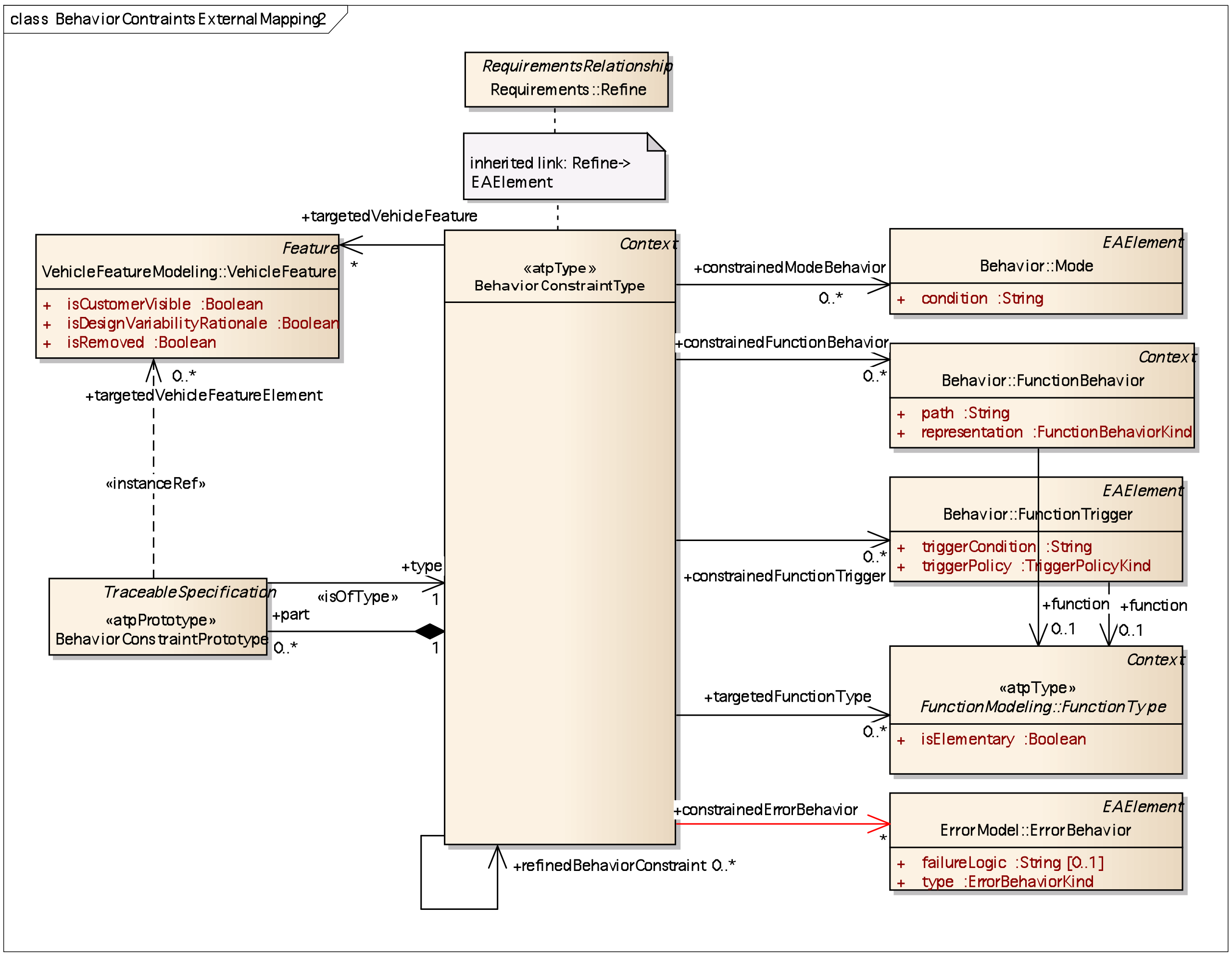

class BehaviorContraintsExternalMapping2
RequirementsRelationship
Requirements::Refine
inherited link: Refine-> EAElement
+targetedVehicleFeature
Feature
VehicleFeatureModeling::VehicleFeature
+ isCustomerVisible :Boolean
+ isDesignVariabilityRationale :Boolean
+ isRemoved :Boolean
0..*
+targetedVehicleFeatureElement
«instanceRef»
TraceableSpecification
«atpPrototype»
BehaviorConstraintPrototype
+type
«isOfType»
+part
0..*
Context
«atpType»
BehaviorConstraintType
+constrainedModeBehavior
EAElement
Behavior::Mode
+ condition :String
+constrainedFunctionBehavior
Context
Behavior::FunctionBehavior
+ path :String
+ representation :FunctionBehaviorKind
EAElement
Behavior::FunctionTrigger
+ triggerCondition :String
+ triggerPolicy :TriggerPolicyKind
+constrainedFunctionTrigger
+function
+function
+targetedFunctionType
Context
«atpType»
FunctionModeling::FunctionType
+ isElementary :Boolean
+constrainedErrorBehavior
EAElement
ErrorModel::ErrorBehavior
+ failureLogic :String [0..1]
+ type :ErrorBehaviorKind
+refinedBehaviorConstraint 0..*

Figure 3. BehaviorConstraintType and the constrained properties in the proposed EAST-ADL2 Behavior Annex.

3.2.1.1 Behavior Constraints for Refinements of Requirements

Through requirement refinement (requirement::refine) relations, behavior constraints can be used to refine the textual statements of requirements, use cases, as well as the assumed operation situations. Such refinements formalize the related behavioral concerns (e.g. the boundary conditions and invariants of variables, states and state transitions) for a more rigorous verification and validation of requirements.

3.2.1.2 Behavior Constraints for Vehicle Features

In EAST-ADL, system functions at the topmost level of abstraction are referred to as vehicle features (VehicleFeatureModeling::VehicleFeature). When assigned to such system functions, behavior constraints are used to capture the related data and behavior characteristics that have to be fulfilled by the target feature. This would constitute a basis for having a more precise reasoning about the configuration of features in terms of feature tree. For example, an assignment of parent-child relation between features may also imply the inheritance of related behavior constraints.

3.2.1.3 Behavior Constraints for Functions and Components

Behavior constraints provide support for specifying the bounds or contracts of acceptable behaviors of functions in a system or its environment. This is achieved by assigning behavior constraints to the function behaviors (which is a container with references to external models and has the run-to-completion semantics), or the function triggers (which declares a triggering policy for the execution of functions) of the target function type. See Figure 4 for a user-model example of applying the behavior constraints to a design function type.

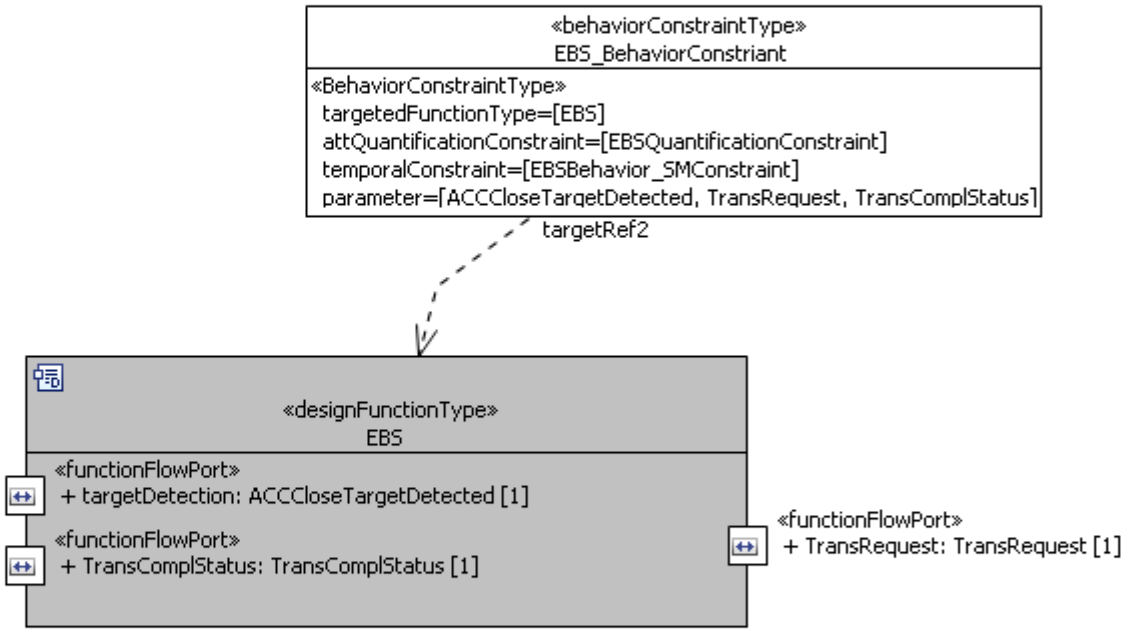

«behaviorConstraintType»
EBS_BehaviorConstriant
«BehaviorConstraintType»
targetedFunctionType=[EBS]
attQuantificationConstraint=[EBSQuantificationConstraint]
temporalConstraint=[EBSBehavior_SMConstraint]
parameter=[ACCCloseTargetDetected, TransRequest, TransComplStatus]
targetRef2
«designFunctionType»
EBS
«functionFlowPort»
+ targetDetection: ACCCloseTargetDetected [1]
«functionFlowPort»
+ TransComplStatus: TransComplStatus [1]
«functionFlowPort»
+ TransRequest: TransRequest [1]

Figure 4. Declaring the behavior constraints of a design function type.

3.2.1.4 Behavior Constraints for Modes

Behavior constraints can also be applied to mode declarations. This modeling feature is not only useful for precisely specifying the mode logics (e.g., to relate modes and the transitions with operational states and resource conditions), but also for specifying the impacts of modes on application behaviors and the system support for quality-of-service management.

3.2.1.5 Behavior Constraints for Error Estimation

While currently focusing on nominal behaviors, the proposed behavior contain extension can also be applied to strengthen the EAST-ADL support for error modeling. When targeting error behaviors, behavior constraints refine the estimated anomalies declared in the error models. This would then allow: precise definitions of faulty conditions in value and time, erroneous states and their transitions, formal analysis of emergent properties due to the compositions. A behavior constraint can be associated to nominal and error behaviors simultaneously. This modeling feature is useful for the specification of fault-injection by allowing transitions across nominal states and errors (for such transitions certain probabilistic attributes will be added).

3.2.2 Attribute Quantification Constraints

Attribute quantifications provide support for the definitions of acausal behavior constraints. They are useful for stating the required value attributes such as the input-, output- and internal variables, as well as expressing the expected value conditions or invariants in terms of equations like F=m*a, U=I*R. This is comparable with the Modelica approach to behavior specification, where system behaviors are primarily declared based on equations instead of data assignment statements. When necessary, the corresponding data transformations for a quantification can be declared through computation constraints, which are introduced in Section 3.2.4. See Figure 5 for an overview of the related meta-model definitions.

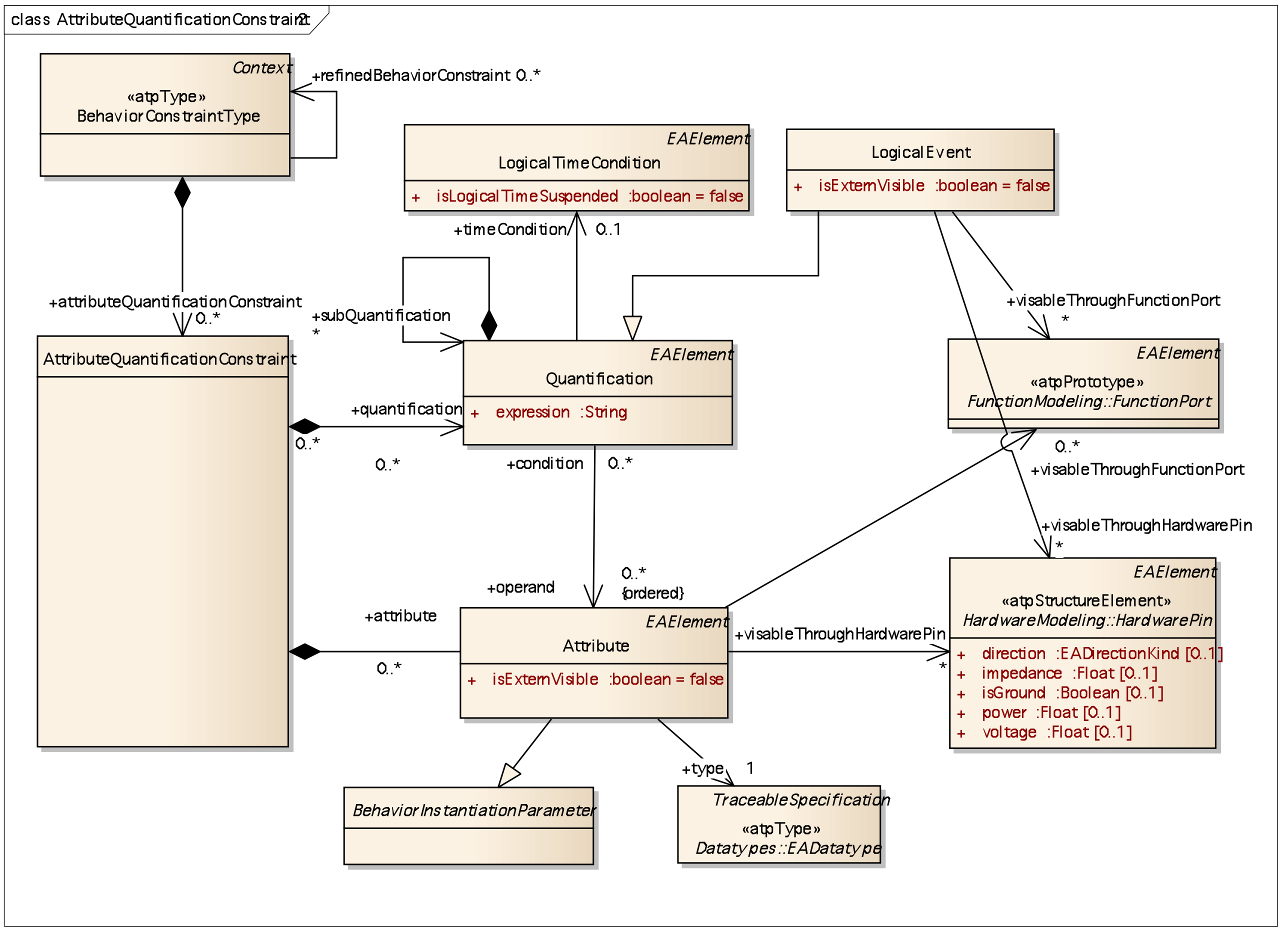

class AttributeQuantificationConstraint
Context
«atpType»
BehaviorConstraintType
+refinedBehaviorConstraint 0..*
EAElement
LogicalTimeCondition
+ isLogicalTimeSuspended :boolean = false
LogicalEvent
+ isExternVisible :boolean = false
+timeCondition 0..1
+attributeQuantificationConstraint 0..*
+subQuantification *
AttributeQuantificationConstraint
EAElement
Quantification
+ expression :String
+quantification
0..*
0..*
+condition 0..*
EAElement
«atpPrototype»
FunctionModeling::FunctionPort
+visableThroughFunctionPort *
0..*
+visableThroughFunctionPort
+visableThroughHardwarePin *
EAElement
«atpStructureElement»
HardwareModeling::HardwarePin
+ direction :EADirectionKind [0..1]
+ impedance :Float [0..1]
+ isGround :Boolean [0..1]
+ power :Float [0..1]
+ voltage :Float [0..1]
+operand
0..* {ordered}
+attribute
0..*
EAElement
Attribute
+ isExternVisible :boolean = false
+visableThroughHardwarePin *
+type 1
BehaviorInstantiationParameter
TraceableSpecification
«atpType»
Datatypes::EADatatype

Figure 5. AttributeQuantificationConstraint and its properties in the proposed EAST-ADL2 Behavior Annex.

A specification of attribute quantification constraints is based on the following constructs:

- **AttributeQuantificationConstraint** – the modeling construct for grouping the attribute and quantification declarations in a behavior constraint.
- **Attribute** – the modeling construct for the declarations of the in-, out-, or local variables to be processed or owned by a behavior. Each Attribute is typed by a data type (DataType), specifying the related meta-information like unit, valid range, required accuracy, etc. If an attribute is externally visible (isExternVisble == true), it denotes an input or output variable and has associated function ports or hardware pins for the external accesses. Attributes are instantiation parameters (BehaviorInstantiation-Parameter), to which certain values can be assigned when behavior constraint types are instantiated as behavior constraint prototypes in a context.
- **Quantification** – the modeling construct for the declarations of the value conditions or invariants that an attribute have to obey. For example, a quantification may state that a monitored environmental variable must fall within particular segments in the spectrum of its possible value range during different modes. Each quantification can also be associated with time conditions for stating the time instances or time intervals where the

quantification is valid or takes place. A quantification can be composed of one or several sub-quantifications. (The expression statement is a placeholder for the upcoming support for logical and arithmetic operators and equations.)

- **Logical Event** – the modeling construct for the declarations of the value conditions that, when fulfilled, may trigger state transitions. If a logical event is externally visible (isExternVisble == true), it is disseminated through function ports or hardware pins.

3.2.3 Temporal Constraints

Temporal constraints provide support for capturing the dependency that a behavior has in regard to its own history and other behaviors on a timeline. They can be expressed by means of temporal logic or state-machines. The semantics is based on timed-automata and thereby comparable with approaches like Promela/Spin and UPPAAL in regard to analysis leverage. Compared to those analytical models, the proposed temporal constraints integrate the existing EAST-ADL support for function, communications, executions, and timing, and provide thereby a more exact definition of semantics in regard to the notions of time, events and events synchronizations. See Figure 5 for an overview of the related meta-model definitions.

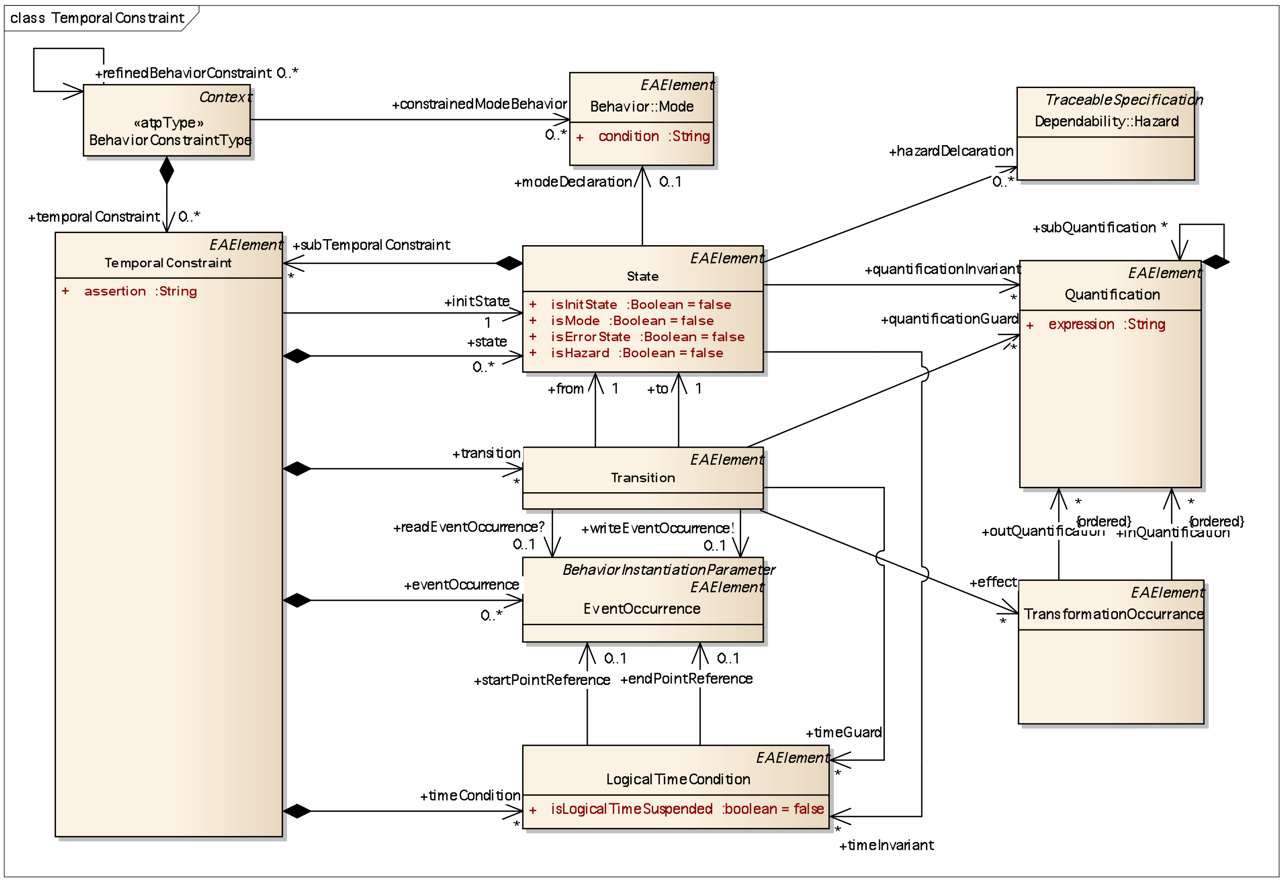


Figure 6. TemporalConstraint and its properties in the proposed EAST-ADL2 Behavior Annex.

