Contract number: ITEA2 – 10039

Safe Automotive soFtware architEcture (SAFE)

ITEA Roadmap application domains:

Major: Services, Systems & Software Creation

Minor: Society

ITEA Roadmap technology categories:

Major: Systems Engineering & Software Engineering

Minor 1: Engineering Process Support

WP3

Deliverable D331a2: Proposal for extension of meta-model for error failure and propagation analysis

Due date of deliverable: 31/12/13

Actual submission date: 18/12/13

Start date of the project: 01/05/2012 **Duration:** 16 months

Project coordinator name: Stefan Voget

Organization name of lead contractor for this deliverable: Valeo

Editor: Florent Meurville (florent.meurville@valeo.com)

Contributors: Philippe Cuenot (Continental) ; Loic Quéran (Dassaut System) ; Andreas Baumgart (OFFIS) ; Markus Oertel (OFFIS);Tilman Ochs (BMW CAR IT) ; Christoph Ainhauser (BMW CAR IT) ; Lukas Bulwahn (BMW CAR IT)

Reviewers: All WT3.3.1 Partners

Revision chart and history log

Version	Date	Reason
0.1	27/11/2013	BMW Car IT update of section 10, Introduction of section 11.2, 11.3, 11.5 and 11.6
0.2	05.12.2013	Conti-F review, updated of Section 11.7 and remove of Annex B
0. 3	11/12/2013	OFFIS update
0.4	13/12/2013	Valeo correction + AltaRica new example provided
0.5	17/12/2013	BMW CAR IT corrections
0.6	18/12/2013	Ready for release proposal
1.0	18/12/2013	Released

1 Table of contents

2 List of figures

3 List of tables

4 Executive Summary

The work task 3.3.1 targets to address the fault modeling and its propagation along the complete development lifecycle. This activity includes the definition of the necessary elements that are needed to capture fault information and propagation concept to produce safety analyses.

Existing fault modeling language candidates such as HiP-HOPS and AltaRica have been deeply analyzed to derive needs for the error modeling as proposed by WT3.3.1.

The starting point for error modeling is the existing modeling approach of EAST-ADLV2.1 tightly coupled with the system model by enriching existing architectural elements with its “fault behavior” in terms of an error model.

The Error model proposed by WT3.3.1 allows to represent the erroneous behavior of a system element as a black box view via the means of ErrorModelTypes (only external visible faults and failures are described) or as a white box view by allowing to a) decompose an ErrorModelType by an arbitrary number of ErrorModelPrototypes and “wiring” the visible malfunctions (faults, failures) between them and b) provide a language for atomic error models to relate internal faults and external faults to their external failures.

In a first step, the mechanisms of error modeling shall be the basis to conduct qualitative safety analyzes. In a second step they shall be extended to conduct quantitative safety analyzes in closed relation with work performed by WT3.2.2.

5 Scope of WT 3.3.1 and structure of the document

5.1 Scope of WT 3.3.1

Embedded in work package 3, work task 3.3.1 deals with failure and cutset analysis. The basis for this work task is the dependability part of EAST-ADLV2.1 which is presented in chapter 9.

WT3.3.1 aims to address the fault modeling and its propagation along the complete development lifecycle and a meta-model extension suitable for the following topics to WT4.2.3.

For the fault modeling language candidates, the needs, regarding fault information and propagation concept to be captured in the model to perform qualitative safety analyzes, will be identified. These artifacts are intended to be attached to each block of an architecture (fault models for inputs, outputs and block propagation), whatever level it is (functional, logical or physical organic or any mix of both). In addition, the same tools shall be used to compute the qualitative safety analyses for evaluation and consolidation of the functional and/or the technical safety concept. The fault and failure context for safety scenarios shall be extracted from safety requirement analysis and then captured using semantics of a fault modeling language. The safety concept will be validated thanks to propagation and analysis of these fault models. At implementation level on the hardware (HW) side, random hardware failure of hardware design and components (failure in time rates) will be considered. In particular, the failures relations to the upper safety concept and theirs contributions to the overall safety analysis will be encompassing. For the hardware architecture, the objective is to extend previous qualitative analyses and to perform quantitative safety analyses with the final goal to work out ISO26262 metrics, such as Single Point Fault Metric (SPFM), Latent Fault Metric (LFM) and Probabilistic Metric HW Failures.

At implementation level on the software (SW) side, failure mode and propagation from the fault modeling language will extend AUTOSAR templates. Relation to the upper safety concept and theirs contributions to analysis will be encompassing. Such failure information will be either captured manually or defined from a tool, as the feasibility study of extraction of Matlab Simulink behavioral model. Additionally, probability of occurrence of the software failure mode will be investigated according to hardware element

Such work will be fertilized by preliminary work performed in the ATESST2 and SPEEDS projects, but also from aeronautic experience regarding the use of Altarica language, with possible use of a subset of it. The final outcomes of this task are an extension of the relevant meta-model to support the failure semantic (this document), and a tool specification for the failure analysis (see D331b document).

5.2 Structure of the document

In a first step, the ISO26262 concepts addressed by WT3.3.1 to evaluate risk of malfunctioning behavior will be explained, including the selection of most relevant safety analyses methods for D331b.

In a second step, the problematic of evaluating malfunctioning behavior in distributed developments mixing OEM, Tier 01 and Tier 02 will be highlighted, and a contract approach will be proposed.

In a third step, HiP-HOPS and AltaRica will be analyzed, and the orientation taken in WT3.3.1 will be justified with some requirements for a simplified SAFE language.

Finally, in a fourth step, the gap between EAST-ADLV2.1 meta-model and previous analysis steps will be documented and an extension of the meta-model will be proposed with application rules.

6 ISO26262 concepts addressed by WT3.3.1 to evaluate risk of malfunctioning behavior

6.1 Short Overview of ISO26262 Chapters of interest for WT3.3.1

During the development of a safety critical E/E product, ISO26262 requires or recommends, depending on the criticality of the product to be developed, to perform a certain number of activities, dealing with risk assessment, that are encompassed in safety analyses. The goal of safety analyses is to help evaluating early during the development phase the potential risks of malfunctioning behavior and find adequate safety measure to eradicate or mitigate their effects. ISO26262 chapters, where the evaluation of potential risks using safety analyses is useful, are illustrated hereafter:

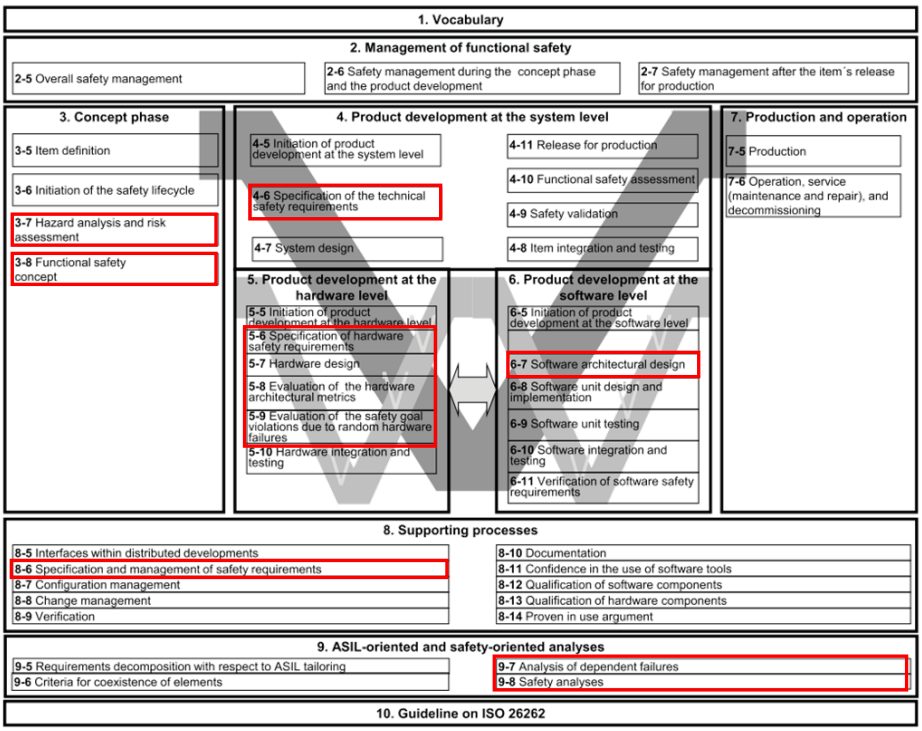


Figure 1: ISO26262 General Overview [1] highlighting where safety analyzes can help

Safety analysis are used to support the concept and development design phase activities during which safety requirements, derived from safety goals, are refined up to HW/SW requirements as illustrated hereafter:

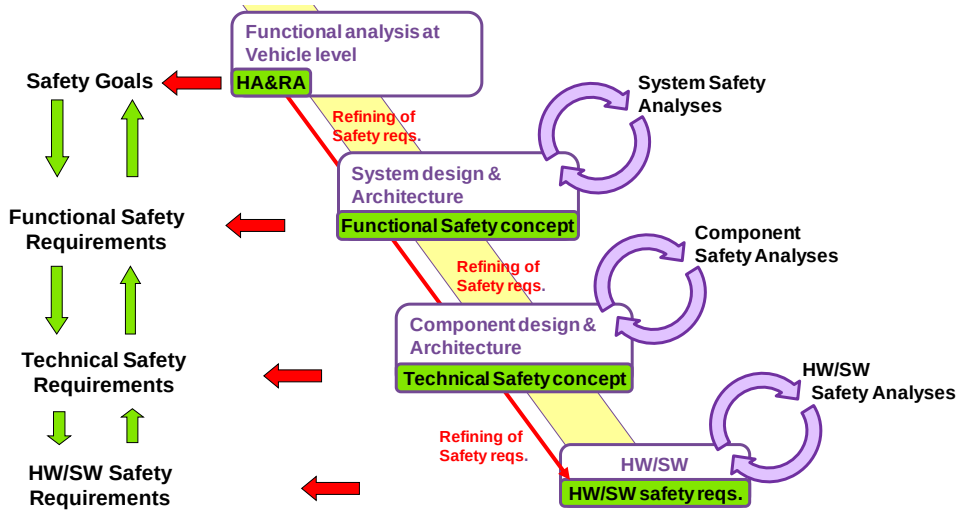


Figure 2: View of safety requirements refinement supported by safety analyses during the concept and development design phases

6.2 ISO26262 and General concept of Fault / Error / Failure for malfunctioning behavior and its propagation

ISO26262 (see [1]) defines **fault** / **error** / **failure** concepts for **malfunctioning behavior**, their interaction and their propagation through different architecture hierarchy levels up to vehicle level:

- A **fault** is an abnormal condition that can cause an element or an item to fail.
- An **error** is defined as the deviation between a computed, observed or measured value or condition from theoretically correct value or condition.
- A **failure** is the termination of the ability of an element, to perform a function as required.
- A **malfunctioning behavior** is a failure or unintended behavior of an item with respect to its design intent.

Therefore an error can be caused by a fault (abnormal condition), and lead to a failure which can be a malfunctioning behavior if appearing at item level.

Faults and failures can be of different types: **systematic or random**.

- Systematic faults or failures are manifested in a deterministic way. They can only be eliminated by a change of the design or the manufacturing process and cannot be quantified.
- Random fault or failures only concern HW elements. They occur unpredictably during the lifetime due to physical causes and follow a probability distribution that allows us to predict Random HW failure rates.

SW faults and failures are always systematic. If you find a scenario that causes a failure, it leads each time to the same failure. In this case, only a design change can eliminate the systematic fault that causes the failure.

HW faults/failures can be either systematic or random.

- Systematic HW: If, as an example, an Electronic Control Unit (ECU) is not protected enough against EMC produced by an external neighbor cable from the system, it always leads to the same failure of the ECU. Only a design change to improve EMC protection would eliminate the systematic faults and failures.
- Random HW: if, as an example, an abnormal oxidation occurs randomly on an HW part belonging to an Electronic Control Unit (ECU), it might lead to a loss of electrical connection and therefore lead to a failure of the ECU.

Note: When systematic faults/failures and HW random faults/failures are mixed together in a same safety analysis such as a Fault Tree Analysis, so to be able to produce quantitative evaluation, it is needed to quantify systematic faults/failures to not produce erroneous probability calculations.

As an example, if a systematic fault/failure is contributing to an AND Gate in a Fault Tree Analysis, its probability of occurrence should be set to 1 to avoid erroneous probability calculation at AND Gate level.

As another example, if a systematic fault/failure is contributing to an OR Gate in a Fault Tree Analysis, its probability of occurrence should be set to 0 to avoid erroneous probability calculation at OR Gate level.

A failure at one architectural level (e.g. ECU level) can become a fault at an upper architectural level (e.g. item level) as shown hereafter.

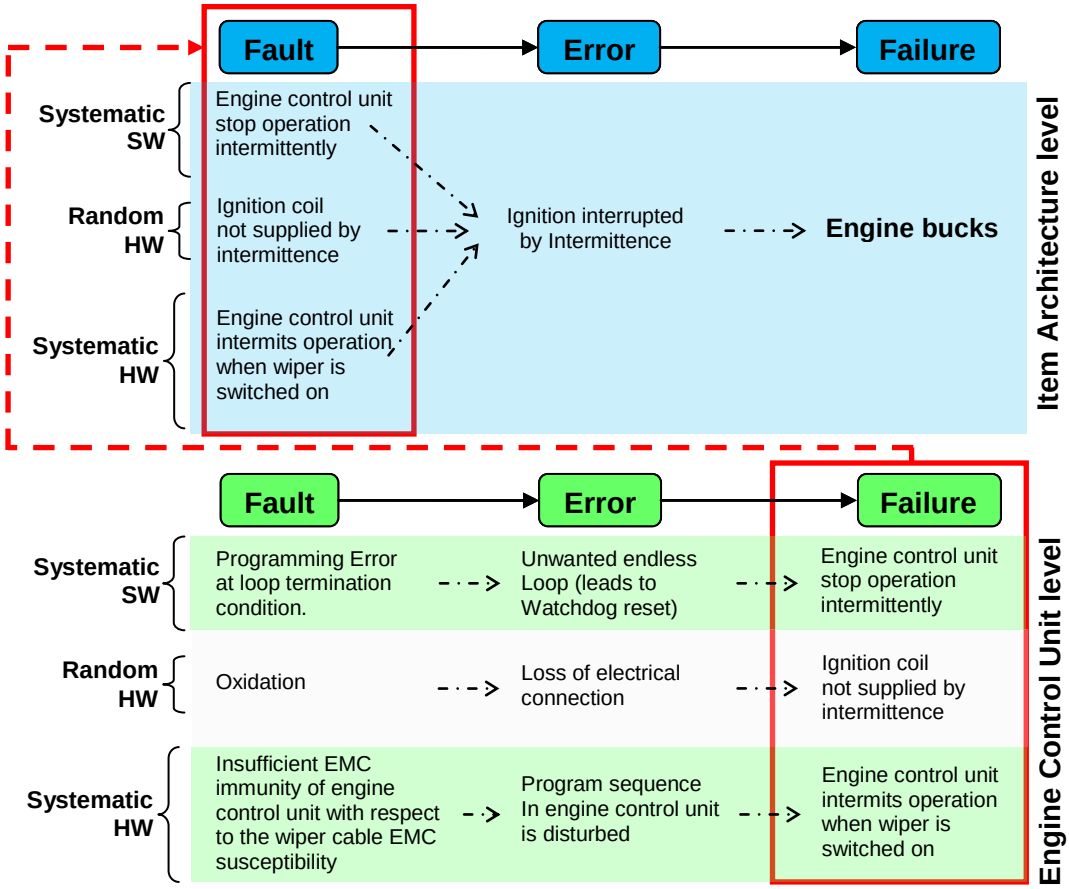
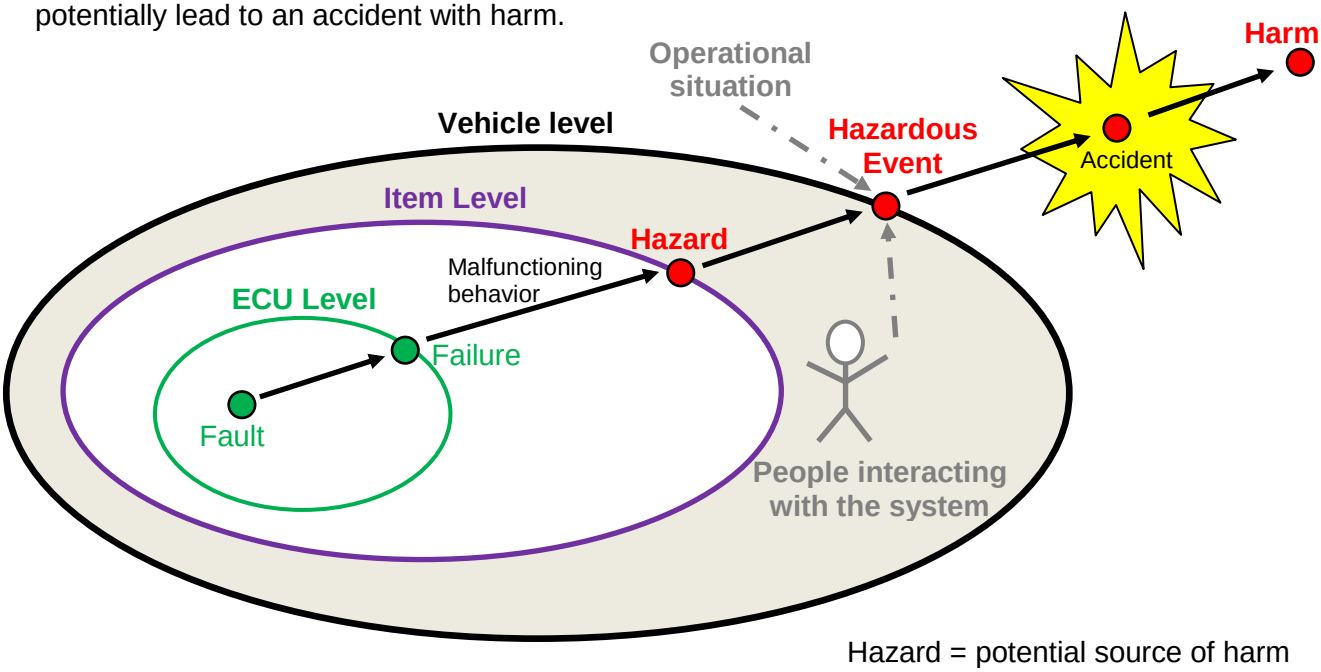


Figure 3: Example of failures at ECU level which become faults at vehicle level

The fault can propagate in the system to produce an hazard at item level, which can become an hazardous event at vehicle level when combined with particular operational situation, and so potentially lead to an accident with harm.



Hazard = potential source of harm

Figure 4: Example of a fault propagating to a hazard

6.3 Types of Safety Analyzes recommended by ISO26262

Through the different concept and development phases from the safety lifecycle, ISO26262 recommends or requires, depending on the criticality of the items or elements to be developed, to perform safety analyses.

The objective of safety analyses is to support the derivation of safety requirements from the safety goals, and to validate and verify their effectiveness and completeness.

Safety analyses help to identify the effect of faults and failures on the functions, behavior and design of items or elements. They also provide information on conditions and causes that could lead to the violation of a safety goal (top-level safety requirement) or a safety requirement. In such a case, additional actions or safety measures shall be determined to eradicate or mitigate the effect of faults and failures.

The fault and failures considered in safety analyses can be either random or systematic, and either internal or external to the items or elements to be developed.

Safety analyses are either inductive or deductive.

- Inductive analysis methods are bottom-up methods that start from known causes and forecast unknown effects. Inductive methods are required by ISO26262 for ASIL A to ASIL D safety goals.
- Deductive analysis methods are top-down methods that start from known effects and seek unknown causes. Deductive methods are required by ISO26262 for ASIL C and ASIL D safety goals and only recommended for ASIL B safety goals.

Safety analyses are qualitative or quantitative:

- Qualitative analyses can be first appropriate and sufficient in most cases to identify failures and when it is not needed to predict the frequency of failure e.g. systematic failures.
- Quantitative analyses extend qualitative safety analyses, in a second step, only when random hardware failures must be predicted as well as the hardware architectural metrics and the evaluation of safety goal violation due to random hardware failures. Quantitative analyses are not required to be applied to systematic failures e.g. software failures.

ISO26262 does not require a specific analysis method but list recognized methods as follows:

Qualitative analysis methods include:	Quantitative analysis methods include:
• Qualitative FMEA[1] (inductive) • Qualitative FTA[2] (deductive) • HAZOP[3](mixed between inductive and deductive) • Qualitative ETA[4] (inductive) • Ishikawa	• Quantitative FMEA[1] (inductive) • Quantitative FTA[2] (deductive) • Quantitative ETA[4] (inductive) • Markov models(inductive) • Reliability Block Diagrams(deductive)
[1]FMEA : **F**ailure **M**ode **E**ffect **A**nalysis [2]FTA : **F**ault **T**ree **A**nalysis [3]HAZOP : **HAZ**ard and **OP**erability analysis [4]ETA : **E**vent **T**ree **A**nalysis	

Table 1 : Example of recognized analyzes methods listed by ISO26262 [1]

Additionally, the safety analyses might also contribute to the identification of new functional or non-functional hazards not previously considered during hazard analysis and risk assessment.

6.4 Considered safety analyzes in WT3.3.1 (D331b)

As explained in chapter 5.1, the scope of WT3.3.1 is a first step to define the concepts needed for fault/failure propagation, documented in the D331a deliverable. In a second step, it is to define a tool specification for most relevant safety analysis methods that will permit to visualize and analyze the results from the fault/failure propagation (D331b deliverable).

Nevertheless to be coherent with fault/failure propagation, it was decided to select the most relevant safety analysis methods during first step and give the results in D331a deliverable.

6.4.1 Criteria-based assessment of most relevant safety analyzes methods using criteria

The different methods were assessed using different criteria as shown in the table hereafter:

YES, **NO** criterion when answer is sure **Maybe** : theoretically possible but never seen **Limited** : when it is not fully capable	Inductive methods			Deductive methods	
	FME(D)A	ETA	Markov	FTA	RBD
Capability to address ISO26262 requirements concerning qualitative / quantitative safety analyzes.					
Does this method allow performing qualitative and quantitative analyzes?	YES Qualitative FMEA Quantitative FMEDA	YES	YES	YES	YES
Can this method be performed at different architectural levels?	YES	YES	YES theoretically but very complex at low level	YES	YES
Can this method address systematic failure?	YES FMEA	YES	YES	YES but Low interest	YES but Low interest
Can this method address random failure?	YES	YES	YES	YES	YES
Can this method be used to calculate architectural metrics (SPFM & LPFM)?	YES FMEDA	Maybe but not direct	Maybe but not direct	Maybe but not direct	Maybe but not direct
Can this method be used to estimate the residual risks of safety goal violation	Yes Failure Class at part level or estimation from FMEDA	Maybe but not direct	Maybe possible but not direct	YES PMHF	Maybe but not direct :PMHF
Does this method support analysis of dependent failure?	YES	YES	YES	YES	YES
Automation capabilities					
Does this method allow mapping with architecture?	NO	NO	NO State machine	Limited Possible but restrictions	Limited (no direct mapping when representing failures)
Can local analyses be generated from models?	YES	Maybe but not direct because of success?	Maybe If state machine behavior defined in blocks	YES	YES
Can this method be transformed into another method without loss of information?	YES ETA but only for failure not success, FTA for cutset 1	Limited (only failure not success) FMEA, FTA with cutset 1 only	NO Only input for other methods	YES FMEA for cutset 1, RBD	YES FMEA for cutset 1, FTA
Can global analysis be build from local analysis?	YES	Maybe but not direct	NO Make no cense	YES Transfert gates	YES
Can this method be coupled with another analysis?	YES FTA, RBD event	YES FTA, RBD, Markov	YES FTA, RBD; ETA	YES FMEA, ETA events	YES FMEA, ETA events
Post-processing capabilities for results					
Can this method allow identifying Single Point Fault?	YES	YES	YES	YES	YES
Can this method allow identifying Safety Mechanism covering a single point Fault?	YES FMEDA	YES Safety Mechanism is a barrier	YES Safe state transition	YES AND Gate	YES Adding of parallel element
Can this method allow identifying Latent Fault?	YES FMEDA	Maybe but not direct	YES Safety Mechanism failure state	YES but not direct	YES but not direct
Does this method allow understanding and visualizing cut-sets?	YES Only cutset 1	YES Only cutset 1	NO cutset computation	YES	YES
Does this method allow understanding and visualizing failure sequence?	YES	YES	YES	YES	Maybe but not direct
Can this method be configurable to analyze and display multiple failure analysis?	NO	NO	YES	YES Cutset analysis and display	YES Cutset analysis and display
Does this method help indentifying path analysis, from Failure mode to end effect, and respective involved elements?	Limited For identification of involved elements	Limited For identification of involved elements	Limited For identification of involved elements	YES	YES

Table 2 : Synthesis table of assessment of most relevant safety analyzes methods using criteria.

The goal here was clearly not to fully describe all the safety analysis methods, because they are well known and already described in [2] and [3] , but to investigate which are the most relevant for the tool specification D331b.

The considered analysis methods in D331b shall permit, first to answer most of ISO26262 requirements concerning qualitative and quantitative analyzes, then to allow semi-automation to help users to generate safety analyzes. It shall finally offer good post-processing capabilities to analyze results and identify weaknesses.

- HAZOP and Ishikawa technique are more qualitative methods for daily life and will not be considered in the tool specification D331b. They are very limited to address ISO26262 requirements concerning safety analyses and are not very compatible with tooling.
- Failure Mode and Effect Analysis (FMEA) is an example of inductive technique, as it starts from known causes and explore possible consequences. FMEA is a well known and accepted technique in the automotive industry. FMEA Analyses in ISO26262 are generally conducted in two steps:

  1. Qualitative analysis during which failure modes and their effects are analyzed.

  2. Quantitative analysis, when dealing with HW random faults, called FME(D)A (Failure Mode Effect and Diagnostic Analysis). FME(D)A permits to calculate the architectural metrics (Single Point Fault Metrics and Latent Fault Metrics) by introducing safety mechanisms with their diagnostic coverage (detection rate of the fault) stopping or mitigating the fault propagation as proposed in the ISO26262 Part 5 Annex E [1].

  Therefore, even if full automation is maybe not reachable, FME(D)A is a serious candidate for the tool specification D331b.

- Event Tree Analysis (ETA) is a second example of inductive technique for identifying and evaluating the sequence of events in a potential accident scenario (failure and success) following the occurrence of an initiating event. This analysis technique is known in the automotive industry but not a current practice as compared with FMEA. It can be used potentially to study a specific event and to demonstrate and visualize the effectiveness of a safety mechanism (seen as barrier). It can permit to quantify results but would not permit to calculate the architectural metrics directly. Moreover the automation capabilities seem reduced.

  Therefore the interest is limited and do not present additional value compared to FME(D)A. It is not a good candidate for the tool specification D331b.

- Markov modeling is a third inductive technique suitable when the dynamic behavior of the system is needed to be studied. It can also be used to model complex interactions within the system when failure of a component can influence behavior of other components. In these two cases, traditional techniques such as FMEA, ETA, RBD or FTA are not relevant.

  Nevertheless Markov analysis technique does not permit to address all qualitative and quantitative analyses required by ISO26262. It has limited automation capabilities and requires high skills for users for results post-processing. Other kinds of methods will be anyway needed and for all these reasons, and therefore it will not be addressed in the tool specification D331b.

- Fault Tree Analysis (FTA) is a deductive analysis technique that starts from known effects and explore possible causes (sometime described as “Top Down” approach). FTA is generally qualitative in a first step, and then quantified in a second step. FTA is composed of events and logical event connectors (OR-gates, AND-gates, etc…).

  Possible results from the analysis are the listing and visualization of all combination of events (cutset) with their importance factor leading to the top event failure and the probability that this critical top event will occur during a specified time interval (when dealing with HW random faults).

FTA is a well known and accepted technique in the automotive industry. It can be used to address most of the ISO26262 requirements concerning safety analyzes, and can offer good post-treatment capabilities. Therefore, even if FTA generation seems difficult to be fully automated, FTA method is a serious candidate for the tool specification D331b.

- Reliability Block Diagram (RBD) is another kind of deductive analysis technique known in automotive but not a current practice. RBD performs the system reliability and availability analyses on large and complex systems using block diagrams to show network relationships. The structure of the reliability block diagram defines the logical interaction of failures, within a system, that are required to sustain system operation (success oriented).

  A lot of people have the preconceived idea that Reliability Block Diagrams always map with the physical arrangement of components in the system but it is not true. In certain cases when elements can have several failure modes, it is not true as illustrated below:

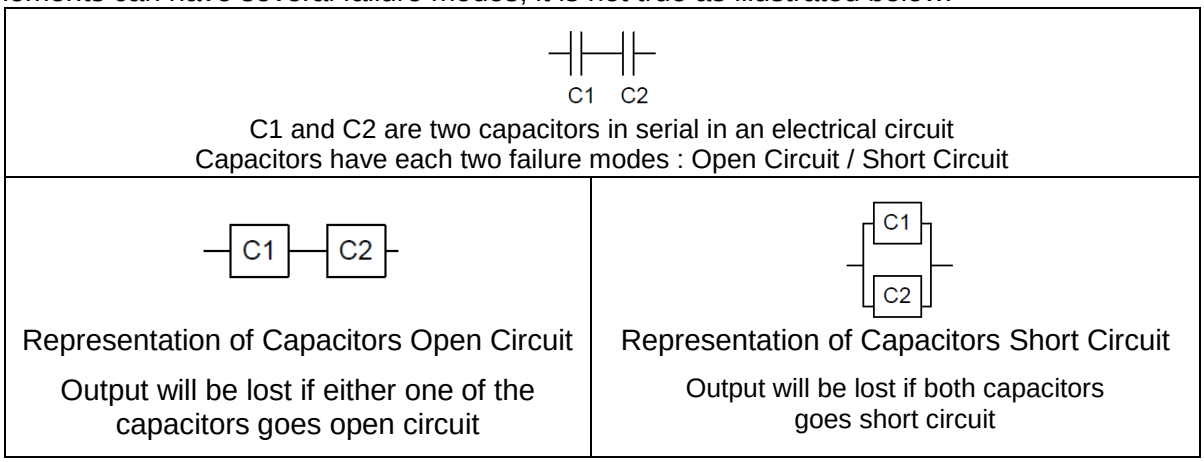


Figure 5: Example of RBD for 2 capacitors with several failure modes

To evaluate an RBD diagram there must be only one failure mode represented for each element. For elements with more than one failure mode, separate RBD diagrams must be drawn, one for each failure mode to avoid dependency problems. As in our systems, there is always more than one failure mode per element, the direct mapping between physical architecture and RBD will be unusual.

Therefore the interest in Reliability Block Diagram is limited and do not present additional value compared to Fault Tree Analysis. It is not a good candidate for the tool specification D331b.

6.4.2 Final choice for D331b

The ISO26262 (see [1]) required that inductive methods have to be used whatever the criticality (ASIL A to ASIL D) and deductive methods for ASIL C and ASIL D as shown in the table hereafter:

	ASIL A	ASIL B	ASIL C	ASIL D
Inductive methods	Required	Required	Required	Required
Deductive methods	Nothing required or recommended	Recommended	Required	Required

Table 3 : Type of safety analysis methods required or recommended by ISO26262 [1]

Therefore for critical systems, we need to select at least one inductive method and one deductive method. Considering the results from chapter 6.4.1, for the tool specification D331b, as best comprise, the methods proposed will be derived from **FME(D)A** for inductive technique and **FTA** for deductive technique.

7 Problematic of evaluating malfunctioning behavior in distributed developments

7.1 Illustration through an example

Fault propagation in complex integrated systems is a challenge. As illustrated in *Figure 3* and *Figure 4* from chapter 6.2, a lot of people think that when we analyzed a fault in a system, we always investigate if this fault can potentially violates a safety goal. In a simplified system such as described in ISO26262 Part 5 Annex E [1] made of a single ECU with sensors and actuators, this is possible, but in reality systems are often made of several ECUs, and therefore investigations are much more complex.

Moreover, most of the time, there is one system responsible (e.g. OEM), and the different ECUs are developed by different Tier 01 suppliers. Tier 01 suppliers themselves can buy SW or HW development from a Tier 02 supplier. It is a so called distributed development. In this context, the propagation of a fault in a HW element developed by a Tier 02 up to the violation of a safety goal is not so obvious.

To illustrate the problematic of distributed development, let us take the example of a system whose desired function should consist in switching ON/OFF the front lights (low beams) of a car. If someone is driving by night in a dark area (operational situation) and the front lights are spuriously lost (malfunctioning behavior leading to hazard), it can be easily understood that this can lead to an hazardous event (ASIL B) for the driver, the other occupants of the car and potentially also people outside of the car. From the hazard analysis and risk assessment, safety goal corresponding to this hazardous event will be defined at our top level safety requirement. As this stage, the system is considered as a “black” box (we do not know how the desired function will be realized).

Then the system responsible will define first a functional architecture (not shown here) which will quickly lead to a preliminary architecture as shown hereafter that can realize the functional architecture. Of course, there is not only one unique technical solution to realize the functional architecture and therefore variants are possible.