A specification of attribute quantification constraints is based on the following constructs:

- **TemporalContraint** – the modeling construct for grouping the declarations of states, transitions, event occurrences, and logical time conditions in a behavior constraint. It can contain assertions in temporal or modal logics when desired. (The assertion attribute (assertion) is a placeholder for the upcoming support for expressions based on temporal/modal logics)

- **State** – the modeling construct for the declarations of states that represent the situations where certain quantifications (Quantification) in terms of value conditions or invariants hold. A state can also have time invariants, representing the time conditions that must be true (e.g., the time duration of a state). In a state-machine based specification, there is always one init state. Each state can have subordinate state machines or other temporal constraint definitions (subTemporalConstraint). Besides nominal operation situations, a state can also represent errors (isError == true), or modes (isMode == true)), or hazards (isMode == true). In the two latter cases, the corresponding declarations of modes (modeDeclaration) and hazards (hazardDeclaration) have to be specified.
- **Transition** – the modeling construct for the declarations of transitions between two states. When the related guard conditions both in time and value domains are met, a transition can be fired to respond to the occurrence of an event (readEventOccurrences?) or to signal the occurrence of an event (writeEventOccurrance!). A transition, when fired, can also invoke one or more logical transformations (TransformationOccurrance).
- **EventOccurrence** – the modeling construct for the declarations of occurrences of events that are either logical events (occurredLogicalEvent), execution specific events (occurred-ExecutionEvent), or fault and failure related (occurredFeatureFlaw, occurredAnomay, occurredHazardEvent). Event-occurrences declared in a behavior constraint type are also instantiation parameters (BehaviorInstantiationParameter), which allow a behavior constraint type to be instantiated as behavior constraint prototypes in different contexts. During the instantiation, such parameters are mapped to some global/external event-occurrences. An occurred event can be purely logical or execution specific.
  - The occurrence of a logical event (LogicalEnvents) denotes a value condition that takes place at a particular time instance and becomes valid in a certain time interval. The semantics is given by the definition of corresponding value condition.
  - The occurrence of an execution event (Timing::Event - an existing EAST-ADL construct from the Timing package) denotes a distinct form of state change in a running system, at distinct points in time, such as at the triggering of a function, or at the receiving/sending of data from/to ports. The definition of execution event itself only provides a description expressing its purpose instead of occurrence.
  - The occurrence of a fault or failure (Dependability::FeatureFlaw, ErrorModel::Anomay, Dependability::HazardEvent – all these are existing EAST-ADL construct from the dependability package) denotes a distinct form of deviation from nominal behaviors at distinct points in time. The definitions of those faults and failures provide the descriptions expressing their estimated existences.

  See Figure 7 for an overview of the related meta-model definitions.
- **LogicalTimeCondition** – the modeling construct for the declarations of time conditions in terms of time instances or time intervals. As shown in Figure 8, such time conditions can be assigned to attribute quantifications, states, transitions, logical transformations or the occurrences of such transformations, in order to characterize the related timing, causality, and synchronization. The logical time condition is an abstraction of real time with the semantics given by the associated occurrences (startPointReference and endPointReference) of execution events (e.g., the triggering event of a function). The value of logical time is defined by a time duration specification (Timing::TimeDuration - an existing EAST-ADL construct from the Timing package) in the format of CseCode as in AUTOSAR and MSR/ASAM.

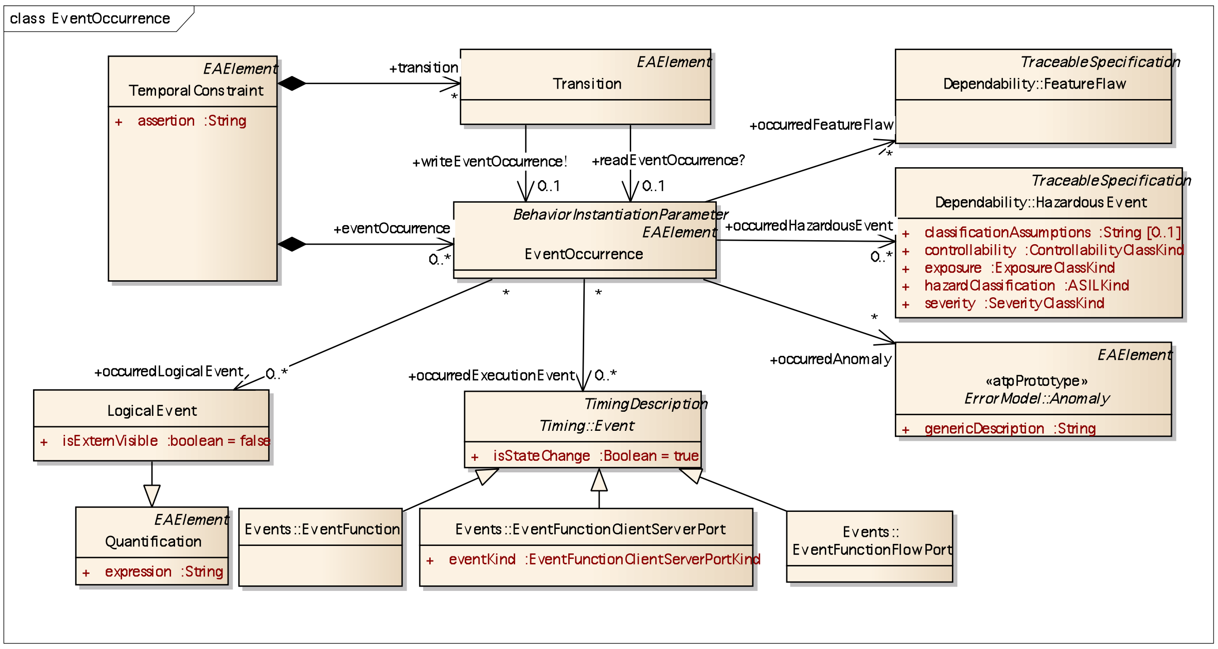

class EventOccurrence
EAElement Temporal Constraint + assertion :String
EAElement Transition
TraceableSpecification Dependability::FeatureFlaw
TraceableSpecification Dependability::HazardousEvent + classificationAssumptions :String [0..1] + controllability :ControllabilityClassKind + exposure :ExposureClassKind + hazardClassification :ASILKind + severity :SeverityClassKind
BehaviorInstantiationParameter EAElement EventOccurrence
EAElement «atpPrototype» ErrorModel::Anomaly + genericDescription :String
LogicalEvent + isExternVisible :boolean = false
TimingDescription Timing::Event + isStateChange :Boolean = true
EAElement Quantification + expression :String
Events::EventFunction
Events::EventFunctionClientServerPort + eventKind :EventFunctionClientServerPortKind
Events::EventFunctionFlowPort
+transition *
+writeEventOccurrence! 0..1
+readEventOccurrence? 0..1
+eventOccurrence 0..*
+occurredFeatureFlaw *
+occurredHazardousEvent 0..*
+occurredAnomaly *
+occurredLogicalEvent 0..*
+occurredExecutionEvent 0..*

Figure 7. EventOccurrence and its properties in the proposed EAST-ADL2 Behavior Annex.

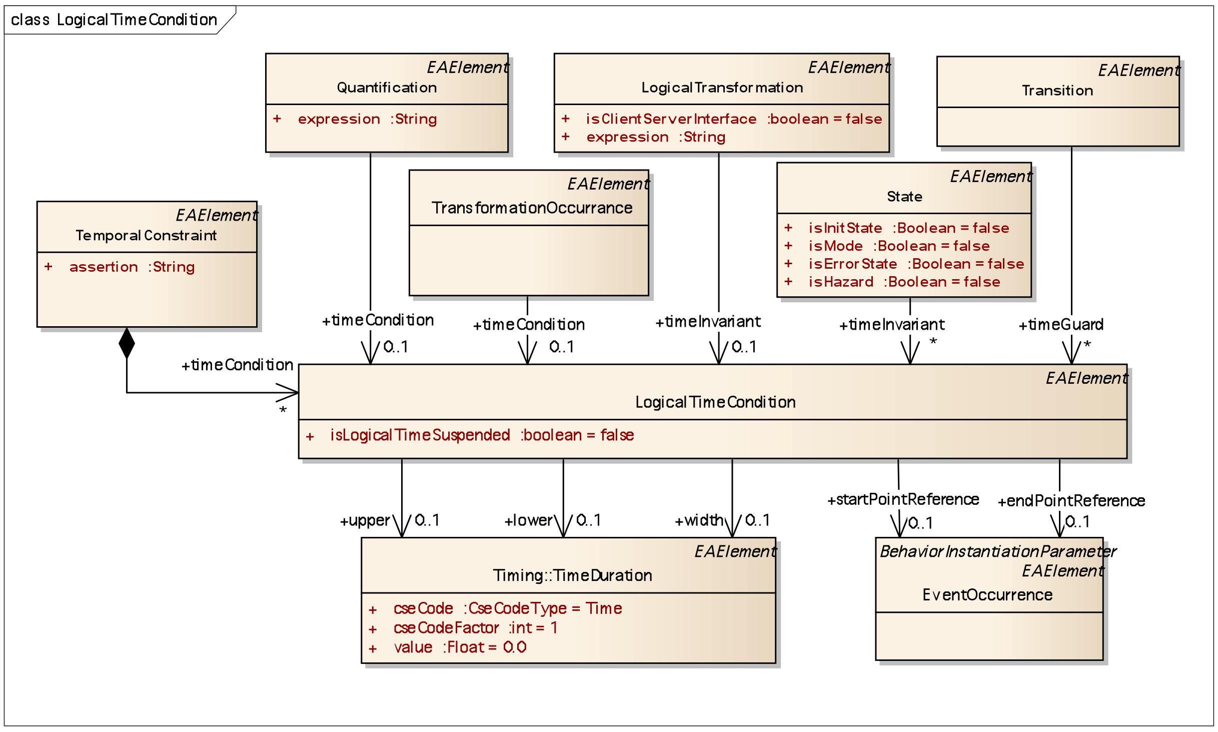

class LogicalTimeCondition
EAElement Quantification + expression :String
EAElement LogicalTransformation + isClientServerInterface :boolean = false + expression :String
EAElement Transition
EAElement TransformationOccurrance
EAElement State + isInitState :Boolean = false + isMode :Boolean = false + isErrorState :Boolean = false + isHazard :Boolean = false
EAElement Temporal Constraint + assertion :String
EAElement LogicalTimeCondition + isLogicalTimeSuspended :boolean = false
EAElement Timing::TimeDuration + cseCode :CseCodeType = Time + cseCodeFactor :int = 1 + value :Float = 0.0
BehaviorInstantiationParameter EAElement EventOccurrence
+timeCondition 0..1
+timeCondition 0..1
+timeInvariant 0..1
+timeInvariant *
+timeGuard *
+timeCondition *
+upper 0..1
+lower 0..1
+width 0..1
+startPointReference 0..1
+endPointReference 0..1

Figure 8. LogicalTimeCondtion and its properties in the proposed EAST-ADL2 Behavior Annex.

See Figure 9 for a user-model example that shows how event occurrences are declared based on execution behaviors. The example system consists of two design functions, a braking control function (pEBS) and a communication transceiver function (pEBSTransiver).

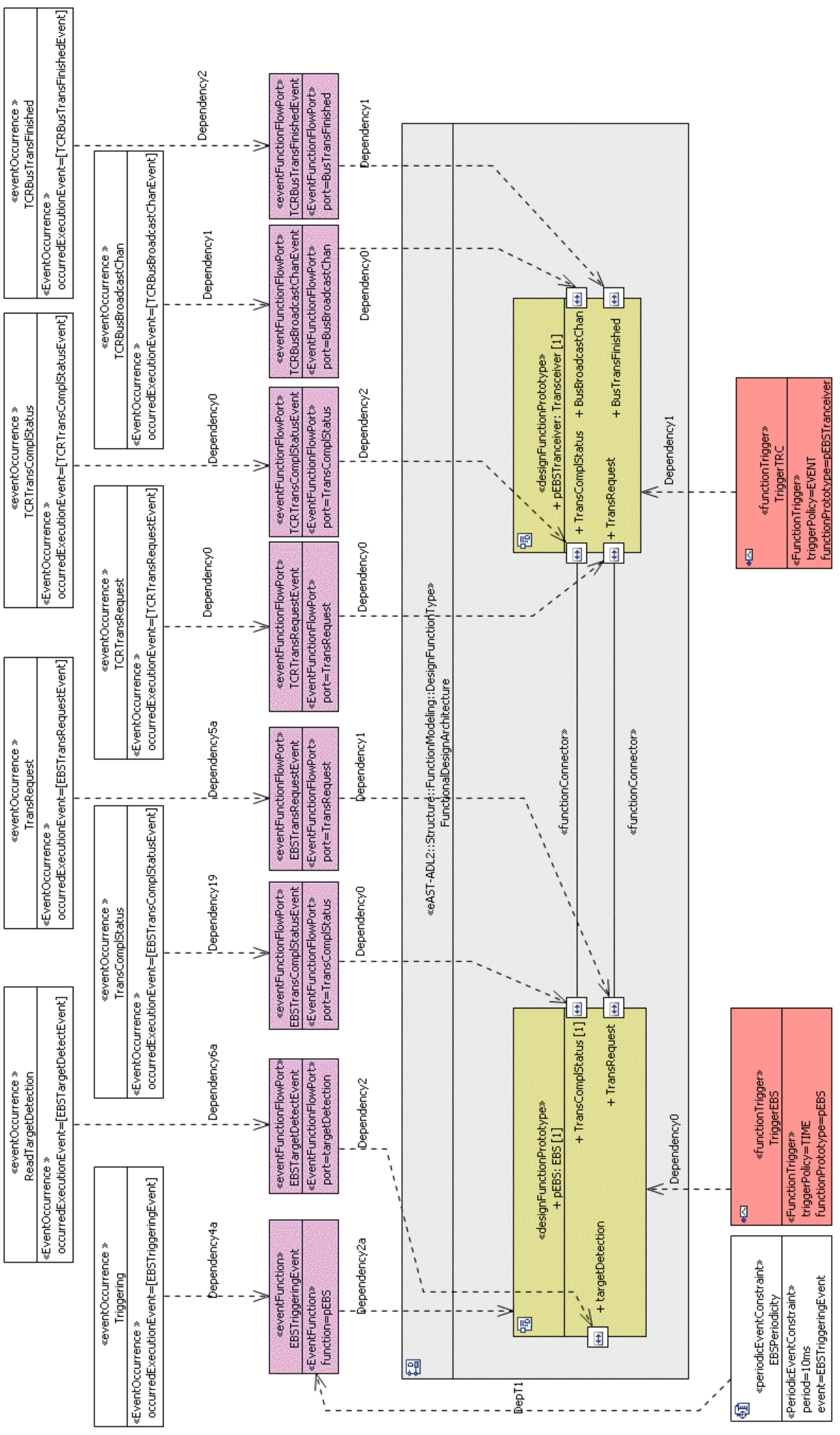

«eventOccurrence » TCRBusTransFinished
«EventOccurrence » occurredExecutionEvent=[TCRBusTransFinishedEvent]
«eventOccurrence » TCRBusBroadcastChan
«EventOccurrence » occurredExecutionEvent=[TCRBusBroadcastChanEvent]
«eventOccurrence » TCRTransComplStatus
«EventOccurrence » occurredExecutionEvent=[TCRTransComplStatusEvent]
«eventOccurrence » TCRTransRequest
«EventOccurrence » occurredExecutionEvent=[TCRTransRequestEvent]
«eventOccurrence » TransRequest
«EventOccurrence » occurredExecutionEvent=[EBSTransRequestEvent]
«eventOccurrence » TransComplStatus
«EventOccurrence » occurredExecutionEvent=[EBSTransComplStatusEvent]
«eventOccurrence » ReadTargetDetection
«EventOccurrence » occurredExecutionEvent=[EBSTargetDetectEvent]
«eventOccurrence » Triggering
«EventOccurrence » occurredExecutionEvent=[EBSTriggeringEvent]
«eventFunctionFlowPort» TCRBusTransFinishedEvent
«EventFunctionFlowPort» port=BusTransFinished
«eventFunctionFlowPort» TCRBusBroadcastChanEvent
«EventFunctionFlowPort» port=BusBroadcastChan
«eventFunctionFlowPort» TCRTransComplStatusEvent
«EventFunctionFlowPort» port=TransComplStatus
«eventFunctionFlowPort» TCRTransRequestEvent
«EventFunctionFlowPort» port=TransRequest
«eventFunctionFlowPort» EBSTransRequestEvent
«EventFunctionFlowPort» port=TransRequest
«eventFunctionFlowPort» EBSTransComplStatusEvent
«EventFunctionFlowPort» port=TransComplStatus
«eventFunctionFlowPort» EBSTargetDetectEvent
«EventFunctionFlowPort» port=targetDetection
«eventFunction» EBSTriggeringEvent
«EventFunction» function=pEBS
Dependency0
Dependency1
Dependency2
Dependency2a
Dependency4a
Dependency5a
Dependency6a
Dependency19
DepT1
«eAST-ADL2::Structure::FunctionModeling::DesignFunctionType» FunctionalDesignArchitecture
«designFunctionPrototype» + pEBSTranceiver: Transceiver [1]
+ TransComplStatus
+ BusBroadcastChan
+ TransRequest
+ BusTransFinished
«designFunctionPrototype» + pEBS: EBS [1]
+ TransComplStatus
+ TransRequest
+ targetDetection
«functionConnector»
«functionTrigger» TriggerTRC
«FunctionTrigger» triggerPolicy=EVENT functionPrototype=pEBSTranceiver
«functionTrigger» TriggerEBS
«FunctionTrigger» triggerPolicy=TIME functionPrototype=pEBS
«periodicEventConstraint» EBSPeriodicity
«PeriodicEventConstraint» period=10ms event=EBSTriggeringEvent

Figure 9. Declaring the occurrences of execution events that express the triggering, data receiving and sending of two functions in a system.

Figure 10 shows a state-machine based specification of the temporal constraint applied to the braking control function (pEBS).

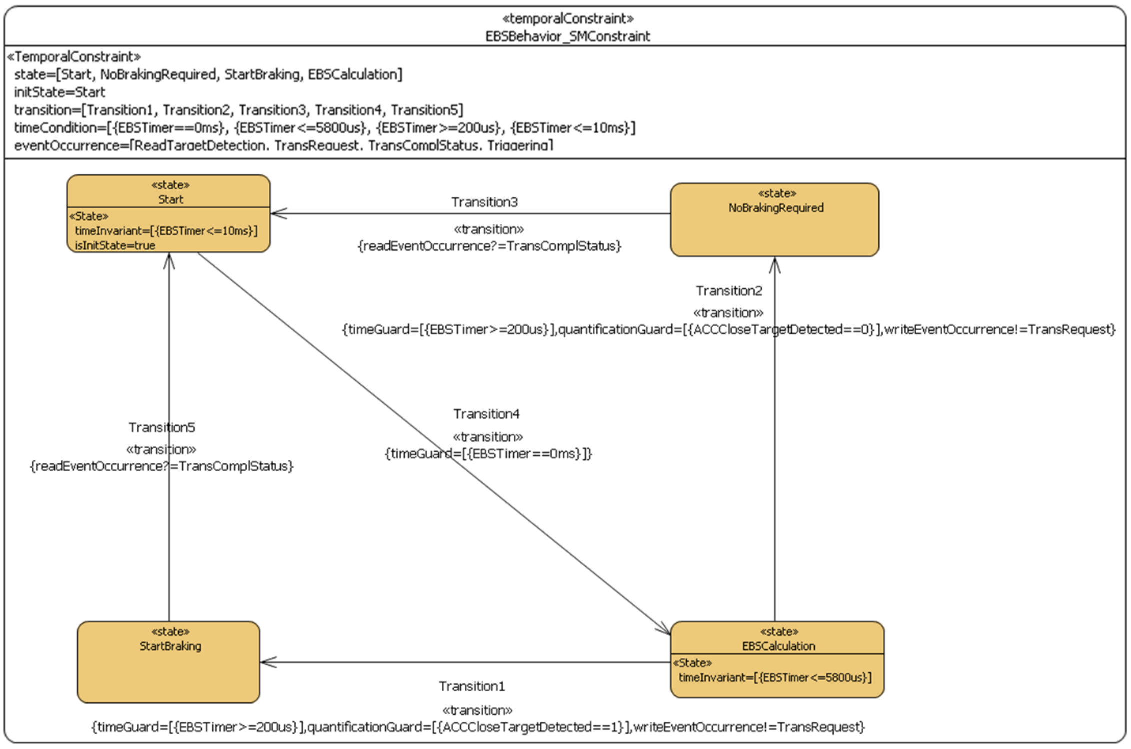

«temporalConstraint»
EBSBehavior_SMConstraint
«TemporalConstraint»
state=[Start, NoBrakingRequired, StartBraking, EBSCalculation]
initState=Start
transition=[Transition1, Transition2, Transition3, Transition4, Transition5]
timeCondition=[{EBSTimer==0ms}, {EBSTimer<=5800us}, {EBSTimer>=200us}, {EBSTimer<=10ms}]
eventOccurrence=[ReadTargetDetection, TransRequest, TransComplStatus, Triggering]
«state» Start
«State» timeInvariant=[{EBSTimer<=10ms}] isInitState=true
«state» NoBrakingRequired
«state» StartBraking
«state» EBSCalculation
«State» timeInvariant=[{EBSTimer<=5800us}]
Transition3 «transition» {readEventOccurrence?=TransComplStatus}
Transition2 «transition» {timeGuard=[{EBSTimer>=200us}],quantificationGuard=[{ACCCloseTargetDetected==0}],writeEventOccurrence!=TransRequest}
Transition4 «transition» {timeGuard=[{EBSTimer==0ms}]}
Transition5 «transition» {readEventOccurrence?=TransComplStatus}
Transition1 «transition» {timeGuard=[{EBSTimer>=200us}],quantificationGuard=[{ACCCloseTargetDetected==1}],writeEventOccurrence!=TransRequest}

Figure 10. Declaring the temporal constraint of a system function.

The logical time conditions and read&write synchronizations in this temporal constraint are defined based on the occurrences of some execution events. See Figure 11 for a snapshot.

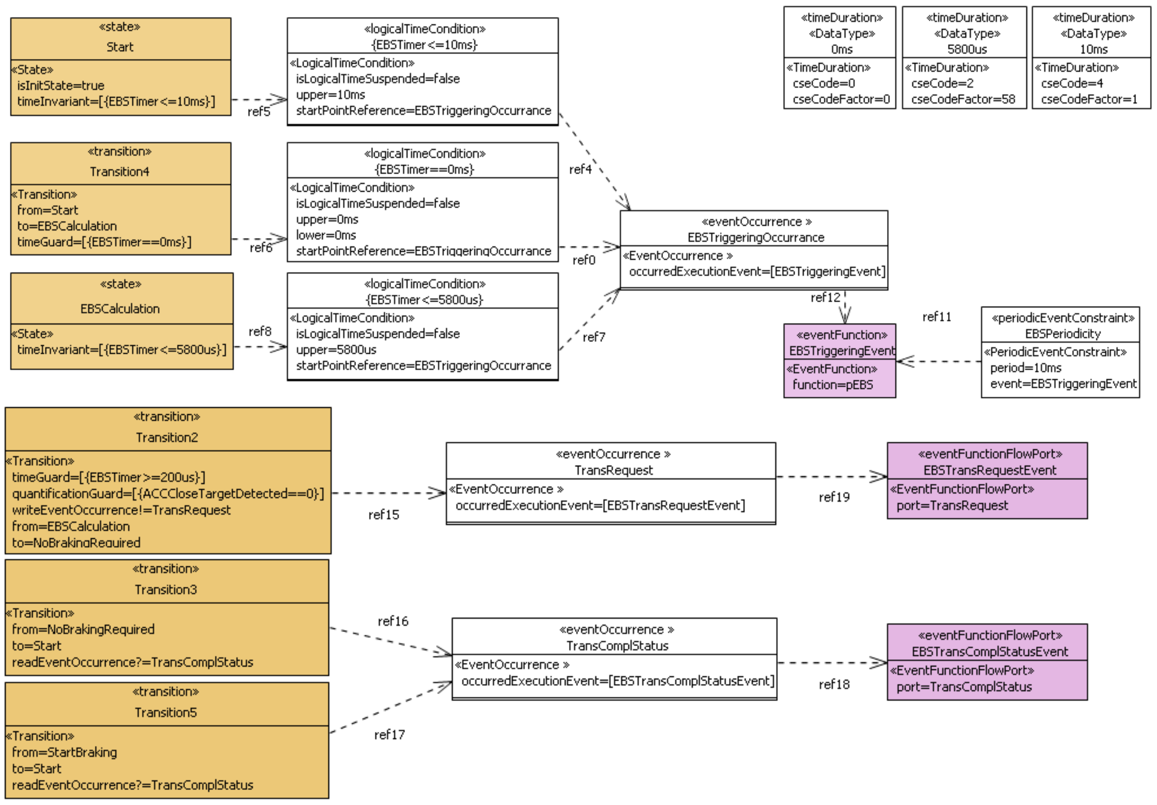

«state» Start
«State» isInitState=true timeInvariant=[{EBSTimer<=10ms}]
«transition» Transition4
«Transition» from=Start to=EBSCalculation timeGuard=[{EBSTimer==0ms}]
«state» EBSCalculation
«State» timeInvariant=[{EBSTimer<=5800us}]
«logicalTimeCondition» {EBSTimer<=10ms}
«LogicalTimeCondition» isLogicalTimeSuspended=false upper=10ms startPointReference=EBSTriggeringOccurrance
«logicalTimeCondition» {EBSTimer==0ms}
«LogicalTimeCondition» isLogicalTimeSuspended=false upper=0ms lower=0ms startPointReference=EBSTriggeringOccurrance
«logicalTimeCondition» {EBSTimer<=5800us}
«LogicalTimeCondition» isLogicalTimeSuspended=false upper=5800us startPointReference=EBSTriggeringOccurrance
«timeDuration» «DataType» 0ms «TimeDuration» cseCode=0 cseCodeFactor=0
«timeDuration» «DataType» 5800us «TimeDuration» cseCode=2 cseCodeFactor=58
«timeDuration» «DataType» 10ms «TimeDuration» cseCode=4 cseCodeFactor=1
«eventOccurrence» EBSTriggeringOccurrance
«EventOccurrence» occurredExecutionEvent=[EBSTriggeringEvent]
«eventFunction» EBSTriggeringEvent
«EventFunction» function=pEBS
«periodicEventConstraint» EBSPeriodicity
«PeriodicEventConstraint» period=10ms event=EBSTriggeringEvent
ref5
ref6
ref8
ref4
ref0
ref7
ref12
ref11
«transition» Transition2
«Transition» timeGuard=[{EBSTimer>=200us}] quantificationGuard=[{ACCCloseTargetDetected==0}] writeEventOccurrence!=TransRequest from=EBSCalculation to=NoBrakingRequired
«eventOccurrence» TransRequest
«EventOccurrence» occurredExecutionEvent=[EBSTransRequestEvent]
«eventFunctionFlowPort» EBSTransRequestEvent
«EventFunctionFlowPort» port=TransRequest
ref15
ref19
«transition» Transition3
«Transition» from=NoBrakingRequired to=Start readEventOccurrence?=TransComplStatus
«transition» Transition5
«Transition» from=StartBraking to=Start readEventOccurrence?=TransComplStatus
«eventOccurrence» TransComplStatus
«EventOccurrence» occurredExecutionEvent=[EBSTransComplStatusEvent]
«eventFunctionFlowPort» EBSTransComplStatusEvent
«EventFunctionFlowPort» port=TransComplStatus
ref16
ref17
ref18

Figure 11. Defining the logical time conditions and read&write synchronizations for a state-machine.

Due to the semantics alignment, the specifications of temporal constraints can be exported and transformed to external models of automata and thereby analyzed through related model-checking engines. Figure 12 shows the corresponding analytical model in UPPAAL for the temporal constraint specification shown in Figure 10. Compared to the analytical model in UPPAAL, the EAST-ADL temporal constraint declaration complements with detailed architecture information and allows an integration of many related architectural aspects for the purpose of architecture design and management.

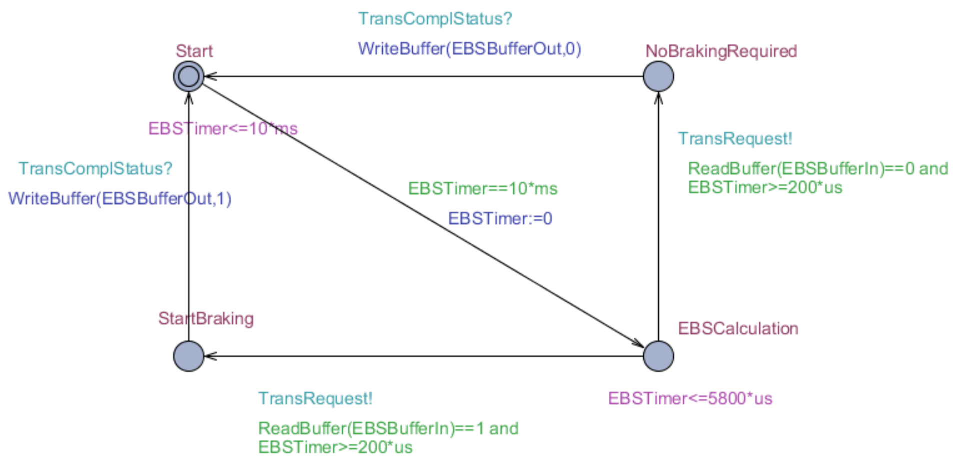


Figure 12. The corresponding UPPAAL analytical model for an EAST-ADL temporal constraint.

3.2.4 Computation Constraint

Computation constraints provide support for the declarations of computation restrictions. They are useful for defining the required logical transformations from input data to output data, as well as the expected causal sequences across such data transformations. Computation constraints are comparable with the UML activity and sequence behavior specification. See Figure 13 for an overview of the related meta-model definitions.

A specification of computation constraints is based on the following constructs:

- **ComputationConstraint** – the modeling construct for grouping the transformation and transformation path declarations in a behavior constraint.
- **LogicalTransformation** – the modeling construct for the declarations of logical computation transformations that determine some out-data (out) by processing some in-data (in) and local-data (contained). Such data are defined in terms of behavior attributes (Attribute). The corresponding value bounds of such data that must be hold before, after, and during the executions of a logical transformation are given by pre-, post-, and invariant conditions respectively. A logical transformation can also have time invariants (timeInvariants), stating the duration bounds when the transformation takes place. If a logical transformation is externally accessible (isClientServerInterface == true), it represents the operations declared in client-server interfaces (clientServerInterfaceOperation). A logical transformation can also have subordinate computation constraints (subComputationConstraint). (The expression attribute (expression) is a placeholder for the upcoming support for expressions of computation logics)
- **TransformationOccurrence** – the modeling construct for the declarations of invocations of logical transformations as the effects of state transitions and control flows. A transformation occurrence can also have a time condition (timeCondition), stating the time instances when the invocation happens. If a logical transformation is invoked, its in-data will be assigned with particular values by the invocation context *(*inQuantification).

As the consequence of transformation, the out-data will also be assigned with particular value (outQuantification).

- **LogicalPath** – the modeling construct for the declarations of the cause-effect paths connecting execution events, logical transformations, and logical events. One main advantage is that the internal causality of functions/components can now be merged explicitly with the related external execution events. When applied to a function/component, a logical path captures the control flow from some logical events (logicalStimulus) or execution events (precedingEventChain) to some other logical events (logicalResponse) or execution events (succeedingEventChain). In each logical path, some logical transformation can be invoked (TransformationOccurrence) either directly or in synchronization with state transitions. By describing the internal causality of a function/component, a logical path may refine an execution event chain (correspondingExecutionEventChain), which is primarily used to capture the causality of triggering, port reading and writing events. Logical paths can be combined in parallel (strand) or in sequence (segment).

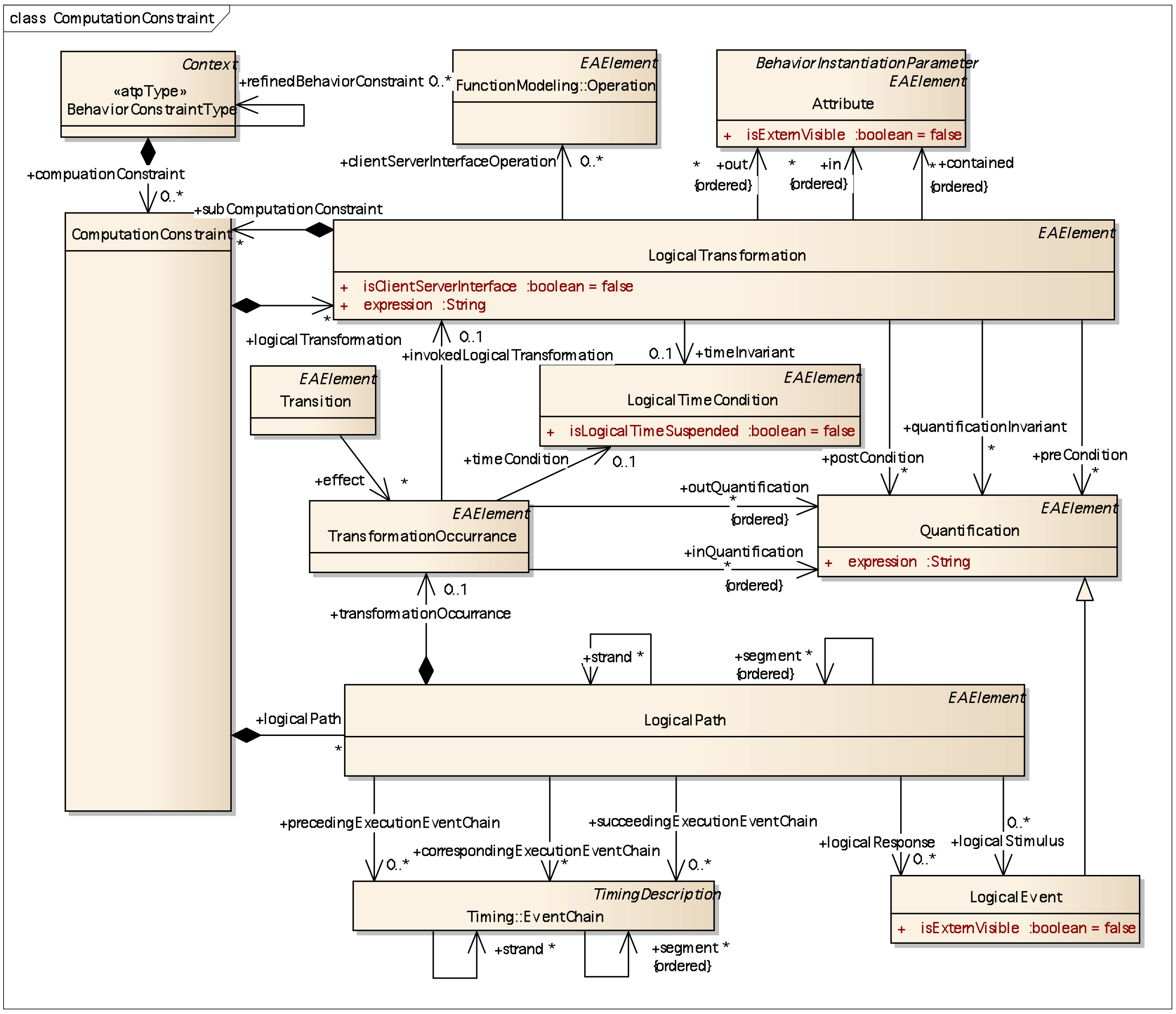

class ComputationConstraint
Context «atpType» BehaviorConstraintType
+refinedBehaviorConstraint 0..*
EAElement FunctionModeling::Operation
BehaviorInstantiationParameter EAElement Attribute
+ isExternVisible :boolean = false
+compuationConstraint
0..*
+subComputationConstraint
ComputationConstraint
+clientServerInterfaceOperation 0..*
* +out {ordered}
* +in {ordered}
* +contained {ordered}
EAElement LogicalTransformation
+ isClientServerInterface :boolean = false
+ expression :String
+logicalTransformation
0..1 +invokedLogicalTransformation
0..1 +timeInvariant
EAElement Transition
EAElement LogicalTimeCondition
+ isLogicalTimeSuspended :boolean = false
+quantificationInvariant
+timeCondition 0..1
+postCondition
+preCondition
+effect *
+outQuantification {ordered}
EAElement TransformationOccurrance
EAElement Quantification
+ expression :String
+inQuantification {ordered}
0..1 +transformationOccurrance
+strand *
+segment * {ordered}
+logicalPath
EAElement LogicalPath
+precedingExecutionEventChain 0..*
+correspondingExecutionEventChain *
+succeedingExecutionEventChain 0..*
+logicalResponse 0..*
+logicalStimulus 0..*
TimingDescription Timing::EventChain
+strand *
+segment * {ordered}
LogicalEvent
+ isExternVisible :boolean = false

Figure 13. ComputationConstraint and its properties in the proposed EAST-ADL2 Behavior Annex.

3.2.5 Instantiations of Behavior Constraint Types

A behavior constraint has both type and prototype(s), following the type-prototype pattern in EAST-ADL. While a type definition provides the template for a range of behaviors, a prototype definition specifies a particular behavior instance in a context. See Figure 13 for an overview of the related meta-model definitions.

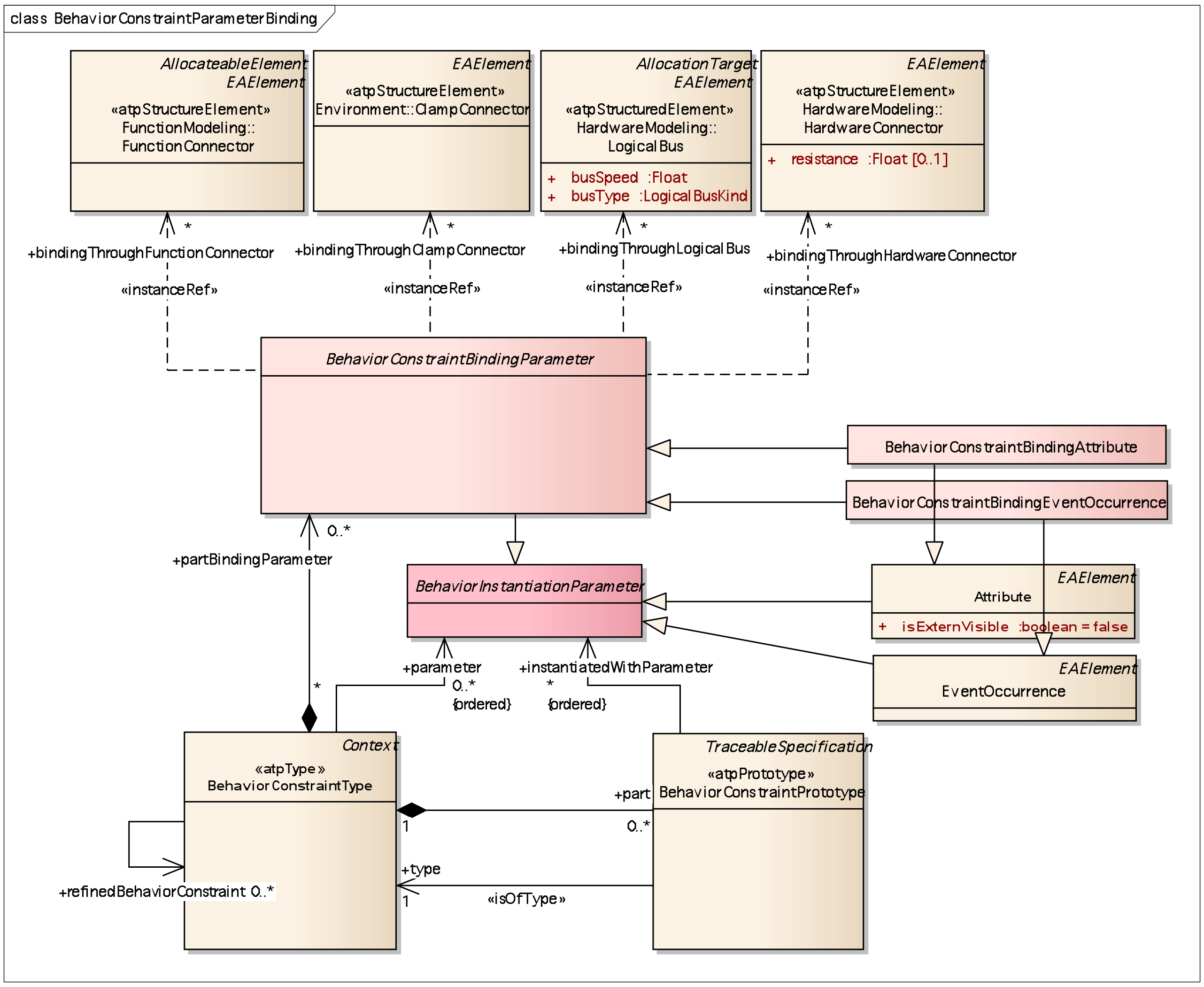

class BehaviorConstraintParameterBinding
AllocateableElement
EAElement
«atpStructureElement»
FunctionModeling::
FunctionConnector
EAElement
«atpStructureElement»
Environment::ClampConnector
AllocationTarget
EAElement
«atpStructuredElement»
HardwareModeling::
LogicalBus
+ busSpeed :Float
+ busType :LogicalBusKind
EAElement
«atpStructureElement»
HardwareModeling::
HardwareConnector
+ resistance :Float [0..1]
+bindingThroughFunctionConnector
+bindingThroughClampConnector
+bindingThroughLogicalBus
+bindingThroughHardwareConnector
«instanceRef»
BehaviorConstraintBindingParameter
BehaviorConstraintBindingAttribute
BehaviorConstraintBindingEventOccurrence
+partBindingParameter
0..*
BehaviorInstantiationParameter
EAElement
Attribute
+ isExternVisible :boolean = false
EAElement
EventOccurrence
+parameter
0..*
{ordered}
+instantiatedWithParameter
{ordered}
Context
«atpType»
BehaviorConstraintType
TraceableSpecification
«atpPrototype»
BehaviorConstraintPrototype
+part
0..*
1
+type
1
«isOfType»
+refinedBehaviorConstraint 0..*

Figure 14. BehaviorConstraintType, BehaviorConstraintPrototype, and the related modeling instantiation constructs in the proposed EAST-ADL2 Behavior Annex.