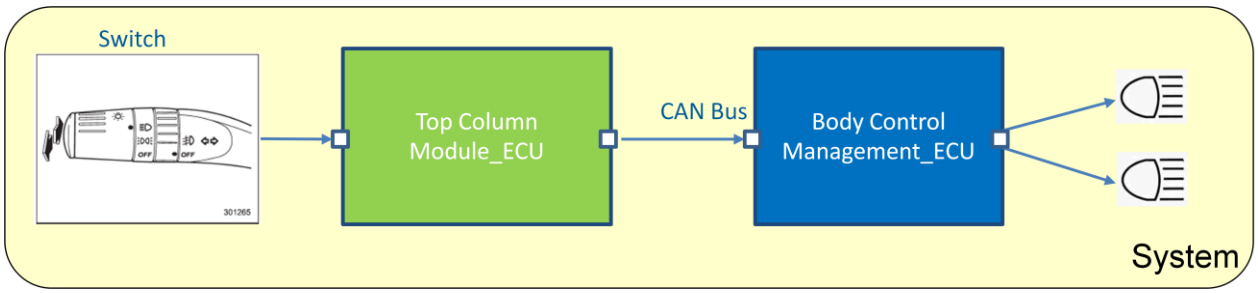
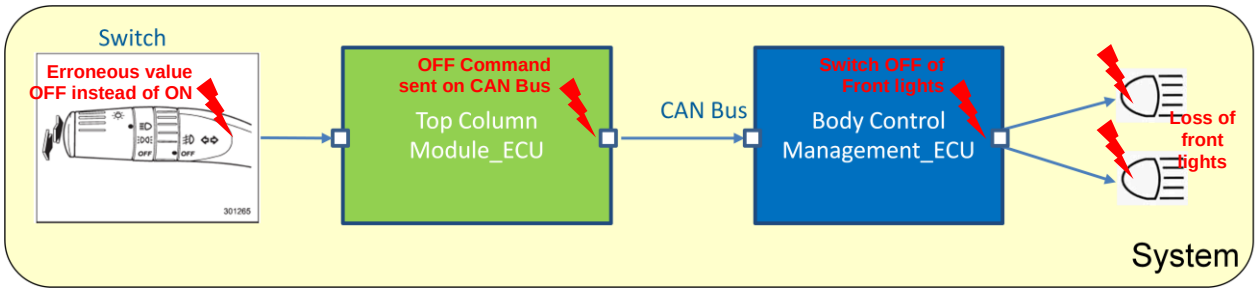


Figure 6: Example of Preliminary Architecture of front lighting switch system

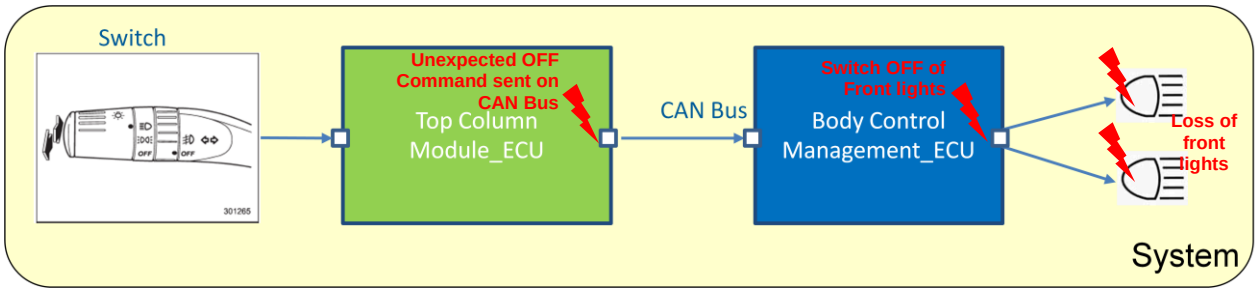
In this example, the driver can activate a switch (ring) on a lever and set ON/OFF the front lights (low beams). The corresponding electrical information is acquired by the Top Column Module ECU which then elaborates a Command that is sent on the CAN Bus. The Body Control Management ECU receives the Command from the CAN Bus and executes it.

Based on a preliminary architecture as defined in *Figure 6*, the system responsible will have to identify, using relevant safety analyzes, the different malfunctions on the output of the components of its system that could propagate within the system and could violate the safety goal.

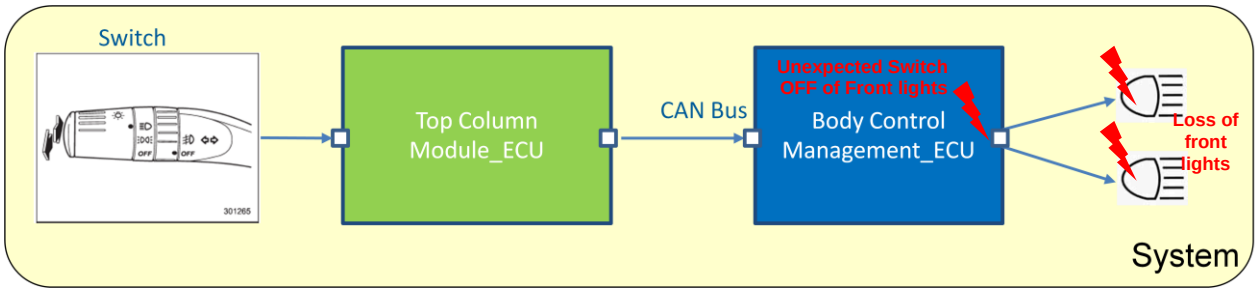
- A malfunction of the output of the switch (e.g. erroneous value: OFF instead of ON) will be propagated to the Top Column Module ECU that will send an OFF value on the CAN Bus. Then the Body Control Management ECU will receive the erroneous value and will switch OFF the front lights. The initial switch malfunction will finally propagate without safety mechanism and lead to the violation of the safety goal.



- In the same manner, a malfunction of the output of the Top Column Module ECU (e.g. unexpected OFF command sent on the CAN bus) will be received by the Body Control Management ECU that will switch OFF the front lights. The initial malfunction will finally propagate without safety mechanism and lead to the violation of the safety goal.



- In the same manner, a malfunction of the output of the Body Control Management ECU (e.g. unexpected OFF command execution) will switch OFF the front lights. The initial malfunction will finally propagate without safety mechanism and lead to the violation of the safety goal.



- And also if both front light modules could have malfunction at the same time, it will lead to a loss of front lights and will lead to the violation of the safety goal without safety mechanism.

In this simple example, a safety mechanism can be implemented in the Top Column Module ECU to detect a switch malfunction. It will be translated into one functional safety requirement:
TCM-FSR_001: TCM shall send a light parameter “Invalid” on the CAN bus in case of malfunction detection of lighting switch acquisition: ASIL B

And also to be sure that it does not lead to a loss of light, another functional safety requirement is needed for the Body Control Management ECU.
BCM_FSR_001 : When ignition switch is ON, BCM shall switch light ON if it receives a light parameter “Invalid” on the CAN bus : ASIL B

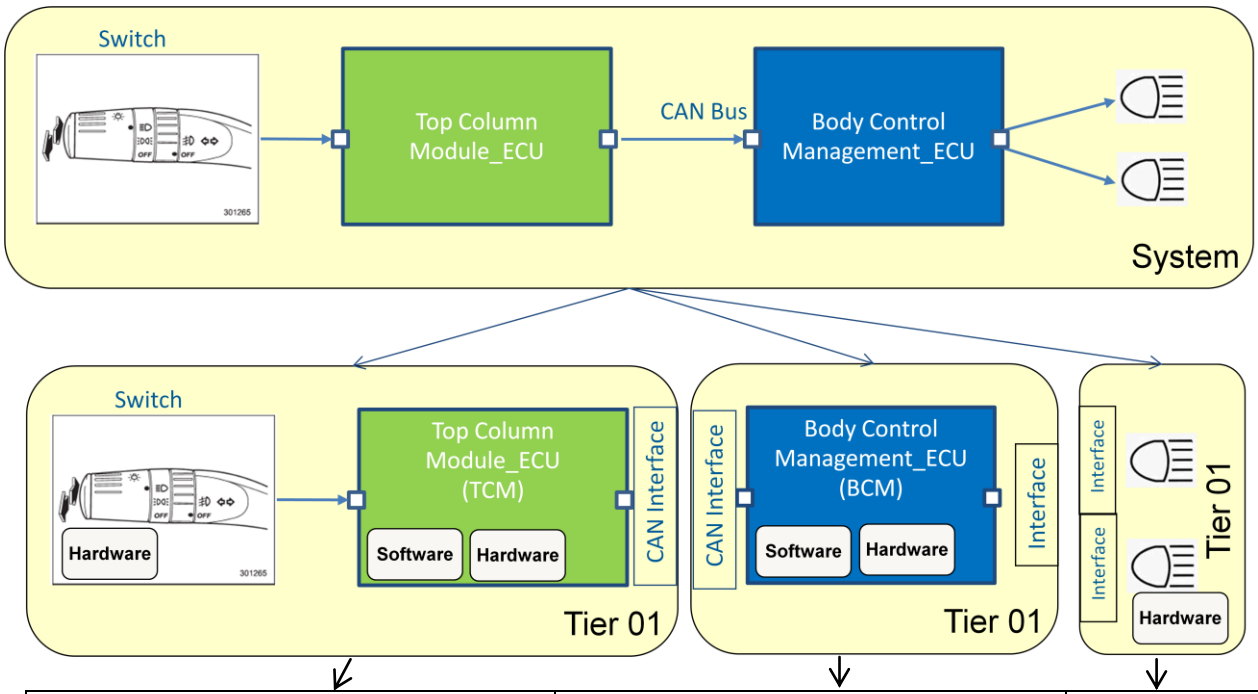
That means that finally a loss of front lights in our system could mainly be due to a malfunction of the output of the Top Column Module ECU that could spuriously send an OFF command on the CAN Bus **OR** due to a malfunction of the output of the Body Management Control ECU that could spuriously switch OFF the front lights **OR** simultaneous malfunction of both Front lights.

As the criticality of the safety goal violated in this example is ASIL B, ISO26262 recommends only some metrics targets as shown in the Table hereafter:

	ASIL A	ASIL B	ASIL C	ASIL D
Single Point Fault Metric (SPFM)	Nothing required or recommended	≥ 90% Recommended	≥ 97% Required	≥ 99% Required
Latent Fault Metric (LFM)	Nothing required or recommended	≥ 60% Recommended	≥ 80% Recommended	≥ 90% Required
Residual risk Metric	Nothing required or recommended	$< 10^{-7}$ / h Recommended	$< 10^{-7}$ / h Required	$< 10^{-8}$ / h Required

Table 4 : Metrics allocation required or recommended by ISO26262 [1]

And if the system responsible (most of the time the OEM) decides to not perform the system development itself, but uses developments distributed to several suppliers (Tier 01). In this situation, it would be necessary to define the different interfaces between elements of the systems, as well as the critical malfunctions with associated allocated metrics.



Malfunction of output of the Top Column Module ECU to control : TCM-F002-MF001 : Unexpected sending of OFF parameter on CAN Bus : ASIL B With max. 5e-8 /h as residual risk : SPFM = 90% ; LFM = 60% **Could be also a functional safety requirement**	**Malfunction of output of the Body control Management ECU to control :** TCM-F002-MF001 : Unexpected switch OFF of both front lights : ASIL B With max. 5e-8 /h as residual risk : SPFM = 90% ; LFM = 60% **Could be also a functional safety requirement**	Reliability targets : 1e-6/h for each front light
TCM Supplier : Valeo	BCM Supplier : Continental	Front Light supplier : Other

Figure 7: Example of requirements allocation from OEM to suppliers in a distributed development

Therefore, when as in the example, the Top Column Module ECU supplier receives the working specification from the OEM, it will have to implement safety mechanisms in its product. These safety mechanisms shall stop or mitigate the propagation of SW and HW faults/failures leading to specified malfunctions outside of its component perimeter as shown hereafter:

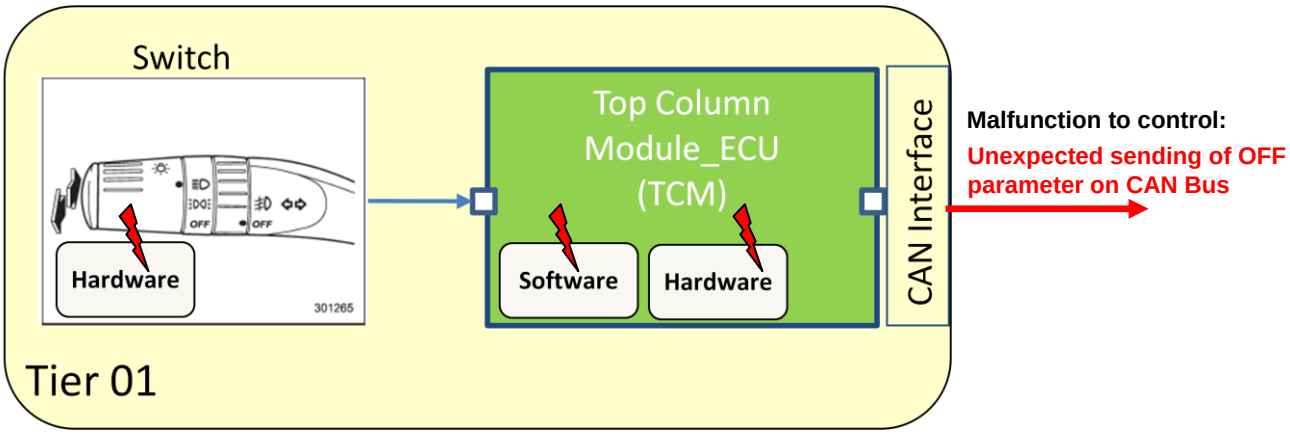


Figure 8: Example of component perimeter known by a Tier 01 in distributed development

And at this level, it is never investigated at supplier level if it can lead or not to the violation of a safety goal. Indeed the system behavior is under the OEM responsibility and is not fully known by the supplier (Tier 01).

Of course, when safety analyses are performed inside the component to be developed and when new malfunctions propagating outside are discovered, the system responsible shall be immediately informed in order to analyze the impact at higher level.

To manage such scenario, a generic contract-based approach is proposed in chapter 7.2 in order to improve the formalism of expected behavior in distributed developments.

7.2 Contracts Approach in distributed developments

Contract-based design is a methodology that allows compositional reasoning. The methodology can be applied for different viewpoints like functional and/or dysfunctional behavior. It allows formal specification and analysis of component characteristics for safety-related systems. Component specifications given by contracts explicitly distinguish between promised behavioral characteristics which are guaranteed as long as behavior assumed for the component context hold. Assumptions and promises of contracts can be formally described e.g. by using a pattern-based specification language. These patterns allow the specification of safety-requirements which guarantee safety-concepts for components under the assumption that specific combinations of defined failures do not occur or are mitigated by safety mechanisms. Combination of contracts can be analyzed for a set of sub-components in a virtual integration test on implying contracts of a parent component composed by these sub-components.

7.2.1 Contracts Historical background

Many of the concepts for contract-based component design are results of the SPEEDS project (Speculative and Exploratory Design in Systems Engineering, EU, 6th Framework) [4], and draw on classic research on compositionality as well as more recent ones. Further activities regarding contract-based requirements engineering using a formal pattern-based requirements specification language (RSL) were performed within the project CESAR (Cost-Efficient methods and processes for SAfety Relevant embedded systems, ARTEMIS JU) [5].

7.2.2 Contracts basic description

Contract based modeling was developed in order to meet the requirements of cooperative systems. The idea is simple. A system is described by a set of components as depicted in *Figure 9*. A component can be decomposed into sub-components as parts of that component. Each component part is a system element that is responsible to provide a number of well-defined services. However, in order to do so they need to rely on the properties of other interfacing components (i.e. they have assumptions on the behavior of the environment in which they are embedded). In turn they provide guarantees to other partners about their own behavior. Contract based specification methods address these issues by distinguishing what a component relies on and what it delivers. This kind of specification is especially useful when no actual implementation exits, for example during early development phases when only requirements and their relations are known, and can be used to establish the preliminary architecture. Due to the defined relationships between context components and sub-components as well as those among sub-components it is possible to evaluate the impact of the overall architecture layout on the different system requirements.

Having a complete and well-defined description of the interface of a component enhances the development of large systems. The definition of such well defined interfaces is a means that improves scalability, compositionality and abstraction. Re-use of components and design patterns, developing libraries of design components and better support for using COTS (Components Off The Shelf) are use-cases that benefit from this approach. Existing designs can be easily changed in order to adapt for new requirements or to support product family development.

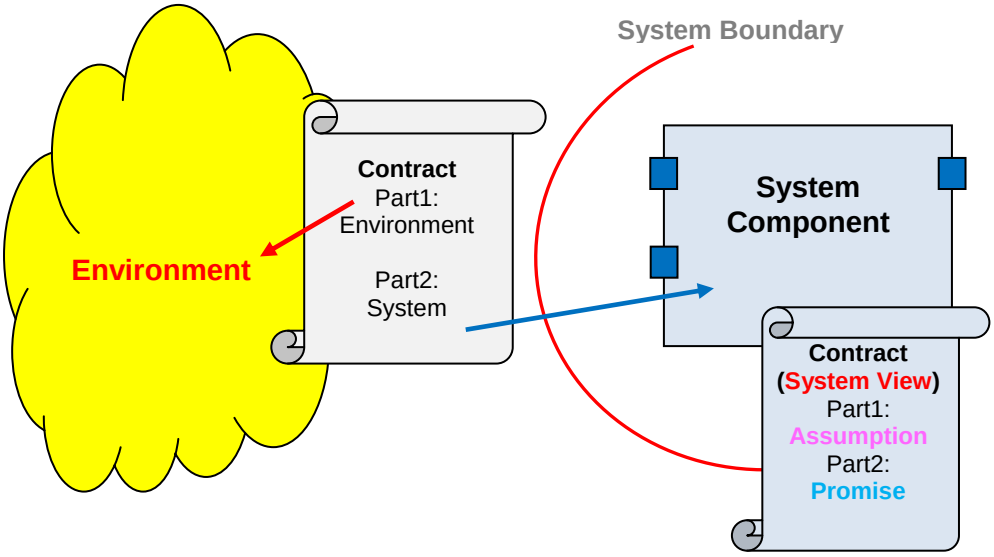


Figure 9: SPEEDS Contract based specification of interface properties [4]

Furthermore contract based modeling provides the necessary infrastructure for efficient compositional analyses, thus avoiding many of the complexity problems otherwise associated with large models. Evaluating the impact of different design choices and alternative implementations of a component helps in avoiding unnecessary cycles in the design process. Compatibility of components can already be tested during the early design phases.

Contract based modeling can be started early in the design process and supports an incremental design evolution with gradual improvements going from abstract models towards more and more refined ones. It enables the specification of well defined interface between components so that:

- each component (possibly collections of components) is associated with a contract that specifies the interface the component uses to interact with the environment
- contracts consists of a number of assumption-promise pairs

- the implementation of each component can be verified on its own, formal verification techniques can be used to validate that the component fulfils its contract
- compositional analysis of system-level properties can be based entirely on the contracts of the individual components, so that issues of complexity and heterogeneity that arise from detailed implementation can be avoided
- functional aspects of the system as well as non-functional properties, such as safety and reliability, can be addressed.

In the SPEEDS methodology [4] a virtual integration test composes the contracts and then verifies whether this assembly is consistent with the contracts of their parent component. This is the fundamental building block underlying the compositional analysis that ensure that the decomposition step was correct, in the sense that the defined sub-components will work together and satisfy the requirements of their parent component.

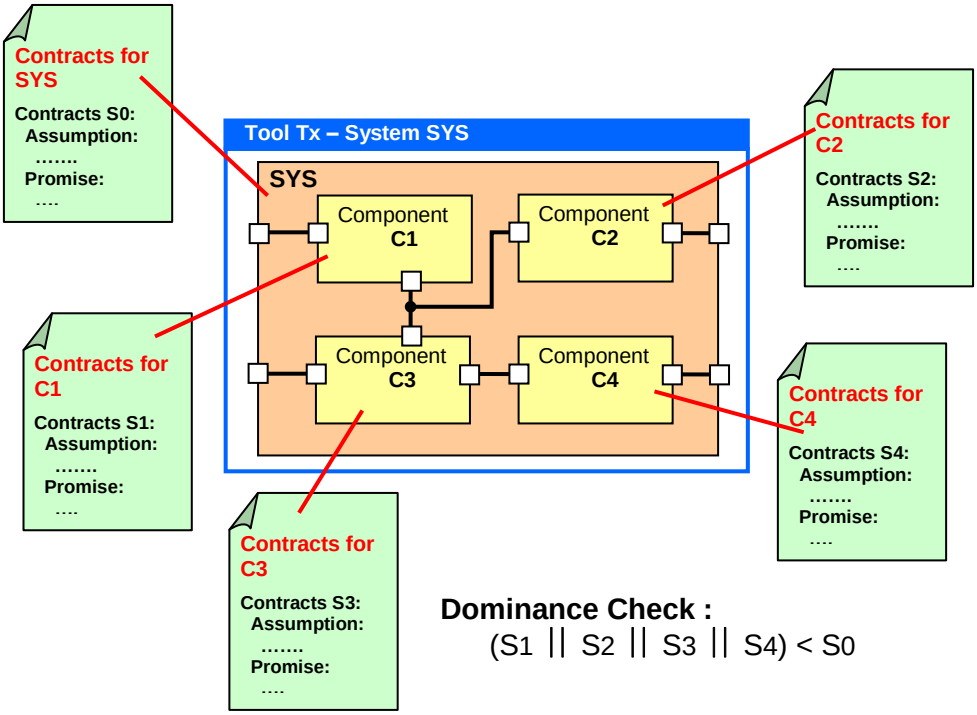


Figure 10: Virtual Integration of Heterogeneous Rich Components (HRC) [4]

Based on contracts, in particular two kinds of analyses are part of the virtual integration:

- **Compatibility Analysis:** This analysis verifies whether the assumptions and promises of interconnected respectively neighboring components are compatible with each other.

- **Entailment Analysis:** This kind of analysis, also known as dominance check, composes the contracts of a set of interconnected components and then verifies whether this assembly is consistent with the contracts of their parent component.

In the case of entailment, one can say that the contracts of the sub-components imply the contracts of their parent component. Both analyses together enable the developer to ensure that the decomposition step was correct, in the sense that the defined sub-components will work together and satisfy the requirements of their parent component, provided that the sub-components satisfy their own contracts. After the incremental verification and validation step, all derived sub-components are sufficiently characterized and can be designed independently. The developer now has the alternatives to iterate the decomposition step again, implement the sub-components or select an existing implementation from a library. The developer must ensure that any implementation that is provided, either newly developed or selected from a library, satisfies the sub-component‘s contracts.

7.2.3 Contracts basic elements

The following chapter will give an overview of the basic elements considered by contract-based component design. Contracts are specifications for components with promised characteristics for an assumed environment of that component. Such specifications address characteristics which are observable at the interface of a component.

Contract

A contract is a requirement which is structured in two parts, a promise, which must hold provided that assumed characteristics of the component's environment are fulfilled, the assumption. Such a contract-based specification therefore distinguishes between assumptions on the usage context of a component and promised characteristics for the specified usage context. This is the basic principle of contract-based design approach in the SPEEDS project and of the HRC meta-model specification [6]. Contracts have two kinds of assertions, namely **assumption** and **promises**. These assertions can be described informally or in a formal way e.g. by using a pattern-based requirement specification language (RSL).

Promise

The promise describes guaranteed functional and non-functional characteristics in a contract-based specification. The promise of a contract, assigned to a component, has to hold provided that the assumptions are satisfied. If an assumption is not fulfilled then the promise does not necessarily hold.

Assumption

An assumption describes the assumed design environment for a contract-based specification. Assumptions characterize the allowed usage context for a component as well as specific use cases within the allowed usage. If a component is used accordingly to its assumptions, it will guarantee the behavior specified by the promise.

Component

A contract is used to specify characteristics of a system, a sub-system or an element with defined interfaces, which is called a component. These component can be of various nature, like logical components that do not distinguish between hardware and software (typical used for functional safety concepts) or technical components (used to describe the technical safety concept). A component defines a set of interfaces which are addressed by the contracts assigned to the component. If a component is considered as a black-box then only its interfaces and its contracts are known. Otherwise a component can be decomposed into a composition of sub-components. Each sub-component can have its own contracts. In a clean architecture design the combination of contracts assigned to the sub-components implies the contracts of the parent component.

7.2.4 Contracts Failure Description

Pattern-based Safety Contracts are a means to define fault containment properties for a system's safety concept. The patterns describe how failures are contained and evaluate the impact on the top-level safety requirements. Containment is meant in terms of failure propagation. Failures can be contained (under given assumptions, defined in the contract) in a part of the system and does not further propagate. This kind of analysis can be done very early in the design process using abstract representation of the component and will be used to derive additional safety requirements. With the pattern presented in this chapter it is possible to create a specification of the containment or propagation of faults.

The main concepts used for this pattern are faults and failures and a combination of them in an expression. The pattern can be used to describe the combinations of faults in an assumption and combinations of failures or malfunctions in a promise of a safety-contract. As long as the specified assumption holds the non-occurrence of the specified failure is guaranteed for the system. Yet, if

the assumption of the contract is violated, i.e. a combination of faults occurs, which is not expected, than the promise cannot be guaranteed anymore and the failure may occur anyway.

The following attributes are used in the pattern:

- Failure-Condition
- Degradation modes
  - A mode expression consists of a mode variable, a mode name and a relational operator("**= =**", "**!=**")
  - Example: DM==normal, DM != detected
- Expression, Expression Sets
  - An expression is either a failure-condition or a mode expression
  - An expression set is a set of expressions
  - Example: {fail1, fail2, fail3 during dm=normal}
- perm()
  - If this operator is applied to an expression, the expression holds for all future states of the path

Pattern S1:	***none of*** *{<expr-set1>, …, <expr-setn>}* ***occur***
This pattern is used to describe the traces that are accepted / not accepted. Any trace that contains all elements of one expr-set is not accepted by the pattern. A trace is a assignment of values to the systems ports over a defined time-frame. **Example Pattern:** ***none of*** *{{f1,f2}, {f3,f4}}* ***occur***	

Figure 11: Example of failure pattern

7.2.5 Contracts Example

A contract is a requirement with a specific structure with an assumption and a promise. The concept of contracts makes assumptions about context explicit, which allows assigning responsibilities in the development processes. Typically contracts are derived from top-level system requirements that may be captured in external requirements management tools like e. g. DOORS. Keeping those requirements separate from an architecture model may be required by the certification processes.

An example of architecture for an electric drive-train is provided hereafter:

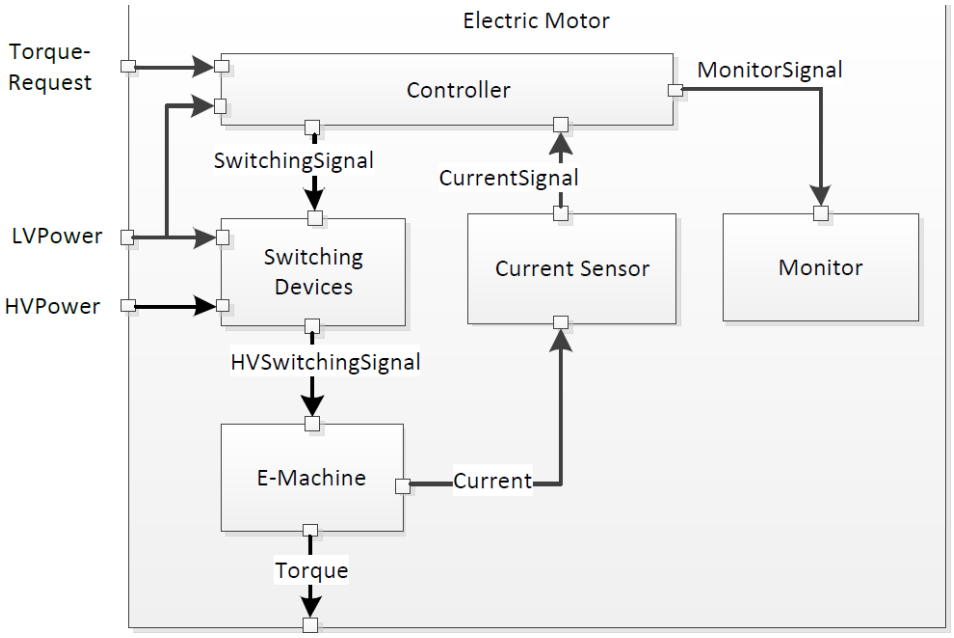


Figure 12: Example an electric drive-train architecture

An example safety contract is the following:

Assumption:	***none of*** *{CurrentSensor_mf}* ***occur***
Guarantee:	***none of*** *{CurrentSignal_fail}* ***occur***

This contract specifies the fault containment property of the current sensor. If there is no internal malfunction in the current sensor, there exists no failure at the output, the current signal. The various faults can be typically obtained from the safety manuals of the components. These faults can be considered as the leaf-nodes in a fault tree. In fact, the cut-sets that are violating a given top-level safety requirement can be calculated and represented as a fault tree.

7.2.6 Contracts and Loop management

A typical issue in system design is the management of control loops. The output of one component is an input of a component that is connected upstream. A failure resulting from the loop behavior (e.g. oscillation) is not detected locally by the components. The combination of the component behaviors connected in the loop leads to failure of the system to which the components are composed. This issue will later be shown for HiP-HOPS (see chapter 8.1.6) and for Altarica (see chapter 8.2.6). Contract-based specifications have semantics defining allowed traces of a system's behavior. According to Hungar [7] the *trace semantics permits to directly relate behaviors and specifications: If all traces of the behavior of a component adhere to its specification, the component is correct*. A trace is the assignment of values to all system ports in a given time frame. Traces do not specify the functional behavior of a system but represent the different states a system can be in, and the temporal relations between these states. A system's implementation that consists of a composition of sub-components connected in a loop can have a behavior with traces that are allowed by a contract-based system specification. If the traces are allowed, then the implementation with the sub-components connected in a loop entails the system contract and is correct from the system's point of view. Whether the actual behavior of a system adheres to the specification is subject to an analysis.

7.2.7 Contracts and failure propagation mitigation with safety mechanism

The pattern-based safety-contract approach allows specifying a safety concept in terms of failure modes, failure rates, their propagation, and the usage of counter measures expressed in assumptions and promises. This method allows verifying decomposition and integration of safety concepts. The safety concept can be seen as requirements on safety that do not want to force a special implementation but requires a defined behavior regarding failure propagation. Typical requirements are the non-existence of a single-point-of-failure. A safety specification can already include partial details about countermeasures like voting or validity checks to realize required fault containment. Expressing such elements is in particular important for verifying if the solution that has been created by a supplier still fits into the overall safety concept. This automated check can prove if the assumptions of the supplier on the behavior of the surrounding components/items are sufficient or already to strict.

A typical way of specifying safety mechanisms is depicted with the following contract:

A:	none of {{mf_11 ..., mf_1n },...,{mf_n1 ...,mf_nn }} occurs.
P:	{output_fail } does not occur.

If all components that take part in the detection and mitigation of the failure do not have any malfunctions, the resulting failure does not occur. Since safety mechanisms usually implements some kind of redundancy the possible malfunction combination are not necessary the Cartesian product of all existent malfunctions, but are limited to some critical combinations where the nominal functionality and the safety mechanisms fail in the same run of the system.

When a safety specification is formalized it is important to distinguish between the assumptions under which a safety concept has to hold and the promise what a component – that later will implement this specification – shall do to keep the system safe. This principle enables the supplier to build a system without having to communicate with the integrator on an informally ambiguous way. E.g. a failure rate for failure modes on a component can only be met by an implementation if there is knowledge about the failure rates of propagated failure modes on the input Ports of the component. Same applies for argumentations not taking failure rates into account: The non existence of a failure mode on a port can only be shown under the assumption that only a known number of faults can occur at the same time.

In order to express the relationships between the failure modes and the counter measures, thus implementing a technical safety concept, formalism is needed that allows the statement of the assumptions as well as the promises in a semantically well defined and unambiguous way. For a pattern-based specification of safety-requirements only few patterns are needed to define error propagation and counter measure functionality.

There are two main scenarios where completeness and consistency of a safety specification needs to be checked:

- On the one hand if the OEM refines a system to distribute the sub-parts to one or more suppliers. In this case is important to prove that the refinement still satisfies the upper level safety goals.
- On the other hand a supplier can offer a solution (that could also have different assumptions as actually needed in the development process) that refines the OEMs view on the system. In this case it is important to prove that this externally developed component fit into the already existing component structure and the top-level safety goals are still satisfied.

7.2.8 Conclusions on Contracts

Contracts can be used to specify and analyze all kinds of safety-requirements required by the ISO26262 in a formalized way [8]. The contract methodology allows the specification and analysis of formal safety-requirements including failure propagation and mitigation with safety mechanism. Safety-contracts can be used to define combinations of faults for which the occurrence of a failure shall be excluded.

Currently no limitation towards the representation of the different ISO 26262 fault and failure types are known. In each contract the faults are considered independently, but the dependence between faults can be expressed in an additional contract which is then respected in all the used analyzes. Also latent faults can be expressed and handled.

Furthermore the contract based approach can be applied to the concept of an “Safety Element out of Context” (SEooC) as defined in the ISO 26262. For these elements the expected behavior of the environment can be formalized in the assumptions of the contracts that describe the element. If the element is placed in an item, the integration can be automatically be checked using the entailment analysis, also called virtual integration check.

The correct implementation of a system’s safety contracts, dealing with faults or failures to be excluded, is subject to a safety analysis. Contract-based methods like entailment or consistency analysis can be applied. The consistency check can detect contradicting requirements. Another possibility is to perform safety analyses generated by fault and propagation languages such as HiP-HOPS and AltaRica as seen in chapter 8.

The approach proposed in the SAFE extension for fault and failure propagation in chapter 10, will be to extend EAST-ADL to perform such contract mind description, and to define failure requirement for failure propagation language as implemented in D331b next document released.

8 Fault and Propagation language overview and considered method in WT3.3.1

The following chapters will describe an overview of the two most interesting model based and safety analysis based methods as state of the art. Both of them provide a fault and propagation language.

8.1 HiP-HOPS

"HiP-HOPS" as Hierarchically Performed Hazard Origin & Propagation Studies, is a safety analysis methodology that allows automating generation of fault trees for fault tree analysis (FTA) and for failure mode and effect representation (FMEA) constructed from system topological models annotated with respective component failure data.

8.1.1 HiP-HOPS Historical background

The "Distributed, Reliable and Intelligent Systems" research group from the University of Hull in United Kingdom has been intensively developing novel techniques and tools supporting the quality and dependability analysis, optimization and improve testing of highly critical system in various industries such as avionic, nuclear plan and process industries.

Since the last decades, the DRIS [9] team builds important contributions to HiP-HOPS techniques, with definition of novel algorithms for bottom up dependability analysis via automatic synthesis of Fault Trees and Failure Models and Effects Analyses (FMEAs). They also defined a method for temporal logic that enables assessment of the effects of sequences of faults in Fault Tree Analysis (FTA) called Pandora [10]. HiP-HOPS methodology can be applied on any type of system design, modeled as a topology of any type of component composed to build a system. HiP-HOPS defines semantic to capture the annotation of appropriate failure description of component and their local effects, and computed propagation of the failure in the system based on the relation defined in the topology of the system. Then it allows automatic generation of common safety analysis like Fault Tree Analysis and Failure Modes and Effects Analysis (FMEA). Different HiP-HOPS prototypes have been implemented in tools like Matlab Simulink and SimulationX by ITI GmbH.

HiP-HOPS was adopted by automotive research consortium of European project (ATESST, ATESST2, MAENAD), as error modeling extension integrated into the EAST-ADL standard (as the architecture description language for design of vehicle control systems).

In 2011, the HiP-HOPS software tool was commercially launched by ITI GmbH, a CAE software house and the author of the SimulationX tool, now integrating HiP-HOPS perspective and toolset. In addition, HiP-HOPS licenses have already been sold to large engineering companies which include Toyota, Honeywell, FEV automotive and ALL4TEC.