The support for the instantiations of behavior constraint types is based on the following constructs:

- **BehaviorConstraintPrototype** – the modeling construct for the declarations of the occurrence(s) of a behavior constraint type (type) in a particular context where it acts as a part (part).
- **BehaviorInstantiationParameter** – the modeling construct for the declarations of the parameters (parameter) that a behavior constraint type offer for its instantiations in terms of prototypes. During the instantiation, the parameters of a behavior constraint type will be bound to some contextual parameters (instantiatedWithParameter) and thereby be assigned with the values of those contextual parameters. A BehaviorInstantiation-Parameter can be a value attribute (Attribute), an event occurrence (EventOccurrence), or an internal binding parameter (BehaviorConstraintBindingParameter).

- **BehaviorConstraintBindingParameter** – the modeling construct for the declaration of parameters (partBindingParameter) that a behavior constraint type has for biding its parts. In effect, such parameters can be shared by all parts in the same context. Each binding parameter can have a structural correspondence (bindingThroughFunctionConnector, bindingThroughClampConnector, bindingThrough-LogicalBus, or bindingThrough-HardwareConnector), stating the structural channels through which the binding takes place. In the meta-model, the abstract binding parameter is further specialized into
    - **BehaviorConstraintBindingAttribute** – the contextual parameters that are value attributes.
    - **BehaviorConstraintBindingEventOccurrence** – the contextual parameters that are event occurrences.

See Figure 15 for a user-model example that shows how the behavior constraint type (EBS_BehaviorConstraint) of the example braking control function (EBS) is instantiated as a prototype (eBS_BehaviorConstraint) in a particular context (FunctionDesignArchitecture_ BehaviorConstraint). The supported part binding parameters in the context are given as a set of sharable event occurrences. Such binding parameters play the roles of synchronization connectors (i.e. rendezvous) with structural correspondences in terms of functional connectors or physical channels.

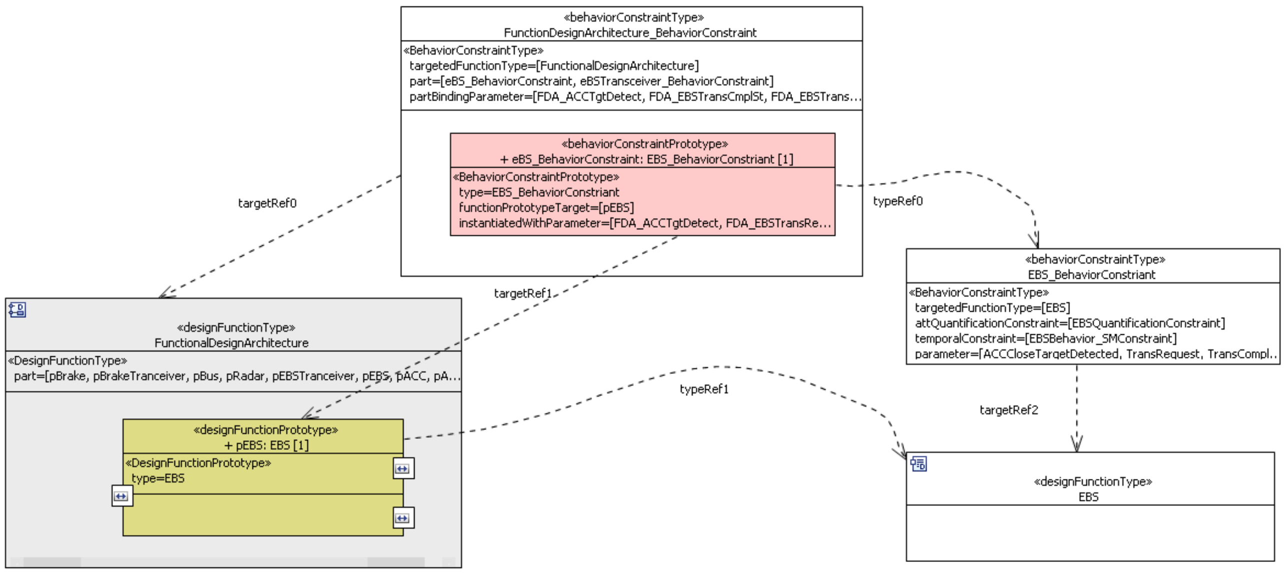

«behaviorConstraintType»
FunctionDesignArchitecture_BehaviorConstraint
«BehaviorConstraintType»
targetedFunctionType=[FunctionalDesignArchitecture]
part=[eBS_BehaviorConstraint, eBSTransceiver_BehaviorConstraint]
partBindingParameter=[FDA_ACCTgtDetect, FDA_EBSTransCmplSt, FDA_EBSTrans...
«behaviorConstraintPrototype»
+ eBS_BehaviorConstraint: EBS_BehaviorConstriant [1]
«BehaviorConstraintPrototype»
type=EBS_BehaviorConstriant
functionPrototypeTarget=[pEBS]
instantiatedWithParameter=[FDA_ACCTgtDetect, FDA_EBSTransRe...
targetRef0
typeRef0
targetRef1
«behaviorConstraintType»
EBS_BehaviorConstriant
«BehaviorConstraintType»
targetedFunctionType=[EBS]
attQuantificationConstraint=[EBSQuantificationConstraint]
temporalConstraint=[EBSBehavior_SMConstraint]
parameter=[ACCCloseTargetDetected, TransRequest, TransCompl..
«designFunctionType»
FunctionalDesignArchitecture
«DesignFunctionType»
part=[pBrake, pBrakeTranceiver, pBus, pRadar, pEBSTranceiver, pEBS, pACC, pA...
typeRef1
targetRef2
«designFunctionPrototype»
+ pEBS: EBS [1]
«DesignFunctionPrototype»
type=EBS
«designFunctionType»
EBS

Figure 15. Declaring the behavior constraint of a system function and instantiating the constraint in an architecture context.

Two or more behavior constraint prototypes in the same context are bound if they share the same binding parameters. One user-model example is shown in Figure 16. The behavior constraint of the braking control function (EBS) declares an ordered set of instantiation parameters:

[ACCCloseTargetDetected, TransRequest, TransComplSt]

In its prototype instantiation, such parameters are assigned through an ordered set of binding parameters:

[FDA_ACCCloseTargetDetected, FDA_TransRequest, FDA_TransComplSt]

In such a way, a behavioral binding of the braking control function (EBS) and the communication transceiver (Transceiver) is established. This is illustrated in Figure 16. Assume that the behavior constraint of transceiver function (EBS) has the instantiation parameter declaration: [TransRequest, TransComplSt]. It is instantiated with: [FDA_TransRequest, FDA_TransComplSt]. Under the circumstance, the read&write event occurrences of transitions in the respective state-machine constraints of these two functions will be sychronized. Figure 16 also shows the declarations of corresponding structural connectors of the binding parameters.

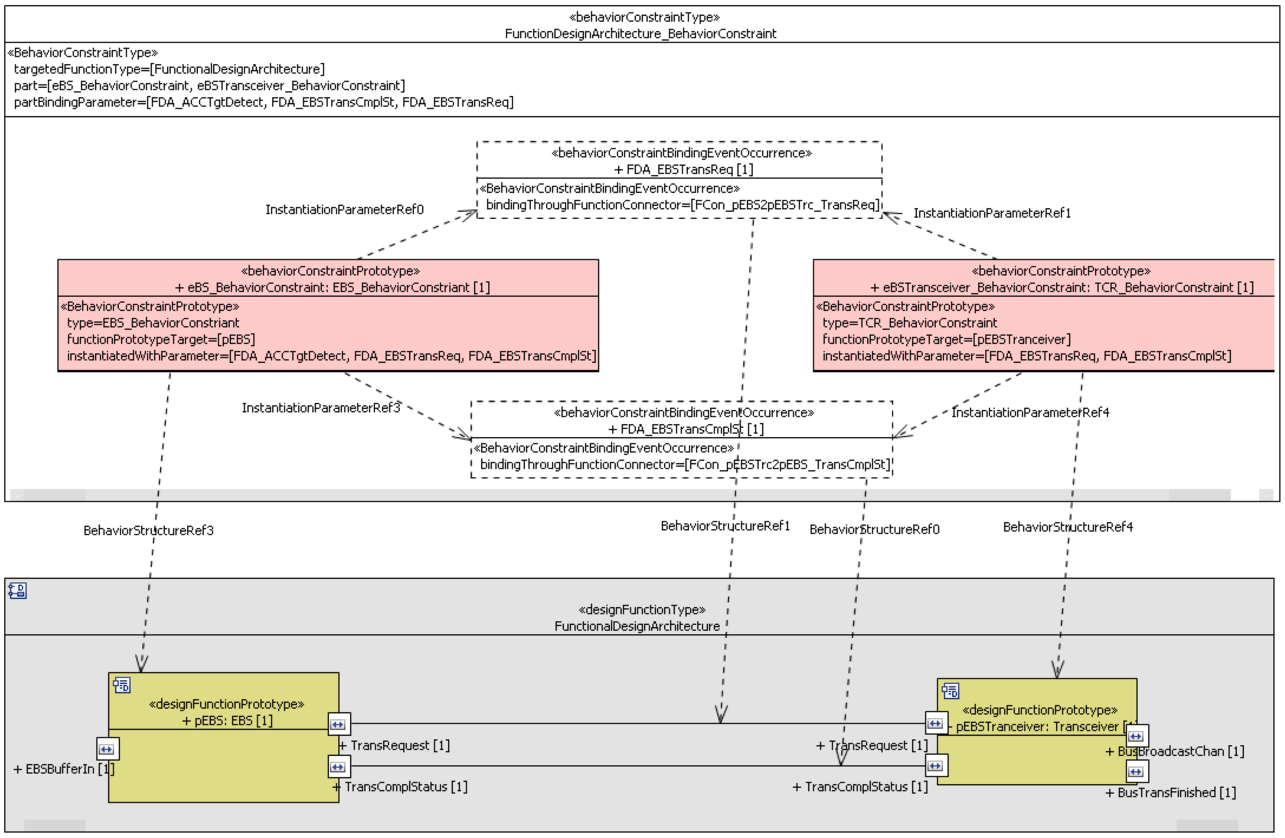

«behaviorConstraintType»
FunctionDesignArchitecture_BehaviorConstraint
«BehaviorConstraintType»
targetedFunctionType=[FunctionalDesignArchitecture]
part=[eBS_BehaviorConstraint, eBSTransceiver_BehaviorConstraint]
partBindingParameter=[FDA_ACCTgtDetect, FDA_EBSTransCmplSt, FDA_EBSTransReq]
«behaviorConstraintBindingEventOccurrence»
+ FDA_EBSTransReq [1]
«BehaviorConstraintBindingEventOccurrence»
bindingThroughFunctionConnector=[FCon_pEBS2pEBSTrc_TransReq]
InstantiationParameterRef0
InstantiationParameterRef1
«behaviorConstraintPrototype»
+ eBS_BehaviorConstraint: EBS_BehaviorConstriant [1]
«BehaviorConstraintPrototype»
type=EBS_BehaviorConstriant
functionPrototypeTarget=[pEBS]
instantiatedWithParameter=[FDA_ACCTgtDetect, FDA_EBSTransReq, FDA_EBSTransCmplSt]
«behaviorConstraintPrototype»
+ eBSTransceiver_BehaviorConstraint: TCR_BehaviorConstraint [1]
«BehaviorConstraintPrototype»
type=TCR_BehaviorConstraint
functionPrototypeTarget=[pEBSTranceiver]
instantiatedWithParameter=[FDA_EBSTransReq, FDA_EBSTransCmplSt]
InstantiationParameterRef3
InstantiationParameterRef4
«behaviorConstraintBindingEventOccurrence»
+ FDA_EBSTransCmplSt [1]
«BehaviorConstraintBindingEventOccurrence»
bindingThroughFunctionConnector=[FCon_pEBSTrc2pEBS_TransCmplSt]
BehaviorStructureRef3
BehaviorStructureRef1
BehaviorStructureRef0
BehaviorStructureRef4
«designFunctionType»
FunctionalDesignArchitecture
«designFunctionPrototype»
+ pEBS: EBS [1]
+ EBSBufferIn [1]
+ TransRequest [1]
+ TransComplStatus [1]
«designFunctionPrototype»
pEBSTranceiver: Transceiver [1]
+ TransRequest [1]
+ TransComplStatus [1]
+ BusBroadcastChan [1]
+ BusTransFinished [1]

Figure 16. Declaring the binding of behavior constraint prototypes in an architecture context.

3.3 Upcoming Activities

EAST-ADL could provide many benefits for analytical and architectural decision-making, information exchange and management in the development of FEV. As an ontology and formalism of embedded systems, the language would also constitute an important basis for the integration of state-of-the-art analysis methods and techniques from computer science, electronics&electrical engineering, and other related disciplines. This would allow a seamlessly integrated analysis support in the entire lifecycle of system development. For many advanced analysis, it is of critical importance that an alignment of EAST-ADL analytical models with the related analysis methods and tools is clearly defined.

Upcoming EAST-ADL support for the analysis of FEV will address the assessment of FEV properties, either of the system itself or of its operational situation, in both logical and physical domains. Such a support will allow enhanced language support for the elicitation of safety goals, the descriptions, verification and validation of safety requirements, as well as for the assessments of mode-sensitive system compositionality and composabiltiy in general. In the upcoming project period, an alignment of the proposed behavior constraint specification support with external formalisms and analysis engines including UPPAAL, SPIN, Modelica will be supported. Current EAST-ADL provides analytical modeling support in regard to timing, faults and error propagation. Future work will investigate the analysis leverage by adopting such external formalisms and engines via the proposed modeling enhancement for behavior constraint description.

3.4 Modeling Concepts for Supporting Timing Analysis

Timing modeling in EAST-ADL results from the work done in the TIMMO project, which produced a dedicated language called TADL (see TIMMO deliverable D6 for instance available from http://www.timmo.org/). TADL concepts were integrated in the course of the ATESST2 project in the EAST-ADL language.

The TIMMO project produced a fair analysis of the needs in this topic, which explains that no language change is envisioned. Rather a pragmatic study of the analysis of models with dedicated tools is at stake.

For the record, in EAST-ADL language, three packages structure the timing concepts: *Timing* which defines core elements and their organization, *Events* which lists various kinds of events that can be associated to structural elements, such arrival of data on ports, and *TimingConstraints* which lists all possible constraints one can model – delays, synchronization, etc.

The modeling principle is the following. *TimingConstraints* are associated to an *EventChain*, which defines the scope of the constraint. The *EventChain* in turn relates to various *Events*, such as arrival of a data on a Port. *Events* point to *structural* elements, such as Ports or Functions.

Based on this modelling concepts timing analysis can be performed. However TADL does not cover implementation level concepts such as tasks, which are usually needed to conduct a detailed schedulability analysis for instance. Thus the analysis that one can perform at this level is restricted to feasibility assessment regarding e.g. response times, age, synchronization and resource load balancing and assessing. This provides an interesting insight on how the software implementation could later be defined. To go beyond this, one needs to go to the implementation level (i.e. Autosar architecture) where the allocation of execution to tasks is defined. For this, MARTE provides a good basis. In the experiments done on timing analysis in ATESST2, the extra information needed in EAST-ADL models were added using MARTE constructs.

In MAENAD a specific task WT4.2 is dedicated to the study of the mapping of EAST-ADL constructs to MARTE. The main objective is to be able to minimize the extra information needed to perform schedulability analysis on EAST-ADL models (see D4.2.1).

A more detailed overview of the timing concepts involved in both EAST-ADL/TADL and MARTE can be found in D3.2.1.

An analysis of the mapping between EAST-ADL and MARTE concepts can be found in D4.2.1.

As a result it is not planned to add any language constructs dedicated to timing analysis, rather map the existing ones onto MARTE, which is a more generic language. The missing part, which was mentioned above: the task level will rather be dealt with different assumptions on function to task allocation or the use of the existing GenericConstraint element in the EAST-ADL language (GenericConstraints extension) to make such assumptions explicit in the model. At most for the time being, a potential revision of GenericConstraintKind with additional Literals might be envisioned (e.g. functionAllocationSameTask, functionAllocationDifferentTask, etc.).

4 Modeling Concepts for Optimization Support

Another objective of MAENAD is to extend the EAST-ADL language with support for multi-objective optimisation, including the definition of standard architectural patterns that can be used in optimisation to improve system characteristics like dependability and performance. This section first provides a brief overview of basic optimisation concepts before then describing current EAST-ADL support for optimisation and detailing the various concepts that still need to be introduced to the language to achieve this objective.

4.1 Overview of general optimisation concepts

Contemporary model-based systems analysis techniques (see section **Fehler! Verweisquelle konnte nicht gefunden werden.**) allow a wealth of information to be obtained about a system. For example, dependability analysis can be used to both determine the probable causes of a system failure and also obtain an estimate of the probability of that failure occurring. Such information can be extremely valuable while designing a system and can be used to guide the future iterations of the design, e.g. to produce a more mature model in which the effects of a critical failure identified in earlier design models are mitigated or avoided.

This capability is enhanced due to the fact that many systems analysis techniques can now be semi-automated by software tools; this is especially true when such tools are compatible with the modelling language used to describe the system model, as the model can then be subjected to analysis directly. Automated analysis allows models to be rapidly evaluated according to a variety of different criteria, e.g. performance, safety, maintainability, cost etc. This allows designers to prototype and test out different design options as part of an iterative design process.

However, modern systems (particularly electronic ones) are typically sufficiently complex that only a small number of potential options can be investigated in this way, since it takes time for a designer to produce a new design variant, analyse it, and evaluate the resulting data. The set of all possible design variants is known as the **design space** or **search space**. This task is made harder when there are multiple design **objectives** - i.e. attributes of the system to be improved - which may conflict, e.g. the goal may be to maximise performance and safety while minimising cost as much as possible. This results in complex trade-offs which can be difficult to evaluate manually. Taken together, the act of searching a large potential design space to obtain a good solution that features a desirable balance of attributes is known as a **multi-objective optimisation problem**.

In multi-objective optimisation, automated algorithms are typically employed to search the design space according to various heuristics that aim to quickly find **solutions** - valid design variants - that offer optimal or near-optimal attributes without having to investigate all possible designs (a task which may be impossible to complete in a reasonable time even for modern computers and software). Different algorithms exist, but typically they operate in an iterative manner, evaluating a given set of solutions to determine which are the best and keeping them while discarding the rest, hopefully leading to a new iteration with a new set of superior solutions.

It is important to note that in a multi-objective optimisation problem, there is normally not one **optimum** solution since the different objectives may be mutually exclusive, i.e. to improve one objective attribute, it may be necessary to sacrifice another objective attribute. Therefore, multi-objective optimisation algorithms typically produce a set of 'optimal' solutions that feature a range of attributes that balance the different objectives in different ways. These are known as the **Pareto**

solutions and are based on the concept of **dominance**; a solution is a Pareto solution if it is better than any other solution in at least one objective attribute and no worse than any other solution in the others, in which case it is said to dominate the other solutions. Thus a Pareto set can include solutions with one objective maximised and the others minimised, or solutions with all objectives more evenly balanced, and all can be described as 'optimal'. This concept can be seen in the figure below:

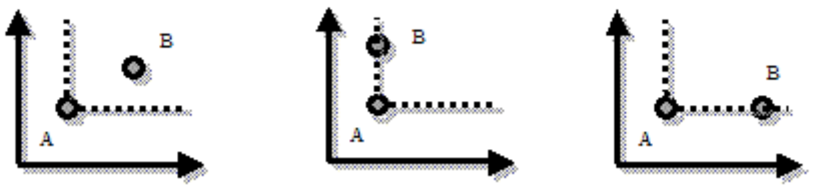


Figure 2. Pareto dominance

Here, solution A is always better in at least one axis (representing a particular optimisation objective, e.g. safety or performance or cost). For example, assuming the X axis is unavailability and the Y axis is cost, then in the middle graph, A has the same unavailability as B but is cheaper, while in the right-most graph, A is no cheaper but has a significantly smaller unavailability. A is therefore said to dominate B.

When the different solutions are plotted on a graph, the dominant Pareto solutions form a curve known as the **Pareto frontier**, as can be seen below:

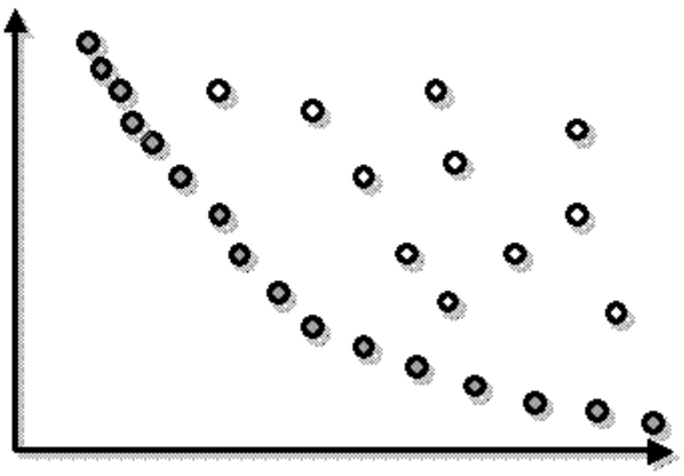


Figure 3: Pareto frontier

Here the shaded dots are Pareto solutions that dominate the non-shaded dots.

One of the key concepts in optimisation of this type, particularly when trying to automate it, is the ability to define and explore the design space. Different design **variants** can be created by altering a particular aspect of a system; for example, dependability may be improved by replicating a critical component to achieve redundancy, but cost may also be increased as a result. The design space therefore contains a set of models that feature all possible design variants. Defining this design space means highlighting areas in the system model where other variants are possible. This can be achieved through the use of **variability** mechanisms, e.g. to indicate that there are different possible implementations of a given subsystem, each with different characteristics (so one may have better performance but cost more, another may be cheaper but may suffer in performance). Typically, variants can be defined in one of three ways:

1. **Substitution** - A given component or subsystem is substituted for another that has different objective attributes (e.g. safety, cost, performance). A substitute must be **functionally equivalent** to be substitutable, i.e. it must perform the same task (thus a substitute can never have less functionality than the element it replaces, only either the same or more functionality). This does not mean that the function must be carried out in the same way, however; for example, an electronic braking subsystem may be replaced by a hydraulic version. The different substitutable options may be thought of different **implementations** of a given element/subsystem.
2. **Replication** - A given component or model element may be duplicated to achieve redundancy. Such replicants are typically connected in a parallel configuration so that failure of one replicant does not lead to failure of the whole subsystem.
3. **Both** - More complex optimisation is possible when the design space features a combination of both substitution and replication. Thus, for example, a design variant may replace a given element with two parallel elements, one of which may be the same as the original, and the other of which is substituted for a different implementation.

These different approaches to creating design variation are illustrated in the figure below:

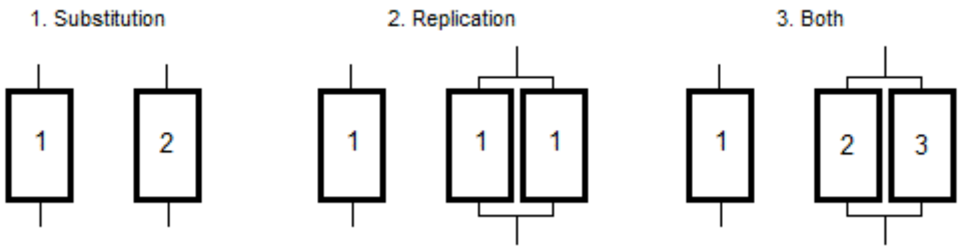


Figure 4. Substitution and Replication strategies

Note that in more complex approaches, substitution can also be hierarchical; for example, a single component may be substituted for another component or may be substituted for a subsystem with an entire sub-architecture, thus allowing for more complex replication strategies (like voters, monitors, or other parallelisms). Such configurations can often be stored in a library and reused.

To be used in the optimisation process, it must be possible to represent these design variants using an **encoding** strategy. A simple encoding may be to merely represent each component implementation with a number, thus 1111 could be a simple model in which no substitution has taken place, and 1112 a model in which the last component has been substituted for a second implementation/version. However, to allow more complex design spaces to be explored (e.g. featuring substitution of subsystems or more sophisticated replication strategies), the encodings become correspondingly more complex. One option is to use a hierarchical encoding such as a **tree encoding**, which echoes the hierarchical structure of the design model.

Once the design space is defined and can be explored using algorithms and encodings, the algorithm must be able to **evaluate** a given design variant according to the optimisation objectives; this requires the use of system analysis techniques. These can range in complexity from simple summations of component costs to more elaborate timing and safety analyses. The algorithm needs to be able to analyse a design variant for each objective attribute (e.g. cost, performance) to allow it to compare that design variant against other variants, and thus determine whether it is a dominant or optimal design and decide whether it should be kept for future optimisation iterations or discarded.

The optimisation process itself can often continue almost indefinitely, assuming the design space is sufficiently large; therefore in general it is set to run for a given number of iterations or a given time period, after which it returns the best solutions it has discovered so far. These can then be examined by the designer and used as the basis for the next iteration of the overall design process.

Thus to be able to perform multi-objective optimisation of a system design, it must be possible to represent and make use of the above concepts; in particular, it has to be possible to:

- Create the design space by defining different possible design variants through the use of replication and substitution.
- The possibility to manually define design variants explicitly.
- Allow the optimisation algorithm to explore the design space by representing the variants as encodings.
- Enable evaluation of the different variants according to the optimisation objectives by means of model-based analysis techniques.
- The product variability space and the take rate of each combination shall be included in the optimization criterion.

4.2 Current EAST-ADL support for optimisation concepts

Although optimisation was investigated as part of ATESST2, for the most part the ideas were not mature enough to be used as input to the language and tool development. However, there are existing areas of the EAST-ADL semantics that may prove valuable in supporting multi-objective optimisation. In particular, EAST-ADL features some sophisticated variability mechanisms; while these were originally designed to be able to represent product line variability, in theory they can also be used to represent different design variants in an optimisation context.

Variability in EAST-ADL is based - at the most abstract level - upon the concept of a **feature**, which can be present in some product lines and absent in others. EAST-ADL provides a range of variability management features to allow more complex configurations of features to be represented, including dependencies between features (possibly hierarchical) and duplication of features. Variability of features is primarily defined using **feature models** and **configuration links**, which connect feature models and define the dependencies between them. Variability management concepts in EAST-ADL are more fully described in the following section of this document, but in the context of optimisation, they provide a promising mechanism for which to define different optimisation design variants by means of substituting one design element for another element or hierarchy of elements. Variability management is also designed to be able to ensure substitutability of features when required, which is vital if it is to be employed in an optimisation context.

The evaluation stage of optimisation consists of a combination of analysis techniques designed to analyse the different optimisation objective attributes. The use of model-based system analysis techniques is described in the preceding section and considerable work has already been done in developing concepts in EAST-ADL to support various analysis techniques, including behavioural analysis, performance and timing analyses, and safety and dependability analyses.

4.3 Concepts and developments required to achieve optimisation support in EAST-ADL

The primary challenge in meeting the optimisation objective in MAENAD is to combine and link existing concepts, such as variability and the various analysis techniques, into an overall optimisation process. This means extending and/or refining the variability mechanisms to allow them to support the definition of optimisation search spaces in addition to their primary role of modelling product line variability and enabling the analysis techniques to analyse these design variants.

There are three main areas which must be further developed if this is to be achieved:

4.3.1 Defining the design space

Firstly, it has to be possible to create an EAST-ADL design model - most likely at the analysis and/or design levels, since sufficient data for evaluation must also be present - that has a number of different variability points and that allows for different design variants to be explored. Existing variability mechanisms may need to be extended or may be already sufficient; this needs to be investigated further. One of the primary concerns here is that the variants must be substitutable, i.e. that one variant is functionally equivalent to another; this is not always the case in normal variability management, where e.g. a feature may be present in one product line but absent in another. Thus, EAST-ADL must be enhanced with means to define which variations represent such optional functionality (a.k.a. “product line variability”, that does not define an optimization choice) and which variations represent alternative but functionally equivalent realizations of a feature (a.k.a. “design space variability”, that defines the optimization space). A potential solution for this is the use of binding time attribute.

Furthermore, it must be possible to encode the design variants so that they can be automatically explored by the optimisation algorithm without unnecessary complexity; in practice one major difficulty in automatic optimisation is the connections between substituted and/or replicated components, e.g. one implementation may not have the same interface (i.e. number and type of inputs & outputs) as another. It will be necessary to resolve this for automatic optimisation to produce valid and sensible results.

Finally, although the original design model used as input to the optimisation process should contain the necessary variability to define the design space (and product space with take rates), the design solutions identified by the optimisation should have had this variability resolved such that the solution represents a single possible configuration of the system. This is perhaps analogous to the concept of binding in variability and if the optimisation algorithm is to be able to perform this automatically, it needs to be able to resolve/bind the variability in the model in such a way that each resulting product variant is still valid and thus must still contain any necessary functionality and meet any potential requirements and constraints. Here there may be potential overlap with the requirements capabilities of EAST-ADL, e.g. safety constraints and requirements (like ASILs) in the safety domain, timing requirements in the performance domain etc.

4.3.2 Evaluating the designs

To be used effectively in optimisation, each design variant has to be evaluated according to each of the objective attributes being optimized (there is a need to specify whether on objective is to be maximized, minimized and/or if it must satisfy a requirement to be “within bounds”; there is a need to identify which constraints are included in the optimization, probably through the VVCase construct). Thus the original design model has to contain all the different attribute data necessary to describe each variant. Note that this does not necessarily mean the model should itself contain all possible outcomes, but should provide enough information for the analysis techniques to determine the final attributes and thus allow for evaluation to take place. For example, it is infeasible for a single design model with, say, 10,000 variants to provide an estimate for unavailability for each variant within the model itself; instead, it should provide unavailability for each design option (i.e. for all possible component/function implementations), so that the safety analysis techniques can use this raw information to arrive at an estimate of the unavailability for that design variant as a whole.

Furthermore, this evaluation information needs to be stored in the resulting solution somehow, so that the optimisation algorithm can both determine its dominance and thus decide whether it should be retained as an optimal solution or not and also so that, if it is retained as a solution, the designer can see what the attributes of the solution are. Note that this does not necessarily mean that a new design model has to be created with all variability resolved, only that the configuration options should be recorded together with the analysis results. Thus for example, a simple design model with only one function that can be implemented in one of three ways may produce three solutions, each of which representing a different trade off. Each solution should provide enough information for the designer to be able to see its overall objective attributes (e.g. timing, safety, cost) and also allow the designer to produce an actual EAST-ADL model by configuring the original model according to the encoding of the given design solution (in this example, by telling the designer which implementation of the function was chosen in each case).

4.3.3 Develop an optimisation algorithm

Although there are different possible optimisation algorithms, each with various strengths and weaknesses, investigation in ATESST2 focused on the use of particular forms of genetic algorithms to perform the multi-objective optimisation. The HiP-HOPS optimisation technique uses genetic algorithms to support optimisation with different discrete objectives (compared to some other techniques that merge objectives into a single evaluation figure, for instance) and thus create a set of Pareto optimal solutions.

However, even if the general algorithm is known, there are many possible tweaks and variations to the algorithm to enhance and optimise its performance and allow it to more efficiently explore the search space and obtain good solutions. Most optimisation algorithms have a variety of parameters that govern their usage, e.g. for genetic algorithms, the number of generations (i.e. iterations) and size of each population (i.e. set of solutions) can be tweaked, as can the rules governing the exploration of new solutions and the rules governing which solutions are kept. Furthermore, these types of options and parameters may need to be altered or refined to ensure they are compatible with the technology used to define the search space and evaluate the resulting design variants.

4.3.4 Language Concepts and Examples

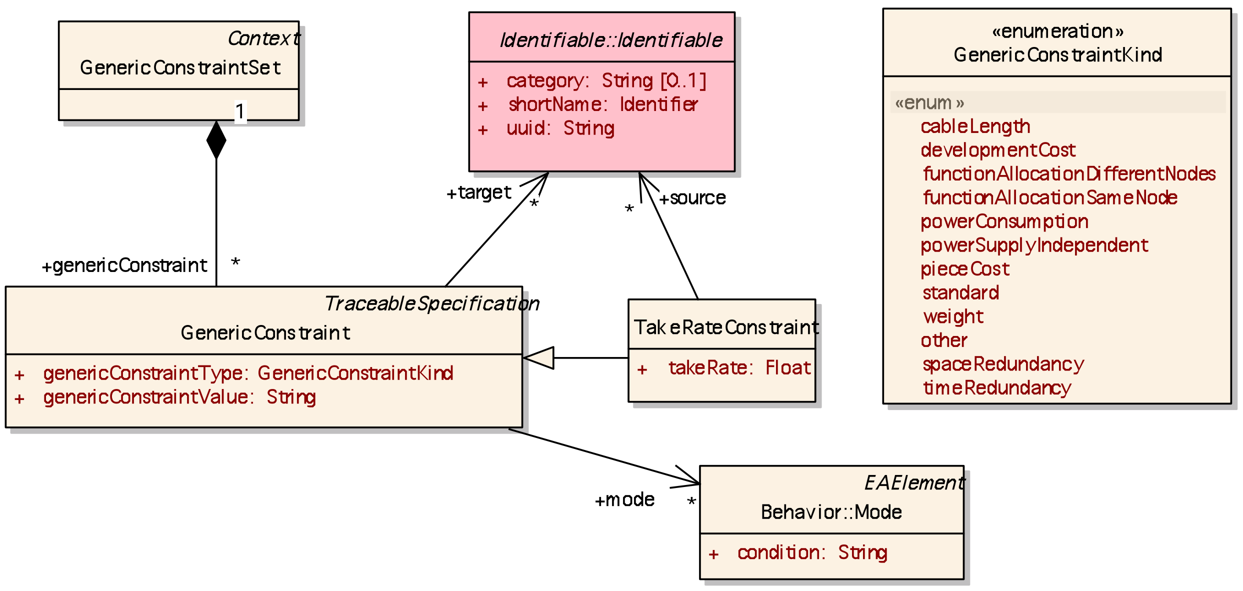


Figure 17. Constraints relevant for optimization

The number of components produced and their cost is a critical parameter for optimization. The total cost is development cost for each component type plus the sum of piececost multipled by piece count summed over all component types.

To know the piece count, it is necessary to define the absolute or relative number of elements in the feature tree or in the artifact model. A constraint solver can compute the number of elements of any given component based on such constraints, and it can also warn if the model is underspecified or inconsistent.

In the example in Figure 19, 20000 vehicles are produced. Since 16% of all vehicles go to the US market, and 50% of those have trim level Prime, it is possible to deduce that 1600 such vehicles with will be produced. 10% of all vehicles will be Plus, so these represent another 2000 vehicles. Both trim levels have ABSPlus which means 3800 PlusECUs will be produced and thus 16200 Standard ECUs.

Having established the number of components produced, it is possible to take this into account during optimization: Solutions with good price and performance for high volume products are favored before solutions that only benefit low volume products.

For example, if a high-volume vehicle needs an advanced component, while the low volume vehicle can do with a simpler component, it may well be better to equip all vehicles with the advanced component. This may also open up for after-sales revenue by selling upgraded functionality that relies on the better ECU. Figure 18 shows that the eliminated fixed cost for the low-end ECU compensates for the higher piece cost. This is consistent with Figure 19, as there is no configuration decision that states whether standard or plusECU shall be used. In the takerate model, it is undefined whether standardECU or PlusECU is used, unless the ABSPlus is selected in which case PlusECU is mandatory.

```
50*3800+300000 + 45*16200+300000=1519000
50*3800+300000 + 50*16200+0=1300000
```

Figure 18. Total Product Line cost for same and different ECU

The assumption is that the stated rate constraints are part of the same GenericConstraintSet and thus consistent. Further, the semantics assumption is that the root element of each product feature tree represent the all vehicles.

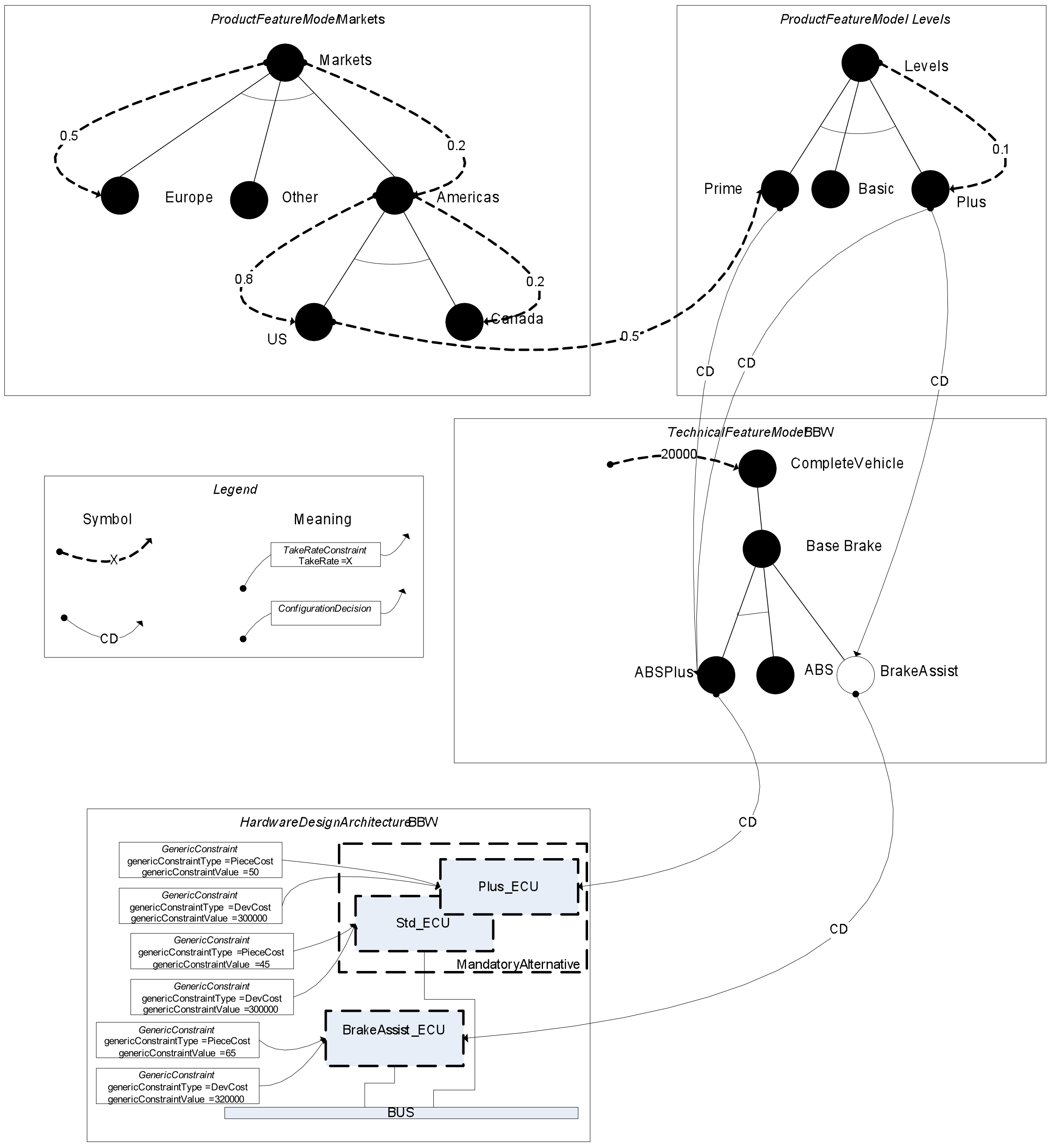

ProductFeatureModel Markets
Markets
0.5
0.2
Europe
Other
Americas
0.8
0.2
US
Canada
0.5
ProductFeatureModel Levels
Levels
0.1
Prime
Basic
Plus
CD
CD
CD
TechnicalFeatureModel BBW
20000
CompleteVehicle
Base Brake
ABSPlus
ABS
BrakeAssist
Legend
Symbol
Meaning
X
TakeRateConstraint
TakeRate =X
CD
ConfigurationDecision
HardwareDesignArchitecture BBW
GenericConstraint
genericConstraintType =PieceCost
genericConstraintValue =50
GenericConstraint
genericConstraintType =DevCost
genericConstraintValue =300000
GenericConstraint
genericConstraintType =PieceCost
genericConstraintValue =45
GenericConstraint
genericConstraintType =DevCost
genericConstraintValue =300000
GenericConstraint
genericConstraintType =PieceCost
genericConstraintValue =65
GenericConstraint
genericConstraintType =DevCost
genericConstraintValue =320000
Plus_ECU
Std_ECU
MandatoryAlternative
BrakeAssist_ECU
BUS
CD
CD