8.1.2 HiP-HOPS basic description

HiP-HOPS technique [11] is a safety analysis methodology based on compositional failure analysis, where the system failure models are constructed from component failure models using a process of composition. The component are modeled according to a dedicated HiP-HOPS failure semantic to represent component output deviation according internal failure and input deviation defined as logical Boolean equation (see next chapter Failure Description for large details) in order to represent the behavior of negative view (also called dysfunctional) of component (in opposition to the positive view representing the normal functional behavior).

The failure behavior of each component is composed according to the component hierarchy and topology organization of the system. The failure propagation between components is then generated in order to automate and simplify standard safety analysis techniques, as depicted in *Figure 13***.** This concept is today applied into the HiP-HOPS toolbox in order to build automatically Fault Tree Analysis and Failure Modes and Effects Analysis (FMEA).

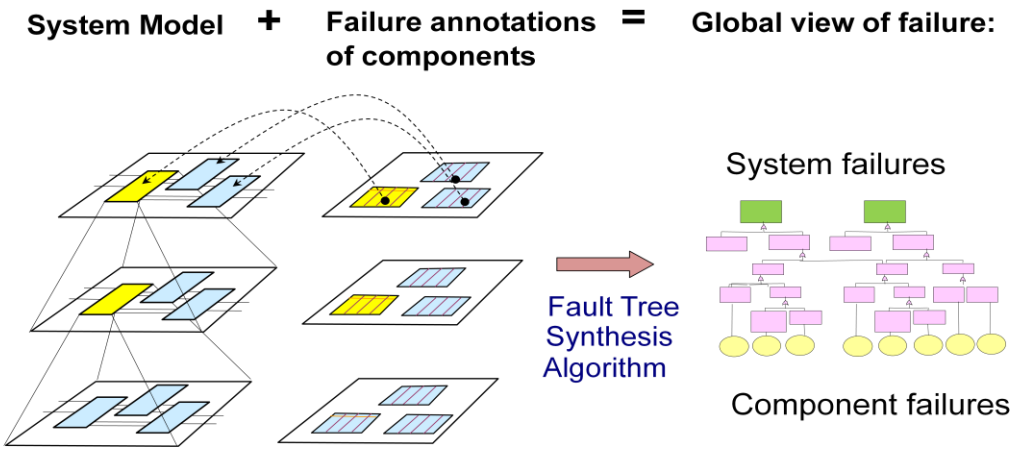


Figure 13: HiP-HOPS methods overview for Fault Tree Synthesis

The basic modeling of the HiP-HOPS tool is independent of any tool implementation. It has been defined according to an XML description in order to interact with the HiP-HOPS engine synthesis. The principle of HiP-HOPS synthesis is to work backward from system's outputs (or the hazard's definition) with combination of miniature component fault trees. A typical miniature component fault tree would be the representation of the internal relation defining the component failure behavior. The top elements are its outputs deviation, the inputs deviation and internal failures represent the leaf nodes. The intermediate node would represent relationship of the various elements defined from the Boolean logic expression of the component failure (as failure data). It is equivalent to the manual capture of a fault tree of a component.

The synthesis algorithm is working backwards through the model from system output, and then combining the miniature fault trees from components, and propagating the input/output relationship recursively within the trees relation. This could end to an error as a missing or incorrect relationship in failure class called dangling deviation situations. Information is available to highlight dangling situations and warn users about possible contradictions. This synthesis is performed using a mixture of classical logical reduction techniques, with application of logical rules to reduce complex expressions, and improved by application of more techniques, as the use of Binary Decision Diagrams (BDDs), to break down the fault trees into a simpler form.

In addition, both qualitative (as logical view and cut-set analysis) and quantitative (numerical-probabilistic based on unavailability formula capturing failure rate or repair rate of basic events) analyses are carried out from the fault trees. FMEAs are then built from extraction of cut-sets of first order that are rearranged. All results are displayed in HTML format as shown hereafter:

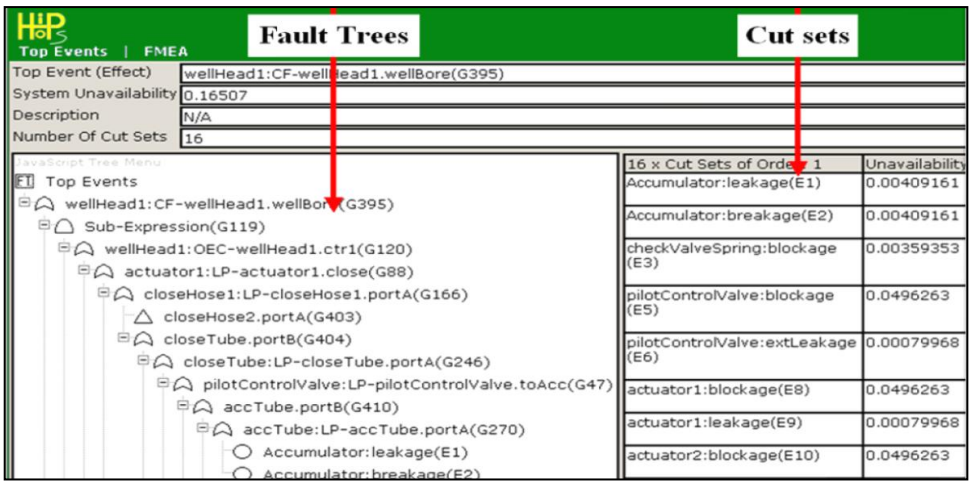


Figure 14: FTA output view from HiP-HOPS toolset

8.1.3 HiP-HOPS basic elements

The following chapter will give an overview of the basic elements managed by HiP-HOPS. Due to copyright and Intellectual property (IP) protection, it will not describe the exact XML format as language definition interpreted by the HiP-HOPS tool. This chapter will explain the concept element useful to be controlled in relation to an architecture language or to a failure language modeling. In addition the concept below has been used from ATESST project to perform transformation from EAST-ADLV2 elements to HiP-HOPS XML format for safety analysis.

Model:
It is the top level of the hierarchy encapsulating all elements for the analysis of an XML file.

Hazard:
It describes the top level failure of the system; it can be a list of hazards. It includes the failure logic of the hazard link to at least one output deviation of a component (see Failure description for more details on the syntax).

System:
It is a hierarchy of elements representing the system to analyze. It is composed of components and lines representing connection between components for failure propagation. Note that a system can be composed of systems.

Component:
It is the elementary artifact of the system hierarchy. Components include a list of ports for component communication that are referenced by lines for definition of propagation of outputs deviation. In addition Components include a reference to the field Implementation describing the definition of the expression of failure component behavior.

Lines:
This element represents the propagation link of the fault via the component port. It is composed by a list of connections being referenced by the component port. Optionally the connection can be directed to causal and non-causal relations. Furthermore a Line representing the connection can include a dedicated failure expression representing failure propagation on the line with the same semantic as Boolean expression for output deviation. Notice that this failure logic expression do not have explicit basic event, as intrinsic Lines failures, but failure relation between ports connected by the line.

Failure Data:
It represents the failure behavior of an implementation of a component. It is composed of basic events representing the intrinsic component failure behavior, of output deviation embedding the logic expression for the fault propagation through the output port of one component, and of **exported propagation** representing direct failure propagation, as for example used for hardware to software propagation (see description hereafter).

Basic Event:
This element represents intrinsic component failure behavior as systematic fault or random fault with possible quantified value for hardware failure rate. They are identified below in the failure expression as “Internal Failure” (see failure description chapter 8.1.4).

Output deviation:
It describes the logical failure of a component as Boolean logic expression that link cause as basic event and/or input deviation to the fault propagated through the output port of the component defined as a failure expression (see failure description chapter 8.1.4 for semantic description). It may include a tag to indicate, as an example for hazard, that the output failure is the top level failure.

Exported Propagation:
It describes the logical failure for any element (such as allocation for example) as Boolean logic expression defines with the same semantic of output deviation.

Furthermore, the syntax offers more concepts than listed above, as for example the concept of ***perspective*** capable to connect different view of system such as hardware and software, joined with a concept of ***allocation*** for multiple perspective and **CCF** for common cause of failure. A concept of **implementation** of component allows defining several implementations for component failure behavior and a field ***Optimization parameter*** permits to control an optimizer engine. Thanks to these advanced features and especially implementation and optimization concept, an optimizer is available in the solver to allow system exploration and ASIL decomposition based on alternative failure behavior [11].

8.1.4 HiP-HOPS Failure Description

The failure logic expression is built with the following syntax:

- Output Deviation = Internal Failures AND/OR Input Deviations
- Operator XOR & NOT are provisioned but not yet supported in the expression.
- Operand support also
    - A jump to an output deviation of a component in the hierarchy of the system defined by LocalGoto(output deviation). Possible jump out a system are possible with GlobalGoto. These two operands shall carefully be used as they induce inconsistency in propagation and may lead to HiP-HOPS engine error.
    - Line failure propagation represented as FromAllocation(propagation), where propagation is the name defined in the exported propagation field.

This component failure shall be expressed as a set of expressions from the above syntax, capturing the deviation of each outputs deviation of a component. The input and output deviations are defined into different failure classes:

- **Omission failure** means failure to provide the data, abbreviated as O
- **Commission failure** for unexpected delivery of the data, abbreviated as C
- Value data corrupted for design malfunction abbreviated as V, LV for low value and HV for high value
- Timing failure of design as T with no temporal indication but simple tag, E for early and L for Late
- Potentially any other classes that may be defined in XML using the correct schema.

The syntax for the definition of input and output deviation is <Failure Class> - <Port name>, where Port name is the name of a port defined in the component. Finally the port can support parameter that can be addressed via the port name as - <Port name> - <Parameter> (O_out1-param1 = O_in1_param1).

The HiP-HOPS propagator pattern requires one expression per failure class with a minimum expression defined below

- O-out1 = O-in; C-out1 = C-in; V-out1 = V-in

A proposal of expansion to describe complex functions has been proposed in [12] with the concept of General Failure Expression that can be introduced in HiP-HOPS. This concept can be generalized for any improvement on the top of the HiP-HOPS XML format in order to bring a large context of extension of the failure expression and facilitate the definition of the failure propagation.

The proposed General Failure Expression helps to abstract the above description with more generic expression of the component failure behavior. The concept of vector and operation has been introduced to support this extension.

The vector denominated FC represents all possible Failure Classes in the system model.
Similar to it, all input and output ports, as well as parameters of a given port, can be generalized as respectively IP (Input Port), OP (Output Port) and PM (Parameter).

It is also possible to define a subset of vector element, as for example failure class by explicitly defining the list of elements in brackets (as for example FC :{O,C}-out = Expression, representing only Omission and Commission of the list of failure class). In addition, it is possible to define exception in the vector element implemented by keyword EXCEPT and the list of concerned element in brackets (as for example FC EXCEPT {V}-out = Expression, representing all failure class except Value).

The operator allows to applied specialized relation on vector of inputs and outputs (IP and OP) in the respect of the correct syntax of the propagation expression. The operator SAME allows to define propagation correspondence of inputs to outputs or inputs, as FC-out = SAME (FC)-in (a typical use case of application is a communication bus).

Another operator ANY helps to represent a logical disjunction on input port as FC-out = FC-ANY (IP) (as for example O-out = O-ANY (IP) where all input port omission will be propagated to the output port similar to an OR between all ports). By extension, the logical conjunction of input port is defined with ALL (as for example O-out = O-ALL (IP) for summarizing an AND between all input ports).

A voter operator MAJ for majority exists and is useful for redundant systems based on majority vote. The typical expression is O-out = O-MAJ(IP) assuming that for n inputs at least (n/2)+1 have to be omission to propagate the failure on the output.

In combination to vector and operator to build complex expressions, the concept of instantiation is used for output generalization, like FC-OP meaning list of output failure deviation expression with O-out1, _O-out2, C-out1, C-out2… For input ports, the applied concept is the expansion, as O-ALL (IP) means a list of input ports expended in the same failure expression as O-in1 AND O-in2 AND…An example can be: ANY (FC)-OP = SAME (FC)-ANY (IP) OR InternalFailure1 where for each failure class of the output port the failure propagation will be given from the same failure class of any input port or an internal failure.

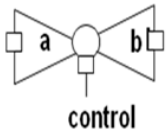
Furthermore one of the most important advantages of the above generalization concept is that it can provide background for object-oriented principles and can be reused in complex system by applying pattern templates and instantiation mechanisms. As for example, one may perform a “generic” component failure behavior defined by a name, and may overload the template by an additional failure expression.

The implementations of all these mechanisms are application dependant and may be transparent to HiP-HOPS XML format. Generic Failure Expression and inheritance mechanism of the failure can created by a front end to capture the failure expression and then be pre-processed to generate existing adequate XML HiP-HOPS formalism. This concept of pre-processing can be applied to any newly defined concept to interface the HiP-HOPS format.

8.1.5 HiP-HOPS Example

The standard use case description of HiP-HOPS is the valve component with “a” as input, “b” as output and valve flows from a to b being controlled by the command “control”. In normal operation, the valve is normally closed and opens only when the computer control signal has a continuously maintained logical value.

See below the description of the malfunction of the valve.



Failure Mode (as Internal failure)	Description (as physical cause)
Blocked	e.g. by debris
Partially Blocked	e.g. by debris
Stuck closed	Mechanically stuck
Stuck open	Mechanically stuck

Table 5 : HiP-HOPS Valve example

The following failure description will then be implementing in the valve component (according to XML formalism not depicted here):

```
Flow Omission : Omission-b = Omission-a OR LowValue-control OR Blocked OR StuckClosed
Flow Commission : Commission-b = Commission-a OR StuckOpen OR HighValue-control
Low Flow : ValueLow-b = ValueLow-a OR PartiallyBlocked
High Flow : ValueHigh-b = ValueHigh-a
Early Flow: Early-b = Early-a OR Early-control
Late Flow: Late-b = Late-a OR Late-control
```

8.1.6 HiP-HOPS and loops management

HiP-HOPS can handle most logical propagation loops in the model by cutting the loop in a deterministic way for loop build with only one entry/exit point.

Example 01 : Let’s imagine three components A, B, C that have basic events or internal failures IFA, IFB and IFC respectively connected to each other in a loop from C to A.

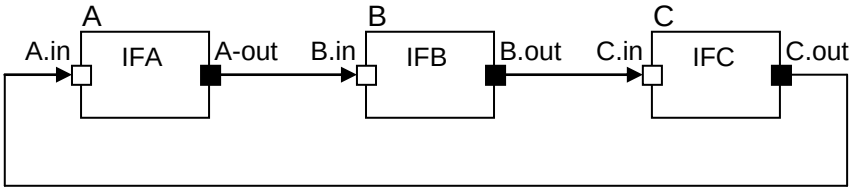


Figure 15: Loop example in HiP-HOPS

The following propagation is built as a logical loop:

```
Omission-A.out = Omission-A.in OR IFA
Omission-B.out = Omission-B.in OR IFB
Omission-C.out = Omission-C.in OR IFC
// link
B.in = A.out
C.in = B.out
A.in = C.out
```

This produces a chain such A causes B to fail, B causes C to fail, and C causes A to fail, whereas a basic failure in any of the component will cause failure of all components.

In practice HiP-HOPS will cut the loop at the point where is starts to repeat. So assuming C is the output where the analysis begins, the loop is cut when we try to go back from C to A. When this cutting happens, HiP-HOPS creates a “circle node” to represent the cut. This has the logical value “always false” (i.e. c contradiction), so any cut set containing it is also always false and can be removed (as if this behavior can be turn off).

So in this case the cut-set might be: IFA OR IFB OR IFC OR CircleTo[C] and the circle node would be removed from normal cut-sets.

Example 02 : Another example is the diagnosis for calculation with respective A, B, and C components, respective basic events or internal failures IFA, IFB and IFC, with A having two inputs as in1 the input of the regulation and in2 the diagnose value controlled by the output of B (as diagnoses component).

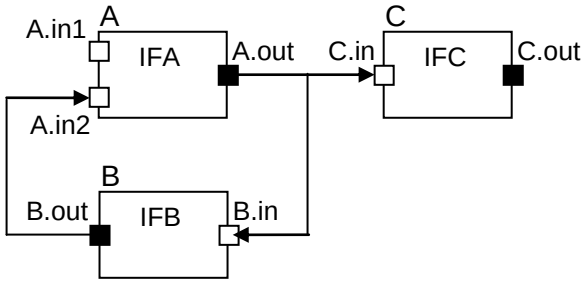


Figure 16: Loop example with diagnosis in HiP-HOPS

Omission-A.out = (Omission-A.in1 AND Omission-A.in2) OR IFA
Omission-B.out = Omission-B.in OR IFB
Omission-C.out = Omission-C.in OR IFC
// link
B.in = A.out
C.in = A.out
A.in2 = B.out
A.in1 as basic event of the system

This construction would cause a loop between A and B and the resulting cut-sets to be: IFB OR IFC OR IFA. As the loop generates a contradiction, the loop trough A.in2 would disappear.

But in case of certain situation called “crazy loops”, mostly when the propagation loop has more than one entry/exit, this behavior becomes invalid, because cutting the loop for one entry affect the results of the loop being entered at a second point. This case is illustrated in example 03.

Example 03: For example let’s imagine a chain having 5 links numbered from 1 to 5.

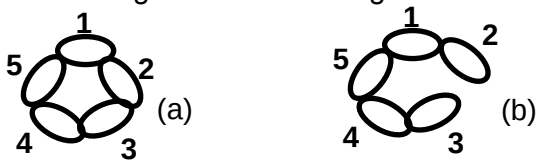


Figure 17: Chain example with 5 links

If you start at any point and move around the chain, you will always count 5 links before reaching your starting point (see Fig.16a).

At one point, if you break the links (see Fig.16b), this affects how many links we can count before you reach the break. So, if we start at 3 and break the chain between 2 and 3, so we still count 5 links as 3, 4, 5, 1, 2 before reaching the break. But if you start at 4, you will get only 4 links as 4, 5, 1, 2. This is now inconsistent because it depends on where you start counting from.

If this chain was propagating through a system and the links are the components or basic events, then we will have the same problem: where we choose to break the loop has an impact on the

apparent causes of the failure, because when we enter in the loop at another location effect is different.

In such scenario, HiP-HOPS is not able to break the loop and will just print out an error message. Since this situation represents a potentially contradictory logic in the model, the modeler has to solve this issue with a deterministic solution.

8.1.7 HiP-HOPS and failure propagation mitigation with safety mechanisms

One of the main goals of the safety analysis is to evaluate the efficiency of the safety mechanism in order to be able to mitigate the effect of a local fault and preventing the propagation of the error. For system application, the effect of the mitigation of a fault is to provide a protection that can be either a default value on the output usually called limp-home value, or an additional output control.

As the concept of failure propagation methods in HiP-HOPS is based on failure classification we may consider defining a dedicated Failure Class to represent the mitigation on a component, by extension proposed as name LimpHome (LH).

So, let's reuse the example 02 from chapter 8.1.6 based on regulation including diagnosis loop. It contains the respective components A for Acquisition, B for Diagnosis and Limp Home and C for Computation. Also the associated basic events as internal faults are IFA, IFB and IFC.

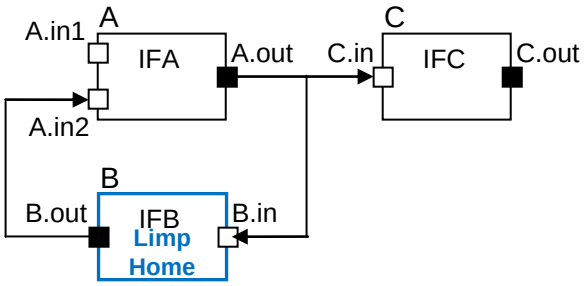


Figure 18: HiP-HOPS example with Limp Home

Compare to the previous definition, a new class of failure LimpHome is introduced and the component description is as follows:

```
Omission-A.out = (Omission-A.in1 AND Omission-A.in2) OR IFA
LimpHome-A.out = LimpHome-A.in1 OR  LimpHome-A.in2
Omission-B.out = IFB
LimpHome-B.out = Omission-B.in
Omission-C.out = Omission-C.in OR IFC
LimpHome-C.out = LimpHome-C.in
// link
B-in = A.out
C-in = A.out
A-in2 = B.out
A-in1 as basic event of the system
```

Compare to the previous loop example, now the component B mitigates the fault on its input, as output of A, meaning that fault on its input is not propagated as an omission but as a limp home indicating that the diagnosis is performed. The omission on diagnosis component is only linked to its internal failure IFB, as Omision-B.in is removed by the mitigation.

Through this basic example, we see that the loop is cut on the failure class Omission and ensure that failure class LimpHome is also not looped.

8.1.8 HiP-HOPS and ISO26262

The following description identifies briefly where the HiP-HOPS analyzer may help to perform safety assessment in respect to the ISO26262 requirements.

The main questions are concerning the level of architecture to which this methods can be applied and also if it can be used to demonstrate the effectiveness of safety mechanisms to eradicate or mitigate failures (systematic or random) toward failure propagation analysis.

The natural matching of HiP-HOPS concept of component, port and line to the architecture description language may help to automate safety analysis at the different levels of architecture and provide results from traditional manual deductive and inductive methods in use today (as respectively FTA and FMEA).

From ISO26262 perspectives, we may expect to perform safety analysis using HiP-HOPS at the following elements:

- On the functional safety concept at the preliminary, functional architecture level.
- On the technical safety concept at the detailed, technical architectural level, considering, HW and SW.
- On probabilistic metrics of hardware design, at least to help their construction.

At low level of architecture like AUTOSAR software and hardware part implementation, it might be very difficult to define such elements with their associated properties and their influence into the overall system. Nevertheless from theoretical point of view it can be possible.

As the objective of this document is to define overall methods, it will help to answer to this question or define relationship between actual or new methods and landscape of associated tools.

8.1.9 EAST-ADL2 experiment with HiP-HOPS, limits and opportunities identified

The ATESST2 project proposed an implementation of HiP-HOPS methods and transformation by mapping the concept of actual EAST-ADLV2 implementation to HiP-HOPS selected concept (see *Figure 19*). Notice that mapping may lightly differ from actual EAST-ADLV2.1 due to meta-model late change.

EAST-ADL	HiPHOPS
ErrorModelType	System
ErrorModelType.errorConnector of type ErrorPropagationLink	System.Lines
ErrorModelType.parts of type ErrorModelPrototype	System.Component
ErrorModelPrototype.type.errorPort of type ErrorPort	System.Component.Ports
ErrorModelPrototype	System.Component.Implementation
ErrorModelPrototype.type.errorBehaviorDescription.internalErrorEvent of type ErrorEvent	System.Component.Implementation.FData. basicEvent
ErrorModelPrototype.type.genericDescription of type String	System.Component.Implementation.FData. outputDeviation
ErrorModelPrototype.type of type ErrorModelType	System.Component.Implementation.System **(recursion)**

Figure 19: ATESST2 HiP-HOPS versus EAST-ADLV2 mapping [11]

This mapping is based on the Error Model defined in EAST-ADLV2, which is separated from the architectural design. This concept gives flexibility for safety assessment but induces more work

during analysis pre-processing as all necessary failure elements from *Figure 19* have to be mapped or related to the architectural elements during model construction. As no 1:1 mapping concept is guaranteed, automation may be limited or complex to be defined.

The multiple perspective capability of EAST-ADL has not been fully exploited in this project and it could be reconsidered in future as it might help to compose different components of the system. As safety mechanisms and coverage mechanisms are often mixed between hardware and software components, the setup of these features shall be carefully designed to allow this close relationship and failure propagation between hardware and software.

The separation of failure class and output propagation with separate flows for the HiP-HOPS analyzer allows precise analysis but requires lot of binary equations to be captured. Thanks to the proposal of the General Failure Expression, Template and Generalization, and pre-processing, we may define failure semantics independent of the final HiP-HOPS implementation. It would allow us to define adequate failure semantics according to the phase of the analysis and to the level of details we want to achieve.

8.1.10 Conclusions on HiP-HOPS

First of all, preliminary safety analysis using mapping of failure class concept from HiP-HOPS to architecture model has been validated in ATESST2 based on prototype and UML domain model definition.

From this initial methodology, several improvements easy to reach have been identified such as:

- Generation of failure class from an above failure language syntax and the possible generalization/specialization of failure class concept,
- Consideration of mitigation with a new failure class,
- Separation of analysis software and hardware safety concept and then merge for an overall technical safety concept analysis based on plain feature of HiP-HOPS concept as perspective and exported propagation for hardware allocation (the architecture elements are present in the SAFE meta-model).

HiP-HOPS derived methods based on Failure Class allows the analysis of semi-formal architectural elements and fault models, from failure propagation and possible mitigation from safety mechanism.

The analysis can be automated for a generation approach, where granularity of analysis for debug has to be specified in the tool interface specification.

Final results are complete FMEA and FTA, allowing local view on component or system parts.

8.2 AltaRica

8.2.1 AltaRica Historical background

The AltaRica project started in 1997 at the Laboratoire Bordelais de Recherche en Informatique (LaBRI, FRANCE). It involved, since the very beginning, a strong partnership between academic laboratories and industries (among which Total and Dassault Aviation played a central role). The primary objective of the project was to give a formal basis to a reliability workbench and to study how reliability engineering and formal methods (model-checking) can be cross-fertilized. Quickly, it became clear that such a formal basis can be obtained only through a dedicated language. The first version of the AltaRica language was designed by the LaBRI team during years 1998-2000 and G. Point's PhD Thesis [13]. This first version was strongly inspired by works done at the LaBRI on model-checking on one hand (with notably the model checker MEC [14]) and constraint logic programming on the other hand.

In the early 2000, Dassault Aviation decided to create its own reliability workbench based on AltaRica (Cécilia OCAS). Severe restrictions were imposed on the language in order to make the compilation into fault trees tractable. With the same objective, ARBoost Technologies (now Dassault System), designed a simplified version of AltaRica called Safety Designer [19]. The idea was mainly to substitute constraint processing by flow propagation, hence transforming AltaRica into a Data-Flow language (and achieving substantial complexity saving). Only minor modifications have been done since then to the language, mainly through normalization of the clause "extern".

8.2.2 AltaRica basic description

The AltaRica Extended language targets model-based safety analysis. This assertion has a few implications:

- AltaRica models are a vision of the real world systems that are oriented towards the tractability of safety analysis.
- AltaRica Extended language allows the composition of hierarchical models.
- AltaRica Extended language is oriented towards the definition of state machines in which transitions are guarded by data flows and events. The events can be both stochastic and deterministic. Stochastic events are the natural means to express random faults while deterministic events are the natural means to express systematic faults.
- AltaRica Extended language, in order to allow the analysis of the consequences of a fault, allows the definition of both the functional behavior and the dysfunctional behavior. The functional behavior is only defined in such a way that it defines the propagation of cascading failures from a failed component to components that are not necessarily crippled by their own faults.

With this last restriction in mind, AltaRica Extended language only defines the functional and dysfunctional behavior of the system. It does not provide the tools that are required to simulate the system, nor to compute the cut-sets or sequences leading to a feared condition or a set of them. The main tools used for that are:

Fault tree compilers; when fault tree compilation is possible, it is the most efficient way to obtain qualitative results (the cut-sets) and quantitative results (the probabilities of reaching a feared condition, the importance factors...). However, this technique is intrinsically limited to problems that match the tree structure. Dynamic systems, in which the order of fault occurrence matters, and looped systems (a tree is by definition an acyclic graph) are out of scope of traditional fault tree analysis.

Sequence generators; sequence generators generate all the possible combinations/permutations of N faults, where N is an integer that is traditionally called “the order” of the sequence. In the automotive industry, the fact that many practitioners only use FMEA demonstrates that N is

generally at most 1 or 2, but rarely more. In aerospace industry on the other hand, as the concept of “safe state” for a plane in flight condition is less applicable, computations are often performed up to the order 4 or 5. For a system where 1 000 events are possible, this leads to millions of billions of simulations. As the order of sequences increases, the performance of these algorithms tends to be paramount. Sequence generators provide qualitative results (the sequence sets); these are used in quantitative analysis by fault tree tools, although this last step can be discussed.

MonteCarlo simulators; MonteCarlo simulators generate a number of paths of evolution for the system in order to obtain average values for some parameters, typically, the probability to reach a feared condition. MonteCarlo simulators are avoided whenever possible because they provide the worst performance.

Due to the combinatorial nature of the problems that exist in the field of functional safety, the performance of the tools is essential in their evaluation.

8.2.3 AltaRica basic elements

The following chapter will give an overview of the basic elements managed by AltaRica.

Node:
The base block in AltaRica is a node. A node is a generic object to describe a behavior, which:
- *Has an internal state,*
- *Reacts on events,*
- *Receives and/or sends data by flows (input and output) which enable to communicate with other components.*

A node may have several sub-nodes which are instances of a node.

In tools, top-level nodes are sometimes referred as “systems”, intermediate nodes are also referred as “equipments” and leaf nodes are referred as “components”.

Each node may:
- *have several input flows and several output flows,*
- *have one or more state variables,*
- *undergo one or more events,*
- *and also have one or more assertions, which are equations that define how inputs are transformed into outputs given the value of the state variable.*

Input *Flows and Output Flows:*
Interface of a Node is defined by Input and Output Flows. These flows are typed. There are mainly three basic types: Boolean, integer and float. Complex types can be built from these 3 elementary types.

Link:
Links can be created between two flow ports, to represent the fact that one end will emit a flow into the other end.

State variable:
A State variable is a variable identifying a component internal state, e.g. a variable with the following values “open/blocked”. State variables have an initial value.

Event:
Event can depend on time or not:

- Timed events: take a non null time.
  Stochastic events with Probability distributions with parameters (exponential, Weibull ...).
  Dirac events.

- Instantaneous events: take no time and may have a priority.
  Immediate events.
  Conditional events.

If an event is declared, a model must contain at least one transition labeled with this event.

Transition:
A transition is composed of a guard that expresses the conditions that allows the transition to be passed if the event is triggered, and a series of affectations of state variables that define the outcome of the transition.

Assertions:
Assertions allow giving a value on output flow variables and may depend on state variables and input flow variables.

Extern clause:
The role of the extern clause is:

- to give some interpretation to the model, e.g. priorities
- to transitions, probability distributions to events,
- to give tools a specific information,
- to provide some mechanism to extend the language.

8.2.4 AltaRica Failure Description and propagation

In AltaRica, the failure description is double.

In one hand, the failure is declared explicitly as an “event”. On the other hand, the state changes induced by the events are declared in transitions.

A transition represents a modification of internal state of a component, depending on the current states value, the value of input flow variables, and occurrence of an event:

Condition |- event -> event -> aff1, …, affn ;

With: **condition** being a Boolean expression depending on the input flow variables and the state(s) of the component,

event being a simple identifier declared in the event tab of the component,

affi are affectations of state variables depending on their current value and the input flow values.

The following table shows example of transitions:

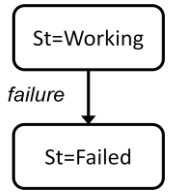
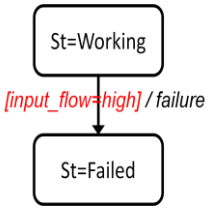
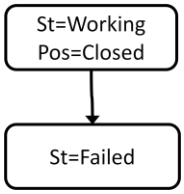
State diagram	AltaRica code
	trans St=Working |- failure -> St := Failed; **Condition on one state variable**
	trans St=Working and input_flow=high |- failure -> St := Failed; **Condition on one state variable and input flow variable**
	trans St=Working and Pos=Closed |- failure -> St := Failed; **Condition on 2 state variables**

Table 6 : Type of analysis methods required or recommended by ISO26262

In AltaRica, propagation of failure is done using assertions.

Assertions are Boolean expressions used to describe invariants on variables. All configurations of a node must satisfy specified assertions. These invariants can be used to describe relations between flow variables as a transfer-function but also they model relationship between states of the nodes and its flows.

3 different forms are possible for assertions:

- Simple affectation: An output flow variable is valuated according an input flow variable.
- If condition then conclusion1.
- If condition then conclusion1 else conclusion2.

with a condition being a Boolean expression depending on input flow variables and component state variables and a conclusion being new values of output flow.

When there is a succession of instructions if-then-else, it can be replaced by (it is equivalent to) a case expression as shown in the following example.

The measure of a sensor (output) depends on internal state of the component

```
assert
        (if sensor_state = nominal then sensor_measure = nominal) ;
        (if sensor_state = degraded then sensor_measure = erroneous) ;
        (if sensor_state = failed then sensor_measure = absent) ;
```

is equivalent to the following statement:

```
assert
      sensor_measure = (case {sensor_state = nominal    : nominal,
      sensor_state = degraded : erroneous,
                        else absent})
```

Figure 20: Example of equivalence between if-then-else expressions and case expression

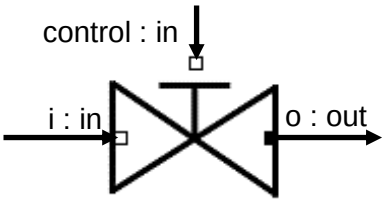
8.2.5 AltaRica Example

The same valve example than used in chapter 8.1.5 with HiP-HOPS will be investigated with AltaRica to highlight some differences. Just to remind the internal failure modes of the valve are:

Failure Mode (as Internal failure)	Description (as physical cause)
Blocked	e.g. by debris
Partially Blocked	e.g. by debris
Stuck closed	Mechanically stuck
Stuck open	Mechanically stuck

Table 7 : Example of Valve Internal failure modes

The corresponding code in AltaRica is the following:




```
node SAFE_WT331Valve
flow
  i : SAFE_MyFlow : in ;
  o : SAFE_MyFlow : out ;
  command : SAFE_MyCommand : in ;
state
  State : {Nominal,StuckOpen,StuckClose,StuckPartiallyOpen};
event
  PartiallyBlocked; StuckOpened; StuckClosed;
init
  State := Nominal;
trans
  State = Nominal |- StuckOpened -> State := StuckOpen;
  State = Nominal |- StuckClosed -> State := StuckClose;
  State = Nominal |- PartiallyBlocked -> State := StuckPartiallyOpen;
assert
  if (State = StuckClose or command = LowValue-control) then o = Omission */ No flow */
  else if (State = StuckOpen or command = HighValue-control) then o = Commission */Unexpected Flow */
  else if (State = StuckPartiallyOpen) then o = ValueLow */Less flow than expected */
  else if (command = EarlyCommand) then o = EarlyFlow */ Flow get out too early */
  else if (command = LateCommand) then o = LateFlow */ Flow get out too late */
  else o = i;
 edon
```

Figure 21: AltaRica Code Example for our Valve

The AltaRica node representing the Valve has two input flows and one output flow defined in the “flow” section.

In the “state” section, 4 states for the valve are defined: Nominal, StuckOpen (meaning always open), StuckClose (meaning always closed) and StuckPartiallyOpen. The initial state of the valve is Nominal (defined in “init” section).

In the “event” section, 3 events corresponding to the internal failure modes (see *Table 7*) of the valve are defined. In this example, one remark is that the internal failure modes Blocked (e.g. by debris) and StuckClosed have the same effect and therefore only one event StuckClosed was considered.

In the “trans” section, a transition from normal state to a failed state is defined: as an example the valve can undergo a “StuckOpened” event, in which case its state becomes “StuckOpen”.

The “assert” section also defines how this failure to operate affects the outflow: the outflow is no longer controlled and lead to “commission” (unexpected flow) if the valve is in state “StuckOpen” or the command has failed (“HighValue-control”).

Moreover in the “assert” section, the functional behavior is also defined: if the state of the valve is nominal and the valve is under control, then the outflow reflects the inflow.

Note: In the AltaRica assertion representing failure propagation description input failures are never considered in comparison with HiP-HOPS. If we want to represent an input failure, we will have to model a new node upstream whose output is linked to the input flow relevant of downstream node (i: see *Figure 21*). In this new upstream node, if output flow might fail in some conditions, it will automatically be propagated into the output of the downstream node. It can be simply explained by the fact that when we are in Nominal state for the valve, in the “assert” section, we have defined that o=i meaning that if everything is OK, the output will simply propagate the input. Therefore if the valve is working well and received a flow that is incorrect, this incorrect input flow will be propagated into the output unless we have a safety mechanism implemented that can detect the failure and stop its propagation. The final behavior is the same but the failure propagation description in HiP-HOPS would need redundant information as output failure is described once in upstream node and a second time as input failure in the downstream node.

8.2.6 AltaRica and Loop management

In the design of complex systems, loops are often introduced to take some feedback into account. For example, a diagnostic may monitor the output of a function and force its transition to a safe state if invalid outputs are detected.

In AltaRica Extended language, the management of loops has long been a problem for various reasons. The first one is that loops make the most effective algorithms for safety analyses – fault trees – at least much harder to use. The impact on Boolean formulae is for example explained in [15]. A second reason is that the semantic of execution of AltaRica must be defined precisely. These difficulties are illustrated in [16].

Two main solutions are used to handle loops.

The first one is to create a fictive “instantaneous” transition, which can affect a state, and consequently take benefit of an initial value for a state. Let us remind that flow variables are not initialized in AltaRica Extended language. This approach is explained in [17] . This workaround is a pain for the end user.

The second solution is to handle the loop as it is. This requires that for each loop in the system, one initial value is provided. A fixed point algorithm is then used to stabilize the loop, with a predefined maximum number of iterations that must detect the potential divergence of the loop. The algorithm has converged for one loop when, starting with the initial condition at the first step or the last stable value during next steps, at the end of an iteration of the loop, the value of the initialized flow remains unchanged.

For a loop management algorithm the following requirements shall be satisfied:

- The loop management algorithm shall be able to handle loops of any complexity.
- The loop management algorithm shall provide stable results, whatever the names of the involved components or the order in which initial values are defined.
- The loop management algorithm shall detect divergence. It shall do it rapidly if achievable, which is often the result of a compromise between memory and CPU consumption.
- The loop management algorithm shall not base its convergence criteria on arbitrary data provided by the end user.

It shall be clear that transient states are not taken into account in the criteria for the feared conditions, as AltaRica Extended language does not handle temporal aspects.

8.2.7 AltaRica and failure propagation mitigation with safety mechanism

In ISO26262, it is required to demonstrate the efficiency of safety mechanisms. As a consequence, their identification could be necessary. This can be easily achieved by the use of extern clauses in the smallest enclosing node. Another way to deal with this requirement is to analyze the cut-sets, which should display an order greater than 1 if the mechanism successfully protects a safety goal.

Safety mechanisms can be modeled with AltaRica Extended Language. It is even one of the goals of the language to support these mechanisms, whatever their complexity may be. In the aerospace industry, some systems contain safety mechanisms that are designed to withstand more than 4 failures at least.

However, safety mechanisms are not identified as such in AltaRica Extended Language. They are nodes, and are not distinguished from the functions they are supposed to protect.

For the sake of illustration, let us consider a sensor which faults are covered by a safety mechansimt, as shown in the next diagram:

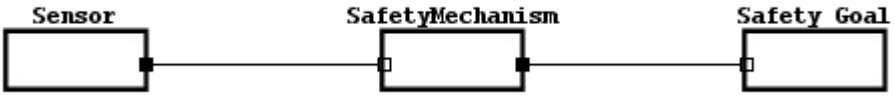


Figure 22: Example of safety mechanism modeled in Safety Designer

The output of the sensor is checked by a safety mechanism module. If this output is wrong, it is detected by the safety mechanism with a certain diagnostic coverage (here 75%).

A wrong sensor output value only violates the safety goal when:

- it is undetected by the safety mechanism (residual fault),
- or it is not detected by the safety mechanism while it was supposed to be because the safety mechanism failed prior the possible detection of the wrong sensor value occurs (latent fault).

In the above example the safety mechanism has different states defined:

- Two for its detection status {Detecting, NotDetecting}.
- Two for its Heath status {Normal, Failed}

And the transition from one state to another one is depending of events that are defined as follows:

- NotDetectingDiagnostic of which probability of failure is 0.1 and represent the non detected portion of a fault by the safety mechanism (meaning a diagnostic coverage of 90%),
- SM_Random_HW_Fault of which probability of failure follows an exponential law with constant failure rate of 3 FIT.

The corresponding code in AltaRica is the following:

```
node ISO26262_SimpleSafetyMechanism_SafetyMechanism
flow
  icone : [1, 2] : local;
  i : bool : in ;
  o : bool : out ;
state
  HealthStatus : {Normal,Failed};
  DetectionCoverage : {Detecting,NotDetecting};
event
  NotDetectingDiagnostic; SM_Random_HW_Fault;
init
  HealthStatus := Normal;
  DetectionCoverage := Detecting;
trans
  DetectionCoverage = Detecting |- NotDetectingDiagnostic -> DetectionCoverage := NotDetecting;
  HealthStatus = Normal |- SM_Random_HW_Fault -> HealthStatus := Failed;
assert
  if (i = true)
  then o = true & icone =1
  else
    if (DetectionCoverage = NotDetecting)
    then o = false & icone = 2
    else if (HealthStatus = Failed and DetectionCoverage = Detecting)
        then o = false & icone = 2
        else o = true & icone =1;
extern
  law <event NotDetectingDiagnostic> = constant(0.10);
  law <event SM_Random_HW_Fault> = exponential(0.000000030);
edon
```

Figure 23: AltaRica Code Example for a safety mechanism

The following Fault Tree generated by Safety Designer is the following:

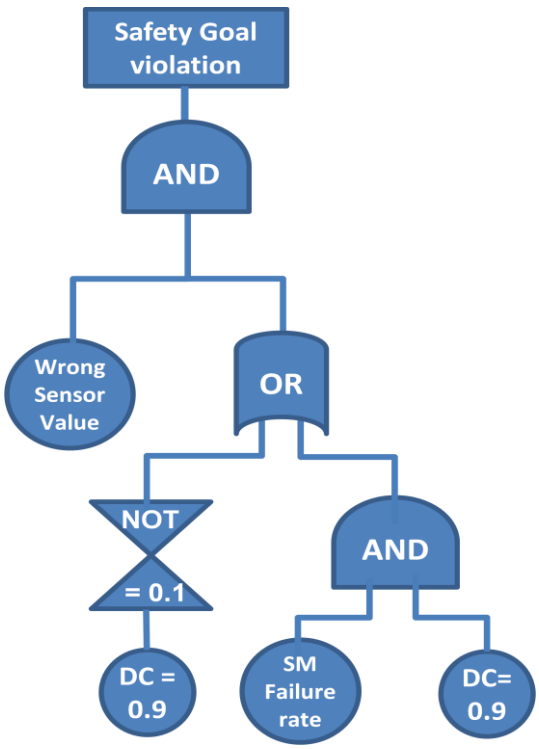


Figure 24: FTA generation by Safety Designer [19] from AltaRIca complete model

8.2.8 AltaRica and ISO26262

The following description identifies briefly where the AltaRica extended language may help to perform safety assessment in respect to ISO26262 requirements.

The natural scope of AltaRica Extended language is to design and validate:

- the functional safety concept at the system architecture level,
- the technical safety concept mixing HW and SW elements.

AltaRica supports FMEA as an inductive method. It also supports the deductive method that is fault tree analysis when the modeling structure of the problem allows it.

AltaRica Extended language can permit to extend its capabilities by adding information in extern clause. Then the tools that are supporting AltaRica Extended language can use these additional information and could provide additional capabilities such as calculation of architectural metrics for a given safety goal.

At low level of architecture like AUTOSAR software and hardware part implementation, it might be very difficult to define such elements with their associated properties and their influence into the overall system. Nevertheless from theoretical point of view it can be possible but would lead to huge model that would need tool modification for solving and analyzing results.

8.2.9 AltaRica concepts versus EAST-ADLV2.1

The EAST-ADLV2.1 concepts of interest are presented in chapter 9.1.

A mapping between the ErrorModel structure and AltaRica Extended language is proposed in the following table:

EAST-ADLV2.1 Concept	AltaRica concept	Comment
ErrorModelType	Node	
ErrorModelPrototype	Sub	The name of the sub (instance) is the target's shortName.
FaultInPort	Flow direction in	Type must be a valid AltaRica identifier (e.g. Boolean)
FailureOutPort	Flow direction out	Type must be a valid AltaRica identifier (e.g. Boolean)
InternalFaultPrototype	Event	In order to keep the semantic of a internal fault, an extern clause must be used in AltaRica.
ProcessFaultPrototype	Event	In order to keep the semantic of a process fault, an extern clause must be used in AltaRica.
FaultFailurePropagationLink	Assert	At node level, assert define the links between sub nodes.

Table 8 : Mapping of AltaRica versus EAST-ADLV2.1 ErrorModel

The failureLogic attribute of an instance of ErrorBehavior may contains AltaRica code if type is ErrorBehaviorKind : ALTARICA. In this case, the AltaRica code shall only contain assertions.

A FaultFailure aggregated by a Dependability is a feared condition in AltaRica. It can be modeled as an extern(al) clause in AltaRica.

As there is no notion of state in EAST-ADL error model, feared condition expressed on state value must be turned into a FaultFailure for an artificial FaultFailurePort.

From concept analysis, it seems that all the concepts from HiP-HOPS are covered by the concepts from AltaRica. It is illustrated through an example in Annex A chapter 17 in which a mapping between AltaRica and HiP-HOPS concepts is proposed. Therefore the translation of HiP-HOPS into AltaRica should be possible.

8.2.10 AltaRica limits

The validation of the safety models developed in AltaRica is not trivial. Results can be obtained from Altarica models, but do these models correspond to the physical phenomena?

The synchronization between AltaRica models and functional architecture or hardware and software architecture is complex especially when there are loops and safety mechanisms modeled with AND structure.

AltaRica cannot handle the dynamics of physical phenomena.

Extern clauses can extend AltaRica, but the semantic of these extern clauses is not standardized by the language itself.

8.2.11 Conclusions on AltaRica

AltaRica Extended language is being used since 2000 in several tools from the market [19][20] to assess complex models in different fields like aeronautics, railways, nuclear and military fields where safety issues are very critical. Therefore its efficiency is recognized.

AltaRica Extended language support debug and simulation and it is clearly a big advantage to validate our functional and technical safety concepts.

A remaining doubt is the difficulty for system/safety engineer to model the dysfunctional behavior using AltaRica Extended languages. Even if tools like SafetyDesigner provide help to generate the AltaRica syntax, the assertions, describing the failure propagation, inside a node are not trivial and might require specific skills.

8.3 Orientation taken by WT3.3.1 in SAFE

8.3.1 Pros and cons analysis of HiP-HOPS and AltaRica languages

In order to help choosing the best orientation for WT3.3.1, a pros and cons analysis was performed based on the different articles read in the literature and also on the experience of some partners with these languages. See Table hereafter:

	HiP-HOPS	AltaRica
Applicability (based on preliminary user tests ; to be verified during use case	Physical architecture validation and possible low level solution	From functional safety concept to technical safety concept.
Pros	- Simple to define as concept is basic (easy to map from an intermediate language as logical equation; near FTA approach). - Allows generation of both FMEA and FTA view. - Use for large scale analysis and synthesis is fast (as no simulation). - Would allow splitting between hardware and software analysis. - Adequate for validation of safety concept.	- Captures architecture blocks. - Supports simulation and debug, which provides an intuitive approach of failure propagation. - Allow generation of both FMEA and FTA. - Validate test scenario. - Used and recognized in other fields: aeronautic, military, railway, nuclear… high maturity. - Adequate for exploration of safety concept. - Export of FTA possible in Open-PSA format that can be imported by other tools. - Library approach.
Cons	- System debug not allowed by simulation, could be complex as no concrete view of the architecture. - No interchange format standardized: neither import nor export (e.g. FTA). - No direct link between component and system element (library concept is linked to tool generation). - Used only recently in few tools from the market and therefore low maturity. - Real-time constraints are hard to model (only sequence is possible).	- The language is rarely mastered by system/safety engineers. - Model validation is difficult. - The synchronization between AltaRica models and functional /physical architecture is complex (loop, safety mechanism modeling...). - Real-time constraints are hard to model, if possible.

Table 9 : Pros and Cons table for HiP-HOPS and AltaRIca

8.3.2 Language choice in WT3.3.1

Even with the pros and cons analysis, the choice of one unique language is not easy and will also depend on the level of granularity that users want to address. Moreover a language like AltaRica is really powerful but also complex to implement for safety engineer and case by case all its capabilities are not fully needed.

Therefore it was decided in WT3.3.1 to define a simplified SAFE language that could be compatible with HiP-HOPS and AltaRica having in mind the generation of FMEA/FTA safety analyzes.

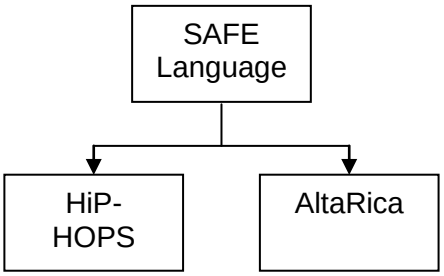


Figure 25: SAFE language proposal

The goal of WT3.3.1 is really not to reinvent a complete language. As HiP-HOPS language expression seems to be less complex for partners than AltaRica, maybe because it is built like local FTAs, it was decided to have something closed to HiP-HOPS in a first step.

Of course it should be possible to transform models of the simplified language towards Altarica for the purpose of a safety analysis and therefore Dassault System partner has provide us some requirements for the simplified language that should ensure that the translation is possible.

8.3.3 General requirements for a simplified SAFE language

Hereafter are the requirements for a simplified language to be transformed in AltaRica language.

Stochastic events shall be connected to their probabilistic distributions

- Faults need to be connected to their probabilistic distributions
- Maintenance events must also be into account to be able to compute availability

It shall be possible to define mutually exclusive failure modes (stochastic events) for a component

- If a resistor has a short circuit, it cannot be simultaneously open

Loops shall be supported

- Monitoring feedback are common practice
- The semantic of these loops shall be explicit and unambiguous
- It shall be possible to simulate the system and the occurrence of faults

Simulation shall be supported

- Simulation provides a better understanding for the designer

Ordering and Timing of safety-relevant events shall be supported

- The timing of events can be important, e.g. to realize the fulfillment of FTTIs

Causality of safety-relevant events shall be supported

- Expression of dependent failures Expression of events that cause other events to occur directly

Expression of safety mechanisms shall be possible in a standardized way

- Expression of dependent failures Safety mechanisms are an essential part of the safety concept and should be directly supported by the language

8.3.4 Hypothesis taken in WT3.3.1

Based on the general requirements from chapter 8.3.3, some hypotheses for WT3.3.1 were considered:

No maintenance considered: *In other fields like Aeronautics, Railway, Military, Nuclear…periodic maintenance is mandatory but not in automotive. If a latent fault is critical, we will implement a safety mechanism that will inform the driver using different warning degrees depending on the criticality of the possible outcome. Moreover this time to discover the latent failure will be taken into account when computing PMHF.*

Constant FIT rate for HW random faults: *Even if AltaRica offers the capability to use different kind of distribution laws with stochastic events, we will consider only constant FIT rate coming from WT3.2.2.*

8.3.5 Refined requirements for a simplified SAFE language

Additionally, some refined requirements were added to precise the content of the simplified SAFE language:

<table>
<tr><td colspan="2">SL_REQ01 : The SAFE language shall support the logical AND operator

SL_REQ02 : The SAFE language shall support the logical OR operator

SL_REQ03 : The SAFE language shall support the logical NOT operator

SL_REQ04 : The SAFE language shall support local symbol or variable

SL_REQ05 : The SAFE language shall be typed for Boolean expression

SL_REQ06 : The SAFE language shall only allow stratified negation
(failure itself shall not be used in its negated form)
e.g failure1 = fault2 or fault3 and not(failure1) expression is forbidden</td></tr>
<tr><td>Covers :</td><td>WT331_REQ_1: The SAFE Meta-model shall provide a fault modeling language to specify fault information and on which element the fault is attached as well as information about fault propagation.</td></tr>
</table>

9 Performing Fault/failure and error propagation based on EAST-ADL V2.1

Within this chapter the current status of the architecture description language EAST-ADL with regard to the fault error failure modeling is described. Furthermore, proposals for an extension of the EAST-ADL concepts are described which could lead to an enhancement of the possibility to perform the fault and propagation analysis.

9.1 Current state of EAST-ADL V2.1 concerning fault/failure and error propagation

EAST-ADL is an architecture description language that has been developed in various European projects in which both, automotive vendors and users are coupled together. The objective is thereby to define an architecture description language tailored to the needs of the automotive industry [18]. The current version published on the website of EAST-ADL (www.east-adl.info) is EAST-ADLV2.1.

EAST-ADL introduces different levels of abstraction, namely:

- Vehicle level (Feature content),
- Analysis level (Abstract functional architecture),
- Design level (Functional architecture, HW architecture, platform abstraction),
- Implementation level (AUTOSAR Software architecture), and
- Operational level (Embedded system in produced vehicle, not in model).

Besides the different abstraction levels, EAST-ADL includes several package extensions of which the dependability package (see *Figure 26*) is of special interest for WT3.3.1, and especially the ErrorModel sub-package (see *Figure 27*).

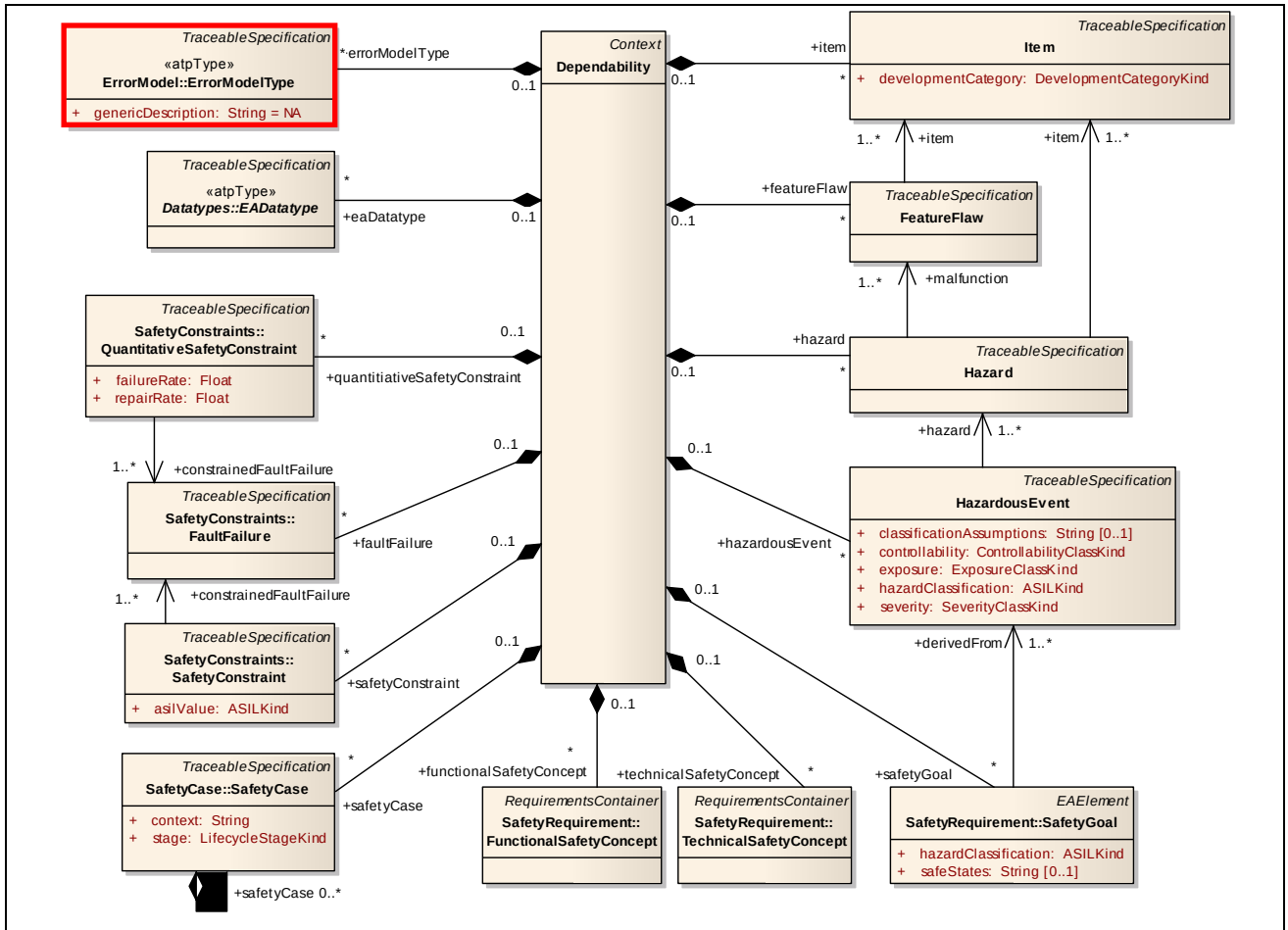
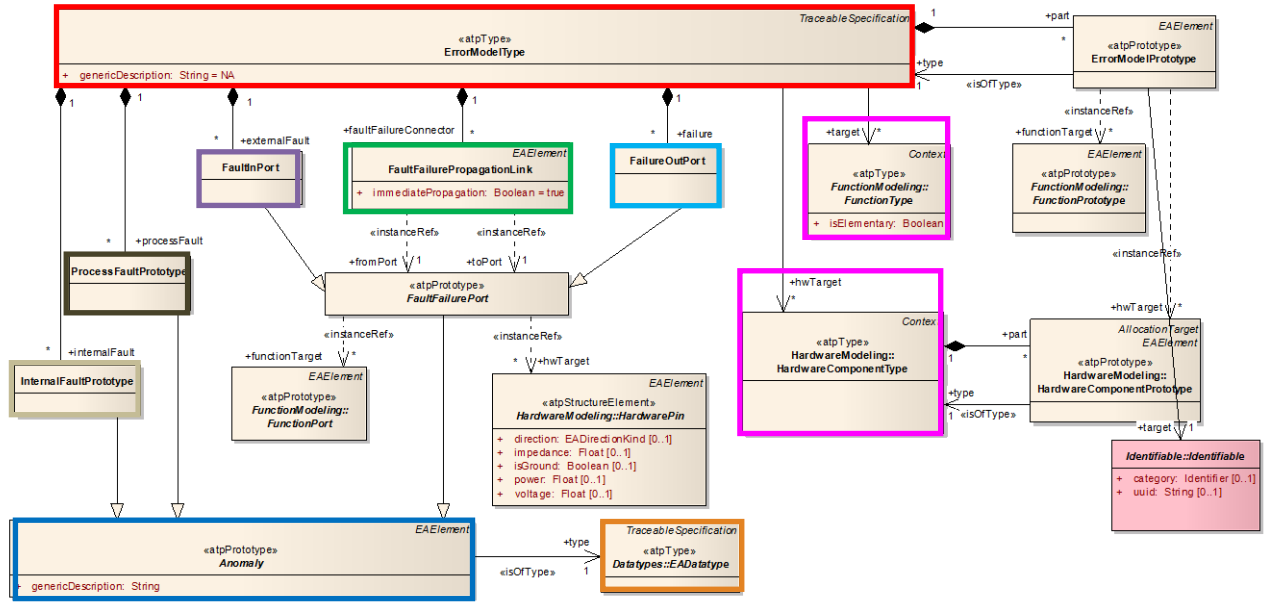


Figure 26: EAST-ADL V2.1 Dependability Package with ErrorModelType class highlighted

The EAST-ADL sub-package for error modeling (see *Figure 27*) provides support for safety engineering by representing possible, incorrect behaviors of a system in its operation (e.g. component errors and their propagations).

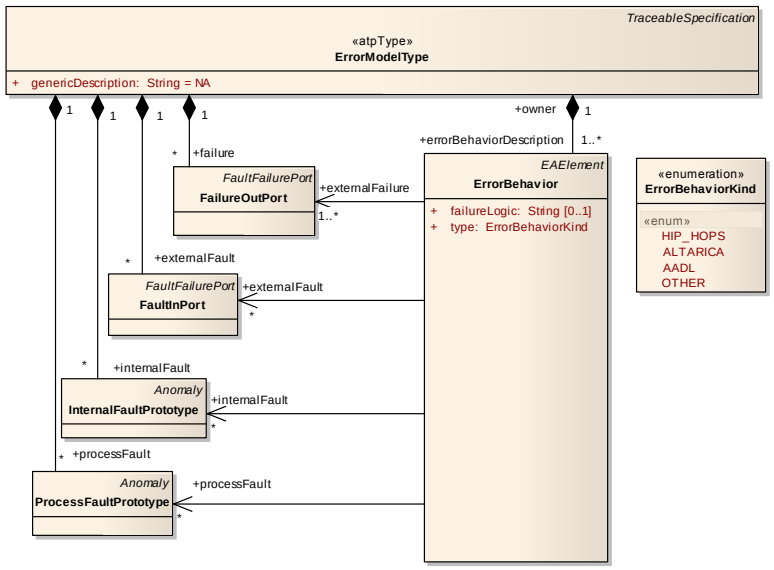
Abnormal behaviors of architectural elements as well as their instantiations in a particular product context can be represented. This forms a basis for safety analysis through external techniques and tools. Through the integration with other language constructs, definitions of error behaviors and hazards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



- **ErrorModelType** (□) specifies possible behaviors of a **target** (□) architectural entity as FunctionType or HardwareComponentType that are of concern when analyzing system anomalies and errors.
- **FaultInPort** (□) represents a propagation point for faults that propagate into the containing ErrorModelType.
- **FailureOutPort** (□) represents a propagation point for failures that propagate out from an ErrorModelType.
- **ProcessFaultPrototype** (□) is a systematic fault that represents the anomalies that the target architectural entities can have due to design or implementation flaws (e.g., incorrect requirements, buffer size configuration, scheduling, etc.).
- **InternalFaultPrototype** (□) represents the particular internal conditions of a target architectural entity that are of particular concern for its fault/failure definition.
- **FaultFailurePropagationLink** (□) connects multiple ErrorModelTypes together via their ports.
- **Anomaly** (□) represents a Fault that may occur internally in an ErrorModel or be propagated to it, or a failure that is propagated out of an ErrorModel. The anomaly may represent different faults or failures depending on the range of its EADatatype (□). Typically the EADatatype is an enumeration. For example, a failure out port may carry a set of failure modes: {Omission, Commission, Value…).

Figure 27: EAST-ADLV2.1 ErrorModelType Content

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



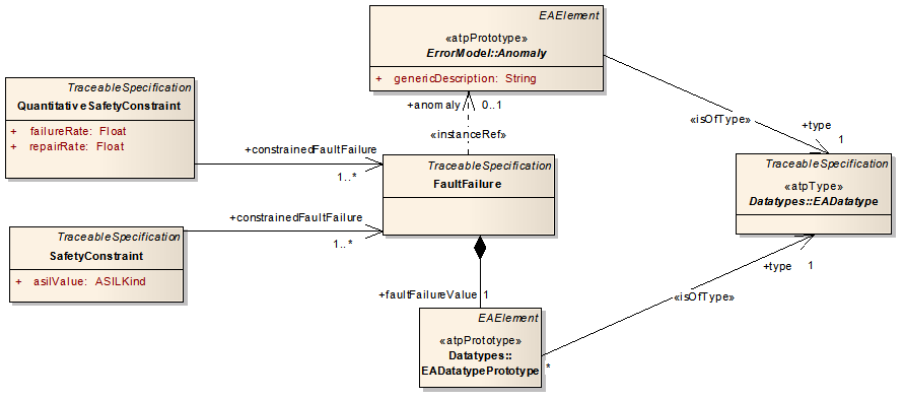
ErrorBehavior defines the error propagation logic of its containing ErrorModelType.

failureLogic attribute: specification of error behavior based on an external formalism or the path to the file containing the external specification.

type:ErrorBehaviorKind attribute : type of formalism, based on enumeration ErrorBehaviorKind, applied for the error behavior description.

Figure 28: EAST-ADLV2.1 ErrorBehavior Content

The SafetyConstraints sub-package is also of special interest for error modeling. It basically contains constructs for defining safety constraints that apply to FaultFailure which itself refer to Anomaly.



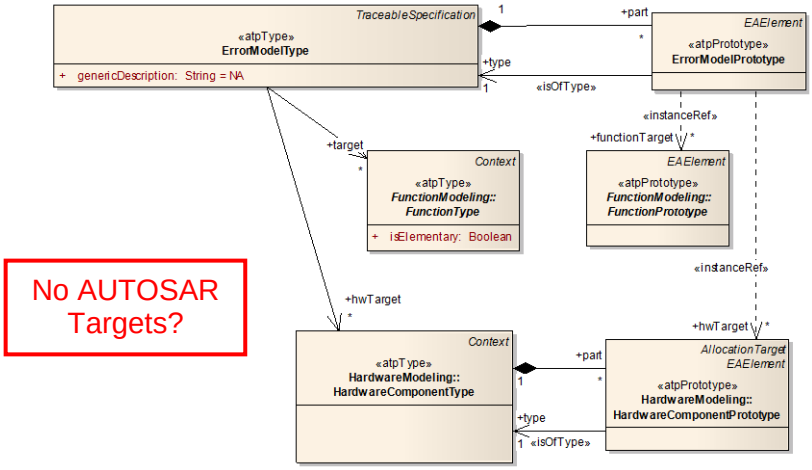
- **FaultFailure** decides the actual value of an anomaly given as a fault in port, failure out port, or internal fault, e.g. {Omission}. It is FaultFailure, instead of Anomaly, to which a safety constraint is assigned. A FaultFailure is defined as a certain value, faultFailureValue, occurring at the referenced Anomaly.
- **SafetyConstraint** represents the qualitative integrity constraints on a fault or failure. Thus, the system has the same or better performance with respect to the constrained fault or failure, and depending on the role this is either a requirement or a property.
- **QuantitativeSafetyConstraint** represents the quantitative integrity constraints on a fault or failure. Thus, the system has the same or better performance with respect to the constrained fault or failure, and depending on the role this is either a requirement or a property. A QuantitativeSafetyConstraint provides information about the probabilistic estimates of target faults/failures, further specified by the failureRate and repairRate attribute.

Figure 29: EAST-ADLV2.1 FaultFailure Content

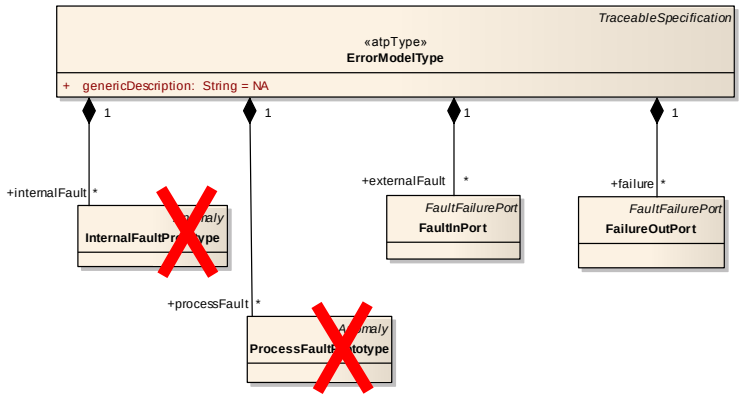
9.2 Analysis of Gap between EAST-ADLV2.1 ErrorModel and our needs

Hereafter are highlighted the gaps between the ErrorModel from EAST-ADLV2.1 and our needs:

- Not possible to address AUTOSAR targets (data element instances, component types, component instances).

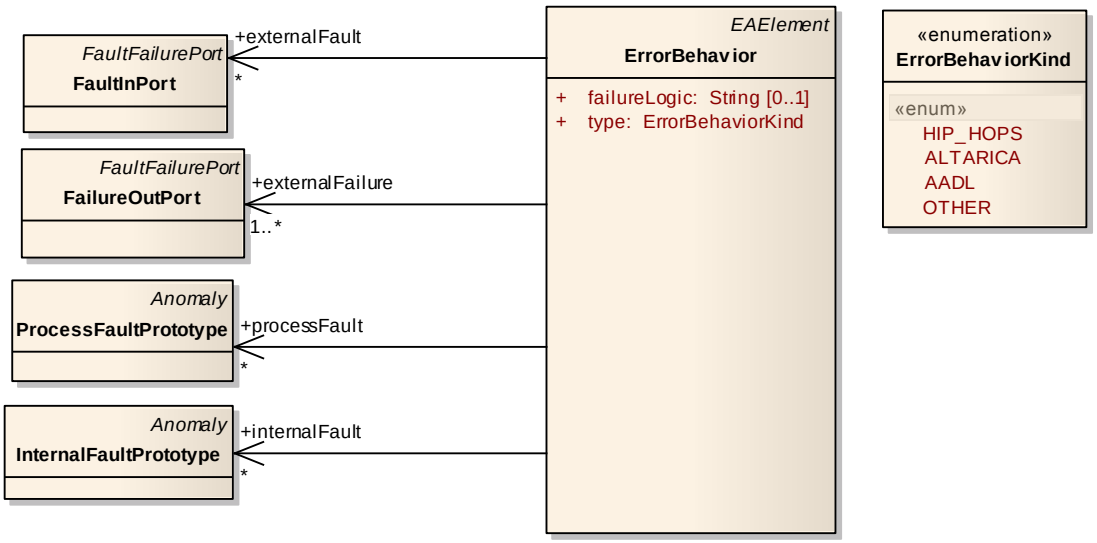


- Internal and external faults are addressed in both ErrorModelType and ErrorBehavior. Distinction is needed to improve visibility. In ErrorModelType, internal details of the target elements should not be visible (black box view abstracting from internal propagation) but only FaultIn and FailureOut. Then in a second step, ErrorBehavior of the ErrorModel should be defined and information about error propagation within the target element (Internal faults) should be attached.