Figure 19. Feature models and constraints to consider in optimization

One way to resolve the take rate constraints is to translate it to a constraint programming problem and use a suitable solver. Figure 20 shows how a constraint programme in Prolog may look. It also contains a minimization criteria for cost.

```
% Prolog CLP(FD) pseudo-code
:- use_module(library(clpfd)).
% The feature tree as constraints
feature_tree(TotVehicles,Cost,Variables):-
        % All variables should be greater than zero
        TotVehicles#>=0,
        VehiclesOther#>=0,
        VehiclesEurope#>=0,
        VehiclesAmerica#>=0,
        VehiclesUS#>=0,
        VehiclesCanada#>=0,
        Prime#>=0,
        Basic#>=0,
        Plus#>=0,
        BaseBrake#>=0,
        ABSPlus#>=0,
        ABS#>=0,
        BrakeAssist#>=0,
        PlusECU#>=0,
        StdECU#>=0,
        BrakeAssistECU#>=0,
        PlusECUYesNo in 0..1,
        StdECUYesNo in 0..1,
        BrakeAssistECUYesNo in 0..1,
        Cost#>=0,
        % Constraints from Markets
        VehiclesEurope#=TotVehicles*0.5,
        VehiclesAmericas#=TotVehicles*0.2,
        TotVehicles#=VehiclesEurope+VehiclesAmericas+VehiclesOther,
        VehiclesUS#=VehiclesAmerica*0.8,
        VehiclesCanada#=VehiclesAmerica*0.2,
        VehiclesAmerica#=VehiclesUS+VehiclesCanada, % redundant constraint
        % Constraints from Levels
        TotVehicles#=Prime+Basic+Plus,
        Prime#=0.5*VehiclesUS,
        Plus#=0.1*TotVehicles,
        % Constraints from BBW feature tree
        BaseBrake#=TotVehicles,
        BaseBrake#=ABSPlus+ABS,
        ABSPlus#=Prime+Plus,
        BrakeAssist#=Plus,
        %Constraints from BBW architecture design
        StdECU#<=ABS
        PlusECU#>=ABSPlus
        PlusECU#<=ABSPlus+ABS
        StdECU#+PlusECU#=ABS+ABSPLus
        BrakeAssisECU#=BrakeAssist,
        % Objective function (single objective)
        PlusECU#>0 #<=> PlusECUYesNo,
        StdECU#>0 #<=> StdECUYesNo,
        BrakeAssistECU#>0 #<=> BrakeAssistECUYesNo,
        Cost#=PlusECUYesNo*300000+PlusECU*50+StdECUYesNo*300000+StdECU*45+BrakeAssistECUYesNo*320000+Brak
eAssistECU*65,
        Variables=[VehiclesEurope,VehiclesAmerica,VehiclesUS,VehiclesCanada,Prime,Basic,Plus,BaseBrake,ABSPlus,ABS,Bra
keAssist,PlusECU,StdECU,BrakeAssistECU].
% Find minimum cost solution for given number of vehicles
find_cost(Cost,Variables):-
        feature_tree(20000,Cost,Variables),
        labeling([minimize(Cost)],Variables).
% Find maximum number of vehicles for maximum given cost
find_vehicles(TotVehicles,Variables):-
        Cost#=<1600000,
        feature_tree(TotVehicles,Cost,Variables),
        labeling([maximize(TotVehicles]),Variables).
```

Figure 20. Prolog code corresponding to the TakeRate constraints

4.3.5 Summary

In summary, to achieve the objective of supporting automatic multi-objective optimisation of EAST-ADL models, it will be necessary to investigate and develop the following concepts further:

- Definition of the design space:
    - Use existing variability mechanisms where possible, with extensions where necessary
    - Ensure substitutability of one design variant for another
    - Develop an encoding strategy to allow the design space to be represented in the optimisation algorithm
    - Allow the variability to be resolved/bound to a particular configuration to provide a design solution with a given set of objective attributes
- Evaluation of the design variants:
    - Make use of existing developments towards system analysis techniques in EAST-ADL where possible, and refine and develop them further where necessary
    - Ensure that the results can be used by the optimisation algorithm to determine dominance
    - Store the results for future reference by the designer
- Development of multi-objective optimisation heuristics to allow the algorithm to efficiently explore the design space and rapidly arrive at suitable Pareto optimal solutions
    - Refine the algorithm to take into account design space and evaluation requirements
    - Experiment with the different algorithm parameters to improve efficiency
    - Develop a initial methodology governing the use of optimisation in an EAST-ADL model

5 Modeling Concepts for Variability Management

5.1 The Role of Variability Management in MAENAD

Variability management is not a primary objective of the MAENAD project. It is not explicitly mentioned in the list of project objectives in the MAENAD description of work (pages 9, 10), because at the end of ATESST2 the variability management in EAST-ADL was considered fairly complete.

However, for several reasons variability is still an important topic for MAENAD:

1. Variability management is an important part of the EAST-ADL and the overall consolidation and maintenance of EAST-ADL is an aim in MAENAD. Therefore, also the maintenance and consolidation of the variability-related concepts is within the project's scope.
2. In automotive industry, most development projects are dealing not with a single system but a whole family of similar but distinct products. The resulting variability in system development poses a significant challenge to most of the primary objectives of MAENAD. Therefore the solutions devised in MAENAD to tackle the project's primary objectives also have to take into account variability.
   For example, the entities for hazard analysis in EAST-ADL also have to be feasible for analyzing variant-rich systems.
3. Variability management concepts in EAST-ADL can be helpful for achieving some of the primary objectives of MAENAD, even though these objectives may not be primarily concerned with variability management. For example, language concepts for defining design variations may also be used to define the optimization space for system optimization (cf. Section 4.3.1).

As a consequence, variability will receive special attention in MAENAD, but the work on variability management will mainly be driven and motivated by the other, primary objectives and the demonstrators (e.g. required refinements of the variability concepts that were identified during demonstrator modeling).

5.2 Prospective Topics Related to Variability

The only exception are some consolidation issues and refinements that were already identified in Q1 of MAENAD even before experience from the case studies was available:

- Dependencies between variants across the containment hierarchy in FAA, FDA, etc.
- Evaluation of the support for storing system configurations in EAST-ADL models.
- Improved documentation of the overall variability management technique in EAST-ADL (in particular the Multi-Level concept).
- Alignment with AR variability management (postponed to end of 2011 when next version of AUTOSAR is expected to be available).
- "Feature Tree Semantics" (see below).

The last item in the above list has been addressed first; the remainder of this chapter summarizes the current status of this discussion as of Feb 2011.

5.3 "Feature Tree Semantics"

The discussion of this topic already started during the consolidation phase at the end of the ATESST2 project, but no final conclusion was taken at that time. It is important to note that despite the name "feature tree semantics", the topic does not deal with the core of the variability-related semantics of EAST-ADL feature modeling itself. Instead, the focus is on the precise semantics of certain Relationships between Features and other entities in the language, in particular Requirements and FunctionPrototypes.

5.3.1 Overview

This issue deals with the precise semantics of the two EAST-ADL relationships **Satisfy** and **Realization** in case there is a Feature on one side. Refer to the next two figures for details on how these relationships are defined in the EAST-ADL domain model.

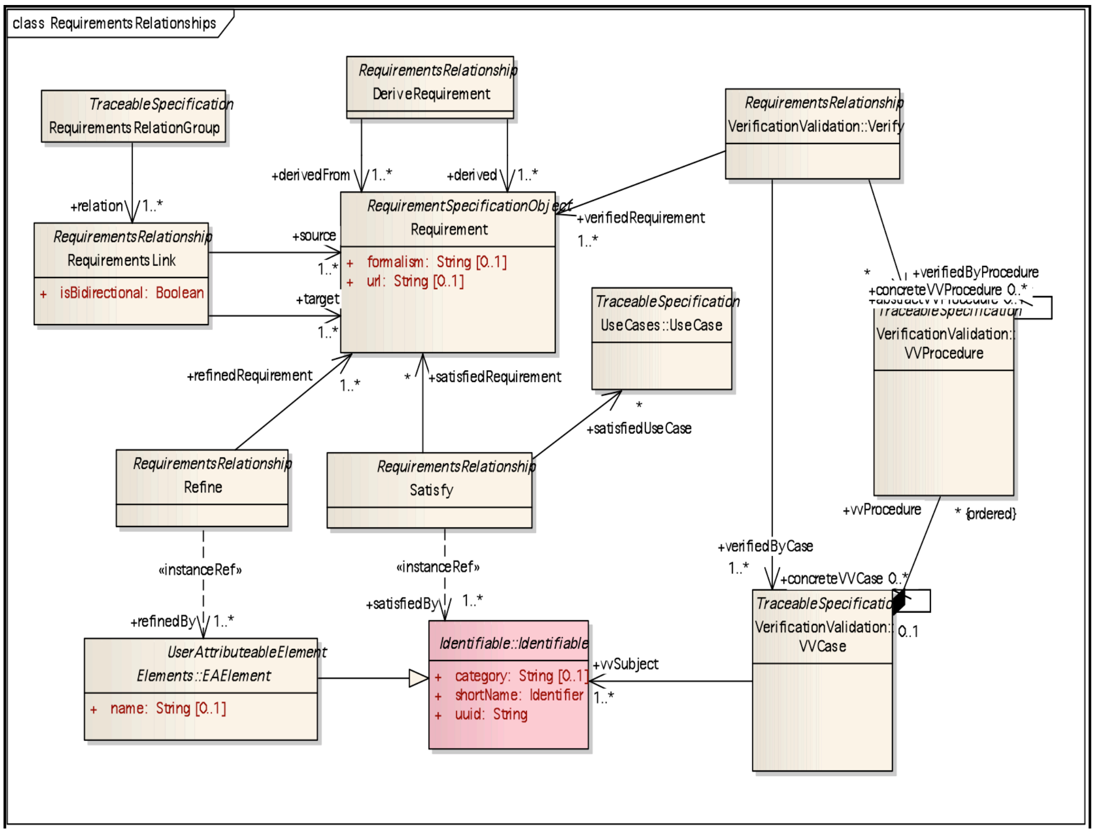


Figure 5. Diagram "Requirements Relationships" from EAST-ADL domain model.

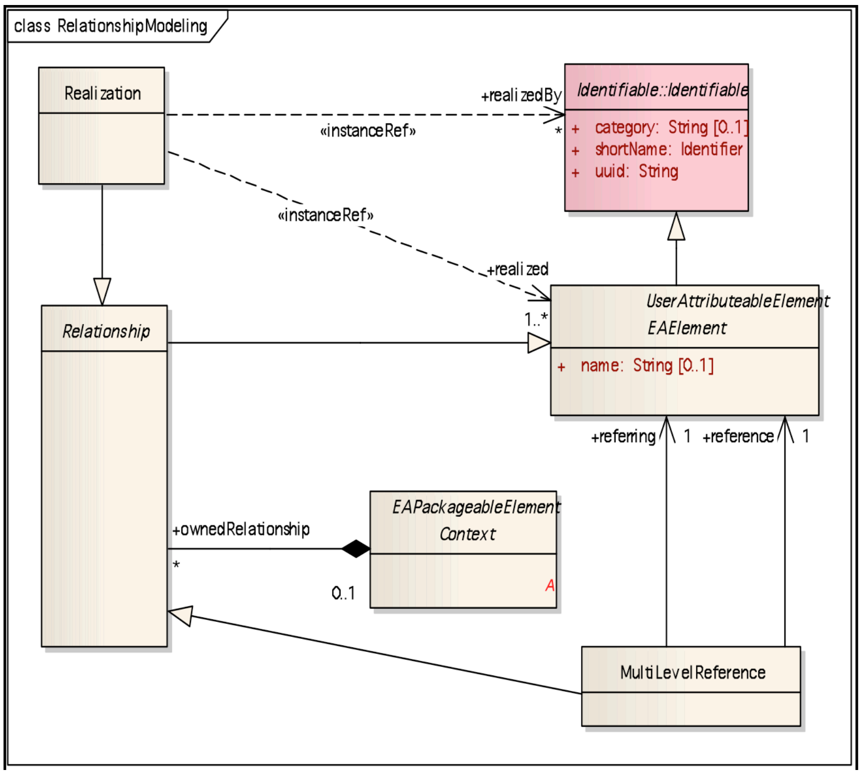


Figure 6. Diagram "Relationship Modeling" from EAST-ADL domain model.

In the remainder of this section we will focus on these cases:

- A Satisfy relationship between a Requirement (role name "satisfiedRequirement") and a Feature (role name "satisfiedBy").
- A Realization relationship between a FunctionPrototype (role name "realizedBy") and a Feature (role name "realized").

The overall semantics of these two relationships – when leaving aside the details – is quite clear in the above cases:

- Feature F → Satisfy → Requirement R:
  Feature F will heed Requirement R, i.e. it is responsible for making sure that Requirement R is fulfilled.
- FunctionPrototype FP → Realization → Feature F:
  FunctionPrototype FP provides an implementation of Feature F (in case of a DesignFunctionPrototype).

However, when inspecting these relationships more closely, the semantics becomes more intricate, especially when taking into consideration the parent/child relations between features. This is discussed in the next section.

5.3.2 Problem Description

In this section we try to highlight the difficulties in the semantics of the two aforementioned relationships by listing a number of questions in each case.

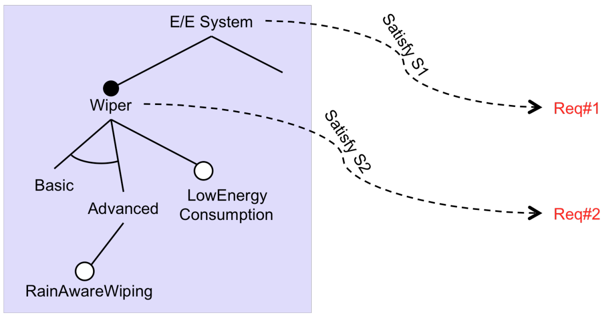


Figure 7. Sample Satisfy relationships.

Some considerations on the fact that Wiper satisfies Req#2, as defined by S2 (in the above figure):

- Does S2 mean that only the wiper-system (i.e. feature Wiper) has to heed requirement Req#2 and the climate-control system (not shown in figure) need not heed Req#2?
- What does S2 imply for predecessors (i.e. parent features, grand-parents, and so on …) and successors (i.e. child features, grand-children, etc.) of feature Wiper? For example, does S2 imply that also feature Advanced satisfies Req#2?

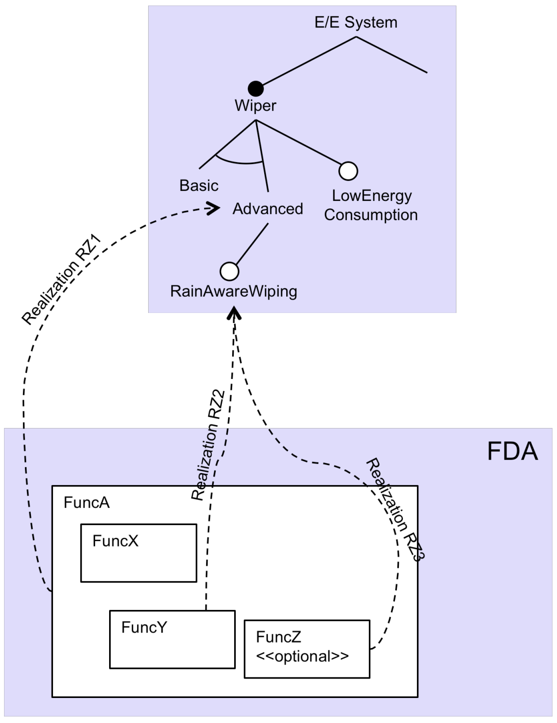


Figure 8. Sample Realization relationships.

Some questions regarding Realization relationships RZ1, RZ2 and RZ3 in the above figure:

- Does RZ2 mean that FuncX does **not** realize RainAwareWiping (i.e. is not at all involved in realizing RainAwareWiping)?
- What does it mean that both FuncY and FuncZ realize feature RainAwareWiping?
- What does RZ1 imply with respect to the realization of features E/E-System, Wiper and RainAwareWiping through FuncA? For example, does RZ1 imply that FuncA also realizes

feature Wiper? Does RZ1 imply that FuncA also realizes feature RainAwareWiping (if it is selected)?

- What does RZ1 imply with respect to FuncX? Does FuncX, as a subfunction of FuncA, also (partly) realize feature Advanced?

5.3.3 Tentative Solution

In this section we provide a tentative semantics definition that may be used as a basis for further investigation and refinement based on more detailed examples and the MAENAD demonstrator models.

Semantics for case “Feature F → Satisfy → Requirement R”:

The Satisfy relationship between a feature and a requirement defines that this particular requirement applies to this feature and its successors, i.e. the functionality and/or non-functional properties represented by the feature and its successors must collectively fulfill the requirement.

Points to note:

- Predecessors of a feature are its parent, grand-parent, and so on. Successors of a feature are its child features, grand-children, etc.
- This might mean that the feature provides some functionality that is required by a functional requirement or that the feature must comply with some constraint, restriction, etc. imposed by the requirement.
- Effect of parent/child relations in the feature tree (still referring to case “Feature F → Satisfy → Requirement R”):
  (a) When looking at the particular requirement R, then this requirement applies to F, the child features of F, the grand children of F, etc.
  (b) when looking at a particular feature F, then all requirements of its parent, those of its grand parent, etc. apply to F.
- The term “collectively” above is still being discussed at time of writing (MS3). Some project partners tend to this view: “Each successor must fulfill the requirement. It may be implemented in different ways but each child feature is individually responsible.”

Semantics for case “FunctionPrototype FP → Realization → Feature F”:

„The Realization relationship denotes the primary responsibility of an architectural element for realizing the functionality and/or non-functional properties represented by a feature. Several architectural elements defined to realize a single feature are collectively responsible for the realization.“

Points to note:

- If several FunctionPrototypes realize the same feature they are all, collectively responsible for realizing the feature. No assumption is made how responsibilities are shared (equally or one function being more significant than the others) and which parts are realized by which function.
- The same applies to a higher-level function that contains subfunctions: the containing function and all its directly or indirectly contained subfunctions collectively realize the feature.

- The explicitly defined realizations (by way of Realization relationships) do not claim completeness in the sense that each and every contribution is explicitly defined. Otherwise also minor, very remote and indirect contributions would have to be defined with a Realization relationship.
  Instead, the Realization relationships define primary / major contributions to realization.

5.3.4 Further Steps

The initial, tentative definitions from the previous section should be evaluated and further refined based on concrete examples and the demonstrator models. Further refinement on this abstract, theoretical level would probably prove very difficult. Also the example in the SAFECOMP paper (one of the ATESST2 publications), where dependent functions are used as examples, can be used as a basis for further exploration. This focuses on the Satisfy relation as that one has a deeper impact on functionality definition on Vehicle level.

Furthermore, it would be advisable to first record the precise purpose / motivation of modeling these relationships (e.g. in the context of ISO26262) to guide the further modifications for finding the most appropriate semantics.

6 Language Consolidation Amendments

This section discusses language refinements related to general consolidation activities that are orthogonal to the specific project objectives. The language concepts that need refinement are discussed below along with proposed modifications. The amendments are related to:

- Type definition
- The Inheritance structure
- Environment Model
- HardwareArchitecture
- Semantics of Realization

Beyond what is documented here, a more extensive consolidation effort is planned for Dec 2011 to Feb 2012. The outcome of this consolidation will be documented in future versions of this deliverable.

6.1 Type definition

The Datatype concept of EAST-ADL 2.1 requires further validation. Both the definition of Datatype and the use of Datatype as a type of various attributes in the language must be changed because…... Figure 21 shows the metamodel of EADatatype.

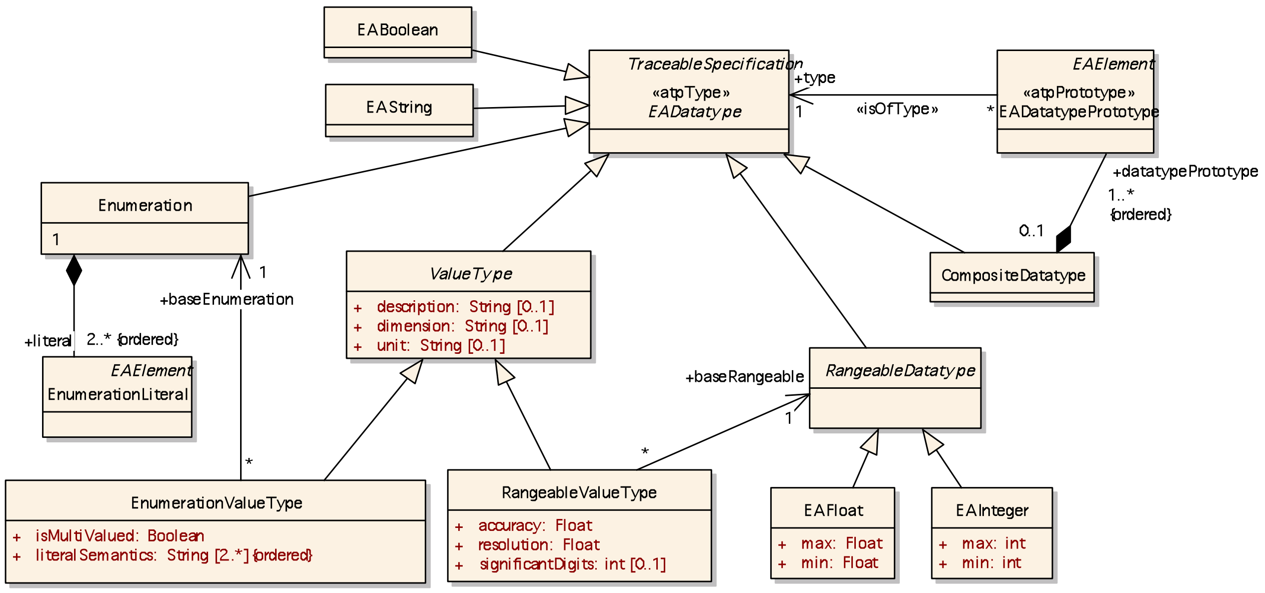


Figure 21. The EADatatype and related elements

6.2 The Inheritance structure

Majority of the language elements are subtypes from main elements like EAElement, AllocationTarget, EAPackeableElement, Context and TraceableSpecification. The inherited attributes and associations needs to be assessed for some language concepts to ensure validity, see Figure 22.

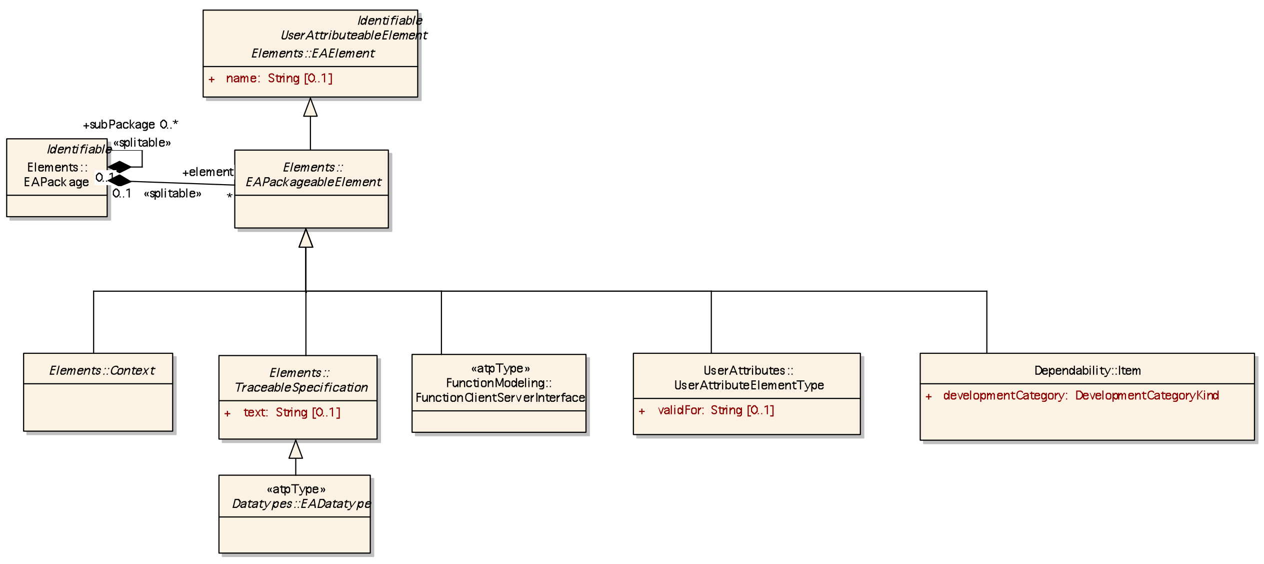


Figure 22. Inheritance structure of some selected elements

Figure 22 shows that FunctionClientServerInterface inherits directly from EAPackableElement. Instead it should inherit from TraceableSpecification like EADatatype. Why, what problem related to validity this solves? What happens to models already made with the "wrong" earlier definition.

Item should have the ability to own TraceableSpecifications and should thus inherit from Context.

UserAttributeElementType should inherit from TraceableSpecification like EADatatype.

6.3 Environment Model

The EnvironmentModel is currently a function hierarchy that is separated from the SystemModel and linked to the FAA and FDA through ClampConnectors, see Figure 23. How the interfacing between system model and environment model is best represented?

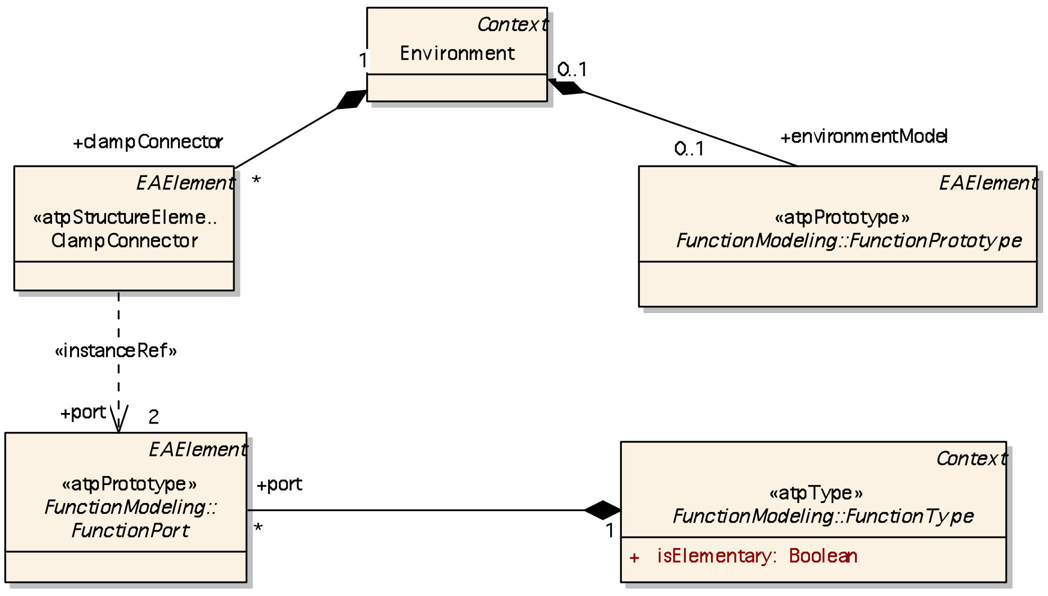


Figure 23. The EnvironmentModel and related elements

6.4 HardwareArchitecture

The HardwareArchitecture need to be refined to support FEV needs. In particular, the specification of electrical I/O need to be added to the metamodel. Currently the metamodel of HardwareArchitecture is as specified in Figure 24. Once this metamodel is entered as a metamodel (see Figure 25) in a modeling tool the language can be applied as shown in Figure 26 - albeit now is not possible to specify electric I/Os.

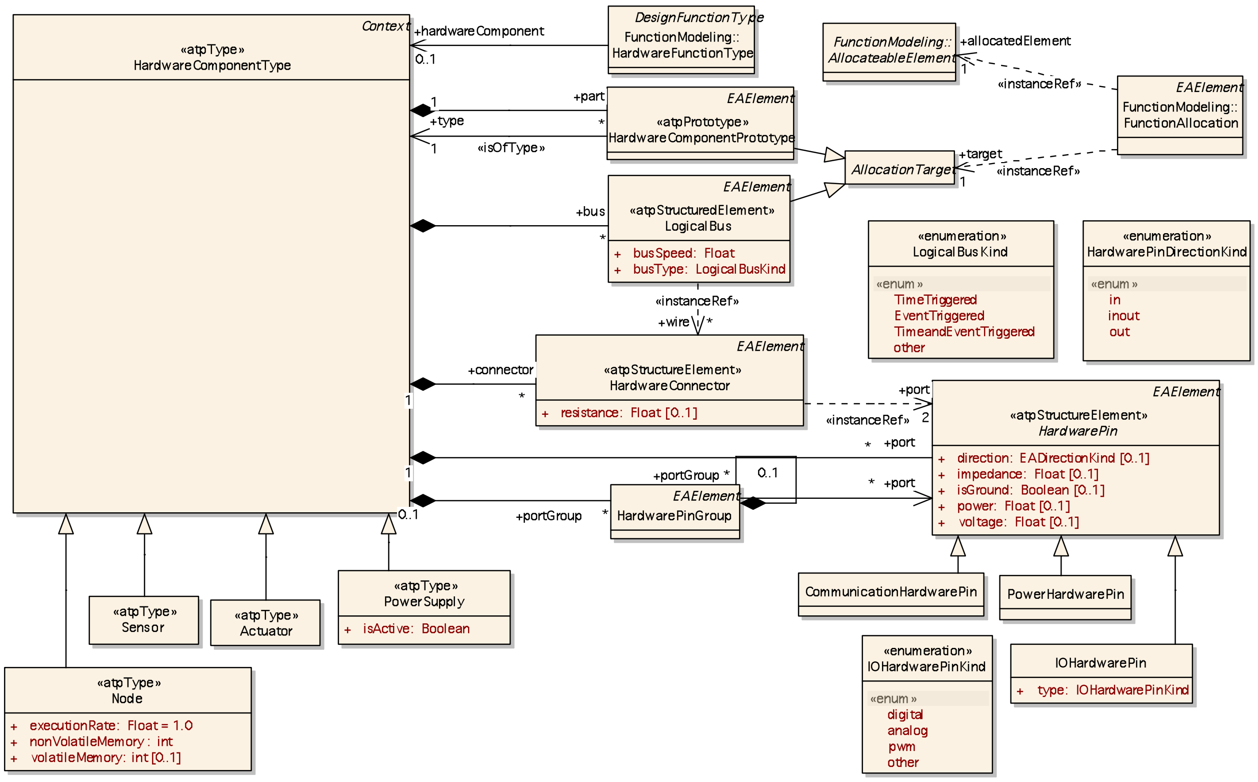
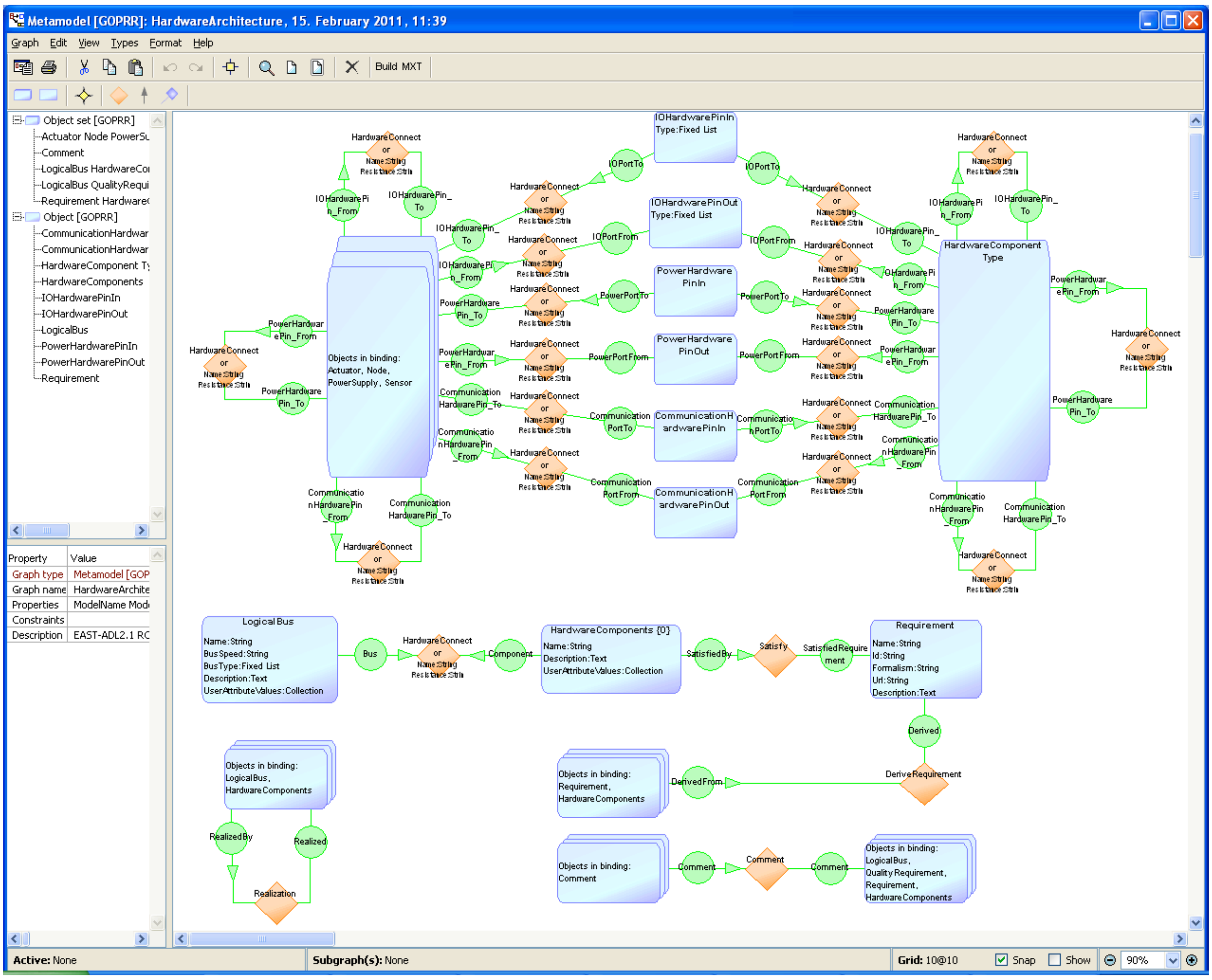


Figure 24. The Hardware Architecture

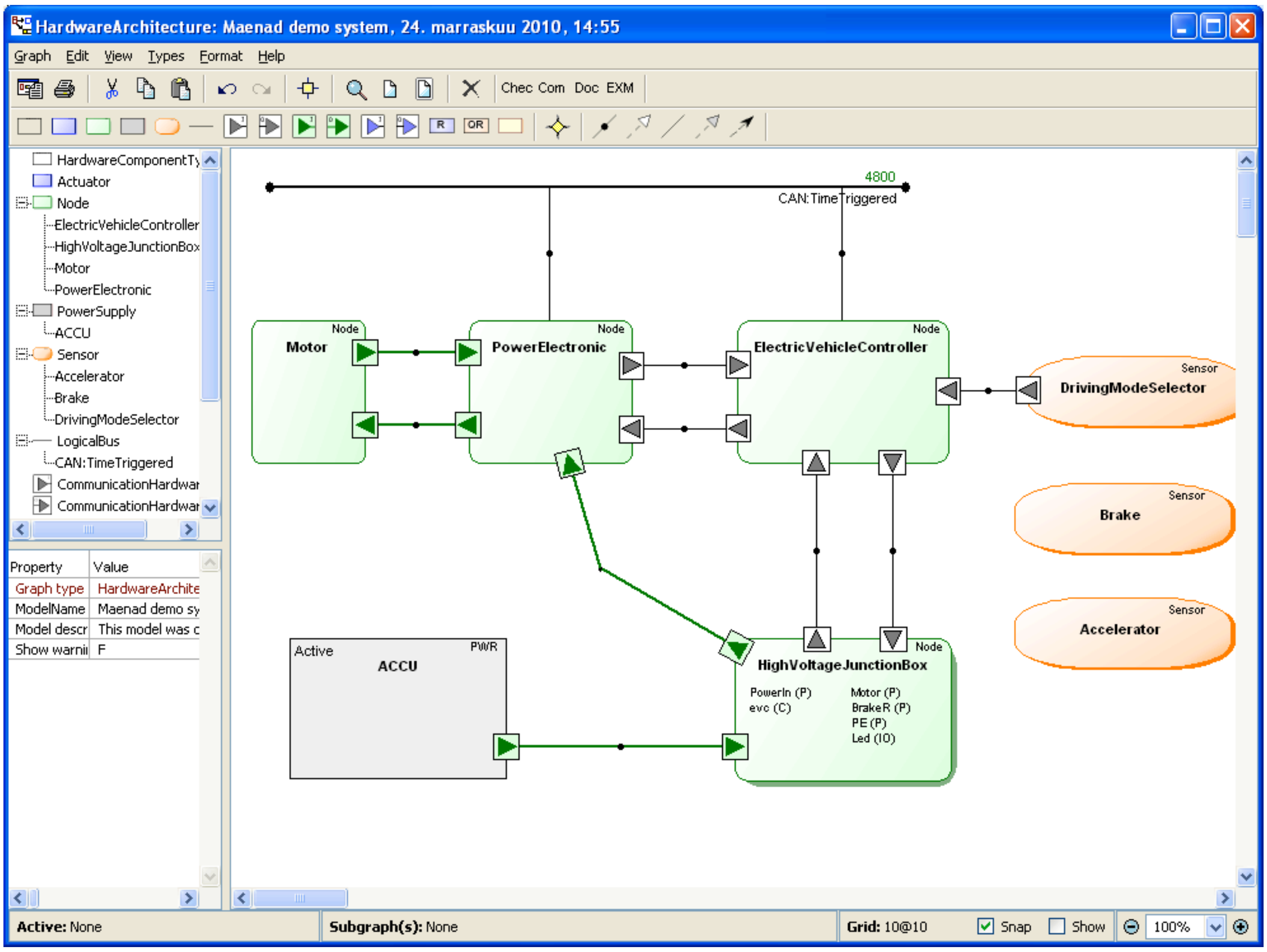

Metamodel [GOPRR]: HardwareArchitecture, 15. February 2011, 11:39
Graph Edit View Types Format Help
Build MXT
Active: None
Subgraph(s): None
Grid: 10@10
Snap
Show
90%
HardwareConnect or Name:String Resistance:String
IOHardwarePinIn Type:Fixed List
IOHardwarePinOut Type:Fixed List
IOPortTo
IOPortFrom
IOHardwarePin_From
IOHardwarePin_To
PowerHardwarePinIn
PowerHardwarePinOut
PowerPortTo
PowerPortFrom
PowerHardwarePin_From
PowerHardwarePin_To
CommunicationHardwarePinIn
CommunicationHardwarePinOut
CommunicationPortTo
CommunicationPortFrom
CommunicationHardwarePin_From
CommunicationHardwarePin_To
HardwareComponentType
Objects in binding: Actuator, Node, PowerSupply, Sensor
LogicalBus Name:String BusSpeed:String BusType:Fixed List Description:Text UserAttributeValues:Collection
Bus
Component
HardwareComponents {0} Name:String Description:Text UserAttributeValues:Collection
SatisfiedBy
Satisfy
SatisfiedRequirement
Requirement Name:String Id:String Formalism:String Url:String Description:Text
Derived
DeriveRequirement
DerivedFrom
Objects in binding: LogicalBus, HardwareComponents
Objects in binding: Requirement, HardwareComponents
RealizedBy
Realized
Realization
Objects in binding: Comment
Comment
Objects in binding: LogicalBus, QualityRequirement, Requirement, HardwareComponents

Object set [GOPRR]
- Actuator Node PowerSu
- Comment
- LogicalBus HardwareCo
- LogicalBus QualityRequi
- Requirement Hardware

Object [GOPRR]
- CommunicationHardwar
- CommunicationHardwar
- HardwareComponent Ty
- HardwareComponents
- IOHardwarePinIn
- IOHardwarePinOut
- LogicalBus
- PowerHardwarePinIn
- PowerHardwarePinOut
- Requirement

Property	Value
Graph type	Metamodel [GOP
Graph name	HardwareArchite
Properties	ModelName Mod
Constraints	
Description	EAST-ADL2.1 RC

Figure 25. The metamodel of Hardware Architecture as implemented in MetaEdit+ for EAST-ADL2



Property	Value
Graph type	HardwareArchite
ModelName	Maenad demo sy
Model descr	This model was c
Show warnii	F

Active: None Subgraph(s): None Grid: 10@10 ☑ Snap ☐ Show 100%

Figure 26. A sample model of an hardware architecture

To support capturing electric I/Os the metamodel must be extended with new kinds of ports and connections that enable specifying electric I/Os. The extension of electric I/Os is similar to other hardware connectors already described in the metamodels, but electric I/O connection and ports have own characteristics as follows:

- electric I/O port has an attribute called ‘Voltage’ to specify ‘Electric voltage used’
- Hardware connection for electric port has ….
- Etc.