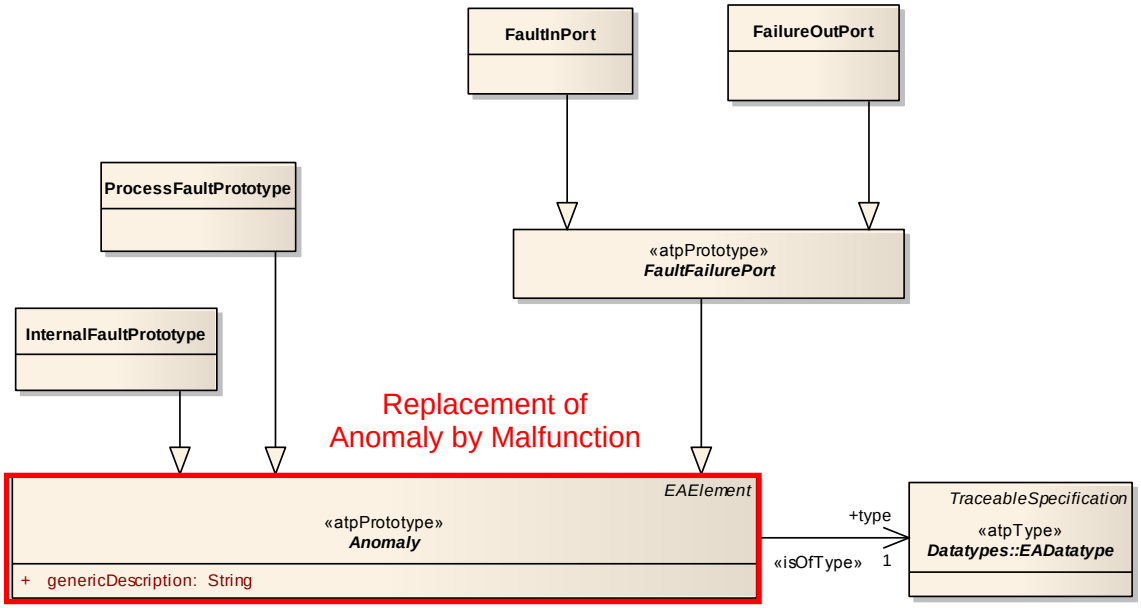


- The ErrorModel Meta-model from EAST-ADLV2.1 is not very constrained and allows lots of freedom in its implementation. As an example, it is possible to associate in an ErrorModelType an HwComponentType and a FunctionType at the same time. This is not correct, but it is still possible. Therefore the ErrorModel should be reworked in order to avoid such scenarios and reduce the risk of applying the meta-model in the wrong way.

- In EAST-ADLV2.1 ErrorBehavior, failureLogic expression permits to express an error behavior language kind other than HiP-HOPS, ALTARICA or AADL by using enumeration OTHER. But this failureLogic notation is only informal. In WT3.3.1, it was decided to specify a well-defined SAFE language including its grammar. Therefore a new meta-model proposal should be done in order to be able to compose our failureLogic expression using formulae and referencing internal faults, process fauls, FaultIn, FailureOut automatically. Then the SAFE language will enforce a semi-formal notation of error propagation.



- In EAST-ADLV2.1 FaultFailure/Anomaly permits to represent different faults or failures depending on the range of its EADatatype which is an enumeration e.g. {Omission, Commission, Value…). It is here proposed to replace Anomaly by a more generic concept as the Malfunction, as it can be useful and easy to exhibit it up to different architecture levels up to the item. A malfunction would be defined as a failure or unintended behavior of the item or element of the item that has the potential to propagate. InternalFaults and ProcessFaults are unintended behavior and therefore Malfunction. FaultIn is propagating to FailureOut and therefore they are also Malfunction.



10 WT3.3.1 Contribution to SAFE Meta-Model

Within this chapter the contribution of WT3.3.1 to the SAFE meta-model is described. At the beginning an overview about meta-modeling approach is given which is followed by the detailed description of the classes and interconnections. Moreover, in another chapter the meta-model is described by means of an example.

10.1 Overview

The error meta-model is aligned with way of describing the system model. An error model can be described for different structural elements of the system model: for *AnalysisFunctionType*s, *DesignFunctionTypes, HardwareComponentType*s, *SwComponentType*s or *BSWModuleDescription*s. In addition, it is possible to define an ErrorModelType for the application environment, which can be an assumed environment or describe the error behavior of a concrete ECU instance or ECU partition respectively.

An *ErrorModelType* describes the black-box view in terms of error propagation for the referenced structural element. Thus, the *externalFault*s and externalFailures typed as *MalfunctionPrototype* are associated with the *ErrorModelType*. In addition, in case the error model is described hierarchically, the meta-model allows connecting externalFailures and externalFaults via the "cause-effect relation" named *FaultFailurePropagationLink*.

To white-box the error behavior of a structural element, the meta-model allows to describe the *ErrorBehavior* for a specific *ErrorModelType*. In this case, also the internal details of the structural element are known, and respective *internalFault*s as well as *processFault*s can be described. In addition, it is possible to describe HOW *externalFault*s, *internalfault*s and *processFault*s are related with e*xternalFailur*es, or with other words: how do those faults contribute to the unintended behavior of the architectural element associated via the *ErrorModelType*. For this purpose, the SAFE meta-model allows to either use existing language to describe the internal error propagation (e.g. via Altarica) or to use the simplified SAFE language for the same purpose. The requirements for the grammar and semantics of the simplified SAFE language are described in chapter 8.3.

Error propagation is either internally described via the *ErrorBehavior* or externally via the *FaultFailurePropagationLink* and shall not to be confused with the data flow of values. Error propagation and data flow of values differ in two aspects: First, error propagate horizontally without following the values' data flow through the application environment. Second, malfunctions in the application layer cannot propagate into malfunctions in the application environment.

The *MalfunctionProtoype*s can by typed with the means of *MalfunctionType*s. A *MalfunctionType* allows describing how the unintended behavior is represented. In addition, with the help of the description capabilities of ErrorBehavior and ErrorModelType, it is also possible to describe how the *MalfunctionPrototype* becomes "active" (e.g. assuming a *MalfunctionPrototype* in the role of externalFailure of an ErrorModelType).

Via the ErrorBehavior means of the meta-model it is possible to describe, how external faults or internal faults can lead to the occurrence of this external failure. In a next step, with the help of the hierarchically error modeling approach, it is then possible to describe, how external faults can be caused from preceding architectural elements (e.g. communication partner, execution environment). This way it is possible to describe a complete error propagation chain from the root fault(s) towards the failure of interest.

10.2 Detailed Description of Classes and Links

Type: **Package**

Package: ClassModel

Notes:

ErrorModel

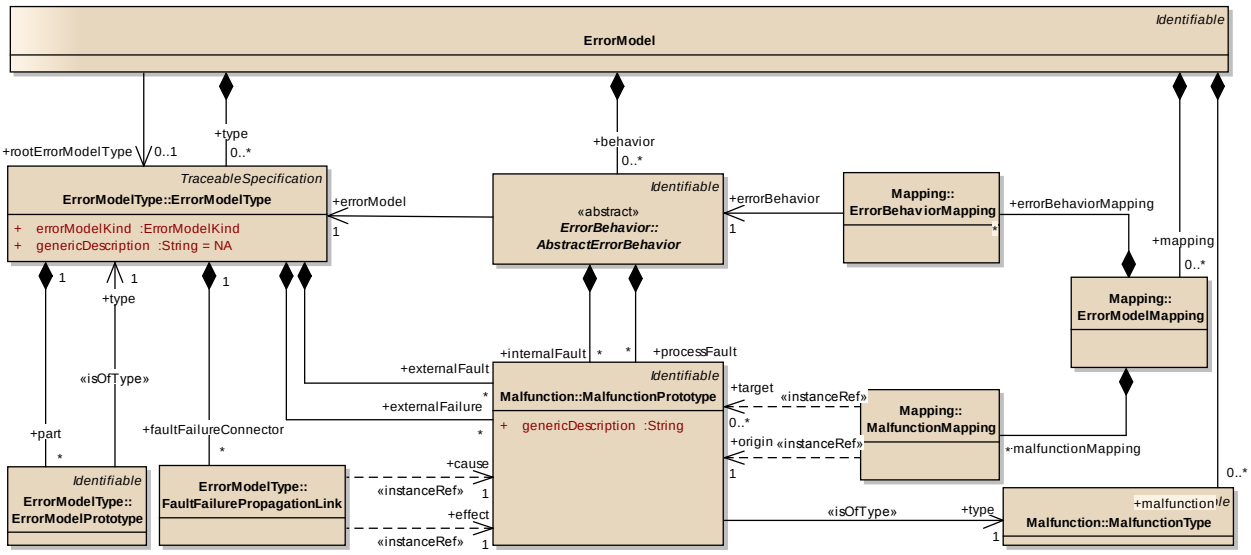


Figure 30 : Overview of WT3.3.1 ErrorModel Package proposal

10.2.1 ErrorModel

Database: Java, *Stereotype:* , *Package:* ErrorModel

Notes: The error model is a container for all artifacts, which are needed to describe the error model of an architectural element: malfunctions, error types and error behaviors.

Relationships

Role	Cardinality	Notes
behavior	**0..***	An arbitrary number of error behaviors.
type	**0..***	An arbitrary number of error model types.
malfunction	**0..***	An arbitrary number of malfunction types.
mapping	**0..***	An arbitrary number of error model mappings.
rootErrorModelType	**0..1**	The root error model type.

10.2.2 ErrorBehavior

Type: **Package**

Package: ErrorModel

Notes:

ErrorBehavior

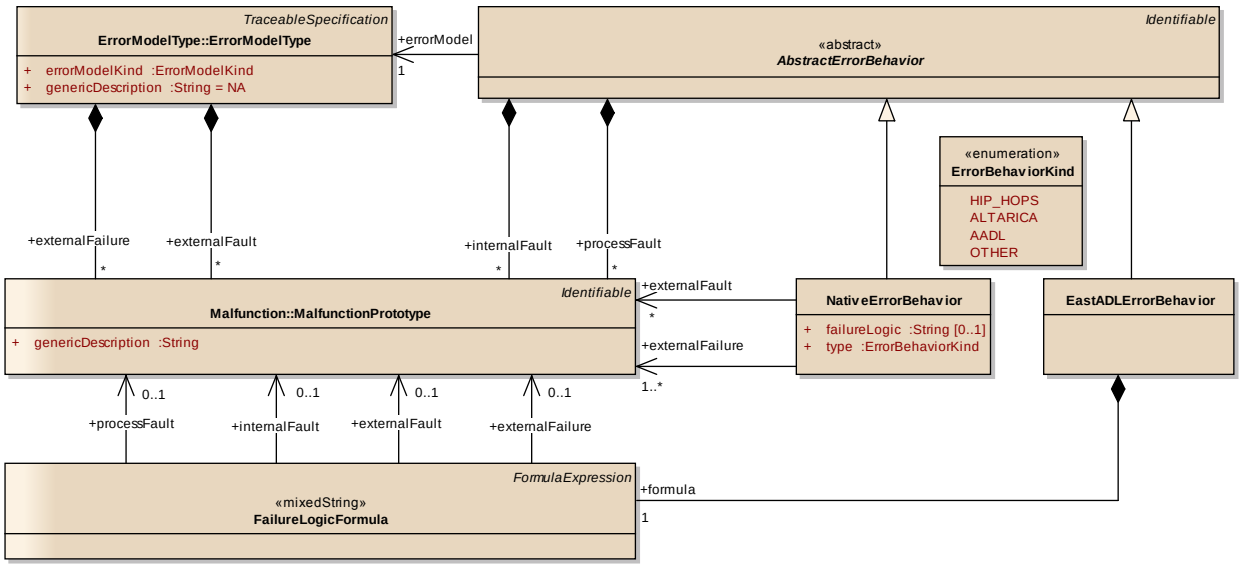


Figure 31 : WT3.3.1 ErrorBehavior proposal

10.2.2.1 AbstractErrorBehavior

Database: Java, *Stereotype:* , *Package:* ErrorBehavior

Notes:

This class contains information about the error behavior independent of concrete behavior descriptions.

The AbstractErrorBehavior contains internalFaults, representing faults that are either propagated to externalFailures of the ErrorModelType or masked, according to the definition of its fault propagation.

A processFault represents a flaw introduced during design, and may lead to any of the failures represented by the ErrorModelType. A processFault therefore has a direct propagation to all externalFailures and cannot be masked.

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

Semantics:

An error behavior describes the error propagation logic of its containing ErrorModelType.

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

Relationships

Role	Cardinality	Notes
processFault	*	processFaults that may affect the ErrorBehavior of the architectural element associated via the ErrorModelType.
internalFault	*	internalFaults that may affect the ErrorBehavior of the architectural element associated via the ErrorModelType.

10.2.2.2 EastADLErrorBehavior

Database: Java, *Stereotype:* , *Package:* ErrorBehavior

Notes: EASTADLErrorBehavior specifies a concrete failure logic description language, which describes the error propagation through the architectural element referenced by the containing ErrorModelType (e.g. function, hw component, sw component).

The failure logic is defined via a formula language called FailureLogicFormula (see "formula" association).

Relationships

Role	Cardinality	Notes
formula	1	Failure logic used to describe the error propagation.

10.2.2.3 ErrorBehaviorKind

Database: Java, *Stereotype:* «enumeration», *Package:* ErrorBehavior

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

Semantics:

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

Extension:

Enumeration, no extension.

Columns

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

10.2.2.4 FailureLogicFormula

Database: Java, *Stereotype:* «atpMixedString», *Package:* ErrorBehavior

Notes: FailureLogicFormula is used to describe the error propagation through the architectural element associated with the containing ErrorModelType. The grammar of the FailureLogicFormula is defined in the respective specification document.

Relationships

Role	Cardinality	Notes
externalFailure	**0..1**	external failures that may result from the ErrorBehavior.
processFault	**0..1**	process faults that influence the errorBehavior .
internalFault	**0..1**	internal faults that influence the errorBehavior .
externalFault	**0..1**	external(incoming) faults that influence the errorBehavior.

10.2.2.5 NativeErrorBehavior

Database: Java, *Stereotype:* , *Package:* ErrorBehavior

Notes: NativeErrorBehavior represents the descriptions of failure logics or semantics that the architectural element associated by the ErrorModelType exhibits.

Semantics:

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

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

Extension:

UML:Behavior

Columns

Name	Type	Notes
failureLogic	String	The specification of error behavior based on an external formalism or the path to the file containing the external specification.
type	ErrorBehaviorKind	The type of formalism applied for the error behavior description.

Relationships

Role	Cardinality	Notes
internalFault	*	internalFaults that influence the errorBehavior.
externalFailure	*	external failures that may result from the ErrorBehavior.
externalFault	*	external(incoming) faults that influence the errorBehavior.
processFault	*	processFaults that may affect the errorBehavior.

10.2.3 ErrorModelType

Type: **Package**

Package: ErrorModel

Notes:

ErrorModelPrototype

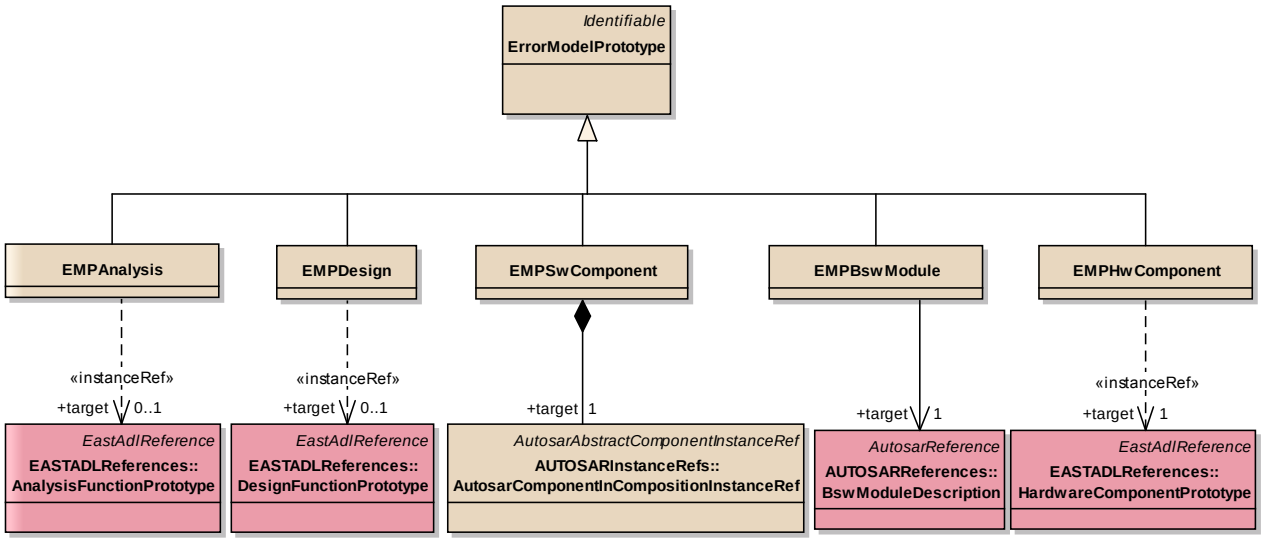


Figure 32 : WT3.3.1 ErrorModelPrototype proposal

ErrorModelType

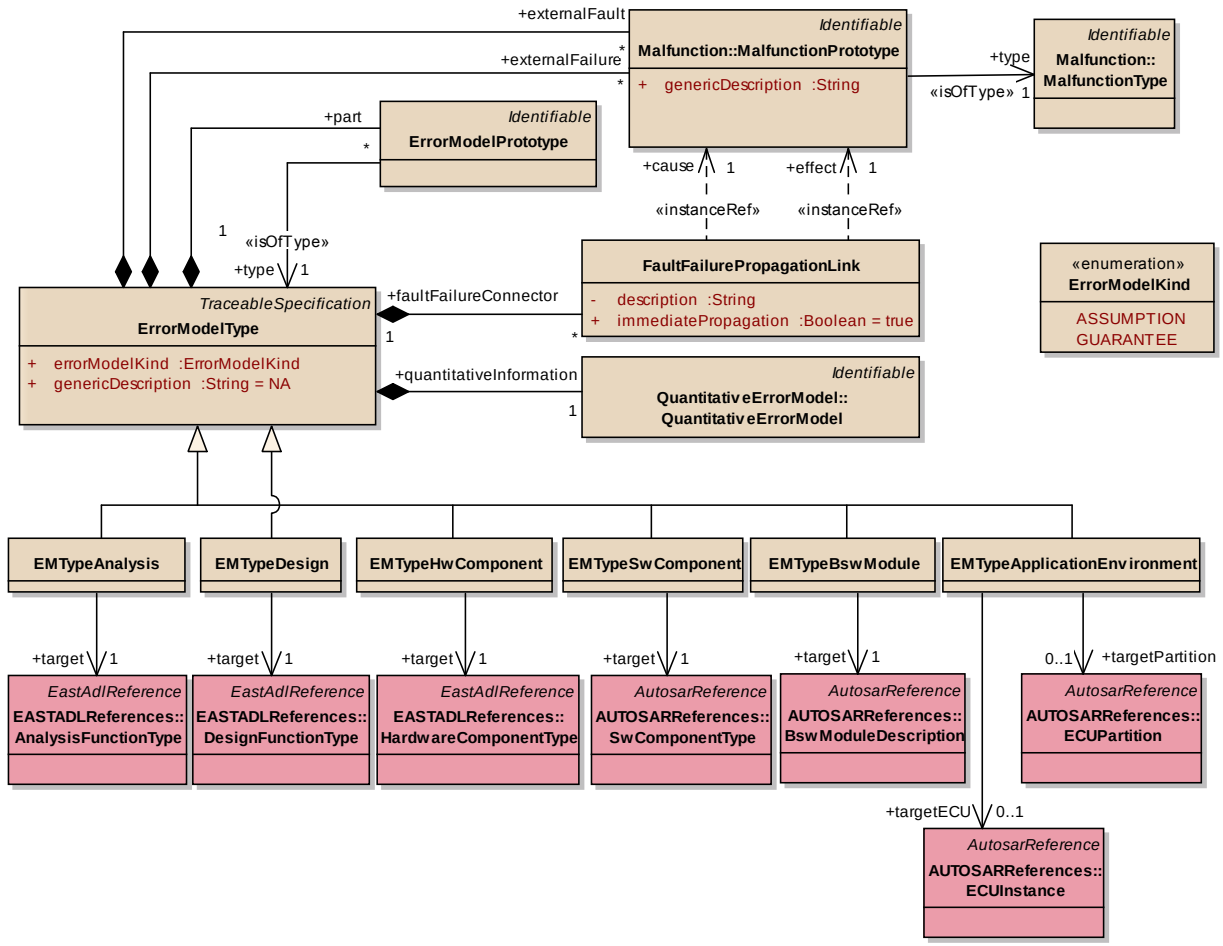


Figure 33 : WT3.3.1 ErrorModelType proposal

10.2.3.1 EMPAnalysis

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model prototype specified for a concrete analysis function instance.

Relationships

Role	Cardinality	Notes
target	**1**	The target function instance.

10.2.3.2 EMPBswModule

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model prototype specified for a concrete bsw software module.

Relationships

Role	Cardinality	Notes
target	**1**	The target basic software module.

10.2.3.3 EMPDesign

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model prototype specified for a concrete function instance.

Relationships

Role	Cardinality	Notes
target	*	A nominal function instance as target of the related error model prototype.

10.2.3.4 EMPHwComponent

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model prototype specified for a concrete hardware component instance.

Relationships

Role	Cardinality	Notes
target	*	A nominal hardware component instance as target of the error model prototype.

10.2.3.5 EMPSwComponent

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model prototype specified for a concrete software component instance.

Relationships

Role	Cardinality	Notes
target	1	the target software component instance.

10.2.3.6 EMTypeAnalysis

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model type specified for a concrete analysis function type.

Relationships

Role	Cardinality	Notes
target	1	the target analysis function

10.2.3.7 EMTypeBswModule

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model type specified for a concrete basic software module.

Relationships

Role	Cardinality	Notes
target	**1**	the target basic software module.

10.2.3.8 EMTypeDesign

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model type specified for a concrete function.

Relationships

Role	Cardinality	Notes
target	**1**	the target design function

10.2.3.9 EMTypeHwComponent

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model type specified for a concrete hardware component.

Relationships

Role	Cardinality	Notes
target	**1**	the target hardware component.

10.2.3.10 EMTypeSwComponent

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes: Error model type specified for a concrete software component.

Relationships

Role	Cardinality	Notes
target	**1**	the target software component.

10.2.3.11 EMTypeSwComponent

Database: Java, *Stereotype:* , *Package:* ErrorModelType

Notes:

The EMTypeApplicationEnvironment allows describing the error model for the runtime environment of application software on AUTOSAR implementation level. The application environment can be virtual, meaning that the modeled error model is an assumption of the environment the application software is running on. In a real system, EMTypeApplicationEnvironment is associated with a concrete ECUInstance in the system and (optional) with an ECU partition within this ECU. In this case, the error model represents the description of the unintended behavior originated by this runtime environment.

Relationships

Role	Cardinality	Notes
targetECU	0..1	the target ECU.

10.2.3.12 ErrorModelPrototype

Database: Java, *Stereotype:* «atpPrototype», *Package:* ErrorModelType

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

There are different subtypes of ErrorModelPrototype specified, allowing adding additional information describe the context of the ErrorModelProtoype.

Semantics:

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

Extension:

(See ADLFunctionPrototype)

Relationships

Role	Cardinality	Notes
type	**1**	The ErrorModelType that types the ErrorModelPrototype.

10.2.3.13 ErrorModelType

Database: <none>, *Stereotype:* «atpType», *Package:* ErrorModelType

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

Independent of the different subtypes of ErrorModelType, this class describes the external faults affecting the element, external failures caused by the element and fault propagations within the nominal element.

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

Constraints:

For an ErrorModelType without part, a respective error behavior shall be defined in the safety model.

Semantics:

The ErrorModelType represents a specification of the faults and fault propagations of its target element.

Both types and prototypes may be targets, and the following cases are relevant:

- One nominal type:

The ErrorModelType represents the identified nominal type wherever this nominal type is instantiated.

- Several nominal types:

The ErrorModelType represents the identified nominal types individually, i.e. the same error model applies to all nominal types and is reused.

- One nominal prototype:

The ErrorModelType represents the identified nominal prototype whenever its context, i.e. its top-level composition is instantiated.

- Several nominal prototypes with instanceref:

The ErrorModelType represents the identified set of nominal prototypes (together) whenever their context, i.e. their top-level composition, is instantiated.

The fault propagation of an errorModelType is defined by its contained parts, the ErrorModelPrototypes and their connections. In case an error behavior is defined for this error model type, the fault propagation information, the error behavior and the parts of the error model shall be consistent.

FaultFailurePropagationLinks define valid propagation paths in the ErrorModelType. In case the contained external faults and external failures reference nominal ports, the connectivity of the nominal model may serve as a pattern for connecting malfunction prototypes in the ErrorModelType.

Extension:

(see ADLTraceableSpecfication)

Columns

Name	Type	Notes
errorModelKind	ErrorModelKind	
genericDescription	String	

Relationships

Role	Cardinality	Notes
faultFailureConnector	*	The contained links for internal propagation of faults/failures between the subordinate error models.
externalFault	*	The external faults affecting the proper execution of the architectural element associated with the error model type.
externalFailure	*	The external failures visible at the borders of the architectural element.
part	*	The contained error models forming a hierarchy.
quantitativeInformation	*	The quantitative information for this ErrorModelType.

10.2.3.14 FaultFailurePropagationLink

Database: <none>, *Stereotype:* , *Package:* ErrorModelType

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

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

A FaultFailurePropagationLink can only connect fault/failures that have compatible types.

Constraints:

[1] Only compatible cause-effect pairs may be connected.

[2] Two fault/failure are compatible if the MalfunctionType of the cause represents a subset of the MalfunctionType set represented by the MalfunctionType of the effect.

Semantics:

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

Extension:

UML::Connector

Columns

Name	Type	Notes
Description	String	
immediatePropagation	Boolean	

Relationships

Role	Cardinality	Notes
effect	1	
cause	1	

10.2.4 QuantItativeErrorModel

Type: **Package**

Notes:

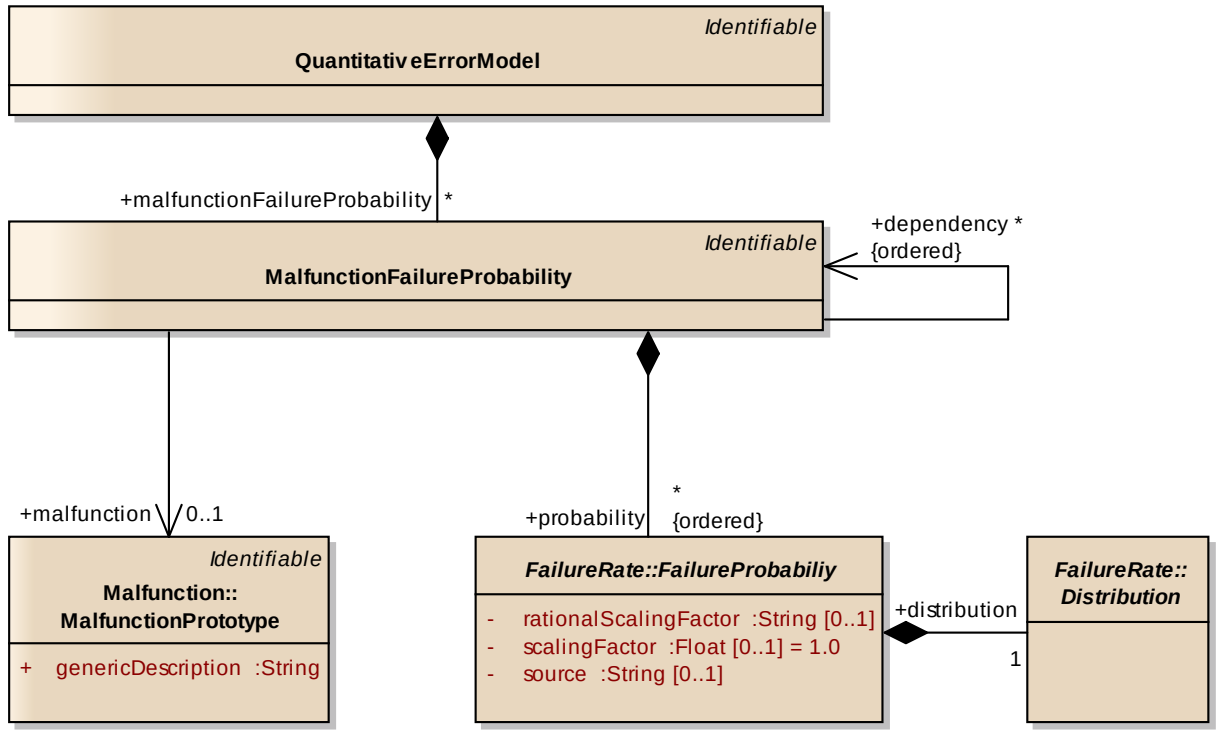


Figure 34 : WT3.3.1 QuantitativeErrorModel proposal

10.2.4.1 QuantitativeErrorModel

Notes:

Semantics:

This class is the container for all quantitative information relevant for the the error model associated with this class.

Relationships

Role	Cardinality	Notes
malfunctionFailureProbability	0..*	The failure probability of the malfunctions defined in the containing ErrorModelType.

10.2.4.2 MalfunctionFailureProbability

Notes:

Semantics:

This class attaches failure probability information to a MalfunctionPrototype.

In case the association "dependency" is empty (no dependencies), a MalfunctionFailureProbability shall contain exactly one FailureProbability (|probability|=1).

In all other cases, the number of FailureProbabilities shall be exactly |probability| = 2^(|dependency|). In this case, it provides the failure probability dependent on the state of the dependencies in a natural order.

Take the following example: Malfunction A depends on Malfunction B and Malfunction C. In this case, the instance of MalfunctionFailureProbability associated with Malfunction A defines four failureProbabilities (via association "probability") with the following semantics:

- The first entry specifies the probability of Malfunction A, if Malfunction B and Malfunction C do NOT occur
- The second entry specifies the probability of Malfunction A, if Malfunction B does NOT occur AND Malfunction C does occur
- The third entry specifies the probability of Malfunction A, if Malfunction B does occur AND Malfunction C does NOT occur
- The fourth entry specifies the probability of Malfunction A, if Malfunction B and Malfunction C do occur

Relationships

Role	Cardinality	Notes
failureProbability	0..*	The concrete failure probabilities of the Malfunction defined via the "malfunction" relation.
malfunction	0..1	The malfunction for which the quantitative information is described.

10.2.5 FailureProbability

Type: **Package**

Notes:

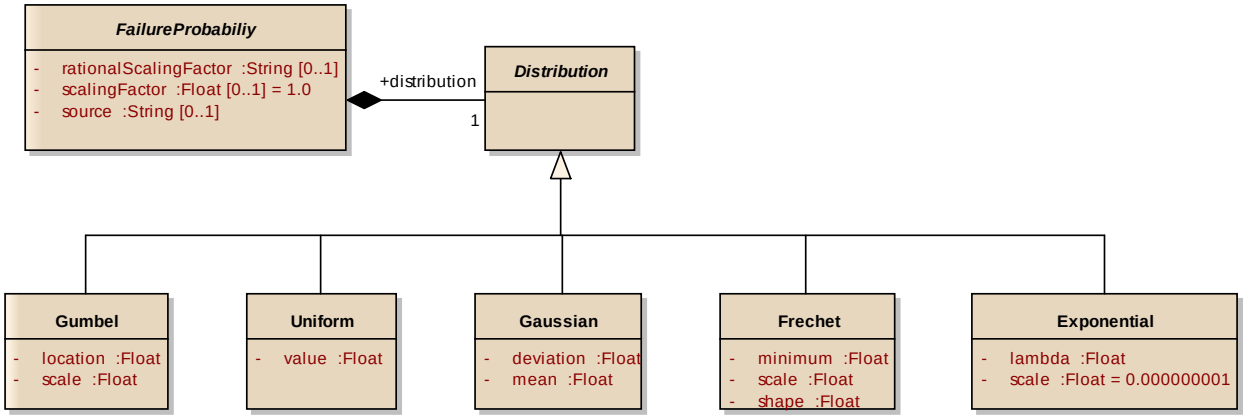


Figure 35 : WT3.3.1 FailureProbability proposal

10.2.5.1 Distribution

Notes:

Abstract super type of the different supported distributions.

10.2.5.2 FailureProbability

Notes:
This class provides generic information required by all the concrete distributions gaussian, gumbel, frechet, exponential and uniform.

Relationships

Role	Cardinality	Notes
probability	0..*	The concrete failure probabilities of the Malfunction defined via the "malfunction" relation.
malfunction	0..1	The malfunction for which the quantitative information is described.

Columns

Name	Type	Notes
rationalScalingFactor	String	The rationaleScalingFactor shall provide a rationale, if a scaling factor different to 1.0 is applied.
source	String	FIT rate source shall documented according to possible source as described in ISO 26262 Part 5 8.4.3.
scalingFactor	Float	The scalingFactor allows potential scaling between different sources of failure probabilities as described in ISO Part 5 Annex F.

10.2.5.3 Frechet

Notes:
Container to model the parameters "location" and "scale", required by the Frechet distribution.

Columns

Name	Type	Notes
minimum	Float	Parameter m of the Frechet distribution.
scale	Float	Parameter s of the Frechet distribution.
shape	Float	Parameter alpha of the Frechet distribution.

10.2.5.4 Exponential

Notes:
Container to model the parameters "lambda" and "scale", required by the exponential distribution: : f(t) = e^(-lambda*scale*t)

Columns

Name	Type	Notes
lambda	Float	Failure rate of the distribution.
scale	Float	The scale applied on the lambda value. The default value equals 10^-9 with the meaning of lambda = FIT.

10.2.5.5 Gaussian

Notes:
Container to model the parameter "deviation" and "mean" required by the Gaussian distribution.

Columns

Name	Type	Notes
deviation	Float	Parameter sigma of the Gaussian distribution.
mean	Float	Parameter mu of the Gaussian distribution.

10.2.5.6 Uniform

Notes:
Container to model the parameter "value" required by the Uniform distribution.

Columns

Name	Type	Notes
Value	Float	Max time interval of the distribution.

10.2.5.7 Gumbel

Notes:
Container to model the parameter location and scale required by the Gumbel distribution.

Columns

Name	Type	Notes
location	Float	Parameter mu of the Gumbel distribution.
scale	Float	Parameter beta of the Gumbel distribution..

10.2.6 Malfunction

Type: **Package**

Package: ErrorModel

Notes:

MalfunctionPrototype

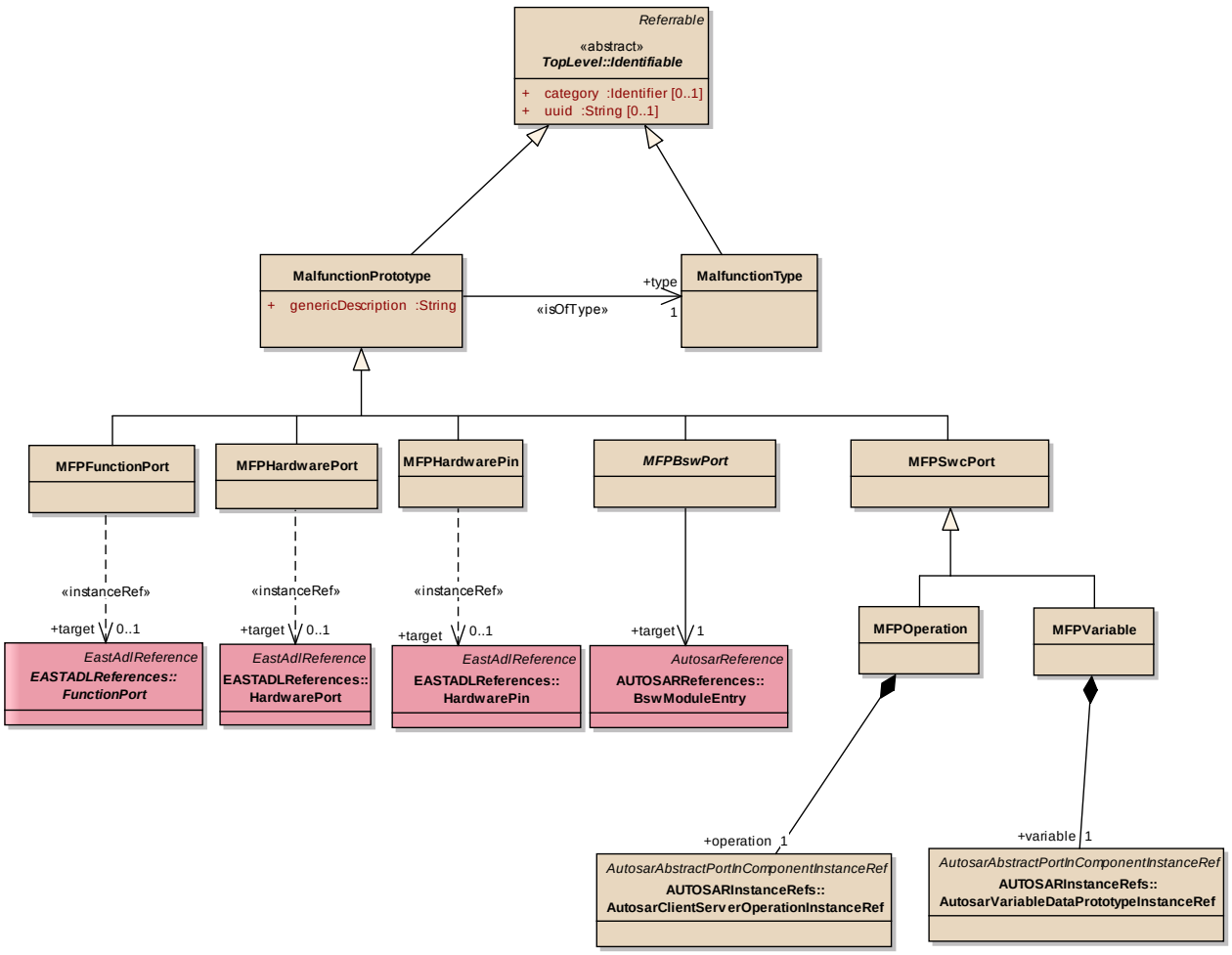


Figure 36 : WT3.3.1 MalfunctionPrototype proposal

MalfunctionType

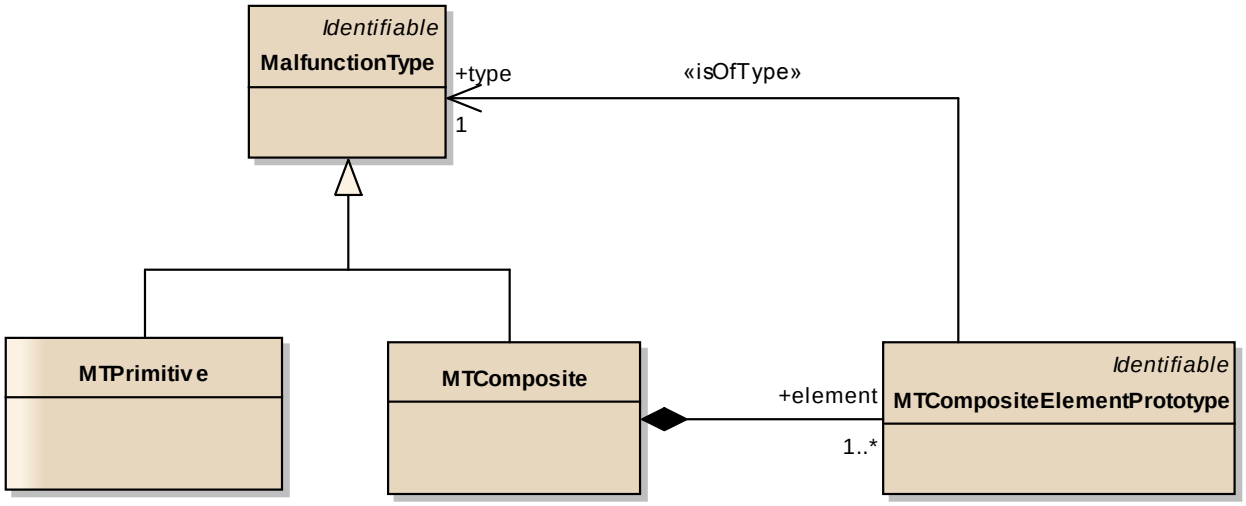


Figure 37 : WT3.3.1 MalfunctionType proposal

10.2.6.1 MFPBswPort

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes:

Semantics:

The MalfunctionPrototype pointing to a basic software module entry.

Relationships

Role	Cardinality	Notes
target	**1**	The target bsw module entry.

10.2.6.2 MFPFunctionPort

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes: The MalfunctionPrototype pointing to a function port instance.

Extension:

UML::Port

Relationships

Role	Cardinality	Notes
functionTarget	0..1	A nominal function port instance as target of the malfunction prototype.

10.2.6.3 MFPHardwarePort

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes: The MalfunctionPrototype pointing to a HardwarPin instance.

Extension:

UML::Port

Relationships

Role	Cardinality	Notes
target	*	A nominal HW port instance as target of the malfunction prototype.

10.2.6.4 MFPHardwarePin

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes: The MalfunctionPrototype pointing to a HardwarPin instance.

Extension:

UML::Port

Relationships

Role	Cardinality	Notes
target	*	A nominal HW pin instance as target of the malfunction prototype.

10.2.6.5 MFPOperation

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes: The MalfunctionPrototype pointing to an AUTOSAR operation instance.

Relationships

Role	Cardinality	Notes
operation	1	The target operation prototype instance.

10.2.6.6 MFPSwcPort

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes: The MalfunctionPrototype pointing to a HardwarPin instance.

10.2.6.7 MFPVariable

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes: The MalfunctionPrototype pointing to an AUTOSAR variable instance.

Relationships

Role	Cardinality	Notes
variable	**1**	The target variable prototype instance.

10.2.6.8 MTComposite

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes: This composite malfunction type allows defining the different ways, how the malfunction becomes visible. As a typical example, a composite malfunction type could have the elements "commission" and "omission".

BrakeMalfunctionType (type: MTComposite):

- BrakePressureTooLow (type: MTPrimitive): Semantics="brake pressure is below 20% of requested value"
- Omission (type: MTPrimitive): Semantics="brake pressure is below 10% of maximal brake pressure"
- Commission (type: MTPrimitive): Semantics="brake pressure exceeds requested value with more than 10% of maximal brake pressure".

Relationships

Role	Cardinality	Notes
element	1..*	Elements of the composite malfunction type..

10.2.6.9 MTCompositeElementPrototype

Database: Java, *Stereotype:* «atpFeature», *Package:* Malfunction

Notes:

This class is used to apply the composite pattern for the definition of MalfunctionTypes. It is contained by the composite MalfunctionTtype and refers to the type of the elements within the composite MalfunctionType.

10.2.6.10 MTPrimitive

Database: Java, *Stereotype:* , *Package:* Malfunction

Notes:

This class is used for the atomic definition of a malfunction type. The description field of the derived Identifiable class shall be used to describe the malfunction type.

Relationships

Role	Cardinality	Notes
type	1	Type of Malfunction.

10.2.6.11 MalfunctionPrototype

Database: Java, *Stereotype:* «atpPrototype», *Package:* Malfunction

Notes: A malfunction is a failure or unintended behavior of the item or element of the item that has the potential to propagate. The MalfunctionPrototype metaclass represents an error that may occur internally in an ErrorModel or be propagated to it, or a failure that is propagated out of an Error Model. The MalfunctionPrototype may represent different errors depending on its type (enumeration of generic description).

Semantics:

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

Extension:

(UML::Part)

Columns

Name	Type	Notes
genericDescription	String	A description of the MalfunctionPrototype

Relationships

Role	Cardinality	Notes
type	**1**	The type of the malfunction prototype. It describes how the malfunction prototype becomes visible.

10.2.6.12 MalfunctionType

Database: Java, *Stereotype:* «atpType», *Package:* Malfunction

Notes: A MalfunctionType describes how a malfunction becomes visible. Currently, it can either be a primitive description of a malfunction or defines as hierarchical composition of different "appearance" possibilities.

10.2.7 Mapping

Type: **Package**

Package: Mapping

Notes:

ErrorBehaviorMapping

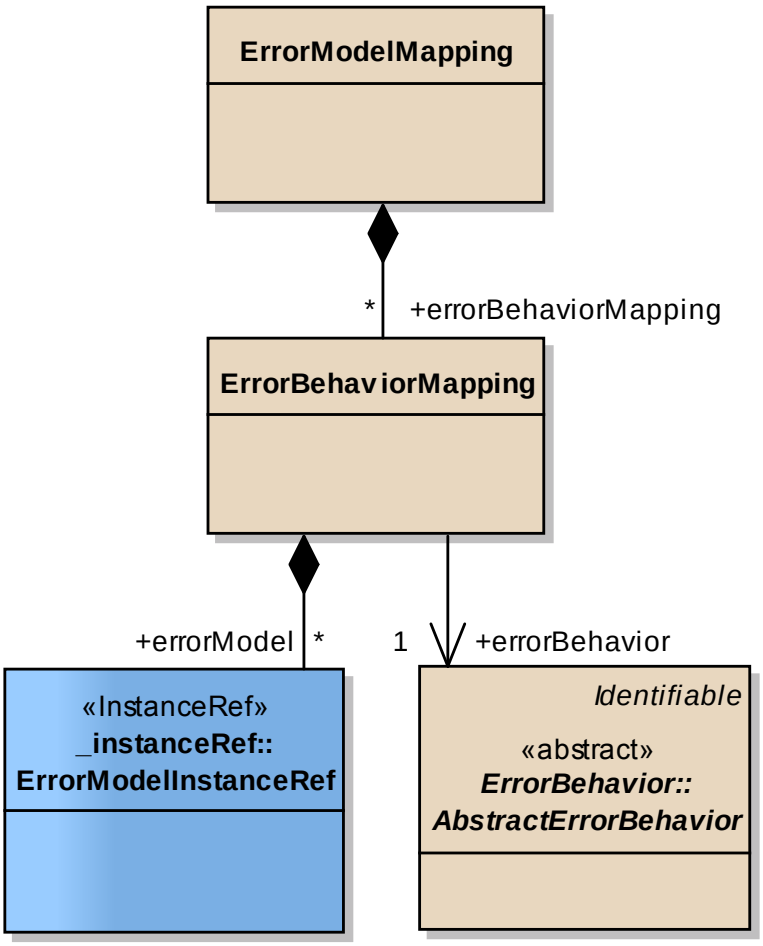


Figure 38 : WT3.3.1 ErrorBehvaiorMapping proposal

10.2.7.1 ErrorBehaviorMapping

Database: Java, *Stereotype:* , *Package:* Mapping

Notes:
The class ErrorBehaviorMapping allows mapping an error behavior to a concrete instance of an ErrorModelType within the ErrorModel. For each instance reference referred by the association named "errorModel", the following rule must be considered:

- The ErrorModelType of the first context element in the instanceRef of type ErrorModelInstanceRef must match with the attribute "rootErrorModelType" of the ErrorModel, where this ErrorBehaviorMapping is contained.

Relationships

Role	Cardinality	Notes
errorBehavior	1	The target error behavior.
errorModel	*	The error model instances, for which this error behavior shall be applied.

10.2.7.2 ErrorModelMapping

Database: Java, *Stereotype:* , *Package:* Mapping

Notes:
This class contains mapping information for error model.

Relationships

Role	Cardinality	Notes
errorBehaviorMapping	1	An arbitrary number of error behavior mappings.

10.2.7.3 MalfunctionMapping

Database: Java, *Stereotype:* , *Package:* Mapping

Notes:
Via the class MalfunctionMapping it is possible to map malfunctions of one abstraction level (e.g. EAST ADL implementation level) to another level of abstraction (e.g. EAST ADL analysis level). This way the correlation between different levels of abstraction can be made explicit.

The MalfunctionMapping shall be attached to the "lower" level of abstraction. Example:

- a malfunction defined for a software component (implementation level) shall be related with a malfunction defined on design level
- in this case, (at least) two instances of SafetyExtensions (TechnicalSafetyExtension, AutosarSystemSafetyExtension) exist. In each of them an ErrorModelType is defined containing the before mentioned malfunctions
- in the AutosarSystemSafetyExtension, we define an MalfunctionMapping and relate the software malfunction (in the role of "origin") to the malfunction defined within the TechnicalSafetyExtension (in the role "target")

Thus, the following rules shall be applied:

- the MalfunctionPrototype referenced via the "origin" relationship shall be defined within the same SafetyExtension as an instance of this class
- the MalfunctionPrototype references via the "target" relationship shall be defined within a SafetyExtension which corresponds to a higher level of abstraction

Relationships

Role	Cardinality	Notes
origin	1	The origin of the mapping, located on the lower level of abstraction.
target	0..*	The targets of the mapping, located on the higher level of abstraction.

10.2.8 _instanceRef

Type: **Package**

Package: ErrorModel

Notes:

EMPAnalysis_analysisFunctionTarget

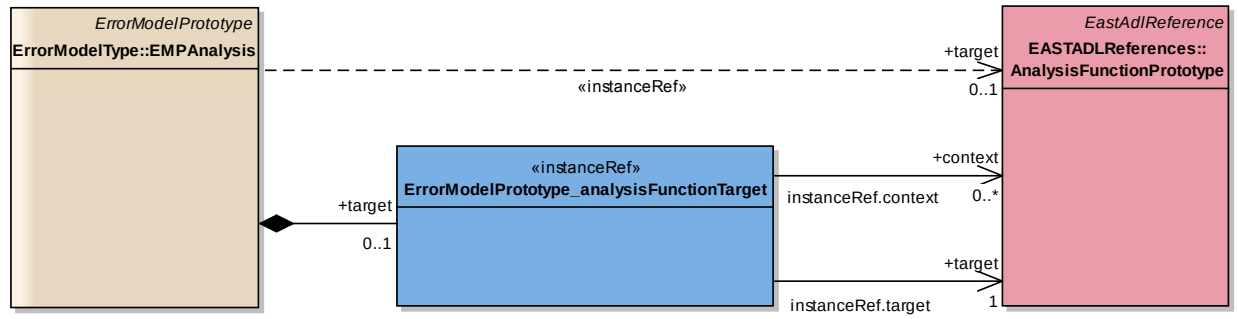


Figure 39 : WT3.3.1 EMPAnalysis InstanceRef proposal

EMPDesign_designFunctionTarget

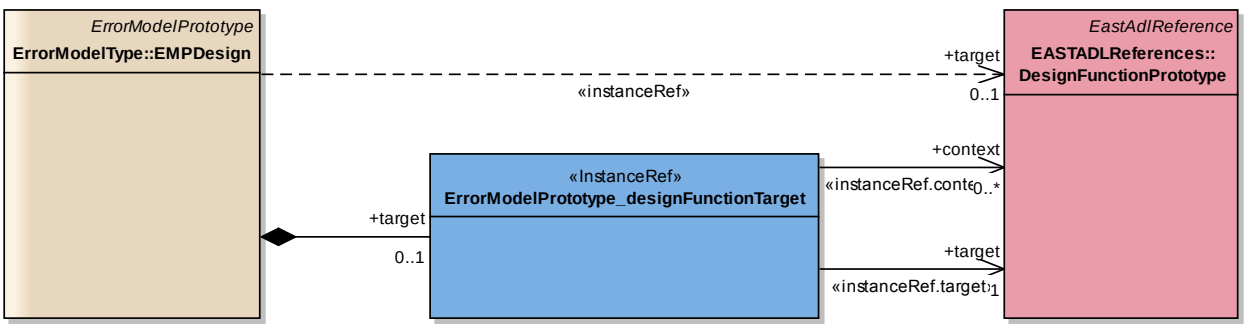


Figure 40 : WT3.3.1 EMPDesign InstanceRef proposal

EMPHwComponent_hwTarget

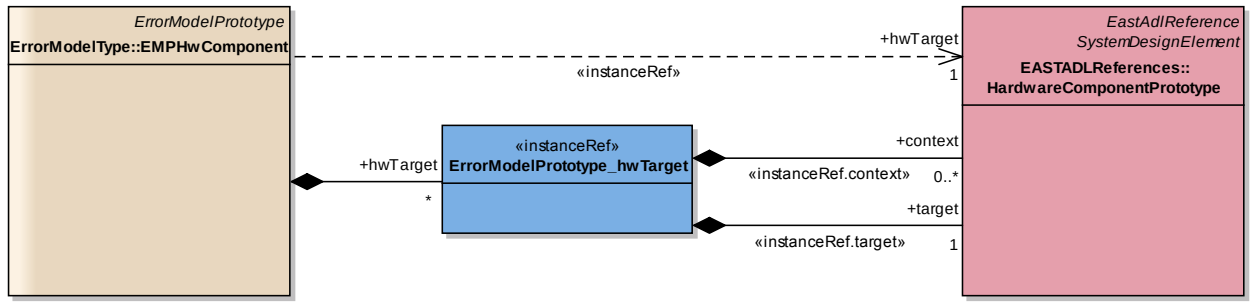


Figure 41 : WT3.3.1 EMPHwComponent InstanceRef proposal

ErrorModelInstance

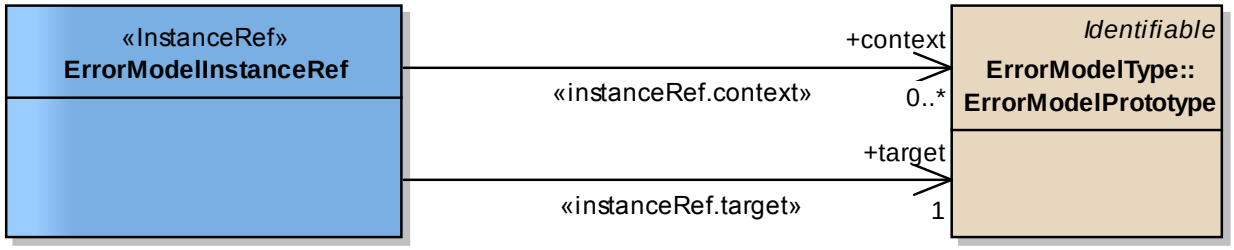


Figure 42 : WT3.3.1 ErrorModelInstance InstanceRef proposal

ErrorModelMapping

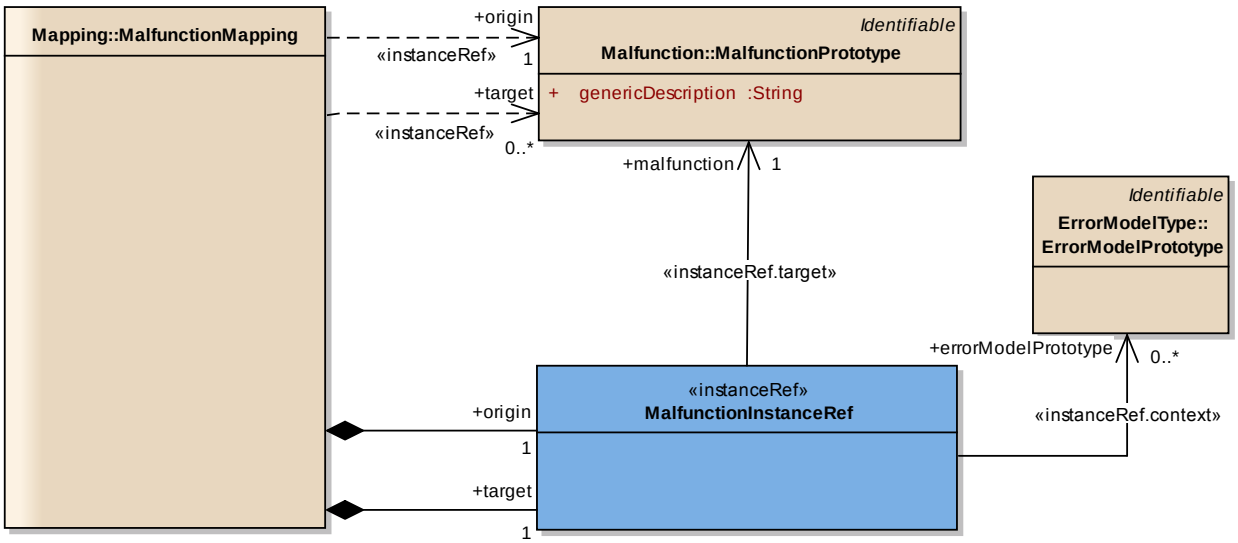


Figure 43 : WT3.3.1 ErrorModelMapping InstanceRef proposal

FaultFailurePropagationLink

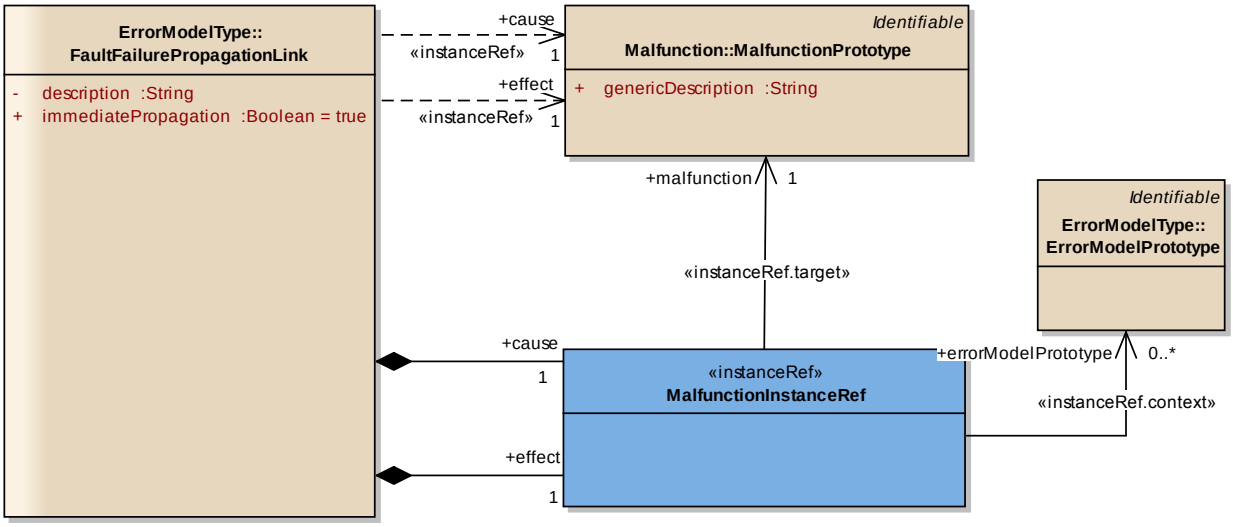


Figure 44 : WT3.3.1 FaultFailurePropagationLink InstanceRef proposal

MFPFunctionPort_functionTarget

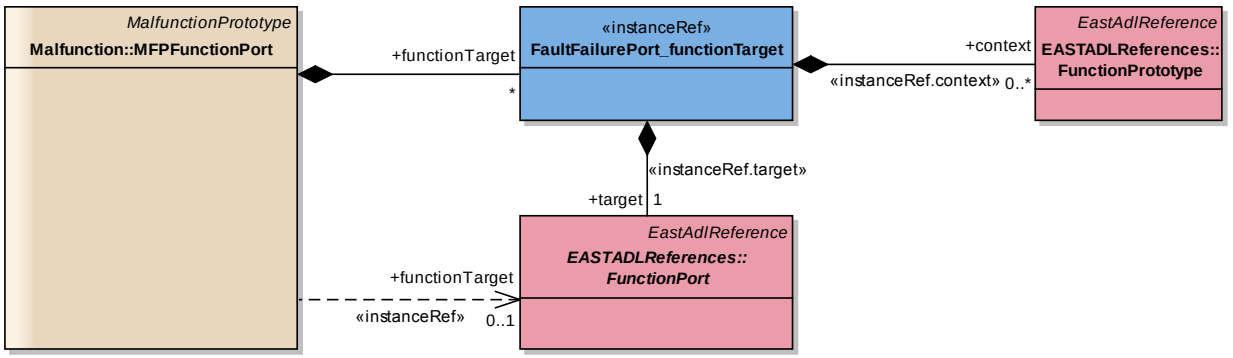


Figure 45 : WT3.3.1 MFPFunctionPort InstanceRef proposal

MFPHardwarePin_hwTarget

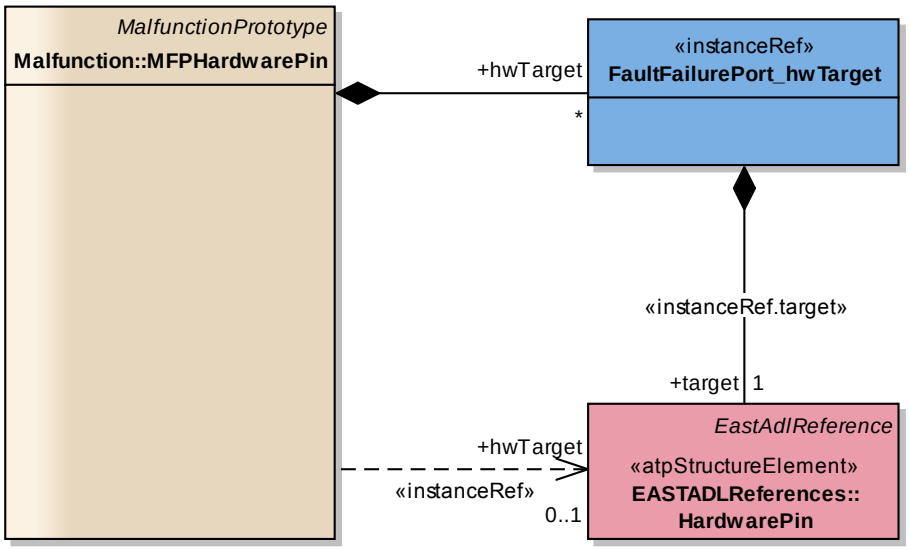


Figure 46 : WT3.3.1 MFPHardwarePin InstanceRef proposal

MFPHardwarePort_hwTarget

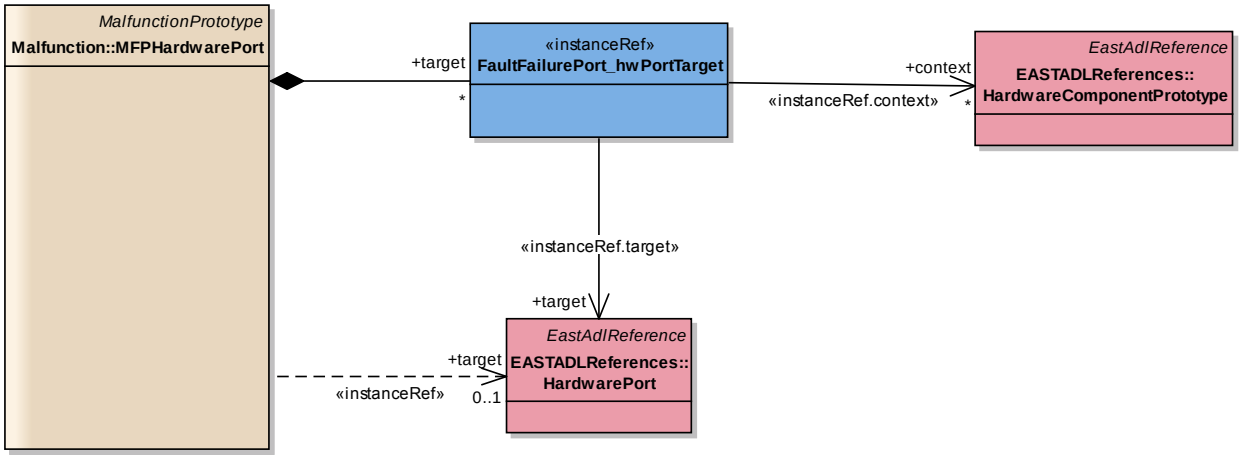


Figure 47 : WT3.3.1 MFPHardwarePort InstanceRef proposal

10.2.8.1 ErrorModelInstanceRef

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

10.2.8.2 ErrorModelPrototype_analysisfunctionTarget

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

Relationships

Role	Cardinality	Notes
target	*	A nominal analysis function instance as target of the related error model prototype.

10.2.8.3 ErrorModelPrototype_designfunctionTarget

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

Relationships

Role	Cardinality	Notes
target	*	A nominal design function instance as target of the related error model prototype.

10.2.8.4 ErrorModelPrototype_hwTarget

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

Relationships

Role	Cardinality	Notes
target	*	A nominal hardware component instance as target of the error model prototype.

10.2.8.5 FaultFailurePort_functionTarget

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

Relationships

Role	Cardinality	Notes
target	*	A nominal function port as target of the malfunction prototype.

10.2.8.6 FaultFailurePort_hwPinTarget

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

Relationships

Role	Cardinality	Notes
target	*	A nominal HW pin instance as target of the malfunction prototype.

10.2.8.7 FaultFailurePort_hwPortTarget

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

Relationships

Role	Cardinality	Notes
target	*	A nominal HW port instance as target of the malfunction prototype.

10.2.8.8 MalfunctionInstanceref

Database: Java, *Stereotype:* «instanceRef», *Package:* _instanceRef

Notes:

Relationships

Role	Cardinality	Notes
target	*	The malfunction instance this instance ref finally refers to.

10.3 WT3.3.1 Meta-model Description Based on an Example

In this chapter, we show some simple examples for the use of the meta-model described in chapter 10.2. We describe how to model a hierarchy of components and how to model malfunctions.

We omit examples for the other aspects of the meta model. In addition, the examples do not show how the meta-model elements for describing error behavior can be used, and the link to the system model is missing as well (e.g. an EMTypeSwComponent is not pointing to an AUTOSAR software component type). This will be subject of upcoming deliverable versions.

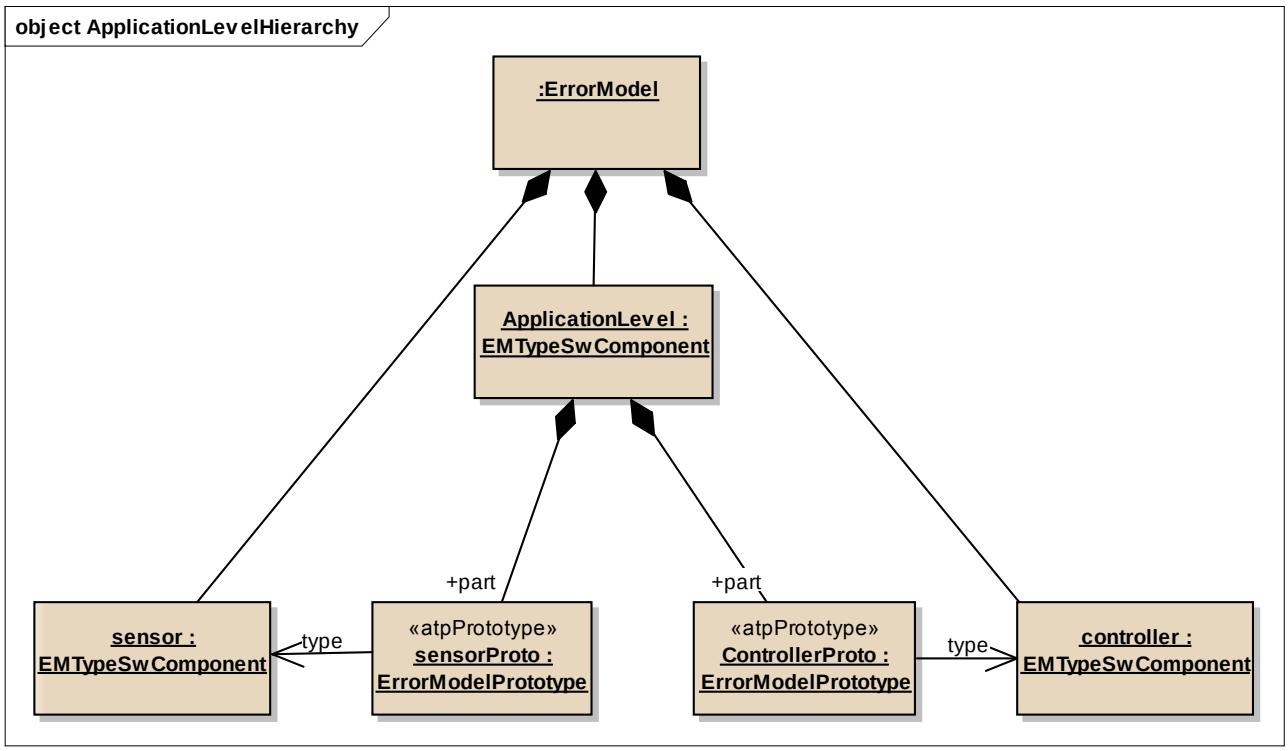


Figure 48 : Application Level Hierarchy diagram highlighting hierarchy modeling capability

This diagram above shows how to model a hierarchy of software components error models. An error model for the software composition “ApplicationLevel” contains two *ErrorModelPrototypes* “sensorProto” and “ControllerProto”.

The two software components are of type “sensor” and “controller”. These two *EMTypeSwComponents* could be again a composite error model type, and hence would allow a hierarchy of error models.

The diagram shown hereafter (see *Figure 49*) refines the application level hierarchy from *Figure 48* and adds four malfunction prototypes.