Electric I/O can be connected only between electric I/O ports and their type are defined by hardware component types and they are used by prototypes similarly to other hardware connectors (power, hardware IO and communication).

After extending the metamodel as shown in Figure 27 the hardware architecture models can be presented in EAST-ADL2.

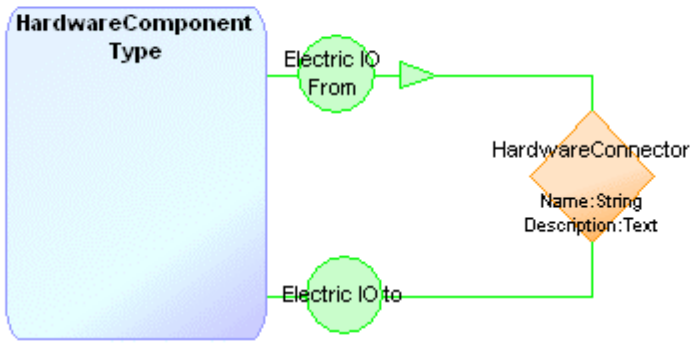


Figure 27. An extended metamodel of Hardware Architecture

A sample of hardware architecture modeling specifying electric I/O is illustrated in Figure 28 where ACCU (PowerSupply Prototype) is connected to HVJB (Node prototype) using the added language concept (electric I/O).

The notation for electric I/O distinguishes it from other ports and connections by using different coloring and line type. The ports and connection may also show relevant information about the electric I/O such as the voltage information as illustrated below (12V).

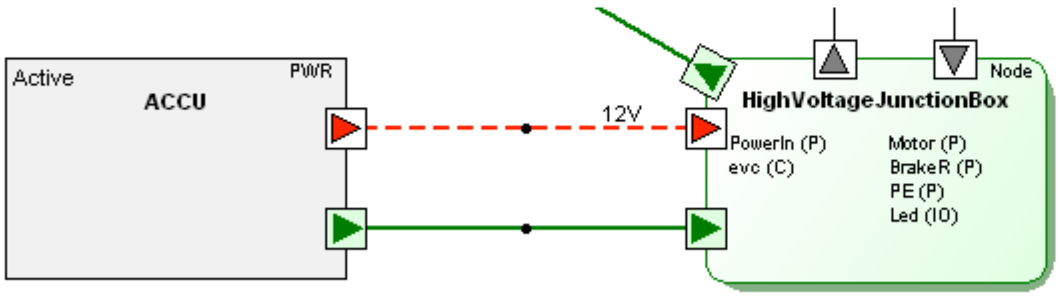


Figure 28. A sample of using the extension: electric I/O

The extended Hardware Architecture language will be tested in the pilots and refined based on the feedback from realistic usage scenarios. If change is accepted it will be incorporated to the next release of the EAST-ADL2 language.

6.5 Semantics of Realization

It shall be defined what the meaning of element X realizing element Y mean. This is particularly important for features referenced by an Item, as they define the scope of the safety element. Figure 29 shows the metamodel of the Realization relation.

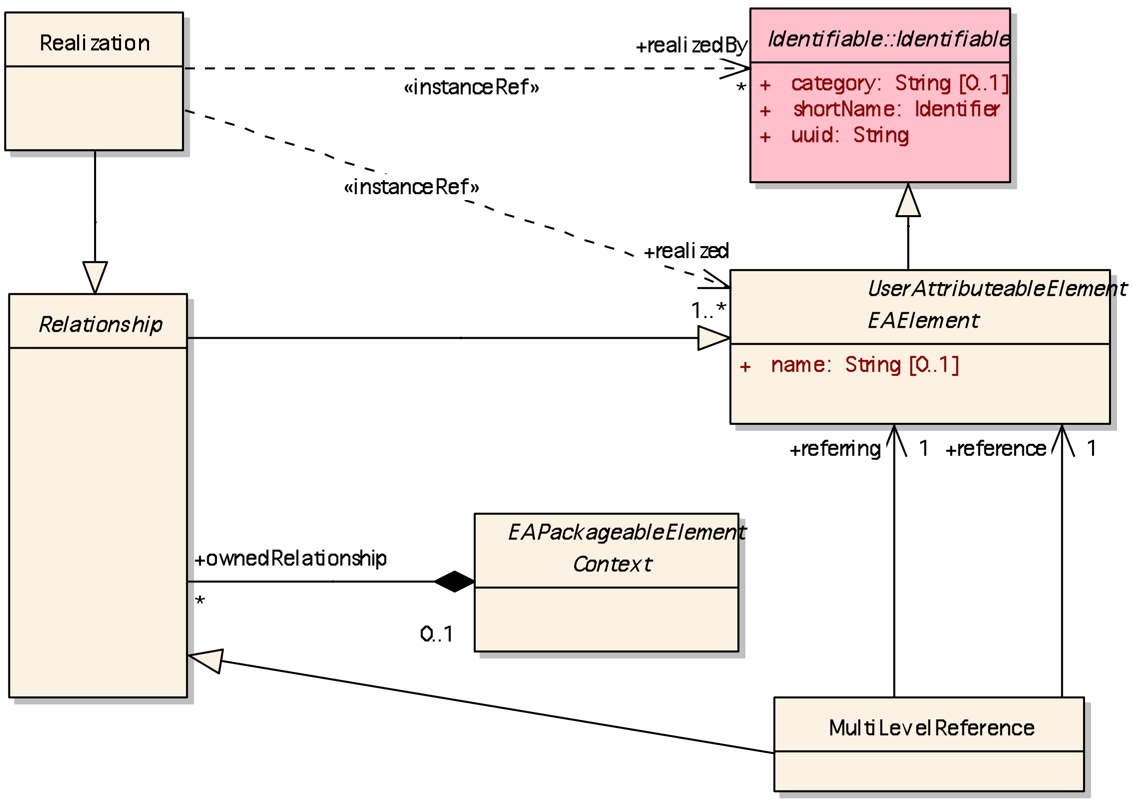


Figure 29. The metamodel of Realization

7 Discussion of other Concepts Related to ISO 26262

7.1 Fault Injection

This section discusses the support that MAENAD language and tools could provide for verification and validation activities during the development phase of the safety lifecycle according to ISO 26262, focusing on fault injection techniques.

ISO 26262 heavily relies on Verification and Validation activities to provide evidence that the obtained product complies with the safety requirements. V&V activities are carried out in a systematic way on each phase of the development: system development, HW development and SW development. At the system development phase, focus is on the integration of the item’s elements and to provide evidence that the integrated elements interact correctly. Integration tests are performed at each stage of integration; software and HW integration, system integration and vehicle integration.

At the HW development level, tests are carried out to check the correctness of the HW safety mechanisms in relation to to the HW safety requirements. The same apply for the SW development level, where tests are performed for SW unit and during the integration of SW unit to form a complete SW architecture.

Goals of the testing activities are

- test compliance with each safety requirement in accordance with its specification and ASIL classification.
- Verify that the "System design" covering the safety requirements are correctly implemented by the entire item
- correct implementation of functional safety and technical safety requirements
- correct functional performance, accuracy and timing of safety mechanisms;
- consistent and correct implementation of interfaces;
- effectiveness of a safety mechanism's diagnostic or failure coverage;
- level of robustness.

ISO 26262 provides a set of testing techniques and methods that address specific goals; more precisely, a given test goal is addressed using different testing techniques. The table below summarizes the relationship between development phases, test goals and fault injection techniques used as a test method to achieve compliance with requirements. Green boxes report when the Fault injection technique is explicitly recommended as a test technique to use to address a test goal, in blue when it is indirectly involved.

Table 3. Applicability of Fault Injection (FI) for test goals and phases (dark fields indicate that FI is recommended and light fields that FI is indirectly involved

			Test Goals				
		Sub phases	correct implementation of functional and safety technical safety requirements	correct functional performance, accuracy and timing of safety mechanisms;	consistent and correct implementation of interfaces	effectiveness of a safety mechanism's diagnostic or failure coverage	level of robustness
Development phases	System	HW/SW integration	▓	░		▓	
		System integration	▓	░		▓	
		Vehicle integration	▓	░		▓	
	HW	Unit	▓	░		▓	
	SW	Unit	▓			▓	
		SW integration	▓	░		▓	

Maenad language and related tools could provide support for experimental V&V activities based on fault injection techniques with different scope. Experimental activities can be structured in three different sub-phases, and model based design could serve each of them in different way

- Test design: models of the systems are used to transfer information useful for the design of a test experiments. In this context, the model of the system is used as a container of data useful to derive information about the System Under Test (SUT), its boundary and to design test vectors
- Test bench setup: information is extracted from the model in order to support a semi-automatic setup of a test experiment and HW test plant. For complex systems to be analyzed, the capability to support the engineers in the semi-automatic setup of a test experiment could save days of works
- Test execution: the SUT is exercised through test equipment. Actual and intended test results are represented in the model using the V&V constructs.

The following section report a gap analysis related to the support that MAENAD language and tools could provide for fault injection experiment.

7.1.1 Test design - gaps analysis

This section is mainly related on the capability of the MAENAD language and tools to support test engineers for the design of experiments. The focus will be on the support provided by the language for the formulation of tests. The first column report the methods to derive test cases as they are expressed by ISO26262. Those methods are applicable to all type of experiment, and for fault injection as well.

Methods	Key point	Gap Analysis
Analysis of requirements		Supported through requirements packages
Analysis of external and internal interfaces	Capability to derive from the model functional/SW structure and decomposition	Full support
Analysis of equivalence classes	Capability to express partitions (valid, invalid) in the model for the input data of functions and SW components. Useful to derive test vectors reducing the total number of test cases that must be developed. Used in Black box testing and Gray box testing.	Equivalence classes in input requires behavioural model that captures the required behavior. Appropriate tooling can then establish the equivalience classes. Interface specifications are also relevant here. This needs to be further analyzed, but there is currently no such tool for EAST-ADL.
Analysis of boundary values	Derived from the equivalence classes	Appropriate tooling can assess behavioural models and interface specifications to identify boundary values. There is currently no such tool for EAST-ADL.
Error guessing based on knowledge or experience	Capability to transfer information on test vector derived from previous experiences	Supported by the V&V package
Analysis of functional dependencies	Capability to transfer information on EE architecture functionalities, their decomposition and the related dependencies	Full support due to the capability of the language to describe HW, functional and SW view of an embedded system and they relationship, functional allocation.
Analysis of common limit conditions, sequences, and sources of dependent failures		Full support for sources of dependent failures (Error modeling) Lack support to express limit conditions in the model
Analysis of environmental conditions and operational use cases		Use cases and behavioural definitions on vehicle level could support this activity. To be further analyzed
Analysis of field experience		Field experience could be interpreted as a special kind of testing, an would then be supported by the V&V constructs.

7.1.2 Test setup - gaps analysis

This section is mainly related on the capability of the MAENAD language and tools to support test engineers for the setup of experiments.

The focus is a gap analysis to derive plug-in that support the semi-automatic setup of an instrumented test.

Needs	Key point	Gap Analysis
Automatic generation of networks related setup	Plug in to automatically derive the information needed to setup network communications and interpretations of network data. This include for each network signals: endianism,length, start bit, factors to obtain the physical value, message packing	Not supported. It is not possible to express and derive this level of details on network connectivity from the model.
Extraction of subsystem test sets	Plug in to automatically derive test vector related to the subsystem under analysis	Implementation possible due to the hierarchical organization of the model and the capability of the model to link architectural elements and their dependencies

7.1.3 Test Execution - gaps analysis

Needs	Key point	Gap Analysis
Capability to support emulation of the environment		Link to external simulation tools capable to realize the necessary emulation is provided through dedicated bridge (Simulink gateway, Modelica exchange, Modelisar FMU import) All those external environments provide the necessary capabilities to execute test vector, emulate plants, emulate missing items of a systems,… To be analyzed the effectiveness and suitability of the gateway starting from the above concerns
Capability to support emulation of the missing items		

7.2 System/Environment model interface implications for ISO26262 support

One of the issues with modeling the system – environment interface and their relation to ISO26262 is the open meaning of ports for functional devices. To enable a clear distinction of what is the target of a hazard when analyzing a model some changes are suggested. The background for the changes are given in section 7.2.1 and the proposal for new structure of Functional Devices are given in 0

7.2.1 Functional devices in current language definition

From a modeling point of view there is no difference between a FunctionalDevice and an AnalysisFunctionType. There are nothing special about it other than the semantics. A lot of this discussion relates to the discussion on the environment model in section 6 and there is some overlap on the issues that relate to functional devices.

The issues that have been detected are the following.

1. Can a FunctionalDevice have an error-behavior or is it just a mapping between physical and logical world.
    a. If a functional device can have logical errors, is it then more than a functional device.
2. Can a FunctionalDevice be monitored by a safety mechanism. (The output is in the physical world.)
    a. It is a modeling challenge to get the right level of detail if functional devices as they can be complex or simple. They are often complete systems that could contain a full system model.
    b. A safety mechanism monitoring the logical output of an actuator is difficult to envision with ISO terminology if there is no error in the functional device and the output to the physical world is a direct mapping of the input.

Some of these views are conceptual discussions based on how to handle sensors and actuators from a safety point of view.

The semantics in the language spec gives some hints on the scope of a functional device: 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).

Does this mean that the logical port of a functional device should encapsulate the unknown implementation of the transfer, or that the transfer function becomes a constraint on how the lower abstraction levels manage the transfer from logical level to environment? It is not sure that the second part of the semantics is necessarily known at the time the functional device is defined and it seems strange that the analysis level even discusses artifacts that are mapped to hardware and software. The first sentence is the basic notion of a functional device on analysis level.

There is nothing that really prevents the current language to address the above mentioned problems but the semantics is too open for comfort. This is especially true for the semantics on the port definitions. Hence after the extract of the relevant parts of the language a proposal for change is made that clarifies the specialization of a functional device from its ancestors.

Extract from the domain model:

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).

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.

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).

7.2.2 Suggested changes to FunctionalDevice.

Since Hazards occur when a failure is propagated to the output of an actuator in a specific situation, they all originate through the *output* of a FunctionalDevice acting as an actuator. But an instance of a FunctionalDevice can be both sensor and actuator and there is no distinction that makes it possible to find locations in a model where propagated failures can cause hazards. Hence it might make more sense to use the roles 'logical' and 'physical' for the ports on a functional device to state where it is connected. The direction of the physical port could then indicate whether it is a sensor or actuator when an analysis is performed. The difference between a functional device and an analysis function could be seen as the fact that the functional device has a connection to the physical world, something that an analysis function cannot have.

The suggested change to the FunctionalDevice class is a new port. The connection to the physical world, the place where hazards occur or where sensors are connected:

Associations

physicalPort : FunctionPort [*] (from FunctionModeling)

This could be both a FlowPort or PowerPort depending on the needs.

This port serves two purposes in the analysis of models. It gives information on the type of functional device given the direction of the port. Secondly it serves as the connection point between Safety goals and the logical architecture, through the error model.

The current port given by the FunctionType attribute 'port' would then be limited to being logical ports not allowed to be connected to the environment. Depending on the type of functional device input ports would be stimuli from the logical world and outputs could be nominal values or logical feedback. This enables the possibility to make functional devices hierarchical as the internal structure could feed not only the physical port with data but also the logical ports which makes it perfectly plausible to use AnalysisFunctionTypes in the decomposition of a FunctionalDevice.

Having the capability to do logical feedback would make it possible to address safety mechanisms as there would be a logical path that could be specialized when decomposing the functional device in a more detailed view.

Functional devices can be seen as either very complex system, especially if you are focusing on them in your modeling. Or as trivial data producers if you are interested in the logic manipulating of data.

8 EV-Specific Needs

As reported in D2.1.1, a process was followed to define the requirements related to FEV development, in order:

- to verify the capability of the current version of EAST-ADL2 to cover the needs related to specific characteristics of FEVs, and to extend its features if necessary; and, similarly,
- to verify the capability of the analysis tools and to give inputs to adapt or, possibly, create specific tools to perform the necessary analyses;
- to define an extension of the basic E/E system development methodology resulted from ATTEST2, in order to help designers to perform the development activities required by the standards and the regulations, or those compliant to best practices or engineering needs for EV development.

Therefore, through a sequence of activities according to a bottom-up approach, three categories of requirements have been defined: language requirements, analysis requirements, and methodology requirements.

The requirements defined have been reported in an Excel sheet and, subsequently, in Enterprise Architect, to comply with the method followed for the collection of MAENAD requirements, thus allowing better traceability, uniform categorization, assignment to WPs.

The following table is an excerpt of the Excel file and includes only the language requirements. Reference are given to the requirement codes used in EA; the field “subject” has been introduced to better identify the related engineering topic and to establish a link with the analysis and methodology requirements related to the same topic.

In addition, an empty field has been here introduced, which will be filled in to specify the technical requirements as to implement the language features. This new definition activity will be performed in the next months, both with the analysis of the requirements to verify which of the requirements can be met with the present EASTADL2 version.

It has to be pointed out that in the following table some language requirements are referred to a specific standard or regulation. However, the requirements, in some cases, can be referred to similar standards (not mentioned here, but only in the Excel sheet, which gives a more global view of the analysis conducted to define the requirements).

First level user requirements		Second level user requirements Language requirements			Technical specification for the implementation of the language features to meet the requirements
Code	**Title**	**Subject**	**Requirement description**	**Code**	
4SG 7	EV safety standards/ ISO 6469-1	Insulation	- Insulation symbols - Insulation attributes (withstand voltage, resistance, presence of DC or AC parts, creepage distance, ref. to standards...) - Insulation devices (to describe the interconnection between isolated and not isolated physical parts, e.g. communication, power supply, drives) - High voltage parts (wrt physical view) in order to take note of the requirements regarding creepage distance, clearance, labeling, wire color, insulation.	4SG76	
		RESS over-current interruption	Modelling of an over-current interruption device	4SG80	
4SG 9	EV safety standards/ ISO 6469-3 Protection of persons against electric hazards	Requirements of potential equalization	Represent bonding/grounding of physical elements (proper symbols)	4SG91	
4SG 18	EV safety standards/ J2289	Key-on discharge	- Modelling the power supply network including fault protection devices with their current-time characteristics - Modelling auxiliary equipment including power requirements/ power profiles	4SG105	
		Key-on Regen operation	- Include voltage limit data/requirements of the drive components - Include recommended battery current and voltage profiles during high SoC - Battery modelling for current-voltage transients analysis	4SG108	

		Key on – Charge	- Include electrical characteristics of the charge system components (e.g. current, voltage)	4SG111	
		Key-Off Parked Off Plug Operating	- Include the power characteristics of the devices running in key-off mode	4SG114	
		Parked Off Plug IDLE/Storage Operation	- Modelling the battery disconnect system (mechanical switch)	4SG117	
		Discharge management - Performance limits	Include the operation limits of the battery (temperature ranges, current, under-voltage)	4SG120	
		Key-on startup diagnostics and warning	Represent different levels of warnings (depending on the fault severity)	4SG123	
4SG 72	FMVSS No. 114 Theft protection	Keylocking device	Model a keylocking device with lock and unlock conditions	4SG127	
		Diagnostics and warning	Modelling HMI interface for visual indicators	4SG134	
4SG 19	EV performance standards/ ISO 8715	Performance testing - Terms and definitions	Define vehicle performance characteristics according to the terms and definitions given by the standard	4SG138	
		Performance testing - Test conditions and procedures	Define the test cases according to the test conditions and test procedures required by the standard. Scope: to define test profiles for simulation	4SG139	
4SG 20	EV performance standards/ ISO 8714	Energy and range testing - Terms and definitions	Define vehicle energy consumption and range characteristics according to the terms and definitions given by the standard	4SG142	
		Energy and range testing - Test conditions and procedures	Define the test cases according to the test conditions and test procedures required by the standard. Scope: to define test profiles for simulation Include standard test cycle (European, Japan, USA cycles)	4SG143	

4SG 23	EV performance standards/ ISO 12405-2	Terms and definitions	Define battery model parameters according to the test purpose (e.g. energy efficiency, charging and discharging resistance)	4SG146	
		Test sequence - Test conditions	Modelling – Comply with test conditions requirements (e.g. battery state of charge, power consumption of the auxiliaries, test mass, etc.)	4SG147	
4SG 74	SAE J2777 Conductive charge coupler	Control pilot	Model communication protocol based on PWM and signal amplitude (by switching a resistor)	4SG150	

9 Summary and Conclusion

As already discussed in the introduction, the contents of this deliverable are work in progress and currently the text is mainly an account of what is currently being discussed and experimented with in the project.

Over the next months, until the final version of this document is due at the end of project month 24, all the topics presented above will be investigated in detail in WT3.1.