These four malfunctions prototypes are ApplicationEnvironmentMalfunctionProto (the malfunction caused by the application environment), SensorApplicationEnvironmentMalfunctionProto (the malfunction from the application environment which affects the sensor), SensorComputationMalfunctionProto (the external fault emitted from the sensor computation), and ReceiveSensorComputationMalfunctionProto (the malfunction that the controller receives from the invalid sensor computations).

The former two malfunctions are connected by the *FaultFailurePropagationLink* named "EnvironmentSensorMalfunctionPropagation", the latter two are connected by the *FaultFailurePropagationLink* named "SensorControllerComputationMalfunctionPropagation".

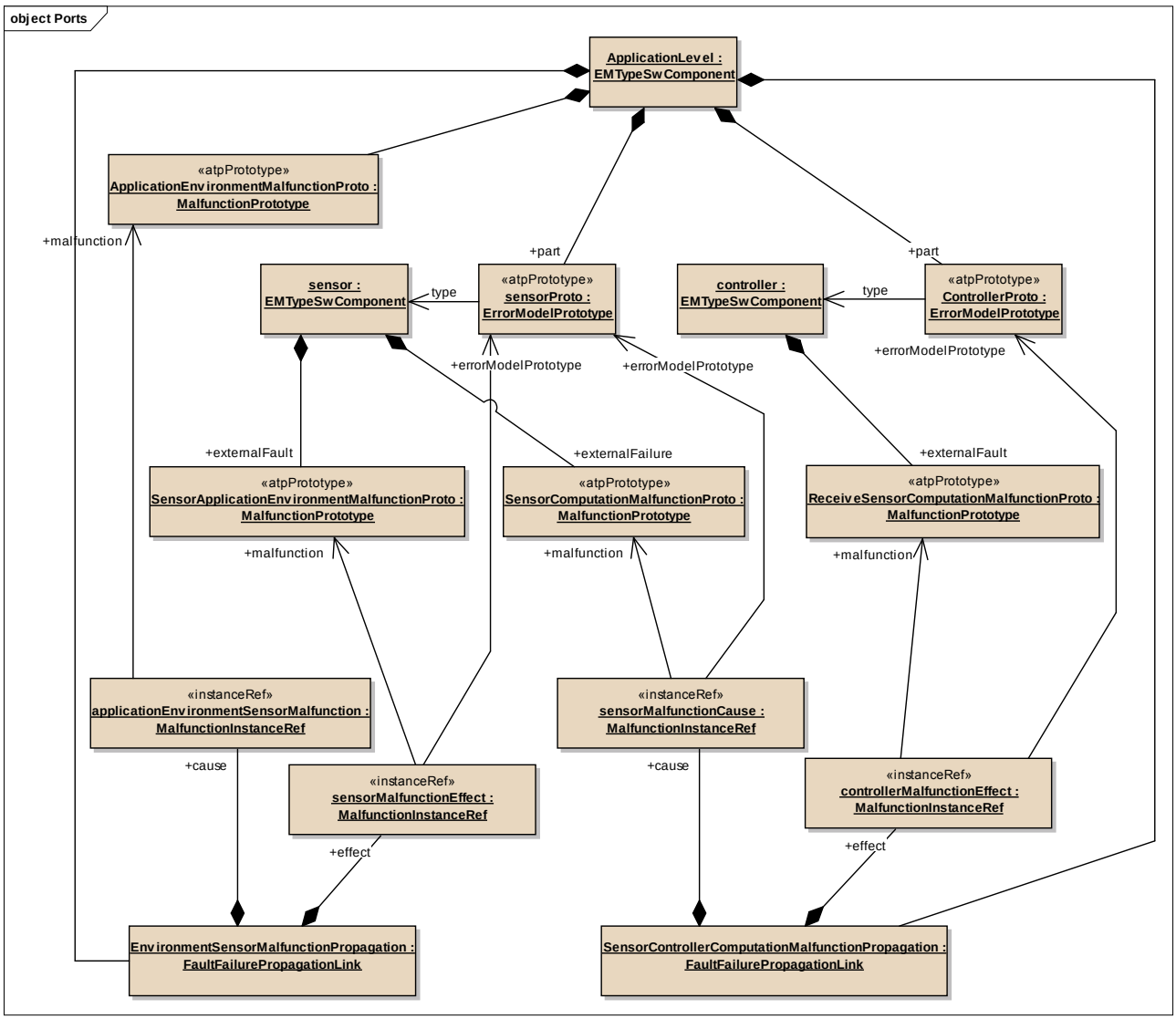


Figure 49 : Application Level Hierarchy refinement with malfunctions added

11 WT3.3.1 Error model Application Rules

The error model as explained in chapter 10 is very flexible and allows many different models for the same system. In order to support exchangeability of analysis models between different tools, SAFE defines a set of patterns that define how the error model shall be used.

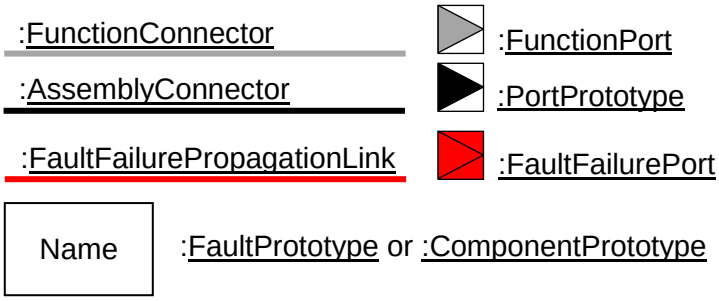


Figure 50: Pattern legend for Applicability

Figure 50 above introduces the set of symbols which are used in the diagrams throughout this chapter. The meta-model elements AssemblyConnector, PortPrototype and ComponentPrototype are defined in the AUTOSAR meta-model, while all others are defined in the SAFE meta-model.

Safety relevant items are normally complex system that consists of hardware elements and software elements. The hardware consists of interconnected Electronic Control Units (ECUs), which can be further decomposed into programmable microcontrollers, other electronic parts and printed circuit boards. The software is composed of many interconnected AUTOSAR software components, which are deployed on the microcontrollers within the ECUs. In addition to AUTOSAR software components, the microcontrollers also contain an AUTOSAR basic software stack, which controls the Microcontroller Unit (MCU) hardware and provides generic services to the software components, like access to input/output channels, persistent memory or partitioning.

While the software architecture for a concrete function is normally defined using the AUTOSAR meta-model, there is currently no widely accepted single meta-model to capture system and hardware architecture. To fill that gap, SAFE uses the hierarchical EAST-ADL FDA and HDA meta-models for the representation of system and hardware architecture.

Once the system architecture and design is modeled in AUTOSAR and EAST-ADL as described above, the model is augmented with a fault and error propagation model, using the error model meta-model of SAFE.

11.1 System Model

In a first step, we focus on the vehicle-network level of abstraction for the system model, which is well suited as a starting point. The software part is represented with the means of AUTOSAR, while the hardware is represented as a network of interconnected ECUs.

This level of abstraction is sufficiently reduced to allow end-to-end analysis while distinction between hardware and software is already visible.

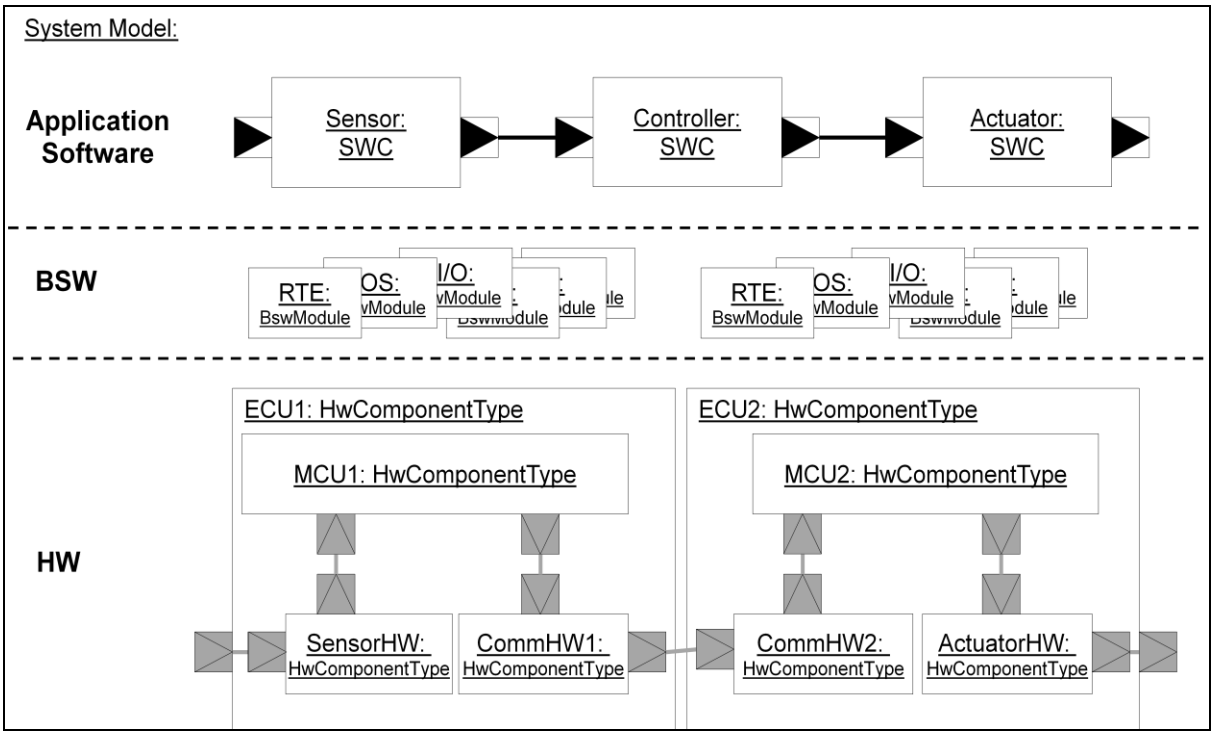


Figure 51 : System model Representation

Figure 51 shows a system model example on implementation level as it would be represented according to the SAFE methodology. The hardware architecture is represented using the HardwareModeling package of the EAST-ADL meta-model, where each ECU, microcontroller and electronic circuit is represented as HardwareComponentType. The software architecture is represented using the AUTOSAR SWC- and System-Template, the basic software is represented using the AUTOSAR BSW Module Template. The mapping of software components on ECUs and the basic software in between is omitted here for simplicity. The AUTOSAR meta-model provides elements to represent this information.

For the sake of completeness: Depending on the level of abstraction, EAST-ADL or AUTOSAR or both may the target for the system model required for safety analysis. As mentioned above, we propose here to use the EAST-ADL capabilities to describe the HW details and use the AUTOSAR SWC- and System-Template to describe the software-relevant information. However, we argue that the demanded system model can be described also by using only one of the mentioned meta-model solutions. E.g. the software-architecture could be described by EAST-ADL facilities like FunctionType, and the hardware architecture could be described via the AUTOSAR ECU Resource Template.

Generally, the system model allows developers to work independently on the different subsystems in the system. In the following, we consider two specific views to the system model and how they are related to the error model.

In a first step (see chapter 11.2), we separate the application layer and the application environment and show how the error model can be used as part of a *safety contract* between those sub-systems.

In a second step (see chapter 11.5), we separate the complete software entities (e.g. basic software, RTE, application software) from the hardware and show how this affects the error model.

11.2 Failure probability of a single error

For an individual malfunction, the failure probability can be expressed via the class MalfunctionFailureProbability. In this case, the class points to the malfunction of interest and contains one instance of FailureProbability, describing the quantitative information via a respective distribution. See chapter 10.2.5**Erreur ! Source du renvoi introuvable.** for further information about the meta-model representation.

11.3 Error model as Assumption or Guarantee

With the current meta-model version, it is possible to express, if a specified error model is a guaranteed behavior of the architectural element under consideration (e.g. gathered bottom up through several safety analysis steps) or an assumption of the (possibly not yet available) implementation (top town).

To do this, the attribute “errorModelKind” of the class “ErrorModelType” must be set accordingly.

11.4 Error model pattern 1 – Separation of application layer and application environment

11.4.1 Introduction

This error model pattern allows engineers to reason independently about the malfunctions in the software components and the underlying system. For this purpose, the error model creates a clear cut in the error model between application layer and application environment (ECU-hardware, basic software and RTE). The malfunctions, their propagation (or isolation) and their compound probability distribution defined within the error model contribute to a *safety contract* between the application software and the application environment. This cuts the two parts of the systems, so that one can reason about malfunctions independently.

11.4.2 Modeling approach

Figure 52 shows the error model corresponding to the system model mentioned in *Figure 51*.

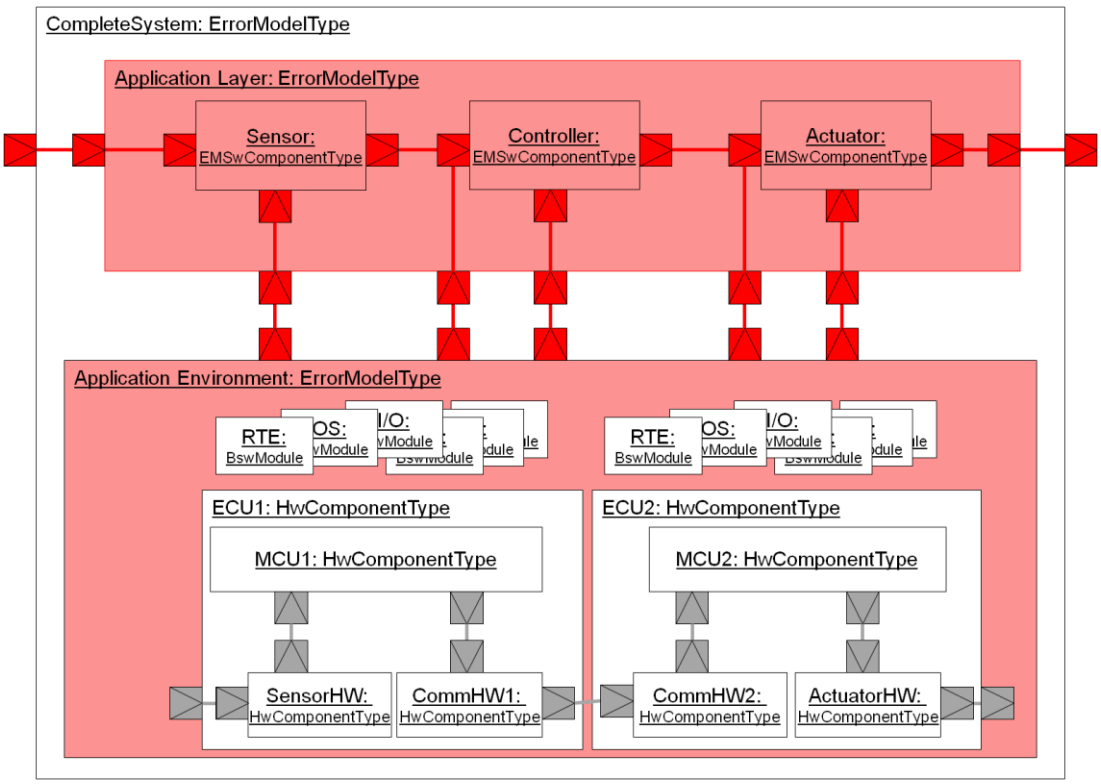


Figure 52 : ErrorModel corresponding to Refined System model

To separate the application layer and the application environment, we do the following steps:

1. We define one error model named “application layer” consisting of all application SWCs and all ECUs including BSW in an application environment. *Figure 52* shows the error model for our example. The boxes in light red are the error model types and error models for the different components.

2. We argue about the different malfunctions from the application environment and how they affect the application software. The set of the different malfunctions, e.g., computing and communication anomalies, in the application environment define the failure ports of the error model of application environment. The failure ports of the application layer match exactly those of the application environment.

   In our example, we identified five malfunctions in the application environment:

   - A computing anomaly in the Sensor SWC,
   - A communication anomaly from the Sensor SWC to the Controller SWC,
   - A computing anomaly in the Controller SWC,
   - A communication anomaly from the Controller SWC to the Actuator SWC,
   - A computing anomaly in the Actuator SWC.

   These five malfunctions are depicted as the five failure ports in *Figure 52*.

3. In a next step, we argue how the error behavior of the application layer shall look like. The error behavior is modeled by horizontal and vertical FaultFailurePropagationLinks.

   Vertical propagation links describe the faults from the application environment that can induce faults in the SWC. The vertical propagation links always link the application layer’s ports and the failure ports of the different software components.

   In our example, the vertical propagation links link the five malfunctions listed above to the affected software component in the application layer.

   Horizontal propagation links describe how errors can propagate from one software component to another on the same level. Every horizontal propagation link is backed by a concrete physical information flow through the application environment (BSW, hardware or communication system). However, the failure propagation due to these concrete data flows is only depicted by the horizontal links. In our example, there are four horizontal propagation links.

   The two internal propagation links model malfunctions that are propagated to the next software components, i.e., a sensor failure is propagated to the controller and a controller failure is propagated to the actuator. The other two horizontal propagation links model the propagation of external malfunctions to internal malfunctions and vice versa.

In these steps, we have followed this general rule for the composition:

To cut the system reasonably, we restrict the direction of fault propagation. Faults propagate only from the software platform to the software components, but never the other way around.

The general principle of failure model decomposition underlying the separation the application layer and the application environment is suitable for any decomposition of a system.

11.4.3 Special case: horizontal error propagation prevented by application environment

In most cases, faults in one software component propagate to another software component without fault detection or fault handling in the application environment. For those cases, the fault propagation is modeled in the error model with a horizontal fault-failure propagation link (see respective description in the meta-model chapter 11) from one software component to the other.

If the application environment has safety mechanisms that handle failures of a SWC, this safety mechanism must be reflected in the application layer of the error model. In this case, horizontal error propagation between two application software components is filtered, as shown in the example below. To reflect the safety mechanism, that is realized by the application environment, in the application layer of the error model, the error model is enriched by an ErrorModelType called *"Virtual SM"*.

Example:

SWC A computes data and sends this data to SWC B through the application environment. The application environment has a safety mechanism that can detect if the data is within a defined range and reacts, so that the data out of bounds is not forwarded.

Assume an error occurs in the SWC A and SWC A sends faulty data to SWC B, e.g. the data is out of a valid range. In this case, the failure mode "data out of range" would directly propagate from SWC A to SWC B. However, if the application environment is able to detect this failure, failure mode is isolated by the mentioned *"Virtual SM"* and does not propagate towards SWC B accordingly.

The *Figure 53* below shows the error model for the described situation. The ErrorModelType *"Virtual SM"* has been introduced in the error model, and the external failure of this error model type does not contain the failure mode "data out of range", because it has been filtered by the application environment.

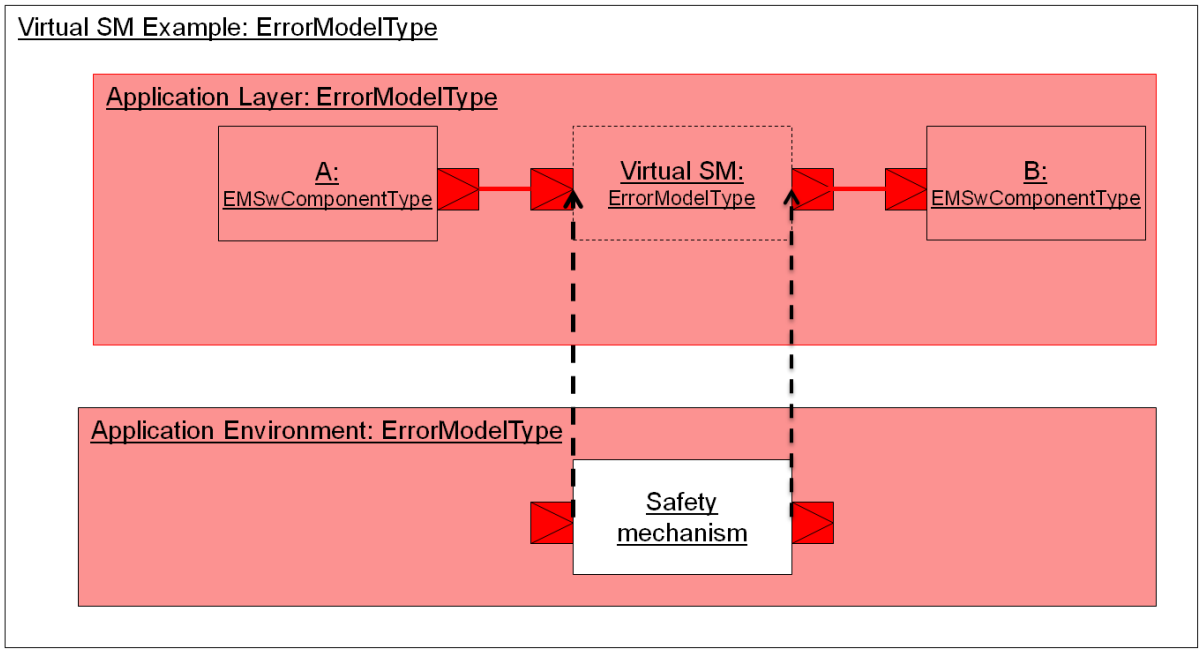


Figure 53 : Example of Error Model modeling Virtual Safety Mechanism

11.4.4 Error Model as Safety Contract

The error model pattern proposed above has the goal to contribute arguments to show the effectiveness of a safety concept. Thus, we propose to see it as part of a *safety contract*. Via the model, the application developer has the ability to specify how the application environment shall or shall NOT affect the execution of its application software. For instance, assuming the error model specifies that memory corruptions in the RAM shall not propagate to the application software (e.g. by storing the same value multiple times in the RAM to detect manipulation). In this case, the safety engineer can use this information to argue about the effectiveness of its safety concept, because he assumes that memory corruptions is not visible at application software level and can therefore not propagate towards possible malfunctions or hazards at top level.

11.4.5 Modeling of Separation of Application Layer and Application Environment

In *Figure 54*, we show how we model the separation of the application layer and the application environment with the meta-model described in chapter 10.

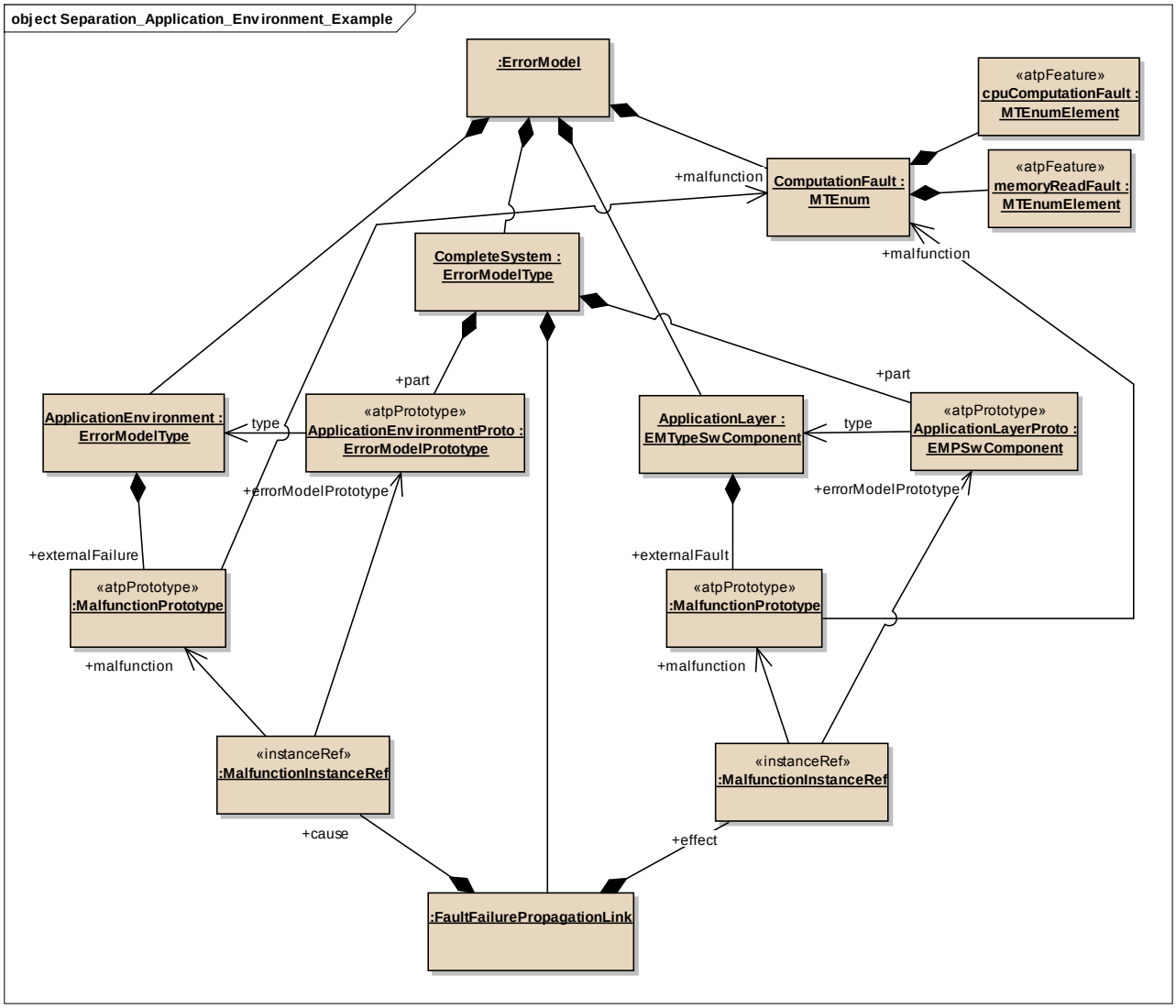


Figure 54 : Example of modeling of the separation between the application layer and the application environment

The error model contains an ErrorModelType for the complete system, which itself is composed of the Application Environment and the Application Layer. In this example, we omit the special case *"Virtual SM"* as mentioned in chapter 11.4.3.

In this diagram, we also show one faultfailurePropagationLink that models a ComputationMalfunction that originates in the application environment and propagates a fault to the application layer.

In the upper right of the diagram, we define computation faults to be either computation faults due to the CPU or due to invalid memory reads.

11.5 Error model pattern 2 – Quantitative Failure Analysis

Motivation:

When doing a safety analysis on application software, it is practically infeasible to analyze the fault propagation and causes of faults down to its root causes in hardware components. Instead a safety analysis on application software ends its fault propagation modeling at the level of the application environment. This cuts the fault propagation modeling at this point with application-specific assumptions about the failure probabilities and fault dependencies of the faults of the application environment. Safety engineers of operating system providers and hardware vendors then continue modeling the fault propagation from hardware components to the level of the application environment. The application software propagation model and application environment propagation model can then be combined to give a complete fault propagation model from hardware components to application software.

However, due to the intermediate split of the fault propagation model, the faults at the intermediate level might still have some dependencies to each other, i.e., they are not all independent of each other.

For a formal description and mathematical view of this statement, we view each intermediate fault as a random variable. Considering this mathematical view, we require means to describe the dependent probability distribution of the random variables in our meta-model.

We assume that the faults only result in Boolean values with a probability, possibly depending on other faults. However, the faults' probability cannot depend on other influences, e.g., time or state of the system.

Motivating Example:

We illustrate the requirement for the SAFE meta-model with the following example:

Assume the three faults, "Communication Failure on Logical Network Connection", "CPU calculation error" and "Memory corruption", can affect a given application software component. We further assume that the CPU calculation error and the memory corruption are independent of each other. The Communication Failure on Logical Network Connection can be caused by the memory corruption (in the network driver's software buffer), or a CPU calculation error (in the network driver's code) or a network transmission error which is not visible to the SW component directly[1]. We use a Bayesian network to describe the failures' probability distribution.

The dependencies and failure rates are captured in the Bayesian network as follows:

[1] Errors caused by the communication software stack are not considered here, as they do not contribute to quantitative failure analysis

First, the dependencies of the faults are as depicted here:

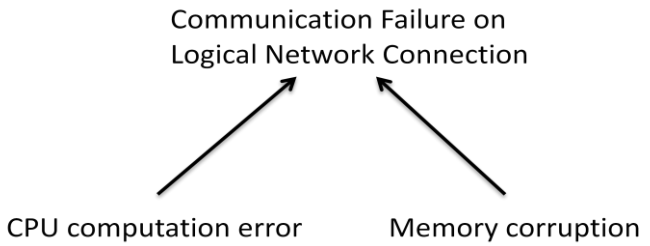


Figure 55 : Fault dependencies of the motivating example

Capturing the dependencies, the Bayesian network consists of three nodes, corresponding to the three faults “Communication Failure on Logical Network Connection”, “CPU calculation error” and “Memory corruption”. The nodes are connected by edges, as shown in Figure 1.

Each node is equipped with a table for the failure probability depending of the fault assignment of its dependencies. If the fault A occurs, the fault assignment for A is 1, otherwise 0.

The table on the right describes the probability of the Communication Failure on Logical Network Connection. As the communication failure depends on the computation error and the memory corruption, we provide its probability depending on the occurrence of these two faults. For example, if the two faults do not occur (fault assignment: comp. err. = 0; mem. corr. = 0), the probability is 0.01, e.g., due to a network transmission error. When only a memory corruption fault occurs (fault assignment: comp. err. = 0; mem. corr. = 1), the probability is 0.51. When a CPU computation occurs (fault assignment: comp. err. = 1; mem. corr. = 0), the probability is 0.26. When both errors occur (fault assignment: comp. err. = 1; mem. corr. = 1), the probability is 0.76.

Further, we assume that the failure probability for a CPU computation error is 10^{-7} / hour (FIT = 100); and for a memory corruption is 10^{-6}/ hour (FIT = 1000). As the CPU computation error and the memory corruption are independent of other nodes, their probabilities are captured in two degenerated tables, as shown in *Table 10****Table 1*****Erreur ! Source du renvoi introuvable.** and *Table 11*.

Fr(Fault occurs)
Exponential(100)

Table 10 : Probability for CPU computation error

Pr(Fault occurs)
Exponential(1000)

Table 11 : Probability for memory corruption

Description of the Meta Model:

In this section, we describe the meta-model to describe these Bayesian networks for our fault modeling.

The formal description of a Bayesian network is well established in the literature, and here, we simply cite one of the definitions in the literature:

“*Formally, Bayesian networks are directed acyclic graphs whose nodes represent random variables. Edges represent conditional dependencies; nodes that are not connected represent*

variables that are conditionally independent of each other. Each node is associated with a probability function that takes as input a particular set of values for the node's parent variables and gives the probability of the variable represented by the node. For example, if the parents are m Boolean variables then the probability function could be represented by a table of 2^m entries, one entry for each of the 2^m possible combinations of its parents being true or false."

[Wikipedia, 2013. http://en.wikipedia.org/wiki/Bayesian_network]

We model this in the meta-model as follows: The Bayesian network is captured in a QuantitativeErrorModel object (see sections 10.2.4 and 10.2.5**Erreur ! Source du renvoi introuvable.** for more detailed information about the meta-model) with an aggregation of MalfunctionFailureProbability objects. Each MalfunctionFailureProbability object represents a node in the Bayesian network. The directed edges of the Bayesian network are represented in the meta-model by the dependency references to other MalfunctionFailureProbability. In other words, each MalfunctionFailureProbability contains an ordered set of references to the MalfunctionFailureProbability objects that it directly depends on. The meta-model is only valid if the dependency graph in the meta-model form a directed acyclic graph.

The MalfunctionFailureProbability object is optionally linked to a Malfunction of the meta-model, and hence is linked to the system model and further to safety requirements. A MalfunctionFailureProbability without a link to a Malfunction is a node in the Bayesian network to model the dependencies without any correspondence to a concrete Malfunction.

The table to each node is modeled by the ordered association of FailureProbability objects. Given the n dependent MalfunctionFailureProbability object, d_0, d_1 … d_{n-1}, and an ordered association of FailureProbability objects, f_0, f_1, …, $f_{(2^n)-1}$, the failure probability associated to the fault assignment [d_0 -> b_0, d_1 -> b_1, …, d_{n-1}->b_{n-1}] for n Boolean values b_0, b_1, …, b_{n-1} is f_m where m is number to the binary value $b_{n-1}…b_1b_0$. For example, to the fault assignment [d_0 -> 1, d_1->1, d_2->0, d_3->1], the binary value is $1011_2 = 11_{10}$, and hence, the failure probability is f_{11} of the ordered association of probability.

The table of a node must cover all assignments, i.e., a node with n dependencies must have 2^n rows with distinct assignments. A current shortcoming of the meta-model is that it requires 2^n entries for the failure probability, even if more compact representation might exist.

Although the meta-model allows to model time-dependent failure probabilities, we limit the FailureProbability objects to only be constant values in the current proposal, i.e., the relation "distribution" of FailureProbability refers to an instance of type "Uniform". This limitation could possibly be removed once the semantics of time-dependent failure probabilities and its interaction with Bayesian networks is better understood.

Example of Use:

The example from our motivation above would be modeled as follows:

The system model contains three malfunctions m_0, m_1 and m_2 with descriptions:

```
Given MalfunctionPrototype mp0, mp1, mp2;
mp0.genericDescription = "Communication Failure on Logical Network Connection";
mp1.genericDescription = "CPU calculation error";
mp2.genericDescription = "Memory corruption";
```

The Bayesian Network with three nodes n_0, n_1 and n_2 is modeled as follows:

```
QuantitativeErrorModel q;
MalfunctionFailureProbability n0, n1, n2;
q.malfunctionFailureProbability = [n0, n1, n2];
```

The three nodes are linked to the malfunctions:

```
n0.malfunction = mp0;
n1.malfunction = mp1;
n2.malfunction = mp2;
```

The dependencies, n_0 is dependent on n_1 and n_2, are modeled by

```
n0.dependency = [n1, n2];
n1.dependency = [];
n2.dependency = [];
```

All three nodes now have tables for their failure rate distribution:

```
n2.probability = [FailureProbability(distribution=Exponential(lambda=100))];
n1.probability = [FailureProbability(distribution=Exponential(lambda=1000))];
n0.probability =
      [FailureProbability(distribution=Uniform(value=0.01)),FailureProbability(distribution=Uniform(
value=0.51)),

FailureProbability(distribution=Uniform(value=0.26)),FailureProbability(distribution=Uniform(value=0
.76))];
```

11.6 Error Model pattern 3 - Independent failures, Error propagation probability

Independence between individual external Fault/external Failures defined for an ErrorModelType is implicitly assumed. According to the completeness of the error model (e.g. knowing the origin for the presence of an external fault), possible common causes can be detected within the complete error model, thus the independence of individual faults/failures can be calculated.

In case of an incomplete error model (e.g. the error model defined for one singular software component without the knowledge of the interfacing software components and the runtime environment), the independence is implicitly assumed, as long as no dependent failure information can be derived from the (incomplete) error model (e.g. as described for the step of quantitative failure analysis in section 11.5).

In some cases, a preceding error may not directly lead to the occurrence of the subsequent error. Thus, it must be possible to express this fact by adding probability information for a propagation link between two errors. This case can be expressed by primitive means of the SAFE error model capability, and will be shown in the following example:

Assume that in 10% of the cases, a memory error in partition A directly leads to a memory error in partition B and vice versa. For this purpose, we would model an error model as an ErrorModelType “EMT_Memory_Partition” as follows:

```
ErrorModelType EMT_Memory_Partition: // assumed to be symmetric for A and B
```

- externalFailures:
  - ExtFailure_memory_error // the error caused by this partition
- externalFaults:
  - ExtFault_memory_error // external error affecting the correct behavior of the partition)

```
EastADLErrorBehavior EMB_Memory_Parition:
```

- errorModel: EMT_Memory_Parition
- internalFaults:
  - IntFault_memory_error

In addition, we define an ErrorModelType “EMT_Memory”, with the following details:

```
ErrorModelType “EMT_Memory”:
```

- externalFailures:
  - ExtFailure_memoryA_error
  - ExtFailure_memoryB_error
- externalFaults:
  - ExtFault_memoryA_error
  - ExtFault_memoryB_error

```
EastADLErrorBehavior EMB_Memory
```

- errorModel: EMT_Memory
- internalFaults:
  - IntFault_memory_protection_error

The internal fault of the “Memory” has a failure rate of 0.1. In addition, the error propagation within “Memory” is given as follows:

```
MalfunctionFailureProbability mfr_memory_protection_error:
```

- malfunction: IntFault_memory_protection_error
- probabilities:
    - FailureProbability(distribution=Uniform(value=0.1))

```
EMB_MemoryX.formula = "ExtFailure_memoryX_error = ExtFault_memoryX_error AND IntFault_memory_protection_error"
```

At ECU level, we have the following three ErrorModelPrototypes:

```
ErrorModelType EMT_ECU:
```

- parts:
    - ErrorModelPrototype EMP_PartitionA:
        - type: EMT_Memory_Partition
    - ErrorModelPrototype EMP_PartitionB:
        - type: EMT_Memory_Partition
    - ErrorModelPrototype EMP_Memory:
        - type: EMT_Memory

Via FaultFailurePropagationLink, we connect the external faults/failures:

```
EMT_ECU.faulFailureConnectors:
```

- FaultFailurePropagationLink FFPL1:
    - cause: EMP_Memory.ExtFailure_memoryA_error
    - effect: EMP_ParitionA.ExtFault_memory_error
- FaultFailurePropagationLink FFPL2:
    - cause: EMP_Memory.ExtFailure_memoryB_error
    - effect: EMP_ParitionB.ExtFault_memory_error
- FaultFailurePropagationLink FFPL3:
    - cause: EMP_ParitionB.ExtFailure_memory_error
    - effect: EMP_Memory.ExtFault_memoryB_error
- FaultFailurePropagationLink FFPL4:
    - cause: EMP_ParitionA.ExtFailure_memory_error
    - effect: EMP_Memory.ExtFault_memoryA_error

Please be aware that the example above introduces loops in the error model (e.g. EMP_PartitionA.ExtFailure_memory_error is transitively related via error propagation with EMP_PartitionB.ExtFailure_memory_error, which itself is again (transitively) related with EMP_ParitionA.ExtFailure_memory_error. However, we decided to neglect this fact in the description here, as it is not of importance to understand the topic of “error propagation probability” described in this section. The respective fault tree for the failure “EMP_ParitionA.ExtFailure_memory_error” is shown hereafter:

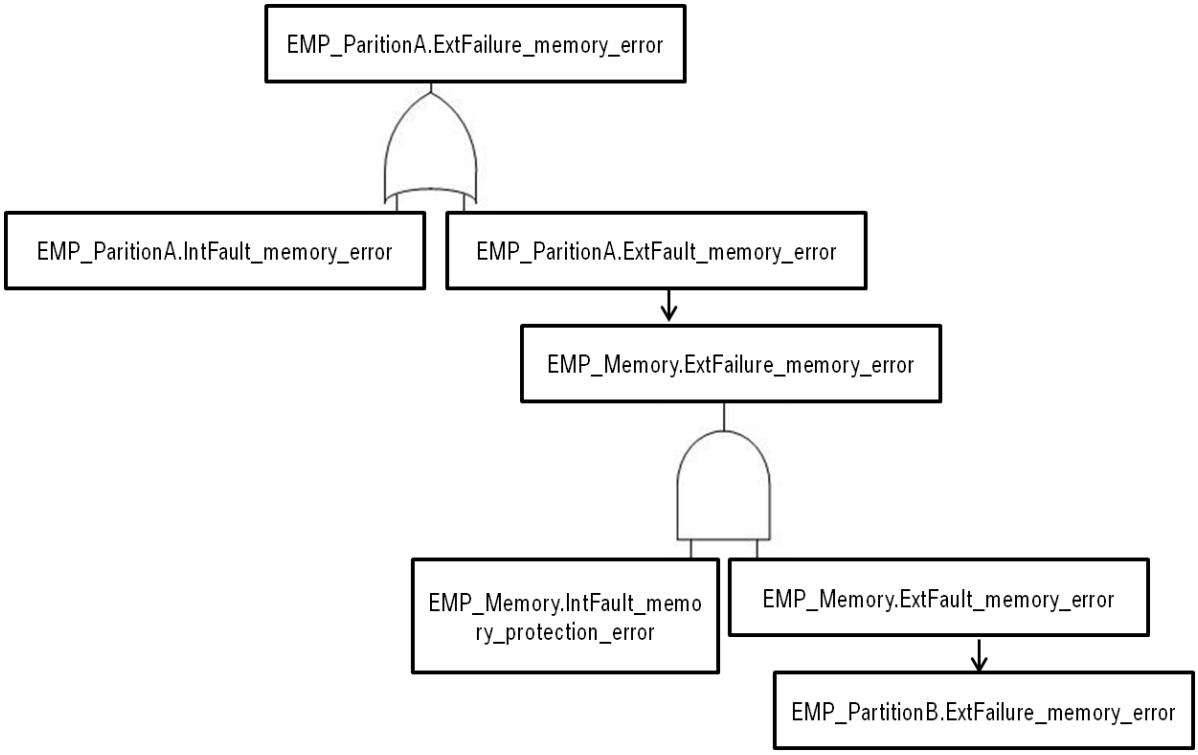


Figure 56 : Fault dependencies of the previously described example [2]

[2] The tree is stopped at EMP_ParttitionB.ExtFailure_memory_error, as the relevant portion (how to express error propagation probability) is already covered in the exemplary fault tree.

11.7 Error model pattern 4 – Separation of Hardware and Software

The separation of hardware and software via a dedicated Hardware Software Interface (HSI) is strongly linked to the abstraction view on the system, and in particular to the representation of the technical safety concept where software and hardware interacts together. On the top of the ISO26262 requirements to identify the HSI interface at the system level, the failure propagation between the hardware and software shall be defined consistently with HSI definition.

Using AUTOSAR scheme, as proposed in the Error model pattern 1 defined in chapter 11.4, the application environment interfaces the application layer via RTE interfaces abstracting the ECU's hardware and BSWs. The application environment, also named AUTOSAR execution platform, is constituted of hardware elements and AUTOSAR software infrastructure such as services, HCAL layer, etc, and MCAL layer. The MCAL software driver interfaces the hardware controller and the peripherals using specialized hardware registers. These hardware registers are physical implementation of the HSI, but do not fit to the abstraction level of the RTE interface.

On the other hand, if EAST-ADL is used for application layer description, the application environment is simplified as the RTE is not visible, as virtual function bus is abstracted by flow port connector. For this abstraction level, the main relevance for HSI is able to define relation between hardware elements of the ECUs and software elements used in the Functional Design of EAST-ADL, embracing the hardware abstraction functionality as proposed in WT3.2.2. The SAFE meta-model declares a *HardwareSoftwareInterfaceElement*, allowing to reference port interfaces between the EAST-ADL elements *FlowPort* from a *DesignFunctionPrototype* playing the role of software function and *HWPin or HWPort* from a *HardwareComponentPrototype* playing the role of hardware element Moreover, the software function as *DesignFunctionPrototype* shall be allocated on a hardware processing units as *HardwareComponentPrototype* using the *Allocation* element of EAST-ADL.

For error propagation purpose, the system error model is built and split into two perspectives, one for the hardware and one for the software. For interfacing the error model, each perspective declares its external faults and/or external failures to describe the incoming faults and propagating failures of the individual perspective. They are modeled as *MFPFunctionPort* for software and *MPFHWPin* or *MPFHWPort* for hardware as declared in 10.2.6. They are linked by a *FaultFailurePropagationLink,* referencing the port malfunctions and the source element of this propagation, respectively an *HSI* element or an *Allocation* net.

We propose below a simple example of a software component "DriverADC" being allocated on a "Mircrontroller" having an HIS for an Analog acquisition of a signal. The error model description limited to an ADC input from a sensor can be built as following:

Error model of SW component

```
ErrorModelType EMT_DriverADC: // assumed to be error model of the software component
```

- externalFailures:
  - ExtFailure_Sw_GetADCValue
- externalFaults:
  - ExtFault_InSw_HSIAdcSens // assumed to be the HSI target
  - InSwDC_AllocationMCError // assumed to be the allocation net target

```
EastADLErrorBehavior EMB_DriverADC
```

- errorModel: EMT_DriverComp
- internalFaults: (empty)
- Formula
  - ExtFailure_Sw_GetADCValue.ERROR = ExtFault_ExtFault_InSw_HSIAdcSens.ERROR OR InSwDC_AllocationMCError.ERROR

Error model of HW component

```
ErrorModelType EMT_Microntroller: // assumed to be error model of the hardware component
```

- externalFailures:
  - ExtFailure_OutHw_HSIAdcSens // assumed to be the HSI source
  - OutAllocationMCError // assumed to be the allocation net source
- externalFaults:
  - ExtFault_HwMC_InADCx

```
EastADLErrorBehavior EMB_Microntroller
```

- errorModel: EMT_Microntroller
- internalFaults:
  - MC_Error
- Formula
  - OutAllocationMCError.ERROR = MC_Error.ERROR
  - Extfailure_OutHw_HSIAdcSens.ERROR = ExtFailure_OutHw_HSIAdcSens

At top level as a system we declare the parts and connect them:

```
ErrorModelType EMT_System:
```

- parts:
  - ErrorModelPrototype EMP_DriverADC:
    - type: EMT_DriverADC
  - ErrorModelPrototype EMP_Microntroller:
    - type: EMT_Microntroller

```
EMT_System.faulFailureConnectors:
```

- FaultFailurePropagationLink Allocation1:
  - cause: EMP_Microntroller.OutAllocationMCError
  - effect: EMP_DriverComp.InSwDC_AllocationMCError
- FaultFailurePropagationLink HSI1:
  - cause EMP_Microntroller.ExtFailure_OutHw_HSIAdcSens
  - effect: EMP_DriverComp.ExtFault_InSw_HSIAdcSens
  -

As it is depicted in the *Figure 57* with the use of the above *ErrorModelType DriverADC* and *Microcontroller*, in a similar example, the error propagation from an incoming hardware failure to software is swap onto the interface of the processing unit (e.g. microcontroller) through the corresponding HSI malfunction port *ExtFailure_OutHw_HSIAdcSens*. The internal fault of the microcontroller propagates through the *OutAllocationMCError* malfunction port to the software error model, alike for the hardware software interface.

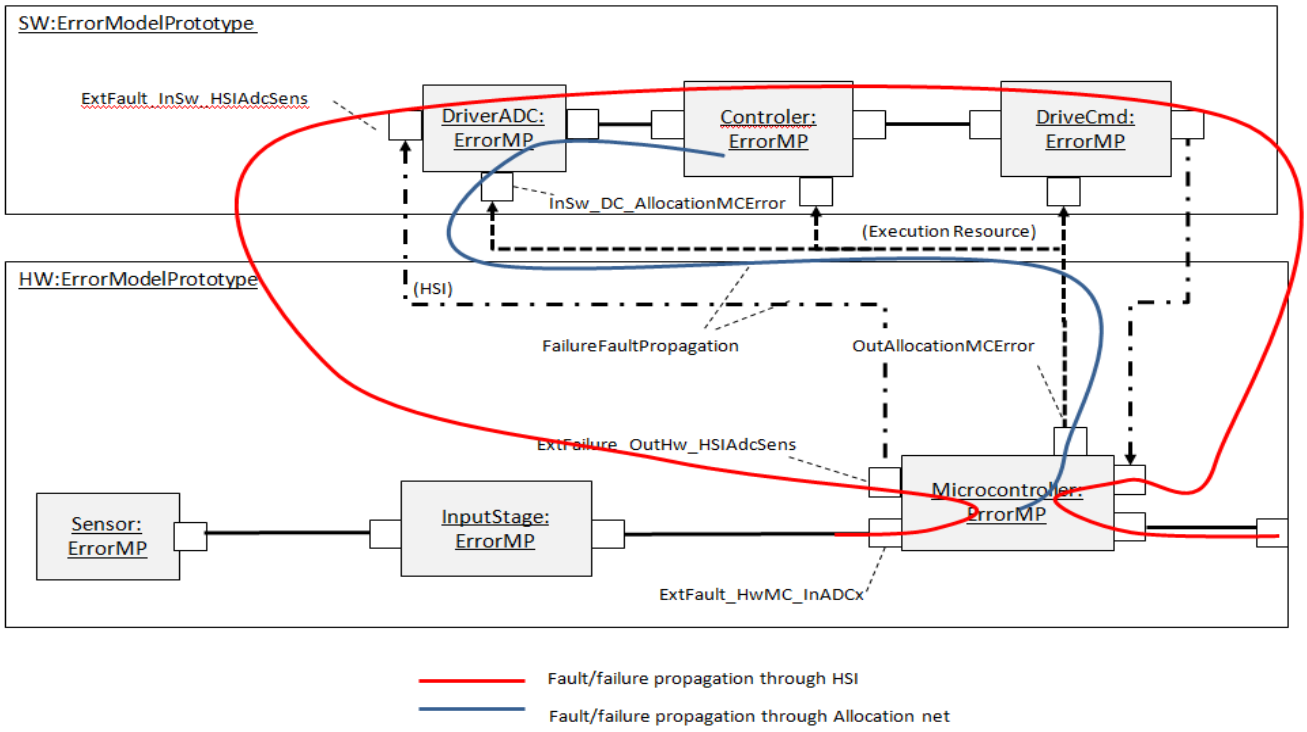


Figure 57 : Hardware – Software Fault propagation

12 Conclusions and next steps

This document is intended to provide information about a proposal for extension of meta-model for error failure and propagation analysis that shall be compliant with the requirements and main concepts addressed by ISO26262.

Also the problematic of distributed development and impact of the fault propagation through the entire item is highlighted in the document. To solve this issue an approach based on pattern-based safety contracts is proposed.

A solid base of information was provided in the document concerning two relevant fault and propagation languages candidate: HiP-HOPS and AltaRica. A final pros and cons analysis did not permit to choose between them. As the priority was to have something simple for the end user, we came to the conclusion that a simplified SAFE language capable to be transformed transparently either in HiP-HOPS or AltaRica was the best compromise. So, we elicited requirements for the grammar and for the semantics of a simplified SAFE language that are now available in the document.

Since it was an objective to reuse EAST-ADL as much as possible, the current version of EAST-ADLV2.1 and more particularly the ErrorModel package was presented in a first step. Then the main gaps compared to our needs were highlighting and finally a proposal for Meta-model extensions was formulated.

Moreover to correctly use and implement our meta-model proposal, a dedicated example with some application rules was provided.

Even if some discussions were already performed between the most relevant work tasks having dependencies with WT3.3.1, the proposed meta-model enhancements for error failure and propagation analysis has to be synchronized with the meta-model extensions of WT3.2.2, WT3.2.1 and WT3.1.1 in order to harmonize the model properties for the description of re-use related information. As a consequence a new release of this document will be performed including clarification of Hardware Software interface.

The next deliverable D331b will provide documentation about Methods and Tool specification for analysis of qualitative and quantitative cut-sets issued from Error Failure propagation analysis. In the document D331a most relevant safety analysis techniques recommended by ISO26262 were assessed and final considered methods for D331b will be qualitative FMEA, quantitative FMEDA and FTA.

13 Glossary useful for D331a document

Hazard	Potential source of harm caused by malfunctioning behavior of the item.
Malfunctioning behavior	Failure or unintended behavior of an item with respect to its design intent.
Fault	Abnormal condition that can cause an element or an item to fail.
Error	Deviation between a computed observed or measured value or condition from theoretically correct value or condition.
Failure	Termination of the ability of an element, to perform a function as required.
Systematic fault	Fault whose failure is manifested in a deterministic way that can only be prevented by applying process or design measures.
Systematic failure	Failure related in a deterministic way to a certain cause, that can only be eliminated by a change of the design or of the manufacturing process, operational procedures, documentation or other relevant factors.
Random hardware failure	Failure that can occur unpredictably during the lifetime of a hardware element and that follows a probability distribution.
Malfunction	Malfunction is a failure or unintended behavior of the item or element of the item that has the potential to propagate.
Horizontal error propagation	Propagation of errors inside a same architectural level.
Vertical error propagation	Propagation of errors through different architectural levels.
Informal Notation	Description technique that does not have its syntax completely defined.
Semi-formal Notation	Description technique whose syntax is completely defined but whose semantics definition can be incomplete.
Formal Notation	Description technique that has both its syntax and semantics completely defined.
Application environment	The application environment includes all entities, in which the application layer is executed. This includes the ECU hardware, the basic software and RTE.
Application layer	The set of all Software Components.
Basic Software	The basic software implements common available services and ECU provided resources.
Virtual fault SWC	A Software Component in the error model that represents a safety mechanism in the application environment. It does not occur in the system model, but only occurs in the error model for software safety analysis.
Composition	A component is refined into subcomponent that together define the behavior of the surrounding component.

14 Abbreviations used in D331a document

ASIL	Automotive Safety Integrity Level
ATTEST	Advancing Traffic Efficiency and Safety through Software Technology
AUTOSAR	AUTomotive Open System ARchitecture
BCM	Body Control Management
BDD	Binary Decision Diagram
CAE	Computer Aided Engineering
CAN	Controller Area Network
CCF	Common Cause of Failure
CESAR	Cost-Efficient methods and processes for SAfety Relevant embedded systems
COTS	Component Off the Shelf
CPU	Central Processing Unit
DM	Degradation Mode
DRIS	Distributed, Reliable and Intelligent control and cognitive Systems
E/E	Electronic and Electrical
EAST-ADL	Electronic Architecture and Software Tools- Architecture Description Language
ECU	Electronic Control Unit
EMC	Electro Magnetic Compatibility
ETA	Event Tree Analysis
FDA	Functional Design Architecture
FIT	Failure In Time
FME(D)A	Failure Mode Effect and Diagnostic Analysis
FMEA	Failure Mode and Effect Analysis
FTA	Fault Tree Analysis
GUI	Graphical User Interface
HAZOP	HAZard and OPerability study
HDA	Hardware Design Architecture
HiP-HOPS	Hierarchically Performed Hazard Origin & Propagation Studies
HRC	Heterogeneous Rich Components
HSI	Hardware Software Interface
HW	Hardware
IP	Intellectual Property
LFM	Latent Fault Metric
LH	Limp Home
MAENAD	Model-based Analysis & Engineering of Novel Architectures for Dependable electric vehicles
MCU	Microcontroller Unit
OEM	Original Equipment Manufacturer
Open-PSA	Open Probabilistic Safety Assessment
RAM	Random Access Memory
RBD	Reliability Block Diagram
RSL	Requirements Specification Language
RTE	Run Time Environment
SAFE	Safe Automotive soFtware architEcture
SM	Safety Mechanism
SPEEDS	Speculative and Exploratory Design in Systems Engineering
SPFM	Single Point Fault Metric
SW	Software
SWC	Software Component
TCM	Top Column Module
WT	Work Task
XML	Extensible Markup Language

15 References

[1] International Organization for Standardization: ISO 26262 Road vehicles - Functional safety. (2011)

[2] Project ATESST2: ATESST2 Partners. Review of relevant Safety Analysis Techniques, http://www.atesst.org/home/liblocal/docs/ATESST2_Deliverable_D2.1_A3.2_V1.1.pdf

[3] http://www.itemuk.com/assets/docs/ToolKit_Manual.pdf

[4] SPEEDS Consortium: SPEEDS Meta-model Syntax and Draft Semantics, D2.1c. (2007)

[5] Project CESAR: CESAR Partners. RE Language Definitions to formalize multi-criteria requirements V2, D_SP2_R2.2_M2, http://www.cesarproject.eu/fileadmin/user_upload/CESAR_D_SP2_R2.2_M2_v1.000_PU.pdf

[6] SPEEDS L-1 Meta-Model, SPEEDS WP2.1 Partners, SPEEDS Project Deliverable D2.1.5, Revision 1.0.1, May 2009, http://speeds.eu.com/downloads/SPEEDS_Meta-Model.pdf

[7] Hungar, H.: Compositionality with Strong Assumptions. In Proceedings of the 23rd Nordic Workshop on Programming Theory. (2011) 19–21

[8] Damm, W., Josko, B., Peikenkamp, T.: Contract based ISO CD 26262 safety analysis. SAE Technical Paper 2009-01-0754, 2009, doi:10.4271/2009-01-0754 (2009)

[9] University of Hull, DRIS research group. The Definitive Guide to the HiP-HOPS XML Input File Format, HiP-HOPS XML Format.doc

[10] Yiannis Papadopoulos, Martin Walker, University of Hull “Qualitative temporal analysis: Towards a full implementation of the Fault tree Handbook”, Control Engineering Practice, Vol.17 Issue 10, Elsevier Editions, 2009.

[11] Project ATESST2: ATESST2 Partners. EAST-ADL update suggestions for Safety Analysis support, http://www.atesst.org/home/liblocal/docs/ATESST2_Deliverable_D3.1_A3.2_V1.1.1.pdf

[12] Yiannis Papadopoulos, Ian Wolfort, Martin Walker, University of Hull “Capture and Reuse of composable failure patterns” , International Journal of Critical Computer Based Systems, Vol 1, Nos. 1/2/3 2010

[13] G. Point. AltaRica: Contribution à l’unification des methods formelles et de la Sûreté de fonctionnement. PhD thesis, Université Bordeaux 1, 2000.

[14] A. Arnold, D. Bégay, and P.Crubillé. Construction and analysis of transition systems with MEC. World Scientific Publishers, 1994.

[15] A. Rauzy: A New Methodology to Handle Boolean Models with Loops In *IEEE Transactions on Reliability*. IEEE Reliability Society. Vol. 52, Num. 1, pp 96–105, 2003.

[16] T. Prosvirnova, and A. Rauzy: Système de Transitions Gardées : formalisme pivot de modélisation pour la Sûreté de Fonctionnement. In J.F. Barbet ed., *Actes du Congrès Lambda-Mu 18*. Octobre, 2012.

[17] Marc BOUISSOU: Gestion de la complexité dans les etudes quantitative de sûreté de fonctionnement de systems. Collection EDF R&D aux éditions LAVOISIER*

[18] Chen, D., Johansson, R., Lönn, H., Papadopoulos, Y., Sandberg, A., Törner, F., Törngren, M.: Modelling Support for Design of Safety-Critical Automotive Embedded Systems. In: Proceedings of SAFECOMP (2008)

[19] Safety Designer tool from Dassault System ; http://www.3ds.com/

[20] SIMFIA ; http://www.apsys.eads.net/en/17/Software

16 Acknowledgments

This document is based on the SAFE project in the framework of the ITEA2, EUREKA cluster programme Σ! 3674. The work has been funded by the German Ministry for Education and Research (BMBF) under the funding ID 01IS11019, and by the French Ministry of the Economy and Finance (DGCIS). The responsibility for the content rests with the authors.

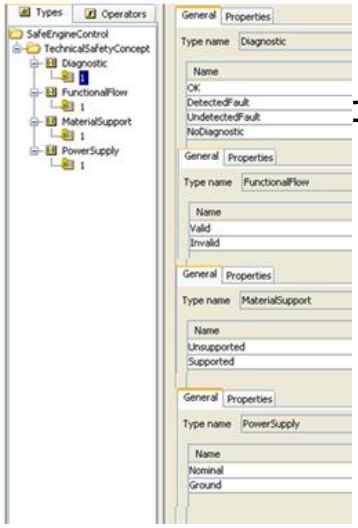
17 Annex A: Mapping between AltaRica and HiP-HOPS

Based on one example provided by Dassault System on SafetyDesigner 9, a mapping with HiP-HOPS was proposed by Continental-France.

Typing / event versus Failure Class

Altarica : Flow Typing

Hip-Hops : Failure Class



FC = DetectedFault

FC : UndetectedFault

FC = Fault

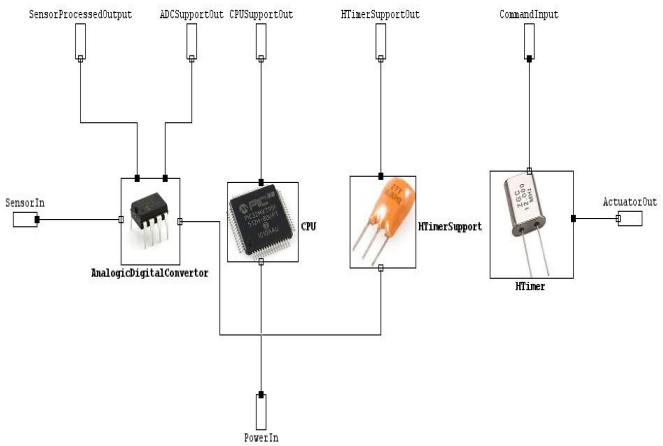
FC = Fault

ITEA 2 ~ 10039

Extract of the exemple

Altarica

```
node SafeEngineControl_TechnicalSafetyConcept_Hardware_MicroController
flow
  icone : [1, 2] : local;
  PowerIn : SafeEngineControl_T
  SensorIn : SafeEngineControl_T
  ActuatorOut : SafeEngineContrc
  CPUSupportOut : SafeEngineC
  CommandInput : SafeEngineCo
  SensorProcessedOutput : SafeE
  ADCSupportOut : SafeEngineC
  HTimerSupportOut : SafeEngine
sub
  HTimerSupport : SafeEngineCo
  HTimer : SafeEngineControl_Te
  CPU : SafeEngineControl_Tech
  AnalogicDigitalConvertor : SafeE
assert
  AnalogicDigitalConvertor.Senso
  SensorProcessedOutput = Anal
  AnalogicDigitalConvertor.Power
  CPU.PowerIn = PowerIn ;
  CPUSupportOut = CPU.CPUSu
  HTimer.CommandIn = Comman
  ActuatorOut = HTimer.Comman
  AD
  HT
  HT
edo
```



As HiP-HOPS interface is XML based; the next slide represent only concept and equation
With removing of sugar information <data> </data>
Knowing that XML file shall be generated from SAFE meta-model Class

ITEA 2 ~ 10039

Analogic Digital Convertor

Altarica

```
node SafeEngineControl_TechnicalSafetyConcept_Hardware_ADC
flow
  icone : [1, 2] : local;
  SensorIn : SafeEngineControl_TechnicalSafetyConcept_FunctionalFlow : in ;
  SensorOut : SafeEngineControl_TechnicalSafetyConcept_FunctionalFlow : out ;
  PowerIn : SafeEngineControl_TechnicalSafetyConcept_PowerSupply : in ;
  ADCSupport : SafeEngineControl_TechnicalSafetyConcept_MaterialSupport :
out ;
state
  State : {OK,KO};
event
  failure;
init
  State := OK;
trans
                State = OK |- failure -> State := KO;
assert
                if (State = OK and PowerIn = Nominal)
                then SensorOut = SensorIn and ADCSupport = Supported
                else SensorOut = Invalid and ADCSupport = Unsupported
extern
  law <event failure> = exponential(5.0E-5);

edon
```

HiP-HOPS

```
Component
        SafeEngineControll_...._ADC
        Ports
                Port  Input SensorIn
                Port Output SensorOut
                Port Input PowerIn
                Port Output ADCSupport

        Implementation
                FailureData
                        BasicEvents
                                Basic Event failure
                                    UnavailabityFormula F1
                                OutputDeviation FailureEq1

FailureEq1
OutputDeviation Fault-SensorOut
        FailureExpression
                Fault-PowerIn OR failure
        Fault-ADCSupport
        FailureExpression
                Fault-PowerIn OR failure

UnavailabilityFormula F1
      Constant  FailureRate 1e-3    //can be Poisson, ...
```


CPU ressource

Altarica

```
node SafeEngineControl_TechnicalSafetyConcept_Hardware_CPU
flow
  icone : [1, 2] : local;
  PowerIn : SafeEngineControl_TechnicalSafetyConcept_PowerSupply : in ;
  CPUSupport : SafeEngineControl_TechnicalSafetyConcept_MaterialSupport : out
;
state
  State : {OK,KO};
event
  failure;
init
  State := OK;

trans
                State = OK |- failure -> State := KO;
assert
                if (PowerIn = Nominal and State=OK) then (CPUSupport =
Supported & icone =2) else (CPUSupport = Unsupported & icone =1);
extern
  law <event failure> = exponential(1.0E-6);

edon
```

HiP-HOPS

```
Component
        SafeEngineControll_...._CPU
        Ports
                Port  Input PowerIn
                Port Output CPUSupport

        Implementation
                FailureData
                        BasicEvents
                                Basic Event failure
                                    UnavailabityFormula F2
                                OutputDeviation FailureEq2
                                ExportedDeviation*

FailureEq2
OutputDeviation Fault-CPUSupport
        FailureExpression
                Fault-PowerIn OR failure

 F2
UnavailabilityFormula
      Constant  FailureRate 1e-3    //can be Poisson, ...
```

ExportedDeviation* can be the construction of a logical Expression of a FailureClass to be reuse across perspective

Hardware Timer Support

Altarica

```
node
SafeEngineControl_TechnicalSafetyConcept_Hardware_HTimerMa
terialSupport
flow
  icone : [1, 2] : local;
  SupplyIn :
SafeEngineControl_TechnicalSafetyConcept_PowerSupply : in ;
  MaterialSupportOut :
SafeEngineControl_TechnicalSafetyConcept_MaterialSupport : out
;
state
  State : {OK,KO};
event
  failure;
init
  State := OK;
trans
                State = OK |- failure -> State := KO;
assert
                if (SupplyIn = Nominal and State=OK) then
(MaterialSupportOut = Supported & icone=2)else
(MaterialSupportOut = Unsupported & icone =1);
extern
  law <event failure> = exponential(3.0E-6);

edon
```

HiP-HOPS

Component
SafeEngineControll_...._HTimerMaterialSupport
Ports
Port *Input* SupplyIn
Port *Output* MaterialSupportOut

Implementation
FailureData
BasicEvents
Basic Event failure
UnavailabityFormula F3
OutputDeviation FailureEq3
ExportedDeviation*

FailureEq3
OutputDeviation
Fault-materialSupportOut
FailureExpression
Fault-SupplyIn OR failure

F3
UnavailabilityFormula
Constant FailureRate 1e-3 //can be Poisson, ...

ITEA 2 ~ 10039

Hardware Timer

Altarica

```
node SafeEngineControl_TechnicalSafetyConcept_Hardware_HTimer
flow
  icone : [1, 2] : local;
  CommandIn :
SafeEngineControl_TechnicalSafetyConcept_FunctionalFlow : in ;
  CommandOut :
SafeEngineControl_TechnicalSafetyConcept_FunctionalFlow : out ;
state
  State : {OK,KO};
event
  failure;
init
  State := OK;
trans
                State = OK |- failure -> State := KO;
assert
                if (State = OK)
                then  CommandOut = CommandIn & icone = 2
                else CommandOut = Invalid & icone = 1;
extern
  law <event failure> = exponential(1.5E-6);

edon
```

HiP-HOPS

Component
SafeEngineControll_...._HTimerMaterialSupport
Ports
Port *Input* CommandIn
Port *Output* CommandOut

Implementation
FailureData
BasicEvents
Basic Event failure
UnavailabityFormula F4
OutputDeviation FailureEq4
ExportedDeviation*

FailureEq4
OutputDeviation Fault-CommandOut
FailureExpression
Fault-CommandIn OR failure

F4
UnavailabilityFormula
Constant FailureRate 1e-3 //can be Poisson, ...

ITEA 2 ~ 10039

Extract of the exemple

Altarica

```
node SafeEngineControl_....._MicroController
flow
  icone : [1, 2] : local;
  PowerIn : SafeEngine….PowerSupply : in ;
  SensorIn : SafeEngine…Flow : in ;
  ActuatorOut : SafeEngine…Flow : out ;
  CPUSupportOut : SafeEngine…Support : out ;
  CommandInput : SafeEngine…Flow : in ;
  SensorProcessedOutput : SafeEngine…Flow : out ;
  ADCSupportOut : Safe….Support : out ;
  HTimerSupportOut : SafeEngine….Support : out ;
sub
  HTimerSupport : SafeEngine…Support;
  HTimer : SafeEngine…_HTimer;
  CPU : SafeEngine…..t_Hardware_CPU;
  AnalogicDigitalConvertor : SafeEngineControl_T…e_ADC;
assert
  AnalogicDigitalConvertor.SensorIn = SensorIn ;
  SensorProcessedOutput = AnalogicDigitalConvertor.SensorOut ;
  AnalogicDigitalConvertor.PowerIn = PowerIn ;
  CPU.PowerIn = PowerIn ;
  CPUSupportOut = CPU.CPUSupport ;
  HTimer.CommandIn = CommandInput ;
  ActuatorOut = HTimer.CommandOut ;
  ADCSupportOut = AnalogicDigitalConvertor.ADCSupport ;
  HTimerSupport.SupplyIn = PowerIn ;
  HTimerSupportOut = HTimerSupport.MaterialSupportOut ;
edon
```

HiP-HOPS

```
System
    SubSystem
    Components
        Component SafeEngineControl_....._MicroController
            Ports
                Port Input  PowerIn
                Port Input SensorIn
            Implementations Impl_SafeEngine…Controller
                FailureData
                    System  mySubComponents
                        Components
                            Component
                            HTimerSupport
                                Ports
                                Implementation
                                    FailureData
                                    ……
                            Component H_Timer
                            …
                        Lines
                            Line SensorADCLin
                            ….
```

On the top level a model has Perspective * that may includes several system

```
SensorADCLin
Line
    Type Directed
    Connections
        Connection
            Port.PowerIn
            PortExpression  CPU.PowerIn
        Connection
```

ITEA 2 ~ 10039